Introduction to dependent type theory

Ali Caglayan

April 16, 2019

Contents

1 Introduction

Dependent types have been around for a while. [[Introduction with citation]]. The fact that they haven't been used widely in programming and mathematics suggests that their exposition is in dire need of attention. This is one of the goals this dissertation aims to achieve. We also note that for type theorists, categorical semantics can be daunting and obscure. For mathematicians, computer scientific ideas seem out of reach.

- 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 Syntax

2.1 The difficulty with syntax

Syntax is difficult to handle rigorously. The syntax of type theory has a long history of proposed solutions and an even longer history of incorrect solutions. The main difficulty lies with the fact that syntax must acount for the deceptively subtle notions of variable binding, capture-free substitution and even multiple derivations of judgements.

Mathematicians therefore have an "*ingenious*" way of dealing with this: Abstract away the key properties to end up with an object with the *desired semantics*. This objects typically fall under names such as *structured categories*, and come up in the subject of *categorical semantics*. Mathematicians can therefore reason about "type theories" by reasoning about these particular objects.

The questions still stands however *what is a type theory?*. We will not claim to solve this problem but rather provide a partial solution. In this thesis we will describe a specific kind of *dependent type theory*. Later we will derived the categorical semantics for our type theory. We cannot however do this all in one step, and so we will begin with what is known as *simply typed lambda calculus*. We will then modify the rules for this to give us a basic *dependent type theory*.

We will discuss in detail the need for some sort of "initiality theorem" for a given type theory. This will make the interpretation of the syntax useful. There have been many attempts in the past to prove some sort of intiality theorem, the most notable by Streicher, but in general there is still much debate on the usefullness of these results. Notable mathematicians such as Vladamir Voevodsky have persuasively argued that this is an unacceptably unrigorous attidue. There is no precise definition of what "suitable type theory" means nor which methods are applicable.

The author will note that there are currently a few attempts at answering this question, but to date, not suitable solution has been proposed.

Now suppose we have some sort of "type theory". It is still not a completely satisfactory situation in terms of describing the syntax of such an object. Many

authors [add citations] have noted that this is the case and more worryingly many other authors have claimed that it is done and dusted. There is as a result a long history of false claims of correct syntax. [add citations]

2.2 Introduction

We will follow the structure of syntax outlined in Harper [14]. 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 [9], 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 [22], 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 [15], 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 [27] 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 [29] 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).

2.3 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 [16, p. 62], [3, Ch. 7]. It's use in the formal semantics of programming languages is not much different either [31, Ch. 3]. There are however more constructive notions of well-foundedness [26, §8] with more careful use of excluded middle. We will follow [28], 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 2.3.1. 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 2.3.2. 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 2.3.3 (Well-founded induction principle)**.** Let X be a well-founded set and P a property of the elements of X (a proposition). Then

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

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 2.3.2 $Y = X$ 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.

2.4 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 2.4.1 (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 2.4.2 (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 2.4.3 (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 2.4.4 (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 2.4.5 (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 2.4.6 (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 2.4.7. 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 2.4.8 (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 2.4.9. 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 2.4.10. An ast can be thought of as a tree whose leaf nodes are variables and branch nodes are operators.

Example 2.4.11 (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.

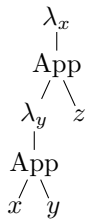

Figure 1: Vertically oriented tree representing the lambda term

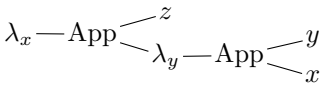

Figure 2: Horizontally oriented tree representing the lambda term

Remark 2.4.12. 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 2.4.13. 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 2.4.14. 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 2.4.15. 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 $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{A}[\mathcal{Y}]$. □

2.5 Substitution in asts

Definition 2.5.1 (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 2.5.2. 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 2.5.3. Note that 2.5.2 was simply about checking Definition 2.5.1. 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 [14] does, with slight modifications that are used in [23, 24], 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.

2.6 Abstract binding trees

Definition 2.6.1 (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 2.6.2. In [14] there is no relation but a function. And each argument has bound variables assigned to it. But as argued in [24] 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 [24] 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 2.6.3 (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 2.6.4. We overload the definitions of arity and operator to mean generalised operator and operator with generalised arity respectively.

Remark 2.6.5. When we wish to specify an operator we need only give the following data:

1. Name - what we wish to call the operator, for example $\to$ or $\times$.
2. Sort - what is the sort of the operator?
3. Variables - What are the variables of the operator?
4. Sorted arguments - What are the arguments and what are their sorts?
5. Scoping - Which arguments are in scope of which variables?
6. Sugared syntax - How do we write down the operator with all the variables and arguments together. By default we have been writing $\mathcal{O}(x$

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 2.6.6 (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 2.6.7. There is a lot going on in the second constructor of Definition 2.6.6. 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.]]

2.7 Substitution in abts

3 Judgements

We will now develop the basic formal tools to describe how our programming languages work. We will first describe judgements and how to specify a type system. Then our first example will be the simply typed lambda calculus. We use the ideas developed in [14] though these ideas are much older. [Probably tracable back to Gentzen]. [There are many more references to be included here]

Definition 3.0.1. The notion of a *judgement* or *assertion* is a logical statement about an abt. The property or relation itself is called a *judgement form*. The judgement that an object or objects have that property or stand in relation is said to be an *instance* of that judgement form. A judgment form has also historically been called a *predicate* and its instances called *subjects*.

Remark 3.0.2. Typically a judgement is denoted J. We can write $a\ \mathsf{J}$, $\mathsf{J}\ a$ to denote the judgment asserting that the judgement form J holds for the abt a. For more abts this can also be written prefix, infix, etc. This will be done for readability. Typically for an unspecified judgement, that is an instance of some judgement form, we will write J.

Definition 3.0.3. An *inductive definition* of a judgement form consists of a collection of rules of the form

$$\frac{J_1 \quad \cdots \quad J_k}{J}$$

in which J and $J_1, \ldots, J_k$ are all judgements of the form being defined. THe judgements above the horizontal line are called the *preimises* of the rules, and the judgement below the line is called its *conclusion*. A rule with no premises is called an *axiom*.

3.1 Inference rules

Remark 3.1.1. An inference rule is read as starting that the premises are *sufficient* for the conclusion: to show J, it is enough to show each of $J_1, \ldots J_k$. Axioms hold unconditionally. If the conclusion of a rule holds it is not necesserily the case that the premises held, in that the conclusion could have been derived by another rule.

