Introduction to dependent type theory

Ali Caglayan

May 1, 2019

Contents

1 Introduction

1.1 What is type theory?

1.2 Summary of dissertation

The goal of this dissertation is to give an introduction to the formal study of lambda calculus and in general type theory. We begin by analysing the intuitive notion of *syntax*, highlighting the many subtleties associated with it. We discuss possible solutions to these issues, but ultimately remark that it is very difficult to be certain of correctness. We will however give a notion of syntax which is correct enough for our purposes.

The next section is to discuss the formality of *judgements*. This is a concept oft overlooked in the study of type theory. We will give a careful and detailed account of derivability and admissibility. We will also remark on inconsistencies of the treatment of certain concepts.

Next we will discuss the technology of *typing*. Even though it is a relatively simple idea, it has many powerful, and subtle, consequences. After a look at this static analysis, we will also discuss the dynamics of programming languages. We will later remark on common solutions to over come incorrect code and run time errors.

This will lead us into studying the *simply type lambda calculus* (STLC), in some ways one of the simplest (functional) programming languages. We will give syntax, judgements and rules. After which, we will prove metaproperties about our type theory and discuss the notion of *type checking*.

We will then analyse the dynamics of the STLC. There is a long history of normalisation results we wish to breifly sketch. We will set up some machinary to prove some of these results. Finally we will discuss notions of canonicity and what these results mean for the design of programming languages.

Next there will be several examples of terms to be type checked. This will show the intricacies that go into desiging a type checker. We will see that typing makes lambda calculus much weaker, in that many terms from the untyped lambda calculus cannot be typed. It is precisely these terms which gave the computational power of the untyped lambda calculus to begin with.

The next section will be a detailed account of the ideas that went in to, what is now known as the *Curry-Howard* correspondance. This is a very deep package of ideas with far reaching consequences, of which we will try to make account of.

We will use the knowledge gained from a study of Curry-Howard to design new types and data structures for our STLC, and turn it into a more powerful programming language, i.e. one that can support recursion. We make a note about encodings of natural numbers in the plain STLC, and why they are insufficient to really be called natural numbers.

Finally we will sketch a dependent type theory with Π and Σ types. We will not prove any formal properties of this type theory but using our previous work we will sketch how one might go about doing so. We will take this time to introduce the workings of dependent types and discuss their advantages over other type theory features.

Our closing remarks will be about future directions in type theory, questions that need to be answered and future of programming language design.

2 Syntax

Definition 2.0.1 (Sorts)**.** Let $\mathcal{S}$ be a finite set of elements called *sorts*.

Definition 2.0.2 (Arity)**.** An *arity* (or *signature*) consists of the following data:

1. A sort $s \in \mathcal{S}$.
2. A natural number n called the *argument arity*.
3. A natural number m called the *binding arity*.
4. A function $\mathrm{soa} : [n] \to \mathcal{S}$ called the *sort of argument* function.
5. A function $\mathrm{sov} : [m] \to \mathcal{S}$ called the *sort of variable* function.
6. A relation $\triangleleft \subseteq [n] \times [m]$ called *scoping*.

We denote the set of arities by $\mathbf{A}$.

Remark 2.0.3. Let $1 \leq k \leq n$ and $1 \leq l \leq m$. We say that the sort of argument k is $\mathrm{soa}(k)$ and the sort of variable l is $\mathrm{sov}(l)$. If $k \triangleleft l$ then we would say that the kth argument is in scope of the lth variable.

Remark 2.0.4. This is a modification to the definition given in [13]. In which each argument has a set of variables. For our purposes we want all arguments to use the same variables. This is achieved with a scoping relation. Details of this idea can be found in [20].

Definition 2.0.5. Let $\mathcal{O}$ be a set of elements called *operators*, and let $\mathrm{ArityOf} : \mathcal{O} \to \mathbf{A}$ be the function picking the *arity of an operator*. The arity of an operator $o \in \mathcal{O}$ is $\mathrm{ArityOf}(o)$.

Definition 2.0.6. A set of *variables* is simply a set $\mathcal{X}$ and a function $\mathrm{SortOf} : \mathcal{X} \to \mathcal{S}$ choosing the sort of the variable. We write $\mathcal{X}_s$ for all the variables $x \in \mathcal{X}$ with $\mathrm{SortOf}(x) = s \in \mathcal{S}$. Observe that $\mathcal{S}$ is the inverse image of SortOf over s.

Remark 2.0.7. Typically a set of variables is endowed with some sort of order. They are also typically countable. We could say that every set of variables should necesserily be equipped with an injection into the natural numbers.

Definition 2.0.8. We say a set of variables $\mathcal{V}$ is *fresh* for a set of variables $\mathcal{X}$ if $\mathcal{V} \cap \mathcal{X} = \varnothing$. We can then take the *union* of sets of variables $\mathcal{V} \cup \mathcal{U}$ with the obvious well-defined definition of SortOf.

Definition 2.0.9. The set of *abstract binding trees* (*abts*) of *sort* $s \in \mathcal{S}$ on a set of variables $\mathcal{X}$, is the least set $\mathcal{B}[\mathcal{X}]_s$ satisfying the following conditions:

1. If $x \in \mathcal{X}_s$ then $x \in \mathcal{B}[\mathcal{X}]_s$.
2. Let $\mathtt{G}$ be an operator of sort s, argument arity n, binding arity m. Let $\mathcal{V} := \{v_1, \dots, v_m\}$ be a finite set of m variables fresh for $\mathcal{X}$. For $1 \leq j \leq n$, let $\mathcal{Y}_j := \{v_k \in \mathcal{V} \mid j \triangleleft k\}$ be the set of variables that the jth argument is in scope of. Now suppose for each $1 \leq j \leq n$, there are $M_j \in \mathcal{B}[\mathcal{X} \cup \mathcal{Y}_j]_{\mathrm{soa}(j)}$. Then $\mathtt{G}(\mathcal{V}; M_1, \dots M_n) \in \mathcal{B}[\mathcal{X}]_s$.

Remark 2.0.10. Harper's notion of *abstract binding tree* is a generalisation of the more common *abstract syntax tree*. The difference is that abts keep track of how their variables are bound. We will later demonstrate this by showing how variable capture is avoided. The above definition may seem complicated but it is simply a tree where branches are operators and nodes are variables. All these trees do not live in the same set however since the bound and free variables are being kept track of.

Definition 2.0.11 (α-equivalence)**.** Let $\mathcal{X}$ and $\mathcal{X}'$ be bijective sets of variables, and let $\rho : \mathcal{X} \to \mathcal{X}'$ be a bijection. Define the following relation $\sim_\rho \subseteq \mathcal{B}[\mathcal{X}]_s \times \mathcal{B}[\mathcal{X}']_s$ by induction on both abts:

- If $x \in \mathcal{X}$ and $y \in \mathcal{X}'$ then $x \sim_\rho y$ if and only if $\rho(x) = y$.
- For bijective sets of variables $\mathcal{V}$ and $\mathcal{V}'$ of size n, free for $\mathcal{X}$ and $\mathcal{X}'$ respectively. By Remark 2.0.7 we give them orders. Let $\xi : \mathcal{V} \to \mathcal{V}'$ be the *unique* order-preserving bijection between them. For $1 \leq j \leq n$, let $\mathcal{Y}_j := \{v_k \in \mathcal{V} \mid j \triangleleft k\}$ and $\mathcal{Y}'_j := \{v'_k \in \mathcal{V}' \mid j \triangleleft k\}$ be the sets of variables the jth argument is in scope of in $\mathcal{V}$ and $\mathcal{V}'$ respectively. Observe that the restriction $\xi_j : \mathcal{Y}_j \to \mathcal{Y}'_j$ is also a bijection. Then $\mathtt{G}(\mathcal{V}; m_1, \ldots, m_n) \sim_\rho \mathtt{G}(\mathcal{V}'; m'_1, \ldots, m'_n)$ if and only if $m_j \sim_{\rho \cup \xi_j} M'_j$ for all $1 \leq j \leq n$.
- In all other cases the relation is false.

