# Introduction to dependent type theory

Ali Caglayan

March 14, 2019

## Contents

## 1 Syntax

### 1.1 Introduction

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

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

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

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

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

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

In Sørensen and Urzyczyn's book [12] 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 [14] 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).

## 1.2 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 [7, p. 62], [3, Ch. 7]. It's use in the formal semantics of programming languages is not much different either [15, Ch. 3]. There are however more constructive notions of well-foundedness [11, §8] with more careful use of excluded middle. We will follow

[13], as this is the simplest to understand, and we won't be using this material much other than an initial justification for induction in classical mathematics.

**Definition 1.2.1.** 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 1.2.2.** 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 1.2.3** (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 1.2.2 $Y = X$ giving us $\forall x \in X, P(x)$. $\square$

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

## 1.3 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 1.3.1** (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 1.3.2** (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 1.3.3** (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 $\alpha$. 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 1.3.4** (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 1.3.5** (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 1.3.6** (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 1.3.7.** 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 1.3.8** (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 1.3.9.** 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 1.3.10.** An ast can be thought of as a tree whose leaf nodes are variables and branch nodes are operators.

**Example 1.3.11** (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 $\lambda_x$ 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.

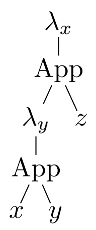

Figure 1: Vertically oriented tree representing the lambda term

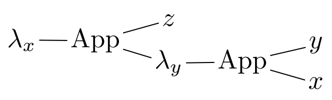

Figure 2: Horizontally oriented tree representing the lambda term

**Remark 1.3.12.** 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 1.3.13.** 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 1.3.14.** 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 1.3.15.** 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 1.3.16** (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. $[b/x]o(a_1; \dots; a_n) = o([b/x]a_1; \dots; [b/x]a_n)$

[Examples of substitution]

**Corollary 1.3.17.** 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 $a := o(a_1; \dots; a_n)$. 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 $a := o(a_1; \dots; a_n)$, we have by induction unique $c_1, \dots, c_n$ such that $c_i := [b/x]a_i$ for $1 \leq i \leq n$. Hence we have a unique $c = o(c_1, \dots, c_n) \in \mathcal{A}[\mathcal{X}]$. $\square$

**Remark 1.3.18.** Note that 1.3.17 was simply about checking Definition 1.3.16. 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 [5] does, with slight modifications that are used in [9, 10], 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 1.3.19** (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 $j$th argument is in scope of the $k$th 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 1.3.20.** In [5] there is no relation but a function. And each argument has bound variables assigned to it. But as argued in [10] 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 [10] 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 1.3.21** (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 $\alpha$. An element $o \in \mathcal{O}_{\alpha \in \mathbf{GA}}$ is called an operator of (generalised) **arity** $\alpha$. If $\alpha$ has sort $s$ then $o$ has **sort** $s$. If $\alpha$ has argument sorts $(s_1, \dots, s_n)$ then we say that $o$ has **argument arity** $n$, with the $j$th argument having **sort** $s_j$. If $\alpha$ has binding sorts $(t_1, \dots, t_m)$ then we say that $o$ has **binding arity** $m$, with the $k$th bound variable having **sort** $s_k$. If the the scoping relation of $\alpha$ has $j \lhd k$ then we say that the $j$th argument of $o$ is in **scope** of the $k$th bound variable of $o$.

**Remark 1.3.22.** 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 1.3.23** (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 $V = \{v_1, \dots, v_m\}$. For $j \in [n]$, let $X_j := \{v_k \in V \mid j \lhd k\}$ be the set of bound variables that the $j$th 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 $j$th argument of $\mathtt{G}$. Then $\mathtt{G}(X; M_1, \dots, M_n) \in \mathcal{B}[\mathcal{X}]_s$.

**Remark 1.3.24.** There is a lot going on in the second constructor of Definition 1.3.23. 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.]]

## 2 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 [5] though these ideas are much older. [Probably tracable back to Gentzen]. [There are many more references to be included here]

**Definition 2.0.1.** 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 2.0.2.** Typically a judgement is denoted $\mathsf{J}$. We can write $a\ \mathsf{J}$, $\mathsf{J}\ a$ to denote the judgment asserting that the judgement form $\mathsf{J}$ 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 2.0.3.** 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.*

**Remark 2.0.4.** 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 2.0.5.** 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 $-$ $\mathsf{nat}$:

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

We can see that an abt $a$ is zero or is of the form $\texttt{succ}(a)$. We see this by induction on the abt, the set of such abts has an operator $\texttt{succ}$. Taking these rules to be exhaustive, it follows that $succ(a)$ is a natural number if and only if $a$ is.

**Remark 2.0.6.** We used the word *exhaustive* without really defining it. By this we mean necessary and sufficient. Which we will define now.

**Definition 2.0.7.** 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 2.0.8.** 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 $\mathsf{nat}$.

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

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

## 2.1 Derivations

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

**Definition 2.1.1.** 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 $\mathtt{nat}$ we can give an example of a derivation.

**Example 2.1.2.** Here is a derivation of the judgement $\mathtt{succ(succ(succ(zero)))\ nat}$:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\mathtt{zero\ nat}}}{\mathtt{succ(zero)\ nat}}}{\mathtt{succ(succ(zero))\ nat}}}{\mathtt{succ(succ(succ(zero)))\ nat}}$$

**Remark 2.1.3.** 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 2.1.4.** 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 2.1.5.** 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 2.1.6.** 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. $\square$

**Remark 2.1.7.** 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!

## 2.2 Rule induction

[[Write about proving properties about a derivable judgement it suffices to show the property is closed under the defining judgement form]]

[[ We should also go back and consider the discussion on forward and backward chaining because I may have made some things up there. We will need to find some references about this]]

After we have rule induction, we can prove a few lemmas about our natural numbers example.

**Lemma 2.2.1.** 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. $\square$

**Lemma 2.2.2** (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. $\square$

**Lemma 2.2.3** (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. $\square$

**Lemma 2.2.4** (Symmetry of $=$)**.** If $a = b$, then $b = a$.

*Proof.* Begin with induction on $a$ and $b$:

- Suppose $a$ is of the form `zero` and $b$ is of the form `zero` then we have `zero` $=$ `zero` as desired.
- Suppose $a$ is of the form `zero` and $b$ is of the form `succ`$(b')$ then our statement is vacously true. The same happens for when $b$ is `zero` and $a$ is of the form `succ`$(a')$.
- Finally when $a$ is of the form `succ`$(a')$ and $b$ is of the form `succ`$(b')$ we have `succ`$(a') =$ `succ`$(b')$. By 2.2.3 we have $a' = b'$ and by our induction hypothesis we have $b' = a'$ as desired.

□

**Lemma 2.2.5** (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 `zero` our statement holds trivially.
- Whne $a$, $b$ and $c$ are of the form `succ`$(a')$, `succ`$(b')$ and `succ`$(c')$ respectively, we can apply 2.2.3 on `succ`$(a') =$ `succ`$(b')$ and `succ`$(b') =$ `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 `succ`$(a') =$ `succ`$(c')$.

□

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

## 3 Simply typed lambda calculus

## References

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

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

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

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

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

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

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

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

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

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

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

[12] 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.

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

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

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