Example 3.1.2. Consider the following judgement from $-\ \mathsf{nat}$, where $a\ \mathsf{nat}$ is read as "a is a natural number". The following rules form an inductive definition of the judgement form $-\ \mathsf{nat}$:

$$\frac{}{\mathtt{zero}\ \mathsf{nat}} \qquad \frac{a\ \mathsf{nat}}{\mathtt{succ}(a)\ \mathsf{nat}}$$

We can see that an abt a is zero or is of the form $\mathtt{succ}(a)$. We see this by induction on the abt, the set of such abts has an operator $\mathtt{succ}$. Taking these

rules to be exhaustive, it follows that $succ(a)$ is a natural number if and only if a is.

Remark 3.1.3. We used the word *exhaustive* without really defining it. By this we mean necessary and sufficient. Which we will define now.

Definition 3.1.4. A collection of rules is considered to define the *strongest* judgement form that *closed under* (or *respects*) those rules. To be closed under the rules means that the rules are *sufficient* to show the validity of a judgement: J holds if there is a way to obtain it using the given rules. To be the *strongest* judgement form closed under the rules means that the rules are also *necessary*: J holds *only if* there is a way to obtain it by applying the rules.

Let's add some more rules to our example, to get a richer structure.

Example 3.1.5. The judgement form $a = b$ expresses the equality of two abts a and b. We define it inductively on our abts as we did for `nat`.

$$\frac{}{\texttt{zero} = \texttt{zero}} \qquad\qquad \frac{a = b}{\texttt{succ}(a) = \texttt{succ}(b)}$$

Our first rule is an axiom declaring that `zero` is equal to itself, and our second rule shows that abts of the form `succ` are equal only if their arguments are. Observe that these are exhaustive rules in that they are necessary and sufficient for the formation of $=$.

3.2 Derivations

To show that an inductively defined judgement holds, we need to exhibit a *derivation* of it.

Definition 3.2.1. A *derivation* of a judgement is a finite composition of rules, starting with axioms and ending with the judgement. It is a tree in which each node is a rule and whose children are derivations of its premises. We sometimes say that a derivation of J is evidence for the validity of an inductively defined judgement J.

Suppose we have a judgement J and

$$\frac{J_1 \quad \cdots \quad J_k}{J}$$

is an inference rule. Suppose $\nabla_1, \ldots, \nabla_k$ are derivations of its premises, then

$$\frac{\nabla_1 \quad \cdots \quad \nabla_k}{J}$$

is a derivation of its conclusion. Notice that if $k = 0$ then the node has no children.

Writing derivations as trees can be very enlightening to how the rules compose. Going back to our example with `nat` we can give an example of a derivation.

Example 3.2.2. Here is a derivation of the judgement $\texttt{succ}(\texttt{succ}(\texttt{succ}(\texttt{zero})))\ \texttt{nat}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{zero nat}}}{\texttt{succ(zero) nat}}}{\texttt{succ(succ(zero)) nat}}}{\texttt{succ(succ(succ(zero))) nat}}$$

Remark 3.2.3. To show that a judgement is *derivable* we need only give a derivation for it. There are two main methods for finding derivations:

- *Forward chaining* or *bottom-up construction*
- *Backward chaining* or *top-down construction*

Forward chaining starts with the axioms and works forward towards the desired conclusion. Backward chaining starts with the desired conclusion and works backwards towards the axioms.

It is easy to observe the *algorithmic* nature of these two processes. In fact this is an important point to think about, since it may become relevent in the future.

Lemma 3.2.4. Given a derivable judgement J, there is an algorithm giving a derivation for J by forward chaining.

Proof. This is not a difficult algorithm to describe. We start with a set of rules $\mathcal{R} := \varnothing$ which we initially set to be empty. Now we consider all the rules that have premises in $\mathcal{R}$, initially this will be all the axioms. We add these rules to $\mathcal{R}$ and repeat this process until J appears as a conclusion of one of the rules in $\mathcal{R}$. It is not difficult to see that this will necesserily give all derivations of all derivable judgements and since J is derivable, it will eventually give a derivation for J. □

Remark 3.2.5. Notice how we had to specify that our judgement is derivable. Since if were not, then our process would not terminate, hence would not be an algorithm. It is also worth noting that this algorithm is very inefficient since the size of $\mathcal{R}$ will grow rapidly, especially when we have more rules available. This is sort of a brute force approach. What we will need is more clever picking of the rules we wish to add. Mathematically this is an algorithm, but not in any practical sense.

Forward chaining does not take into account any of the information given by the judgement J. The algorithm is in a sense blind.

Lemma 3.2.6. Given a derivable judgement J, we can give a derivation for J by backward chaining.

Proof. Backward chaining maintains a queue of goals, judgements whose derivations are to be sought. Initially this consists of the sole judgement we want to derive. At each step, we pick a goal, then we pick a rule whose conclusion is our picked goal and add the premises of the rule to our list of goals. Since J is derivable there must be a derivation that can be chosen. □

Remark 3.2.7. We could as before consider all possible goals generated by all possible rules which would technically give us an algorithm like in the case for forward chaining. But it would also be as useless as that algorithm. What backward chaining allows us to do however is better pick to rules at each stage. This is the structure that type checkers will take later on and even proof assitants, programs that assist a user in proving a statement formally. Due to each stage giving us information about the kind of rule we ought to pick, backward chaining is more suitable for algorithmically proving something. In face if we set up our rules in such a way that for each goal there is only one such rule to pick, we have an algorithm!

3.3 Rule induction

Conveniently our notion of inductive definition of a judgement form is actually an inductive definition. In that the set of derivable judgements forms a well-founded tree as defined earlier. This means we can apply our more general notion of well-founded induction when proving properties of a judgement.

Definition 3.3.1. We say that a property $\mathcal{P}$ is *closed under* or *respects* the rules defining a judgement form J. A property $\mathcal{P}$ respects the rule

$$\frac{a_1\ \mathsf{J} \quad \cdots \quad a_k\ \mathsf{J}}{a\ \mathsf{J}}$$

if $\mathcal{P}(a)$ holds whenever $\mathcal{P}(a_1), \ldots, \mathcal{P}(a_k)$ do.

Remark 3.3.2. This is nothing more than a rephrasing of well-founded trees which is classically more common. This style of inductive definition fits more closely with what is actually going on, and we would argue is easier to work with.

We will now give some examples detailing how rule induction can be used.

Example 3.3.3. Continuing our nat example, if we want to show $\mathcal{P}(a)$ for some $a\ \mathsf{nat}$ it is enough to show the following:

- $\mathcal{P}(\mathtt{zero})$.
- for all a, of $\mathcal{P}(a)$, then $\mathcal{P}(\mathtt{succ}(a))$.