We say that an abt $a \in \mathcal{B}[\mathcal{X}]_s$ is *α-equivalent* to an abt $b \in \mathcal{B}[\mathcal{X}']_s$, written $a \simeq_\alpha b$, if there exists a bijection $\rho : \mathcal{X} \to \mathcal{X}'$ such that $a \sim_\rho b$.

We quickly sketch some routine proofs showing α-equivalence is in fact an equivalence relation.

Lemma 2.0.12 (Reflexivity)**.** α-equivalence is reflexive.

Proof. Observe that for any $m \in \mathcal{B}[\mathcal{X}]_s$ we have $m \sim_{\mathrm{id}} m$. □

Lemma 2.0.13 (Symmetry)**.** α-equivalence is symmetric.

Proof. Suppose $a \simeq_\alpha b$ then $a \sim_\rho b$ for some bijection ρ. The inverse ρ^{-1} is also a bijection, and observe that $b \sim_{\rho^{-1}} a$. □

Lemma 2.0.14. α-equivalence is transitive.

Proof. Suppose $a \simeq_\alpha b$ and $b \simeq_\alpha c$ then $a \sim_\rho b$ and $b \sim_{\rho'} c$ for some bijections ρ and ρ'. Observe that the composite $\rho' \cdot \rho$ is also a bijection, and that as a result $a \sim_{\rho' \cdot \rho} b$. It can then easily be checked that $a \simeq_\alpha c$. □

Corollary 2.0.15. α-equivalence is an equivalence relation.

Definition 2.0.16. Given $\sigma : \mathcal{X} \to \mathcal{Y}$ such that σ preserves sorts, i.e. $\mathrm{SortOf} \cdot \sigma = \mathrm{SortOf}$, we define a function $\mathcal{B}[\mathcal{X}]_s \to \mathcal{B}[\mathcal{Y}]_s$ denoted $a \mapsto a[\sigma]$ by induction on a:

- If $a = x \in \mathcal{X}$, then $x[\sigma] = \sigma(x)$.
- If $a = \mathtt{G}(\mathcal{V}; m_1, \ldots, m_n)$ we would like to define $a[\sigma]$ as

$$\mathtt{G}(\mathcal{V}, m_1[\sigma], \ldots, m_n[\sigma])$$

but this is not possible since $\mathcal{V}$ may not be disjoint from $\mathcal{Y}$. Therefore we observe that we can accomadate for this by first freshening up our variables in $\mathcal{V}$ with respect to $\mathcal{Y}$ by finding another $\mathcal{V}'$ whose elements are all fresh in $\mathcal{Y}$ and $\mathcal{V} \simeq_\alpha \mathcal{V}'$. We will call such an operation $\mathcal{V}^{[\mathcal{Y}]}$ and then define $\mathtt{G}(\mathcal{V}; m_1, \dots, m_n)[\sigma] := \mathtt{G}(\mathcal{V}^{[\mathcal{Y}]}; m_1[\sigma], \dots, m_n[\sigma])$.

If σ is an inclusion, then the operation is *weakening* (at the level of syntax). If σ is a permutation (a self-bijection) then this is known as *exchange* (at the level of syntax). If σ is a surjection, then the operation is *contraction* (at the level of syntax).

Remark 2.0.17. We must be weary not to get confused later on with the *structural rules* with the same names. These operations are intrinsic to syntax, and are not directly related with rules we will look at later.

Remark 2.0.18. It can be seen that α-equivalence between a and b can be stated as the existence of a bijection ρ such taht $a[\rho] = b$.

Lemma 2.0.19. Given functions σ and σ', we have $a[\sigma][\sigma'] = a[\sigma' \cdot \sigma]$.

Proof. Expanding the definition of $a[\sigma]$ and $a[\sigma][\sigma']$ this can be observed. □

Lemma 2.0.20. $\mathrm{sub}_a lphaIf\rho, \rho'$ are bijections, then $a \sim_\rho b$ implies $a[\sigma] \sim_{\rho'} b[\rho' \cdot \sigma \cdot \rho^{-1}]$. Hence the operation $-[\sigma]$ respects α-equivalence.

Proof. By Remark 2.0.18 and Lemma 2.0.19, we have $a \sim_\rho b \iff a[\rho] = b \iff a = b[\rho^{-1}]$. So $a[\sigma] = b[\rho^{-1}][\sigma] = b[\sigma \cdot \rho^{-1}]$ and hence $a[\sigma][\rho'] = b[\rho' \cdot \sigma \cdot \rho^{-1}] \iff a[\sigma] \sim_{\rho'} b[\rho' \cdot \sigma \cdot \rho^{-1}]$. □

Definition 2.0.21. We override our definition of abstract binding tree by defining the set of all abts of sort s over a set of variables $\mathcal{X}$ as $\mathcal{B}[\mathcal{X}]_s / \simeq_\alpha$.

Remark 2.0.22. Whenever we refer to an abt we typically write it as some representing element of the equivalence class.

Remark 2.0.23. Due to Lemma **??**, Definition 2.0.16 makes sense for equivalence classes too. Thus we do not need to change the meaning of $-[\sigma]$, by simply noting that it acts on representatives of the equivalcen class in a well-defined way.

Definition 2.0.24. We call the disjoint union $\sqcup_{s \in \mathcal{S}} \mathcal{B}[\mathcal{X}]_s$ of abts over $\mathcal{X}$ with sort s the set of all abts over $\mathcal{X}$. We could have defined this first and then defined $\mathcal{B}[\mathcal{X}]_s$ as the inverse image of some sort choosing function over a sort s like in Definition 2.0.6. When we talk about the sort of an abt $a \in \mathcal{B}[\mathcal{X}]$ we refer to the s which corresponds to the set in which a lives.

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 [13] though these ideas are much older. [Probably traceable 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 judgement 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 judgement 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 $J_1, \dots, J_k$ are all judgements of the form being defined. The judgements above the horizontal line are called the *premises* 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 $J_1, \dots J_k$. Axioms hold unconditionally. If the conclusion of a rule holds it is not necessarily 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 $-\ \mathsf{nat}$:

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

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 `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`.

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

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 $\triangledown_1, \dots, \triangledown_k$ are derivations of its premises, then

$$\frac{\triangledown_1 \quad \cdots \quad \triangledown_k}{J}$$

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 $\texttt{succ}(\texttt{succ}(\texttt{succ}(\texttt{zero})))\ \mathsf{nat}$:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\texttt{zero}\ \mathsf{nat}}}{\texttt{succ}(\texttt{zero})\ \mathsf{nat}}}{\texttt{succ}(\texttt{succ}(\texttt{zero}))\ \mathsf{nat}}}{\texttt{succ}(\texttt{succ}(\texttt{succ}(\texttt{zero})))\ \mathsf{nat}}$$

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 relevant 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 necessarily 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 assistants, 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 algorithmic ally 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:

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

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:

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

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

$$\frac{\dfrac{}{\mathsf{empty\ tree}} \qquad \dfrac{\dfrac{}{\mathsf{empty\ tree}} \quad \dfrac{}{\mathsf{empty\ tree}}}{\mathsf{node}(\texttt{empty}; \texttt{empty})\ \mathsf{tree}}}{\mathsf{node}(\texttt{empty}; \mathsf{node}(\texttt{empty}; \texttt{empty}))\ \mathsf{tree}}$$

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

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

This is the familiar 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 $\texttt{succ}(a)\ \mathsf{nat}$, then $a\ \mathsf{nat}$.