This is the familiar notion of mathematical induction on the natural numbers.

Now for another example where we combine all the things we have just discussed.

Example 3.3.4. Consider the judgement form tree defined inductively by the following rules:

$$\frac{}{\mathtt{empty}\ \mathsf{tree}} \qquad \frac{a_1\ \mathsf{tree} \quad a_2\ \mathsf{tree}}{\mathsf{node}(a_1; a_2)\ \mathsf{tree}}$$

Here is a derivation of the judgement $\mathtt{node}(\mathtt{empty}; \mathtt{node}(\mathtt{empty}; \mathtt{empty}))\ \mathsf{tree}$:

$$\dfrac{\dfrac{}{\mathtt{empty}\ \mathsf{tree}} \quad \dfrac{\dfrac{}{\mathtt{empty}\ \mathsf{tree}} \quad \dfrac{}{\mathtt{empty}\ \mathsf{tree}}}{\mathtt{node}(\mathtt{empty}; \mathtt{empty})\ \mathsf{tree}}}{\mathtt{node}(\mathtt{empty}; \mathtt{node}(\mathtt{empty}; \mathtt{empty}))\ \mathsf{tree}}$$

Now rule induction for the judgement form tree states that, to show $\mathcal{P}(a)$ it is enough to show the following:

- $\mathcal{P}(\mathtt{empty})$.
- for all a_1 and a_2, if both $\mathcal{P}(a_1)$ and $\mathcal{P}(a_2)$ then, $\mathcal{P}(\mathtt{node}(a_1; a_2))$.

This is the familar notion of tree induction.

Now that we have induction on our inductive definitions we can prove some results about our examples.

Lemma 3.3.5. If $\mathtt{succ}(a)\ \mathsf{nat}$, then $a\ \mathsf{nat}$.

Proof. By induction on $\mathtt{succ}(a)$, when $\mathtt{succ}(a)$ is $\mathtt{zero}$ this is vacously true. Otherwise when $\mathtt{succ}(a)$ is succ(b), what we want to prove is $\mathtt{succ}(b)\ \mathsf{nat} \implies b\ \mathsf{nat}$ but this is exactly our induction hypothesis. □

Lemma 3.3.6 (Reflexivity of $=$)**.** If $a\ \mathsf{nat}$, then $a = a$.

Proof. By induction on a we have two cases which are exactly the two rules about $=$ to begin with. □

Lemma 3.3.7 (Injectivity of $\mathtt{succ}$)**.** If $\mathtt{succ}(a_1) = \mathtt{succ}(a_2)$, then $a_1 = a_2$.

Proof. We perform induction on $\mathtt{succ}(a_1)$ and $\mathtt{succ}(a_2)$. Note that if any of the two are of the form $\mathtt{zero}$ then the statement is true vacously. When $\mathtt{succ}(a_1)$ is of the form $\mathtt{succ}(b_1)$ and $\mathtt{succ}(a_2)$ is of the form $\mathtt{succ}(b_2)$ our statement that we want to prove is exactly what we get from the induction hypothesis. □

Lemma 3.3.8 (Symmetry of $=$)**.** If $a = b$, then $b = a$.

Proof. Begin with induction on a and b:

- Suppose a is of the form $\mathtt{zero}$ and b is of the form $\mathtt{zero}$ then we have $\mathtt{zero} = \mathtt{zero}$ as desired.
- Suppose a is of the form $\mathtt{zero}$ and b is of the form $\mathtt{succ}(b')$ then our statement is vacously true. The same happens for when b is $\mathtt{zero}$ and a is of the form $\mathtt{succ}(a')$.

- Finally when a is of the form $\texttt{succ}(a')$ and b is of the form $\texttt{succ}(b')$ we have $\texttt{succ}(a') = \texttt{succ}(b')$. By 3.3.7 we have $a' = b'$ and by our induction hypothesis we have $b' = a'$ as desired.

□

Lemma 3.3.9 (Transitivity of $=$)**.** If $a = b$ and $b = c$ then $a = c$.

Proof. By induction on a, b and c we see that we have eight cases. Clearly six of these are vacously true, so we will prove the other two:

- When a, b and c are of the form $\texttt{zero}$ our statement holds trivially.
- Whne a, b and c are of the form $\texttt{succ}(a')$, $\texttt{succ}(b')$ and $\texttt{succ}(c')$ respectively, we can apply 3.3.7 on $\texttt{succ}(a') = \texttt{succ}(b')$ and $\texttt{succ}(b') = \texttt{succ}(c')$ to get $a' = b'$ and $b' = c'$. Then applying our induction hypothesis we have $a' = c'$, finally applying the second rule for $=$ we have $\texttt{succ}(a') = \texttt{succ}(c')$.

□

Finally we can say our four rules correspond to Peano arithmetic!

[[Now talk about how classically peano arithmetic requires many more axioms, we only have four rules and the notion of induction!]] [[Talk about what we have proven about peano arithematic is actually a meta statement, a statement in the metalanguage, later we will have richer logics where we can prove things like this internally]].

[[References include Aczel 1977 who provides a thorough account of inductive definitions and judgement based logic is inspired by Martin-Lof's logic of judgements 1983, 1987]]

[[Talk about iterated and simultaneous inductive definitions]]

3.4 Hypothetical judgements

A *hypothetical judgement* expresses an entailment between one or more hypthesis and a conclusion. There are two main notions of entailment in logic: *derivability* and *admissibility*. We first begin by defining derivability.

Definition 3.4.1. Given a set $\mathcal{R}$ of rules, define the *derivability* judgement, $J_1, \ldots, J_k \vdash_{\mathcal{R}} K$ where each J_i and K are basic judgements, to mean that we may derive K from the *expansion* $\mathcal{R} \cup \{J_1, \ldots, J_k\}$ of the rules $\mathcal{R}$ with the axioms

$$\frac{}{J_1} \quad \cdots \quad \frac{}{J_k}$$

We treat the *hypotheses* or *antecedents* $J_1, \ldots, J_k$ of the judgement $J_1, \ldots, J_k \vdash_{\mathcal{R}} K$ as axioms and derive the *conclusion* or *consequent*, by composing rules in $\mathcal{R}$. Thus $J_1, \ldots, J_k \vdash_{\mathcal{R}} K$ means the judgement K is derivable from the expanded rules $\mathcal{R} \cup \{J_1, \ldots, J_k\}$.

Remark 3.4.2. We will typically denote a list of basic judgements by a captial greek letter such as Γ or Δ. The expansion $\mathcal{R} \cup \{J_1, \ldots, J_k\}$ may also be written as $\mathcal{R} \cup \Gamma$ where $\Gamma := J_1, dots, J_k$. The judgement $\Gamma \vdash_{\mathcal{R}} K$ means K is derivable from the rules $\mathcal{R} \cup \Gamma$, and the judgement $\vdash_{\mathcal{R}} \Gamma$ means that $\vdash_{\mathcal{R}} J$ for each J in Γ. We may also extend lists of basic judgements like this: Γ, J, which would correspond to the list of basic judgements $J_1, \ldots, J_k, J$, similarly for J, Γ. We can then concatenate two lists of basic judgements in the obvious way, through list concatenation written Γ, Δ.

Example 3.4.3. Let Peano be the set of four rules for our nat example. Consider the following derivability judgement:

$$a\ \mathsf{nat} \vdash_{\mathsf{Peano}} \mathtt{succ}(\mathtt{succ}(a))\ \mathsf{nat}$$

This can be shown to be true by exhibiting the following derivation:

$$\dfrac{\dfrac{a\ \mathsf{nat}}{\mathtt{succ}(a)\ \mathsf{nat}}}{\mathtt{succ}(\mathtt{succ}(a))\ \mathsf{nat}}$$

We now show that derivability doesn't get affected by expansion.

Lemma 3.4.4 (Stability)**.** If $\Gamma \vdash_{\mathcal{R}} J$, then $\Gamma \vdash_{\mathcal{R} \cup \mathcal{R}'} J$.

Proof. Any derivation if J from $\mathcal{R} \cup \Gamma$ is also a derivation from $(\mathcal{R} \cup \mathcal{R}') \cup \Gamma$ since $\mathcal{R} \subseteq \mathcal{R} \cup \mathcal{R}'$. □

There are a number of structural properties that derivability satisfies:

Lemma 3.4.5 (Reflexivity)**.** Every judgement is a consequence of itself: $\Gamma, J \vdash_{\mathcal{R}} J$.

Proof. Since J becomes an axiom, the proof is trivial. □

Lemma 3.4.6 (Weakening)**.** If $\Gamma \vdash_{\mathcal{R}} J$, then $\Gamma, K \vdash_{\mathcal{R}} J$. Entailiment is not influenced by unused premises.

Proof. The proof is trivial. □

Lemma 3.4.7 (Transitivity)**.** If $\Gamma, K \vdash_{\mathcal{R}} J$ and $\Gamma \vdash_{\mathcal{R}} K$, then $\Gamma \vdash_{\mathcal{R}} J$. If we repalce an axiom by a derivation of it, the result is a derivation of the consequent without the hypothesis.

Proof. It is clear that if there is a derivation for J from $\Gamma, K \cup \mathcal{R}$ and a derivation for K from $\Gamma \cup \mathcal{R}$, then there is clearly a derivation for J from $\Gamma \cup \mathcal{R}$. For the first case it is clear how to compose two derivations to give the desired derivation. □

Definition 3.4.8. Another form of entailment, *admissibility*, written $\Gamma \vDash_{\mathcal{R}} J$, is a weaker form of hypothetical judgement stating that $\vdash_{\mathcal{R}} \Gamma$ implies $\vdash_{\mathcal{R}} J$. That is, the conclusion J is derivable from the rules $\mathcal{R}$ when the assumptions are all derivable from the rules $\mathcal{R}$.

Remark 3.4.9. In particular, if any of the hypotheses are *not* derivable relative to $\mathcal{R}$, then the judgement is vacously true.

The admissibility judgement is *not* stable under expansion of the rules.

Lemma 3.4.10. If $\Gamma \vdash_{\mathcal{R}} J$, then $\Gamma \vDash_{\mathcal{R}} J$.

Proof. By definition of admissibility we need to show that $\vdash_{\mathcal{R}} \Gamma$ implies $\vdash_{\mathcal{R}} J$. It can be seen that repeated applciation of transitivity allows us to form a similar statement for when K is a list of basic judgements in reference to 3.4.7. This repeated transitivity gives us the desired result. □

We will now give an example of some inadmissible rules.

Example 3.4.11. Consider the collection of rules Parity consisting of the rules in Peano and the following:

$$\frac{}{\texttt{zero}\ \mathsf{even}} \qquad \frac{b\ \mathsf{odd}}{\texttt{succ}(b)\ \mathsf{even}} \qquad \frac{a\ \mathsf{even}}{\texttt{succ}(a)\ \mathsf{odd}}$$

This is a similtaneous inductive definition. Clearly we have the following admissibility judgement

$$\texttt{succ}(a)\ \mathsf{even} \vDash_{\mathsf{Parity}} a\ \mathsf{odd}$$

But by adding the following rule to Parity, and calling it Parity′

$$\frac{}{\texttt{succ}(\texttt{zero})\ \mathsf{even}}$$

we see that the following is no longer true:

$$\texttt{succ}(a)\ \mathsf{even} \vDash_{\mathsf{Parity}'} a\ \mathsf{odd}$$

since there is no composition of rules deriving `zero` odd. Hence admissibility is not stable under expansion.

Remark 3.4.12. Admissibility is a useful property of a rule. It essentially checks whether we can get rid of a rules, knowing that we can derive it anyway. Hence by identifying inadmissible rules we can streamline our rule set.

3.5 Hypothetical inductive definitions

Our inductive definitions give us a rich and expressive way to define and use rules. We wish to enrich it further by introducing rules whose premises and conclusions are derivability judgements.

Definition 3.5.1. A *hypothetical inductive definition* consists of a set of *hypothetical rules* of the following form:

$$\frac{\Gamma, \Gamma_1 \vdash J_1 \quad \cdots \quad \Gamma, \Gamma_n \vdash J_n}{\Gamma \vdash J}$$

We call the hypotheses Γ, the *global hypotheses* of the rule, and Γ_i are called the local hypotheses of the ith premise of the rule. We will require that all rules in a hypothetical inductive definition be *uniform* in the following sense.

Definition 3.5.2. A hyptothetical rules is said to be *uniform* if it holds for *all* global contexts.

Remark 3.5.3. When we have uniformity, we can present the rule in an *implicit* or *local* form:

$$\frac{\Gamma_1 \vdash J_1 \quad \cdots \quad \Gamma_n \vdash J_n}{J}$$

with the understanding that the rule applies for any choice of global hypotheses.