Proof. By induction on $\texttt{succ}(a)$, when $\texttt{succ}(a)$ is $\texttt{zero}$ this is vacuously true. Otherwise when $\texttt{succ}(a)$ is succ(b), what we want to prove is $\texttt{succ}(b)\ \mathsf{nat} \implies b\ \mathsf{nat}$ 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 $\texttt{succ}$)**.** If $\texttt{succ}(a_1) = \texttt{succ}(a_2)$, then $a_1 = a_2$.

Proof. We perform induction on $\texttt{succ}(a_1)$ and $\texttt{succ}(a_2)$. Note that if any of the two are of the form $\texttt{zero}$ then the statement is true vacuously. When $\texttt{succ}(a_1)$ is of the form $\texttt{succ}(b_1)$ and $\texttt{succ}(a_2)$ is of the form $\texttt{succ}(b_2)$ 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 `zero` and b is of the form `zero` then we have $\texttt{zero} = \texttt{zero}$ as desired.
- Suppose a is of the form `zero` and b is of the form $\texttt{succ}(b')$ then our statement is vacuously true. The same happens for when b is `zero` and a is of the form $\texttt{succ}(a')$.
- 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 vacuously true, so we will prove the other two:

- When a, b and c are of the form `zero` our statement holds trivially.
- When 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 arithmetic 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-Löf'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 hypothesis 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, $J_1, \dots, J_k \vdash_{\mathcal{R}} K$ where each J_i and K are basic judgements, to mean that we may derive K from the *expansion* $\mathcal{R} \cup \{J_1, \dots, J_k\}$ 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 capital 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:

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

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 $\Gamma, K \vdash_{\mathcal{R}} J$. Entailment 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 replace 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 vacuously 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 application 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 simultaneous 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 $\texttt{zero}\ \mathsf{odd}$. 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 hypothetical 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 hypothetical 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 hypothetical 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 ascertain 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 correspondence 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 in fact not necessarily needed. In fact 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 favourite dependent types. Show how useful they are for programmers and mathematicians

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 languages 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 guaranteed 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 type checking 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

Things 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 judgmentally 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 Plotkin's 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 $e\ \mathsf{val}$, 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 exceptional case when dividing 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 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 briefly 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 5.0.1. The sorts of simply typed lambda calculus are terms and types $\mathcal{S} := \{\mathrm{tm}, \mathrm{ty}\}$.

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

Definition 5.0.2. The operators in the syntax of simply typed lambda calculus are given by the following table:

Op	Sort	Vars	Type args	Term args	Scoping	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 5.0.3. Note that some of the 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 $\mathrm{App}_{A,B}(M; N)$. Having this information in the syntax will be useful when we want to induct over syntax, for example when proving an initiality theorem. But in practice we will save ourselves from having to write it out.

Definition 5.0.4. We can now construct our raw terms and types as the collection of abts (see definition **??**) over the previously defined data $\mathrm{Term} := \mathcal{B}[\varnothing]_{\mathrm{tm}}$ and $\mathrm{Type} := \mathcal{B}[\varnothing]_{\mathrm{ty}}$.

Remark 5.0.5. 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 sub-terms (sub-trees 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]].

5.1 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 type checking, 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 becoming what is known as an "equational theory". This will be a useful feature for when we want to derive 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:

- $\Gamma \vdash A \ \mathsf{type}$ - "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".
- $\Gamma \vdash A \equiv B \ \mathsf{type}$ - "A and B are judgmentally equal types in context Γ".
- $\Gamma \vdash S \equiv T : A$ - "S and T are judgmentally equal terms of type A in context Γ".

5.2 Structural rules

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

Definition 5.2.1. 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 succinctly as:

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

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!]]

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:

$$\frac{\Gamma \vdash t \Rightarrow A \qquad \Gamma \vdash A \equiv B \ \mathsf{type}}{\Gamma \vdash t \Leftarrow B}\ (\text{cswitch})$$

Remark 5.2.2. This rule has been specially set up in that it will be the *only way* to derive $\Gamma \vdash T \Leftarrow B$. These are the kinds of properties we would like our syntax to have. A careful analysis will be done under the name of *inversion lemma*. [[Link to inversion lemma?]]

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]]

Remark 5.2.3. Occasionally, we will simply mode-switch using reflexivity $\Gamma \vdash A \equiv A \ \mathsf{type}$, in which case we will abbreviate the rule as follows:

$$\frac{\Gamma \vdash t \Rightarrow A}{\Gamma \vdash t \Leftarrow A}\ (\text{switch})$$

5.3 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 each other.

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

Definition 5.3.1. We wish for our judgemental equality of types to be reflexive:

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

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:

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

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:

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

5.4 Type formers

What we have constructed thus far is essentially an "empty type theory". What we have included which other authors typically gloss over is a clean way of constructing a type checking algorithm: bidirectional type checking 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 derived 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 5.4.1. Our formation rules tell us how to construct arrow types from other types:

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

Our introduction rule tells us how to construct terms of our type. This is also known as λ-abstraction:

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

Our elimination rule tells us how to use terms of this type. For function types this corresponds to application:

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

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 applied functions compute:

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

Then we introduce η-conversion which tells us if two functions applied to the same term and are judgmentally equal then the functions are judgmentally equal. This is "function 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.

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

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

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

Remark 5.4.2. Notice that we don't ensure that types compute the same way. This is because the computation rules will not be used in the type checking process and are therefore irrelevant to the inversion lemmas. Later we will prove that "fully reduced" computations are in fact equal. This is known as the Church-Rosser theorem.

We define the product type as follows.

Definition 5.4.3 (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 give two eliminators for pairs, the first and second elements:

$$\frac{\Gamma \vdash t \Leftarrow A \times B}{\Gamma \vdash \text{fst}(t) \Rightarrow A} \ (\times\text{-elim}_1) \qquad \frac{\Gamma \vdash t \Leftarrow A \times B}{\Gamma \vdash \text{snd}(t) \Rightarrow B} \ (\times\text{-elim}_2)$$

And we finally need to dictate how this is computed:

$$\frac{\Gamma \vdash x \Leftarrow A \qquad \Gamma \vdash y \Leftarrow B}{\Gamma \vdash \text{fst}(x, y) \equiv x : A} \ (\times\text{-}\beta_1)$$
$$\frac{\Gamma \vdash x \Leftarrow A \qquad \Gamma \vdash y \Leftarrow B}{\Gamma \vdash \text{snd}(x, y) \equiv y : B} \ (\times\text{-}\beta_2)$$

However we need to be careful since there is a nontrivial equality we must also add as a rule:

$$\frac{\Gamma \vdash \text{fst}(t) \equiv \text{fst}(t') : A \qquad \Gamma \vdash \text{snd}(t) \equiv \text{snd}(t') : B}{\Gamma \vdash t \equiv t' : A \times B} \ (\times\text{-}\eta)$$

Remark 5.4.4. There are many other ways to present product types, the eliminators are in a sense not unique. Typically in presentations of type theory

[[LIKE IN MARTIN-LOF]] an inductive principle is given. This is simply just a way to build functions out of the type, the elimination principle is stated like that. What we note is that rule is in fact admissible in the presence of our fst and snd eliminators. We also argue that the fst and snd approach more closely matches what a programmer will do with the type theory. Elimination principles in general correspond to left[[or right I need to check]] universal properties of the categorical semantic counterparts.

Remark 5.4.5. Our presentation of η-reduction is unconventional. The traditional η

$$\frac{\Gamma \vdash t \Leftarrow A \times B}{\Gamma \vdash (\mathrm{fst}(t), \mathrm{snd}(t)) \equiv t}$$

Is in fact admissible by observing the following proof tree: [[Include admissibility tree]] We choose our presentation because it more clearly display what η really means and why it is there.

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

Definition 5.4.6 (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:

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

Remark 5.4.7. We don't need to give any more rules since the unit type has all the properties we need. Our rules for $\to$ allow us to build constant functions anyway. And we note that all functions $\mathbf{1} \to A$ are constant functions!