Remark 3.5.4. A hypothetical inductive definition can be regarded as an ordinary inductive definition of a *formal derivability judgement* $\Gamma \vdash J$ consisting of a list of basic judgements Γ and a basic judgement J.

Definition 3.5.5. A *formal derivability judgement* $\Gamma \vdash J$ is closed under a set of hypothetical rules $\mathcal{R}$ and the judgement is *structural* is that it is closed under the following rules

$$\frac{}{\Gamma, J \vdash J} \qquad \frac{\Gamma \vdash J}{\Gamma, K \vdash J} \qquad \frac{\Gamma \vdash K \quad \Gamma, K \vdash J}{\Gamma \vdash J}$$

These rules ensure that formal derivability behaves like a hyptothetical judgement. We write $\Gamma \vdash_{\mathcal{R}} J$ to denote that $\Gamma \vdash J$ is derivable from rules $\mathcal{R}$.

[[This bit is very confusing, and we abuse notation (with reason)]]

Remark 3.5.6. This definition is perhaps quite confusing, this is because we have two layers of derivability. What a formal derivability judgement shows is that the judgement of being derivable is itself derivable. This also means that we do not have to define what hyptothetical induction on a hypothetical inductive definition is, since the formal derivability judgement is itself a judgement. So the principle of *hypothetical rule induction* is just the principle of rule induction applied to the formal hypothetical rule induction.

[[TODO: talk about admissibility of structural rules]]

3.6 General judgements

[[Talk about generic judgements and parametric judgements]]

4 Statics and Dynamics

How can we in general design programming languages to assertain certain behaviours. Static and dynamic typing for instance. Different constructs and data types such as products and sums. Later we will look at a deep correspondance between programming and logic which gives us an indication of what a programming language ought to have.

Statics: Type checking Dynamics: Computation, equational rules, transition systems (reduction with betas and etas)

We will introduce typing and think carefully about another structural rule: The exchange rule, we will see that it is inadmissible and infact not necesserily needed. Infact later when we think about dependent types we will see that it is in general "complete nonsense". HOWEVER it is essential for some models of STLC.

We will end up with STLC. But we will also show how to add sum types.

We will also model the semantics of such programming languages (at least the statics of) using categories.

Later we will see that Curry-Howard is very suggestive about quantifiers, can we add these? YES!

Then we can introduce our favorite dependent types. Show how useful they are for programmers and mathematicans

We will now try to design programming languages that can have types, types allow us to restrict what terms we can apply functions to. Something take for granted very often in mathematics and to a lesser extent in programming. Programming langauges such as C don't really type check, which means functions that should be applied can be. There are different strengths to type checking, some check at compilation (which is arguably to most sensible) but others check during run time but this means a program cannot be garanteed to be safe.

The ideas of types are very deep, so when combined with a flexibly expressible type system (dependent types) it leads to a powerful correctness tool.

4.1 Typing and Type systems

We will need to discuss the modern idea of bidirectional typechecking which has advantages over choosing a single one. This is closely related to the subtle difference between Church's lambda calculus and Curry's lambda calculus.

Things to discuss:

- Typing judgements
- typing contexts
- Type checking
  - Different kinds of type checking (bidirectional?)
  - Program safety
- Unicity of typing (every term has one type)
- Inversion is a form of type inference fitting into the more general framework of bidirectional type checking
- Exchange rule (discussion of and implications thereof)
- Other structural properties including substitution, decomposition, weakening

4.2 Dynamics

Thigns to discuss:

Two formulations of dynamics in type theory:

- Transition systems
- Equational dynamics

Arguably the first is more reminiscent of what a programming language ought to do, and the second is more reminiscent of what a mathematician would want.

- Judgemental equality has some issues, some terms that ought to be judgementally equal are only so for particular instances but not in general. There is a discussion of *semantic* equivalence to solve this issue.
- Should be based off of Plotkins work on structural semantics

Need a discussion of

- structural dynamics
- contextual dynamics

Which are basically the same thing.

4.3 Type safety

Type safety is the notion of being strongly typed. It is typically given as a theorem consisting of two parts: *preservation* and *progress*.

Definition 4.3.1 (Preservation)**.** If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.

Evaluation preserves typing.
Discuss canonical forms and canonicity of a type theory.

Definition 4.3.2 (Progress)**.** If $e : \tau$, then either e val, or there exists e' such that $e \mapsto e'$.

A term is either evaluated or can be evaluated.

Definition 4.3.3 (Type safety)**.** A language is said to be *safely typed* or *strongly typed* if it satisfies preservation and progress.

4.4 Run time errors

Talk about division yielding an excpetional case when divding by zero. There are two solutions:

1. Enhance the typing systems
2. Add dynamic checks

Typically the second is more common, but we will argue the case for dependent types which essentially allows us to solve the first.

4.5 Evaluation dynamics

There can be a discussion of richer judgements of evaluation that also take into account of cost and such things, but these might be out of scope.

They can be related back to structural dynamics. Evaluation dynamics are more expressive yet limited since now we cannot check type safety. But if we have run time errors we can get quite close. These ideas are developed when designing standard ML, Milner. Cost dynamics are used by Blelloch and Greiner in a study of parallel computation.

5 Curry-Howard correspondance

5.1 Mathematical logic

At the beginning of the 20th century, Whitehead and Russell pubished their *Principia Mathematica* [25], 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 [13], 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 [7]; *recursive functions*, proposed by Gödel in 1934 later published in 1936 by Stephen Kleene [21]; and finally *Turing machines* in 1937 by Alan Turing [30].

5.2 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.z It is closely related to Curry's idea of combinatory logic [10, 11] 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." [4, 5]. 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. [17, 18]. Church later proposed 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.

5.3 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 [**?**]. Church later outlined a proof [6] and Kleene later published it in detail [19]. This however did not have the intended effect on Gödel, whereby he then became convinced that his own definition was incorrect!

5.4 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".

5.5 Russells paradox

[Talk about the origin of types and stuff]

5.6 The problem with lambda calculus as a logic

Church's students Kleene and Rosser quickly discovered that lambda calculus was inconsistent as a logic [20]. Curry later simplified the result which became known as Curry's paradox [12]. Curry's paradox 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 when formulating *Principia Mathematica*: use types [**?**]. What was discovered is now known today as *simply-typed lambda calculus* [8]. 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. [30]

[CITATION NEEDED] From this consistency of Church's STLC as a logic could be established.

5.7 Types to the rescue

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

5.8 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]

5.9 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.

5.10 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".]

5.11 Predicates [CHANGE] as types?

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

5.12 Dependent types

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

6 Simply typed lambda calculus

First develop the features needed. Discuss the arbitrary nature of such features, then use Curry-Howard as motivation for "the language that ought to be". Develop STLC, discuss in detail the implications, give categorical semantics. Discuss breifly the dynamics of simply typed lambda calculus. A big disadvantage of STLC over the untyped version (which we ought to discuss since we have the tools to) is that there is no recursion. There are many ways to fix this, see Gödel for example. In order to fix this we will introduce dependent types.

We begin by discussing the syntax of our type theory. We will start by specifying the sorts $\mathcal{S}$ of our type theory.

Definition 6.0.1. The sorts of simply typed lambda calculus are terms and types $\mathcal{S} := \{\text{tm}, \text{ty}\}$.

We now specify the operators (with generalized arities) that we defined in definition 2.6.3. In remark 2.6.5 we discussed the data needed to give an operator, therefore we will present all our operators in the following table.

Definition 6.0.2. The operators in the syntax of simply typed lambda calculus are given by the following table:

Operator	Sort	Vars	Type args	Term args	Scoping	Sugared syntax
$\to$	ty	—	A, B	—	—	$A \to B$
$\times$	ty	—	A, B	—	—	$A \times B$
$(-,-)$	tm	—	—	x, y	—	(x, y)
λ	tm	x	A, B	—	M	$\lambda(x : A).M$
App	tm	—	A, B	—	M, N	MN

Remark 6.0.3. Note that some of the sugared syntax loses information that was put in. The application is the main example of this. In practice if we know the type of M and N we can deduce the type of MN just from the rules we will define later. The syntax is sugared or *syntactic sugar* so we do not have to write so much. If done incorrectly it could be considered an abuse of notation. It should be possible to *desugar* the syntax by adding an *annotated* version of an operator. For example for application instead of MN we could write $\text{App}_{A,B}(M; N)$. Having this information in the syntax will be useful when we want to induct over syntax, for example when proving an intiality theorem. But in practice we will save ourselves from having to write it out.

Definition 6.0.4. We can now construct our raw terms and types as the collection of abts (see definition 2.6.6) over the previously defined data $\text{Term} := \mathcal{B}[\varnothing]_{\text{tm}}$ and $\text{Type} := \mathcal{B}[\varnothing]_{\text{ty}}$.

Remark 6.0.5. Note that we have no variables. This is because if we set the definition of abt up correctly we don't need any, but terms can have subterms (subtrees of the abt) which have variables. The sets Term and Type become *all* the types and terms we ought to be able to write down from scratch.

We now need to define judgements about our syntax and write down the rules to write them down. [[Make a note about substitution because afik we haven't defined it properly yet]].

6.1 Judgements

[[TODO: Clean up this whole paragraph(s)]] We begin with our basic judgements. Of which there will be 5. Our STLC will have bidirectional typechecking, in that we will distinguish between the direction of type checking. There are several advantages of this and historically the two main systems called STLC are Curry's and Church's which simply differ in the direction of type checking. By

having both directions and a sort of "mode-switching rule" we have far greater control and ease when describing type checking properties. We will also need to have a notion of *judgemental equality* since we wish to do some computation. There are variations of this theme discussed in the statics chapter that allow us to have transition systems instead but we will use an equational style since transition systems can be derived from this. This also has the advantage of STLC becomming what is known as an "equational theory". This will be a useful feature for when we want to derrive categorical semantics.

A context is a list of basic judgements. Our basic judgements are $x : A$. [[No it is not fix this]]

There are 5 judgements that we have:

- $\Gamma \vdash A$ type - "A is a type in context Γ".
- $\Gamma \vdash T \Leftarrow A$ - "T can be checked to have type A in context Γ".
- $\Gamma \vdash T \Rightarrow A$ - "T synthesises the type A in context Γ".
- $\Gamma \vdash A \equiv B$ type - "A and B are jdugementally equal types in context Γ".
- $\Gamma \vdash S \equiv T : A$ - "S and T are judgementally equal terms of type A in context Γ".

6.2 Structural rules

Structural rules will dictate how our judgements interact with eachother, how different contexts can be formed and how substitution works. This is all roughly what a "type theory" ought to provide.

We begin with the *variable* rule, this says that if a term x appears with a type A as an element in a context Γ then x synthesises a type A in context Γ. Or written more succiently as:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}$$

Other structural rules: weakening, contraction and substitution are all admissible. [[What does it mean for a rule to be admissible? We have defined this previously but we need to carefully state these facts, and prove them too!]]

6.3 Mode-switching

One of the features of bidirectional type checking is that we can switch the mode we are in. This is expressed as the mode switching rule:

$$\frac{\Gamma \vdash T \Rightarrow A \qquad \Gamma \vdash A \equiv B \text{ type}}{\Gamma \vdash T \Leftarrow B}$$

This rule has been specially set up in that it will be the *only way* to derive $\Gamma \vdash T \Leftarrow B$. [[TODO: talk more about this]]

In a unidirectional type system, the judgements $\Gamma \vdash T \Rightarrow A$ and $\Gamma \vdash T \Leftarrow B$ are collapsed into one: $\Gamma \vdash T : A$. And now the mode-switching rule may have a more familiar form:

$$\frac{\Gamma \vdash T : A \qquad \Gamma \vdash A \equiv B \ \mathsf{type}}{\Gamma \vdash T : B}$$

Which shows that it is actually a rule about substituting along a judgemental equality! But this is a problem since a type checking algorithm will have to decide when to stop doing this. This is one of the big advantages that bidirectional type checking has over unidirectional type checking. The type checking algorithm will be simpler! [[TODO: Clean up and discuss type checking in more detail]]

6.4 Equality rules

Finally we have some structural rules for our two judgemental equality judgements. We wish for these to be an equivalence relation and that they are compatible with eachother.

First we begin with the structural rules for the judgement form $- \equiv - \ \mathsf{type}$:

We wish for our judgemental equality of types to be reflexive:

$$\frac{}{\Gamma \vdash A \equiv A \ \mathsf{type}} \ (\equiv_{\mathsf{type}}\text{-reflexivity})$$

We want our judgemental equality of types to be symmetric:

$$\frac{\Gamma \vdash A \equiv B \ \mathsf{type}}{\Gamma \vdash B \equiv A \ \mathsf{type}} \ (\equiv_{\mathsf{type}}\text{-symmetry})$$

and our judgemental equality of types to be transitive:

$$\frac{\Gamma \vdash B \ \mathsf{type} \qquad \Gamma \vdash A \equiv B \ \mathsf{type} \qquad \Gamma \vdash B \equiv C \ \mathsf{type}}{\Gamma \vdash A \equiv C \ \mathsf{type}} \ (\equiv_{\mathsf{type}}\text{-transitivity})$$

Notice how the previous rule also checks that B is a type. This is because if we did not do this, we could insert any symbol in. This is clearly undesirable. It also demonstrates how subtly sensitive rules are.