[[TODO: Clear up wording maybe?]]

Remark 5.4.8. 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.

5.5 Inversion lemmas

Having listed all these rules, we need *Inversion lemmas* detailing how different judgements can *only* come from a set of given judgements. 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 nonetheless is essential if we want to induct over terms. These are also known as *Generation lemmas* [25, 13].

Luckily we set up syntax in such a way that we only need induct over the syntax. So we pick a syntactic form and the inversion lemma will tell us exactly how we can arrive at that conclusion. Let us list all term syntax we can create in STLC. We will write them in Backus-Naur form (BNF) [CITATION] which is a common and clear way to write inductive generators:

$$\text{Term} ::= x \mid \lambda x.a \mid (a, b) \mid ab \mid c$$

Where x is a variable, a, b are terms and c is a constant, in this case any of $*$, fst, snd. We may also list the types that we have:

$$\text{Type} ::= A \times B \mid A \to B \mid \mathbf{1}$$

Where A, B are types.

- x where x is a variable.
- $\lambda x.M$ where M is a term.
- (x, y) where x and y are terms.
- fst, snd the eliminators of $\times$
- $*$ the element of $\mathbf{1}$
- $\text{ind}_{\mathbf{1}}$ the eliminator of $\mathbf{1}$

Lemma 5.5.1.

[[TODO: State this beast]]

Lemma 5.5.2. In the STLC the following term forms are generated by certain rules...

6 Normalisation of STLC

6.1 Introduction

We now wish to analyse the computational power of our type theory. When designing the type checking algorithm we made a point not to invoke any computational rules, since this will give us a decidable type checking algorithm. We now wish to show that successive applications of mode-switching, betas and eta will always terminate and to the same term, this will be known as the *normal form*. The theorem is known as the Church-Rosser theorem [[CITE]]. This is a subtle property of the type theory and is determined by the computational rules we have added. Further addition of term constructors and type formers should leave this property untouched.

Our proof will follow the proof in [25, p. 67] albeit with modifications to make it work here. [[TODO rewrite and add good citations]]

6.2 Properties of relations

First we define what we mean by a binary relation being *compatible* with the syntax of the STLC.

Definition 6.2.1. A binary relation $\succ$ on Term the set of all terms, is said to be *compatible with the syntax of STLC* (or just simply *compatible*) if the following conditions hold:

1. If $M \succ N$ then $\lambda x.M \succ \lambda x.N$.
2. If $M \succ N$ then $MZ \succ NZ$.
3. If $M \succ N$ then $ZM \succ ZN$.
4. If $M \succ N$ then $(Z, M) \succ (Z, N)$.
5. If $M \succ N$ then $(M, Z) \succ (N, Z)$.

Remark 6.2.2. The notion of compatibility allows us to make sure a relation also considers sub-terms. This is a tricky thing to get right but due to our focus on the correct structure of syntax we are fine.

Remark 6.2.3 ([CLEAN THIS UP)**.**] The reader may ask what relations have to do with normalisation, but it is a formalism that we have chosen. This is definitely not the only way to prove properties like Church-Rosser. The main reason we have chosen this method is for its simplicity. In fact earlier we discussed the dynamics of languages, this is exactly that. There are many ways to go about dynamics including transition systems and equational dynamics. Our approach corresponds to the more classical and simple transition systems approach. It can be shown that this is equivalent to equational dynamics in that a reduction step will be justified by application of rules from STLC.

We will demonstrate our last remark by considering the following relation:

Definition 6.2.4. Let $\sim_{\text{ty}}$ denote the relation amond terms of having the same type. Suppose $\Gamma \vdash s \Leftarrow S$ and $\Gamma \vdash t \Leftarrow T$, then:

$$s \sim_{\text{ty}} t \iff \Gamma \vdash S \equiv T \ \mathsf{type}$$

Lemma 6.2.5. The relation $\sim_{\text{ty}}$ is a compatible relation.

Proof. Suppose $M \sim_{\text{ty}} N$, then we have $\Gamma \vdash M \Leftarrow S$, $\Gamma \vdash N \Leftarrow T$ and $\Gamma \vdash S \equiv T$ type.

□

Definition 6.2.6. Given a relation $\succ$ on a set X, we denote by $\succ^+$ the *transitive closure* of $\succ$. This is the smallest relation which coincides with $\succ$ and is transitive. We also consider the *reflexive-transitive closure* $\succ^*$ of $\succ$ which is simply the relation $\Delta(X) \cup \succ^+$ where $\Delta(X)$ is the image of the diagonal function $x \mapsto (x, x)$. (We've simply added that $x \succ^* x$)

Remark 6.2.7. Transitive closures correspond to chains of the relation, and reflexive-transitive closures allow for chains of length 0. It should also be noted that we took the *union* of a relation. This is a well-defined notion and can easily be seen to be a relation.

Let $\to$ be a binary relation on a set A, $\twoheadrightarrow^+$ be its transitive closure and $\twoheadrightarrow$ be its reflexive-transitive closure.

Now we define (very generally) what it means for an element of a set to be in *normal form* and *normalising* with respect to some relation.

Definition 6.2.8. An element $a \in A$ is said to be of *normal form* if $\forall b \in A$, $a \not\twoheadrightarrow b$.

Definition 6.2.9. An element $a \in A$ is said to be *normalising* (or *weakly normalising*) if there is a reduction sequence $a \to a_1 \to a_2 \to \cdots \to a_n$ where a_n is in normal form, for some n. We call a_n a *normal form* or *reduct* of a.

Remark 6.2.10. Note that not every reduction sequence is guaranteed to be finite. We also note that if $\to$ a relation is Church-Rosser (to be defined below) then a_n is *the* normal form or reduct.

We discuss what it means for a relation to be Church-Rosser:

Definition 6.2.11. A relation $\to$ has the *Church-Rosser* (CR) property if and only if for all $a, b, c \in A$ such that $a \twoheadrightarrow b$ and $a \twoheadrightarrow c$, there exists $d \in A$ with $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$.

Remark 6.2.12. This says no matter what path we take along a relation, there will always be elements at which the paths cross.

We will also need a slightly weaker version called weak Church-Rosser, for reasons we will see later:

Definition 6.2.13. A relation $\to$ has the *weak Church-Rosser* (WCR) property if and only if for all $a, b, c \in A$ such that $a \to b$ and $a \to c$, there exists $d \in A$ with $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$.

We now state the obvious:

Corollary 6.2.14. If $\to$ is CR then $\to$ is WCR.

[TODO.] □

The converse to this is in general *false* but it is true when another condition holds, namely that $\to$ is *strongly normalising.*

Definition 6.2.15. A binary relation $\to$ is *strongly normalising* (SN) if and only if there is no infinite sequence $a_0 \to a_1 \to a_2 \to \cdots$.

Remark 6.2.16. In other words, a relation $\to$ is strongly normalising if and only if *every* sequence $a_0 \to a_1 \to a_2 \to \cdots$ terminates after a finite number of steps.

Remark 6.2.17. We typically also say an element is strongly normalising if the condition holds for that element. This allows us to state SN in a different (and perhaps more correct) way: A relation $\to$ is strongly normalising if each element is strongly normalising with respect to $\to$. Then we can define an element to be strongly normalising if all of it's reducts are strongly normalising. The nice thing about this definition is that we have seen it before, this is precisely what it means to be a *well-founded relation* from Defintion **??**. So $\to$ is strongly normalising if and only if it is well-founded. This is good because we can induct over it!

Corollary 6.2.18. If a relation $\to$ is strongly normalising then every element is normalising.

Proof. By induction on $\to$ we see that either an element is in normal form, or it reduces to normal form. This is precisely what it means to be normalising. □

We now state a lemma which will be very useful. It is a sufficient condition for the converse of Corollary 6.2.14 to hold.