Now we list the rules making the judgement form $- \equiv - : A$ into an equivalence relation:

We wish for our judgemental equality of terms to be reflexive:

$$\frac{}{\Gamma \vdash T \equiv T : A} \ (\equiv_{\mathsf{term}}\text{-reflexivity})$$

We want our judgemental equality of terms to be symmetric:

$$\frac{\Gamma \vdash S \equiv T : A}{\Gamma \vdash T \equiv S : A} \ (\equiv_{\mathsf{term}}\text{-symmetry})$$

and our judgemental equality of terms to be transitive:

$$\frac{\Gamma \vdash T \Leftarrow A \qquad \Gamma \vdash S \equiv T : A \qquad \Gamma \vdash T \equiv R : A}{\Gamma \vdash S \equiv R : A} \; (\equiv_{\mathsf{term}}\text{-transitivity})$$

as we stated before for transitivity judgemental equality of types we need to also check that the middle term T is actually a term.

Finally we need a rule that will make that judgemental equality of types and judgemental equality of terms interact the way we expect them to:

$$\frac{\Gamma \vdash A \; \mathsf{type} \qquad \Gamma \vdash S \equiv T : A \qquad \Gamma \vdash A \equiv B \; \mathsf{type}}{\Gamma \vdash S \equiv T : B} \; (\equiv_{\mathsf{term}}\text{-}\equiv_{\mathsf{type}}\text{-compat})$$

6.5 Type formers

What we have constructed thusfar is essentially an "empty type theory". What we have included which other authors typcially gloss over is a clean way of constructing a typechecking algorithm: bidirectional typechecking and an account of judgemental equality. We now study what are known as type formers, typically when we wish to add a new type to a type theory we need to think about a collection of rules. These can roughly be sorted into 5 kinds of rules:

- Formation rules - How can I construct my type?
- Introduction rules - Which terms synthesise this type?
- Elimination rules - How can terms of this type be used?
- Computation (or equality) rules - How do terms of this type compute? (Normalise, etc.)
- Congruence rules - How do all the previous rules interact with judgemental equality

We make a note that although we will be providing all the rules, the congruence rules can be typically derrived from the others. Although we do not know exactly how to do this so we will provide them explicitly. We also note that not every type need computation rules.

Building on top of our "empty type theory" we introduce $\to$ the function type former:

Definition 6.5.1 (Formation rules)**.** Our formation rules tell us how to construct arrow types from other types:

$$\frac{\Gamma \vdash A \; \mathsf{type} \qquad \Gamma \vdash B \; \mathsf{type}}{\Gamma \vdash A \to B \; \mathsf{type}} \; (\to\text{-form})$$

Definition 6.5.2 (Introduction rules)**.** Our introduction rule tells us how to construct terms of our type. This is also known as λ-abstraction:

$$\frac{\Gamma, x : A \vdash M \Leftarrow B}{\Gamma \vdash \lambda x.M \Rightarrow A \to B} \ (\to\text{-intro})$$

Definition 6.5.3 (Elimination rules)**.** Our elimination rule tells us how to use terms of this type. For function types this corresponds to application:

$$\frac{\Gamma \vdash M \Leftarrow A \to B \qquad \Gamma \vdash N \Leftarrow A}{\Gamma \vdash MN \Rightarrow A \to B} \ (\to\text{-elim})$$

Definition 6.5.4 (Computation rules)**.** And finally we have computation rules which tell us how to compute our terms. We will later prove results about normalisation of the lambda calculus. We start with β-reduction which tells us how applicated functions compute:

$$\frac{\Gamma, x : A \vdash y : B \qquad \Gamma \vdash t : A}{\Gamma \vdash (\lambda x.y)t \equiv y[t/x] : B} \ (\to\text{-}\beta)$$

Then we introduce η-conversion which tells us if two functions applied to the same term and are judgementally equal then the functions are judgementally equal. This is "functional extensionality" for judgemental equality.

$$\frac{\Gamma, y : A \vdash My \equiv M'y : B}{\Gamma \vdash M \equiv M' : A \to B} \ (\to\text{-}\eta)$$

Finally we have to make sure all our rules respect judgemental equality. This means showing that $\to$ respects judgemental equality of types and that λ-terms and applications respect judgemental equality of terms.

$$\frac{\Gamma \vdash A \equiv A' \ \mathsf{type} \qquad \Gamma \vdash B \equiv B' \ \mathsf{type}}{\Gamma \vdash A \to B \equiv A' \to B' \ \mathsf{type}} \ (\to\text{-}\equiv_{\mathsf{type}}\text{-cong})$$

$$\frac{\Gamma, x : A \vdash M \equiv M' : B}{\Gamma \vdash \lambda x.M \equiv \lambda x.M' : A \to B} \ (\to\text{-}\equiv_{\mathsf{term}}\text{-cong})$$

$$\frac{\Gamma \vdash M \equiv M' : A \to B \qquad \Gamma \vdash N \equiv N' : A}{\Gamma \vdash MN \equiv M'N' : A \to B} \ (\to\text{-elim-cong})$$

We define the product type as follows.

Definition 6.5.6 (Product type)**.** Given two types, we have their product type:

$$\frac{\Gamma \vdash A \ \mathsf{type} \qquad \Gamma \vdash B \ \mathsf{type}}{\Gamma \vdash A \times B \ \mathsf{type}} \ (\times\text{-form})$$

We define ordered pairs as taking a term of each type:

$$\frac{\Gamma \vdash a \Leftarrow A \qquad \Gamma \vdash b \Leftarrow B}{\Gamma \vdash (a, b) \Rightarrow A \times B} \ (\times\text{-intro})$$

We given an eliminator for the product type. Which is going to be uncurrying. This is equivalent to giving two "accessor" functions that project.

$$\frac{\Gamma \vdash f \Leftarrow A \to (B \to C)}{\Gamma \vdash \mathrm{ind}_{A\times B}(f) \Rightarrow A \times B \to C}\ (\times\text{-elim})$$

And we finally need to dictate how this is computed:

$$\frac{\Gamma \vdash a \Leftarrow A \qquad \Gamma \vdash b \Leftarrow B \qquad \Gamma \vdash f \Leftarrow A \to (B \to C)}{\Gamma \vdash \mathrm{ind}_{A\times B}(f)(a,b) \equiv f(a)(b) : C}\ (\times\text{-}\beta)$$

We will sometimes talk about the first and second elements of a pair, so it can be convenient to introduce the following functions with names. But we will not really consider this as part of the type theory, more rather a feature. There are various ways to formalise what we have done here, usually going by the name of a let statement or something.

$$\frac{}{\Gamma \vdash \mathrm{fst} \equiv \mathrm{ind}_{A\times B}(\lambda x.(\lambda y.x)) : A \times B \to C}$$

$$\frac{}{\Gamma \vdash \mathrm{snd} \equiv \mathrm{ind}_{A\times B}(\lambda x.(\lambda y.y)) : A \times B \to C}$$

It can be shown that $a : A, b : B \vdash \mathrm{fst}(a,b) \equiv a : A$ and $a : A, b : B \vdash \mathrm{snd}(a,b) \equiv b : B$. [[However I am not so sure that $t : A \times B \vdash (\mathrm{fst}(t), \mathrm{snd}(t)) \equiv t : A \times B$... Maybe $(\times\text{-}\eta)$ or something needs to be added]]

We will also need to add a unit type. This will be the simplest type, with only one term.

Definition 6.5.7 (Unit type)**.** We begin with the formation rules, essentially saying that the unit type exists.

$$\frac{}{\mathbf{1}\ \mathsf{type}}\ (\mathbf{1}\text{-form})$$

We then say that the unit type has a term:

$$\frac{}{\Gamma \vdash * \Rightarrow \mathbf{1}}\ (\mathbf{1}\text{-intro})$$

Next we specifiy that in order to create a function out of the unit type, it suffices to give a term $a : A$. This function is then called $\mathrm{ind}_{\mathbf{1}}(a) : \mathbf{1} \to A$.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathrm{ind}_{\mathbf{1}}(a) \Rightarrow \mathbf{1} \to A}\ (\mathbf{1}\text{-elim})$$

Next we show how this function computes, which is that for any $x \Leftarrow \mathbf{1}$ (of which there are not many) we have it being judgementally equal to $a : A$.

$$\frac{\Gamma vdash x \Leftarrow \mathbf{1}}{\Gamma \vdash \mathrm{ind}_{\mathbf{1}}(a)x \equiv a : A}\ (\mathbf{1}\text{-comp})$$

[[TODO: Clear up wording maybe?]]

Remark 6.5.8. We make an important note that this is not the simplest presentation of the STLC of which there are many variations thereof. We chose judgemental equality and bidirectional type checking because these are features we will need if we are to enrich our type system with dependent types.

6.6 Inversion lemmas

Having listed all these rules we need some lemmas detailing how different terms can *only* come from a set of specified rules. This is a crucial analysis if we wish to construct a type checking algorithm. An inversion lemma for a type theory is typically very difficult to state, and extremely tedious to prove. But nontheless is essential if we want to induct over terms.

[[TODO: State this beast]]

Lemma 6.6.1. In the STLC the following term forms are generated by certain rules...

7 Category theory

7.1 Introduction

7.2 Categories

We begin with the notion of a category. This can be thought of as a place mathematical objects live and interact with eachother. We have the eventual goal of deriving categorical semantics for simply typed lambda calculus.

Definition 7.2.1. 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 7.2.2. One typically writes composition as juxtaposition and omits the symbol $\circ$.

Example 7.2.3. 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 7.2.4. 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 7.2.5. It is a simple exercise to check that for any category $\mathcal{C}$

[TODO: Note on commutative diagrams]

7.3 Functors

Definition 7.3.1. 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*:

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

- *H respects units*:

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

7.4 Natural transformations

Definition 7.4.1. 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:

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

7.5 Having a left or right 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 7.5.1. 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 $A \in \mathcal{C}$ and for all $f : X \to UA$, there exists a *unique* map $g : FX \to A$ such that the following diagram commutes:

$$\begin{array}{ccc} X & \xrightarrow{\eta_X} & UFX \\ & \searrow^{f} & \downarrow{\scriptstyle Ug} \\ & & UA \end{array}$$

The definition of having a right adjoint is very similar, in fact it is a special case of being a left adjoint, however we will reproduce it here for simplicity:

Definition 7.5.2. A functor $U : \mathcal{C} \to D$ *has a right adjoint* if for all $X \in \mathcal{D}$, there exists an $RX \in C$ and $\epsilon_X : URX \to X$ in $\mathcal{D}$ such that for all $A \in \mathcal{C}$ and for all $h : UA \to X$, there exists a *unique* map $k : A \to RX$ such that teh following diagram commutes:

$$\begin{array}{ccc} UA & \xrightarrow{Uk} & URX \\ & \searrow^{h} & \downarrow{\scriptstyle \epsilon_X} \\ & & X \end{array}$$

Definition 7.5.3. These definitions are very minimalistic, in that there are not that many conditions to check. The notation we have used is very suggestive, for example greek letters

7.6 Products, Terminal objects and Cartesian closedness

Definition 7.6.1. A terminal object in a category $\mathcal{C}$ is an object T of $\mathcal{C}$ such that, for every object X of $\mathcal{C}$, there is a unique map from X to T.

Here we see the utility a

7.7 Display object categories

8 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 8.0.1. 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. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[5] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 34(4):839–864, 1933.

[6] Alonzo Church. A note on the entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936.

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

[8] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

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

[10] H. B. Curry. Grundlagen der kombinatorischen logik. *American Journal of Mathematics*, 52(3):509–536, 1930.

[11] H. B. Curry. Grundlagen der kombinatorischen logik. *American Journal of Mathematics*, 52(4):789–834, 1930.

[12] Haskell B. Curry. The inconsistency of certain formal logic. *The Journal of Symbolic Logic*, 7(3):115–117, 1942.

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

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

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

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

[17] S. C. Kleene. A theory of positive integers in formal logic. part i. *American Journal of Mathematics*, 57(1):153–173, 1935.

[18] S. C. Kleene. A theory of positive integers in formal logic. part ii. *American Journal of Mathematics*, 57(2):219–244, 1935.

[19] S. C. Kleene. λ -definability and recursiveness. *Duke Math. J.*, 2(2):340–353, 06 1936.

[20] S. C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36(3):630–636, 1935.

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

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

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

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

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

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

[27] 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.

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

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

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

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