Lemma 6.2.19 (Newman's Lemma)**.** If $\to$ is strongly normalising and WCR then it is CR.

Proof. Since $\to$ is strongly normalising, any $a \in A$ has a normal form. Call an element *ambiguous* if a reduces to two distinct normal forms. Clearly $\to$ is CR if there are no ambiguous elements of A. Assume, for contradiction, that there is an ambiguous a. We will show that there is another ambiguous a' where $a \to a'$. Suppose we have $a \twoheadrightarrow b_1$ and $a \twoheadrightarrow b_2$ where b_1 and b_2 are two different normal forms. Both reductions must make at least one step, thus both reductions can be written as $a \to a_1 \twoheadrightarrow b_1$ and $a \to a_2 \twoheadrightarrow b_2$. Suppose $a_1 = a_2$ then we can choose $a' = a_1 = a_2$. Now suppose $a_1 \neq a_2$, we know by WCR that $a_1 \twoheadrightarrow b_3$ and $a_2 \twoheadrightarrow b_3$ for some b_3. We can assume that b_3 is a normal form. Since b_1

and b_2 are distinct, b_3 is different from b_1 or b_2 so we can choose $a' = a_1$ or $a' = a_2$. Since we can always choose an a', we can repeat this process and get an infinite chain of ambiguous elements. It is clear that this contradicts strongly normalising, hence A has no ambiguous elements. $\square$

6.3 Normalisation

Now we define what we mean by β-reduction and β-normal form.

Definition 6.3.1. We define *β-reduction* to be the least compatible relation $\to_\beta$ on Term satisfying the following conditions:

1. $(\lambda x.y)t \to_\beta y[t/x]$
2. $\mathrm{fst}(x, y) \to_\beta x$
3. $\mathrm{snd}(x, y) \to_\beta y$

A term on the left hand side of any of the above is called a *β-redex* (reducible expression) and the right hand sides are said to *arise by contracting the redex.*

Remark 6.3.2 ([Clear up wording)**.**] Observe that these are very similar to our β rules, in fact they are exactly those. So the question may arise: why haven't we defined β-reduction using the rules that we already have? The answer is that we could but we would have a much harder time, the rules also take into account typing information but we are explicitly not worried about that since we will show later β-reduction doesn't change a typed terms type. It is somewhat simpler and clearer to focus purely on terms. We will later justify calling this β-reduction.

Definition 6.3.3. A term M is said to be in *β-normal form* if it is in normal form with respect to $\to_\beta$.

Remark 6.3.4. That is to say a term is in β-normal form if there is no β-reduction to any other term. Or better yet, M does not contain a β-redex.

Definition 6.3.5. Let $\twoheadrightarrow_\beta$ be the transitive and reflexive closure of $\to_\beta$ called a *multi-step β-reduction.*

Remark 6.3.6. Not every term is normalising. Take for example the term $\Omega = (\lambda x.xx)(\lambda x.xx)$ which cannot be typed as we will see later. There is an infinite reduction sequence:

$$\Omega \to_\beta \Omega \to_\beta \Omega \to_\beta \Omega \to_\beta \cdots$$

Since Ω cannot be given a type, it is deemed *ill-typed.*

This means we have to be careful which terms we are talking about. When talking about terms of the STLC we should add that we expect them to be well-typed (derivable). We will see later there are many syntactically valid terms that are ill-typed.

We want to now prove that every derivable term is β-normalising. In order to do this we need to keep track of available redexes and bound them. We will then show there is a reduction strategy that decreases this bound yielding our result.

This proof is usually attributed to an unpublished note of Turing [[CITE]] but it has been rediscovered by various authors. We will follow the proof in Girard's book [10].

Definition 6.3.7. The *degree* $\partial(T)$ *of a type* T is defined by:

- $\partial(T) := 1$ if T is atomic.
- $\partial(U \times V), \partial(U \to V) := \max(\partial(U), \partial(V)) + 1$.

Definition 6.3.8. The *(β-)degree* $\partial_\beta(t)$ *of a redex* is defined by:

- $\partial_\beta(\mathrm{fst}(u, v)), \partial_\beta(\mathrm{snd}(u, v)) := \partial(U \times V)$ where $\Gamma \vdash (u, v) \Leftarrow U \times V$.
- $\partial_\beta((\lambda x.v)u) := \partial(U \to V)$ where $\Gamma \vdash \lambda x.v \Leftarrow U \to V$.

Definition 6.3.9. The *(β-)degree* $d_\beta(t)$ *of a term* is the maximum of the degrees of its redexes:

$$d_\beta(t) := \max\{\partial_\beta(s) \mid s \text{ is a redex in } t\}$$

Remark 6.3.10. A redex is associated to two degrees, one as a redex and another as a term. Since a redex r may contain other redexes we have that $\partial(r) \leq d(r)$. It should be noted we have defined degree to mean 3 different things here, but as long as we are careful we should not get confused.

Lemma 6.3.11. If r is a redex of type T then $\partial(T) < \partial_\beta(r)$.

Proof. Checking the cases for r:

- $\partial(T) < \partial_\beta(\mathrm{fst}(t, u)) = \max(\partial(T), \partial(U)) + 1$.
- $\partial(T) < \partial_\beta(\mathrm{snd}(u, t)) = \max(\partial(U), \partial(T)) + 1$.
- $\partial(T) < \partial_\beta((\lambda x.t)u) = \max(\partial(U), \partial(T)) + 1$.

□

Lemma 6.3.12. If $\Gamma, x : T \vdash t \Leftarrow U$ then $d_\beta(t[u/x]) \leq \max(d_\beta(t), d_\beta(u), \partial(T))$.

Proof. Analysing the redexes of $t[u/x]$ we find that they fall into the following cases:

- They are redexes of t (in which u has become x).
- They are redexes of u, proliferating due to each occurence of x in t.
- They are formed when t is of the form $\mathrm{fst}(x)$, $\mathrm{snd}(x)$, or xv for u of the form (u', u''), (u', u''), or $\lambda y.u'$ respectively. These new redexes have degree $\partial(T)$.

□

Lemma 6.3.13. If $t \to_\beta u$ then $d_\beta(u) \leq d_\beta(t)$.

Proof. Consider the reduction where u is obtained from t by replacing the redex r in u by c. Now we consider all the redexes of u where we find:

- redexes which were originally in t, but not in r, and have been modified by the replacement of r by c. Observe that their degree does not change.
- redexes which were originally in c. But c is obtained by reducing r, or in other words a substitution in r. Notice $(\lambda x.s)s'$ becomes $s[s'/x]$ and Lemma 6.3.12 tells us that $d_\beta(c) \leq \max(d_\beta(s), d_\beta(s'), \partial(T))$, where T is the type of x. But by Lemma 6.3.11 we have $\partial(T) \leq \partial(r)$. Applying max gives us $\max(d(s), d(s'), \partial(T)) \leq \max(d_\beta(s), d_\beta(s'), \partial_\beta(r))$ and hence $d_\beta(c) \leq \max(d_\beta(s), d_\beta(s'), \partial(r)) = d(r)$.
- redexes which come from replacing r by c. These redexes have degree equal to $\partial(T)$ where T is the type of r. By Lemma 6.3.11 we have $\partial(T) \leq \partial(r)$.

□

Next we will prove a lemma bounding the number of redexes of a certain degree.

Lemma 6.3.14. Let r be a redex of maximal degree n in t, and suppose that all redexes strictly contained in r have degree less than n. If u is obtained from t by reducing r to c. Then u has strictly fewer redexes of degree n.

Proof. When the reduction happens we make the following observations:

- The redexes outside r in t remain u.
- The redexes strictly inside r are in general conserved but sometimes become more prolific. Take for example $(\lambda x.(x, x))s \to_\beta (s, s)$. The number of redexes in the reduct are double that of redex on the left. However the degree of the proliferated redexes must be strictly less than n.

- The redex r is destroyed and possibly replaced by redexes of strictly smaller degree.

□

Remark 6.3.15. Although not defined, we take the meaning of *a redex strictly inside* to be a redex that is not the whole redex.

We now have all the machinary needed to prove that typed terms in the STLC are weakly β-normalising.

Theorem 6.3.16. Every derivable term $\Gamma \vdash t \Leftarrow A$ in the STLC is β-normalising.

Proof. Consider the function $\mu : \mathbf{Term} \to \mathbb{N} \times \mathbb{N}$ which takes $t \mapsto (n, m)$ where $n = d_\beta(t)$ and m is the number of redexes in t of degree n. By Lemma 6.3.14 it is possible to choose a redex r of t in such a way that, after reduction of r to c, the reduct t' satisfies $\mu(t') < \mu(t)$. Thus by double induction on n and m it is possible to see that $\mu(t)$ can always be decreased until t is normal. □

Remark 6.3.17. The ordering in $\mu(t') < \mu(t)$ on $\mathbb{N} \times \mathbb{N}$ is the lexicographic ordering. Meaning $(n', m') < (n, m)$ if and only if $n' < n$ or $n' = n$ and $m' < m$. (Think Alphabetical order).

Lemma 6.3.18. Suppose $\Gamma \vdash M \Leftarrow T$ and $M \twoheadrightarrow_\beta N$, then $\Gamma \vdash M \equiv N : T$.

[TODO.] □

Definition 6.3.19. We define *η-reduction* to be the least compatible relation $\to_\eta$ on Term satisfying the following conditions:

1. $\lambda x.fx \to_\eta f$
2. $(\mathrm{fst}(t), \mathrm{snd}(t)) \to_\eta t$

Just like for β-reduction we have the notions of *η-redex* and terms that *arise by contracting the redex.*

Definition 6.3.20. A term is said to be in η-normal form if it is in normal form with respect to $\to_\eta$.

Definition 6.3.21. Let $\twoheadrightarrow_\eta$ be the transitive and reflexive closure of $\to_\eta$ called a *multi-step η-reduction.*

We will now show that $\to_\eta$ is strongly normalising.

Remark 6.3.22. Originally we had thought to modify the proof of β-normalisation, and make it work for η. However, this is where the difference between the two is key. β-normalisation has the power to create new β-redexes whereas η-normalisation never does. In fact η-normalisation is strongly normalising even in the untyped lambda calculus. This suggests that talking about degrees is not the correct approach and there ought to be some other metric for which can be used to bound η-reducible terms. Based off of work in [9], the authors of [25, Ex. 3.21] define a *depth* function for terms. We belive this to be the actual depth of the underlying tree of the abstract binding tree of the syntax of the term. But that is not a relevent result for now.

Definition 6.3.23. Given a term t we define the *depth* $\delta(\mathsf{t})$ *of* t by induction on terms:

- $\delta(x) := 0$ for x a variable or constant.
- $\delta(ab) := 1 + \max(\delta(a), \delta(b))$.
- $\delta(\lambda x.y) := 1 + \delta(y)$.
- $\delta((a, b)) := 1 + \max(\delta(a), \delta(b))$.

Lemma 6.3.24. If $t \to_\eta u$ then $\delta(u) < \delta(t)$.

Proof. Observe that since $\to_\eta$ is a compatible relation, we need only prove the statement for a redex. We do this by cases:

- $$\begin{aligned}\delta((\mathrm{fst}(s), \mathrm{snd}(s))) &= 1 + \max(\delta(\mathrm{fst}(s)), \delta(\mathrm{snd}(s)))\\ &= 1 + \max(1 + \delta(s), 1 + \delta(s))\\ &= \delta(s) + 2\end{aligned}$$
- $$\begin{aligned}\delta(\lambda x.sx) &= 1 + \delta(sx)\\ &= 2 + \max(\delta(s), \delta(x))\\ &= \delta(s) + 2\end{aligned}$$

Observe that in both cases we have that the depth of a redex s is $\delta(s) = \delta(r) + 2$ where r is the reduct of s. However at the level of terms we cannot garantee equality due to the nature of depth and compatibility. ☐

Lemma 6.3.25. η-reduction is strongly normalising.

Proof. By Lemma 6.3.24 we have that the depth of any η-reduction sequence is strictly decreasing. Hence there may only be finitely many steps in any given η-reduction sequence. ☐

Lemma 6.3.26. Suppose $\Gamma \vdash M \Leftarrow T$ and $M \twoheadrightarrow_\eta N$, then $\Gamma \vdash M \equiv N : T$.

Proof. Observe that in the definition of □

Now we take some results from Takahashi [] [[NEED CITTIOONS]], who considers *parallel reductions*. This is a stronger relation than $\to_\beta$ and weaker than $\twoheadrightarrow_\beta$. This might not seem like much but, parallel (β-)reduction, denoted $\Rightarrow_\beta$ (not to be confused with our typing judgements), satisfies the diamond in Church-Rosser. And since $\twoheadrightarrow_\beta$ is the transitive closure of $\Rightarrow_\beta$, it too satisfies the diamond in Church-Rosser hence $\to_\beta$ has the Church-Rosser property. We will formalise this argument as follows and consider β, η and $\beta\eta$ reductions along the way.

Definition 6.3.27. *Parallel β-reduction*, $\Rightarrow_\beta$, is defined inductively on terms by the following rules:

1. $x \Rightarrow_\beta x$ for a variable x.
2. $\lambda x.M \Rightarrow_\beta \lambda x.M'$ if $M \Rightarrow_\beta M'$.
3. $MN \Rightarrow_\beta M'N'$ if $M \Rightarrow_\beta M'$ and $N \Rightarrow_\beta N'$.
4. $(M, N) \Rightarrow_\beta (M', N')$ if $M \Rightarrow_\beta M'$ and $N \Rightarrow_\beta N'$.
5. $(\lambda x.M)N \Rightarrow_\beta M'[N'/x]$ if $M \Rightarrow_\beta M'$ and $N \Rightarrow_\beta N'$.
6. $\mathrm{fst}(M, N) \Rightarrow_\beta M'$ if $M \Rightarrow_\beta M'$.
7. $\mathrm{snd}(M, N) \Rightarrow_\beta N'$ if $N \Rightarrow_\beta N'$.

Remark 6.3.28. If we expand the definition of compatible in the definition of $\to_\beta$ it may appear to be identical to the definition of $\Rightarrow_\beta$. The key difference is the direction in which we are building up the terms. In the above definition we are breaking down the syntax and making sure that *all* components also satisfy the relation. We will see later the relation with $\to_\beta$.

Remark 6.3.29. The name comes from the fact that parallel reduction can reduce many β-redexes at once, unlike usual reduction.

Corollary 6.3.30. The relation $\Rightarrow_\beta$ is reflexive.

Proof. Observe that ignoring the last three rules in the definition of $\Rightarrow_\beta$ we still cover all the syntax. □

Lemma 6.3.31. We have the following implications:

$$M \to_\beta M' \quad \Longrightarrow \quad M \Rightarrow_\beta M' \quad \Longrightarrow \quad M \twoheadrightarrow_\beta M'$$

Proof. For the first implication, observe that a redex in M is being contracted to such that $M \to_\beta M'$. We can also contract the redex in the definition of $M \Rightarrow_\beta M'$ by choosing the correct rule. For the second implication, proceed by induction on M:

- If $x \Rightarrow_\beta M'$ then clearly $M' = x$ hence $x \twoheadrightarrow x'$.
- If

□

Now we need a small technical lemma that will show the utility of being strongly normalising.

Lemma 6.3.32. If there is an infinite $\beta\eta$-reduction sequence starting from M, then there is an infinite β-reduction sequence starting from M.

[TODO.] □

We are interested in the contrapositive form of this lemma:

Corollary 6.3.33. If there is no infinite β-reduction sequence starting from M, then there is no infinite $\beta\eta$-reduction sequence starting from M.

Remark 6.3.34. In particular this means that $\rightarrow_\beta$ being strongly normalising implies that $\rightarrow_{\beta\eta}$ is strongly normalising.

Theorem 6.3.35. β-reduction is strongly normalising.

[TODO.] □

Corollary 6.3.36. $\beta\eta$-reduction is strongly normalising.

Lemma 6.3.37. $\beta\eta$-reduction is WCR.

[TODO.] □

Theorem 6.3.38. The Church-Rosser property holds for $\beta\eta$-reduction.

[TODO.] □

Remark 6.3.39. So not only does every well-typed term have a normal-form, but it is in fact unique!

6.4 Canonicity

[[These two concepts are very related, we should find some way to talk about it, including Church-Rosser]]

7 STLC Examples

7.1 Identity function $\lambda x.x$

Example 7.1.1 (Identity function)**.** Let's consider the following lambda term $\lambda x.x$. We wish to find a type T such that given some context Γ we have $\Gamma \vdash \lambda x.x \Leftarrow T$. The only rule that allows us to get to this judgement is the mode-switching rule (switch). We also have the oppurtunity to add some structure to the type, so we keep this in mind.

$$\dfrac{\Gamma \vdash \lambda x.x \Rightarrow \boxed{?} \qquad \Gamma \vdash T \equiv \boxed{?}\ \mathsf{type}}{\Gamma \vdash \lambda x.x \Leftarrow T}\ \text{(switch)}$$

A first guess on what $\boxed{?}$ is could be T. But this would be a baseless claim to make. Since our syntax has some structure, we can infer what the type ought to be. Checking the conclusions of our rules, we need to find something that will roughly match $\Gamma \vdash \lambda x.x \Rightarrow \boxed{?}$. We see that there is only one rule that fits: ($\to$-intro). We also see that $\boxed{?}$ will have to be $A \to B$ for some types A and B. So we *must* have the following hypothesis in order to progress:

$$\Gamma \vdash T \equiv A \to B\ \mathsf{type} \tag{$*$}$$

Hence we must have the following derivation:

$$\dfrac{(\to\text{-intro})\ \dfrac{\Gamma, x : A \vdash x \Leftarrow B}{\Gamma \vdash \lambda x.x \Rightarrow A \to B} \qquad \Gamma \vdash T \equiv A \to B\ \mathsf{type}}{\Gamma \vdash \lambda x.x \Leftarrow T}\ \text{(switch)}$$

We need to now resolve the hypothesis $\Gamma, x : A \vdash x \Leftarrow B$. Observing the conclusions of our rules we see that we must mode-switch. As before we have a chance to change our type, so we play the same game with the holes:

$$\dfrac{(\to\text{-intro})\ \dfrac{\dfrac{\Gamma, x : A \vdash x \Rightarrow \boxed{?} \qquad \Gamma, x : A \vdash B \equiv \boxed{?}\ \mathsf{type}}{\Gamma, x : A \vdash x \Leftarrow B}\ \text{(switch)}}{\Gamma \vdash \lambda x.x \Rightarrow A \to B} \qquad \Gamma \vdash T \equiv A \to B\ \mathsf{type}}{\Gamma \vdash \lambda x.x \Leftarrow T}\ \text{(switch)}$$

Now observe that there is precisely one rule, the variable rule (var), with a hypothesis in the form of $\Gamma, x : A \vdash x \Rightarrow \boxed{?}$, but for this to be correct we have to place A into $\boxed{?}$. This means we will have to assume:

$$\Gamma, x : A \vdash B \equiv A\ \mathsf{type} \tag{$**$}$$

But since the hypothesis of (var) is quite clearly true, namely that $(x : A) \in \Gamma, x : A$ we are done! Here is the full derivation tree:

$$\dfrac{(\to\text{-intro})\ \dfrac{\dfrac{(\text{var})\ \dfrac{(x:A)\in\Gamma,x:A}{\Gamma,x:A\vdash x\Rightarrow A} \qquad \Gamma,x:A\vdash B\equiv A\ \mathsf{type}}{\Gamma,x:A\vdash x\Leftarrow B}\,(\text{switch})}{\Gamma\vdash\lambda x.x\Rightarrow A\to B} \quad \Gamma\vdash T\equiv A\to B\ \mathsf{type}}{\Gamma\vdash\lambda x.x\Leftarrow T}\,(\text{switch})$$

However we are not quite done yet. We have two type equations $(*)$ and $(**)$ to resolve. It is clear that if we choose $B := A$ and $T := A \to A$ we can resolve all our equational hypotheses. So in actual fact a derivation tree would look like this:

$$\dfrac{\dfrac{\dfrac{\dfrac{(x:A)\in\Gamma,x:A}{\Gamma,x:A\vdash x\Rightarrow A}\,(\text{var})}{\Gamma,x:A\vdash x\Leftarrow A}\,(\text{cswitch})}{\Gamma\vdash\lambda x.x\Rightarrow A\to A}\,(\to\text{-intro})}{\Gamma\vdash\lambda x.x\Leftarrow A\to A}\,(\text{cswitch})$$

It is clear when using the compact mode-switching, the derivation tree is much easier to understand and read. But when searching for a type we necessarily have to use regular mode-switching.

7.2 Function application $\lambda x.\lambda y.xy$

Example 7.2.1 (Function application)**.** Here is another example of a term that type checks. We want to find a type T such that $\Gamma \vdash \lambda x.\lambda y.xy \Leftarrow T$ is true. A derivation tree can be found in Appendix B.2. Here is a proof:

Proof. We begin with the judgement $\Gamma \vdash \lambda x.\lambda y.xy \Leftarrow T$, now the only way to arrive at this judgement is via the mode-switching rule. Whilst doing this we add type variables A and B which can easily be seen to form into $A \to B$ and let $T \equiv A \to B$. We can come back later and validate this judgement. The mode-switching should have given us $\Gamma \vdash \lambda x.\lambda y.xy \Rightarrow A \to B$ which we can only arrive at by applying the ($\to$-intro) rule. This gives us $\Gamma, x : A \vdash \lambda y.xy \Leftarrow B$. Which we have to mode-switch, and as before we take this chance to introduce type variables C and D in order to arrive at the judgement $\Gamma, x : A \vdash \lambda y.xy \Rightarrow C \to D$. This allows us to apply ($\to$-intro) giving us $\Gamma, x : A, y : C \vdash xy \Leftarrow D$. Now we apply the ($\to$-elim) rule since we have an application. For this we need $\Gamma, x : A, y : C \vdash y \Leftarrow C$, which is marked as ($\dagger$), and observe that a simple application of mode-switching and the variable rule allows us to derive this judgement. The other hypothesis we need is $\Gamma, x : A, y : C \vdash x \Leftarrow C \to D$. Again by mode-switching and setting $C \to D \equiv A$ we get $\Gamma, x : A, y : C \vdash x \Rightarrow A$ which is clearly derivable by the variable rule.

Now we have 3 type equations $(*)$, $(**)$ and $(***)$, substituting back in we get $\Gamma \vdash T \equiv (C \to D) \to C \to D$ for some types C and D. So $\Gamma \vdash \lambda x.y.xy \Leftarrow T$ if we have types C and D. $\square$

Remark 7.2.2. The density of information in the previous proof is one reason why it is sometimes clearer to draw a derivation tree. The crucial lesson is the choices we have to make at each step. We have set up our rules in such a manner that there is very often only *one* choice that can be made. When being ambiguous about the type we start with, we are in essense *inferring* typing information. Simply typed lambda calculus where the terms do not have typing information is typically referred to as Curry-style. Where as when terms are annotated with their types it is referred to as Church-style [25].

7.3 Mockingbird $\lambda x.xx$

In the untyped lambda calculus, λ-terms with no free variables can be called combinators. By combining combinators interesting expressions can be written in a readable way. This is related to the idea of combinatory logic which was mostly developed by Haskell Curry. Many combinators have been recorded and one of the best known references is [24], *"To Mock a Mockingbird"*. We will take a look at some other combinators later but we will start with the Mockingbird, also known as $\lambda x.xx$.

Example 7.3.1. We wish to find a type T such that $\Gamma \vdash \lambda x.xx \Leftarrow T$ for some context Γ. We begin, as with every begining, with the (switch) rule. We take this time to assume that $\Gamma \vdash T \equiv A \to B$ type given that there is no other way for a λ-term, as discussed in previous examples. This gives us $\Gamma \vdash \lambda x.xx \Rightarrow A \to B$. Observing that we can only apply ($\to$-intro) we arrive at $\Gamma, x : A \vdash xx \Leftarrow B$. First performing a (cswitch) we get $\Gamma, x : A \vdash xx \Rightarrow B$ which then points us to ($\to$-elim). This gives us $\Gamma, x : A \vdash x \Leftarrow \boxed{?}$ and $\Gamma, x : A \vdash x \Leftarrow \boxed{?} \to B$. Now we need to resolve both of these, the first is easier. We can see that we will have to (cswitch) and then use the variable rule, since this is the only judgement that matches with our hole. This also allows us to deduce that $\boxed{?}$ can be filled with A, yielding $\Gamma, x : A \vdash x \Leftarrow A \to B$. This is where things become problematic. We can of course apply the switch rule. But the only way to do this is with the hypothesis $\Gamma, x : A \vdash A \equiv A \to B$ type. And we see that $\Gamma, x : A \vdash x \Rightarrow A$ resolves via (var). At this point it would seem we are done, but now we will show the importance of checking the type equations we hypothesised. We set up judgemental equality in such a way that if $\Gamma \vdash A \equiv B$ type then the abts A and B where equal as abts. Thus we have an equation $A = A \to B$, but this is impossible! This means that the term $\lambda x.xx$ cannot be typed! This is the first stark difference we have seen compared to the untyped lambda calculus. It is typical to assume that by adding typing information to lambda calculus we will break nothing. But as we can clearly see, this is not the case.

Remark 7.3.2. This also presents an oppurtunity to show why we can *only normalise typed terms.* Using the notion of β-reduction we define back in 6.3.1, it appears that $\lambda x.xx$ is in *(β-)normal form.* This is misleading since any application of this function to some other term will not be able to reach normal form:

$$(\lambda x.xx)(\lambda x.xx) \to_\beta yy[\lambda x.xx/y] = (\lambda x.xx)(\lambda x.xx) \to_\beta \cdots$$

This is very similar to the example in Remark 6.3.6. Of course here, we have stayed fixed, but it is not too difficult to see how a term such as $\lambda x.xxx$ can get very much out of hand when attempting to normalise it. So it is not as if typing is a proof trick, which allows us to prove normalisation, but a property of computation in STLC. Only well-typed programs can run.

7.4 Aye-aye $(\lambda x.x)(\lambda x.x)$

It doesn't mean however any expression containing an ill-typed term is ill-typed. Take for instance $(\lambda x.x)(\lambda x.x)$ which may be written as $(\lambda x.xx)(\lambda x.x)$. As we saw in 7.3.1, $\lambda x.xx$ cannot be typed.

Example 7.4.1. Now suppose we want to show $\Gamma \vdash (\lambda x.x)(\lambda x.x) \Leftarrow T$ for some type T. We begin with (cswitch) noting that we will later use ($\to$-elim) so there is no reason to introduce an equality. From $\Gamma \vdash (\lambda x.x)(\lambda x.x) \Rightarrow T$ we use ($\to$-elim) to arive with two hypotheses $\Gamma \vdash \lambda x.x \Leftarrow A \to T$ and $\Gamma \vdash \lambda x.x \Leftarrow A$ for some type A. Here we might be inclined to think something has gone wrong, since we have the same term being typed in two different ways! But this is not the case.

We noted in [[ALPHA EQUIVALENCE]] that variables were really just considered up to α-equivalence and that it is always sensible to change when things get confusing. We also noted that such intricacies are the source of many problems in theory and implementation of type theories. It's not hard to see that $\lambda x.x$ can have any type $A \to A$ we give it. This is because it is simply the identity function. Therefore we could have equally written the judgements as $\Gamma \vdash \lambda x.x \Leftarrow A \to T$ and $\Gamma \vdash \lambda y.y \Leftarrow A$ and this would not have been confusing.

Working on the first we see that after a (cswitch) we get $\Gamma \vdash \lambda x.x \Rightarrow A \to T$ which allows us to use ($\to$-intro) giving us $\Gamma, x : A \vdash x \Leftarrow T$. We see that switching with $\Gamma, x : A \vdash T \equiv A \ \mathsf{type}$ leads to $\Gamma, x : A \vdash x \Rightarrow A$ which is obviously true by (var). Applying the weakening rule on our type equation $\Gamma, x : A \vdash T \equiv A \ \mathsf{type}$ gives us $\Gamma \vdash T \equiv A \ \mathsf{type}$ hence going back to $\Gamma \vdash \lambda y.y \Leftarrow A$ we can switch with $\Gamma \vdash A \equiv C \to D$ in order to be able to progress with ($\to$-intro). Now applying ($\to$-intro) to $\Gamma \vdash \lambda y.y \Rightarrow C \to D$ we get $\Gamma, y : C \vdash y \Leftarrow D$. Mode switching with $\Gamma, y : C, \vdash D \equiv C \ \mathsf{type}$ we get $\Gamma, y : C \vdash y \Rightarrow C$ which is true by (var). We finally see that $\Gamma \vdash T \equiv C \to C \ \mathsf{type}$, and that $\Gamma \vdash (\lambda x.x)(\lambda x.x) \Rightarrow A \to A$ for some type A. An important thing to note, is that even though we have two syntactically identical terms that look like $\lambda x.x$ the type information we gave them had to be different. In this case $(A \to A) \to (A \to A)$ for the first occurance and $A \to A$ for the second.

$$\text{(cswitch)}\,\dfrac{\text{(}{\to}\text{-elim)}\,\dfrac{\text{(cswitch)}\,\dfrac{\text{(}{\to}\text{-intro)}\,\dfrac{\text{(cswitch)}\,\dfrac{\text{(var)}\,\dfrac{(x:A\to A)\in\Gamma,x:A\to A}{\Gamma,x:A\to A\vdash x\Rightarrow A\to A}}{\Gamma,x:A\to A\vdash x\Leftarrow A\to A}}{\Gamma\vdash\lambda x.x\Rightarrow(A\to A)\to(A\to A)}}{\Gamma\vdash\lambda x.x\Leftarrow(A\to A)\to(A\to A)} \quad \dfrac{\dfrac{\dfrac{\dfrac{(x:A)\in\Gamma,x:A}{\Gamma,x:A\vdash x\Rightarrow A}\,\text{(var)}}{\Gamma,x:A\vdash x\Leftarrow A}\,\text{(cswitch)}}{\Gamma\vdash\lambda x.x\Rightarrow A\to A}\,\text{(}{\to}\text{-intro)}}{\Gamma\vdash\lambda x.x\Leftarrow A\to A}\,\text{(cswitch)}}{\Gamma\vdash(\lambda x.x)(\lambda x.x)\Rightarrow A\to A}}{\Gamma\vdash(\lambda x.x)(\lambda x.x)\Leftarrow A\to A}$$

7.5 Bluebird mocks the bluebird $\lambda x.\lambda y.(xy)(xy)$

7.6 Y-combinator $\lambda x.(\lambda y.x(yy))(\lambda y.x(yy))$

7.7 Function composition $\lambda x.\lambda y.\lambda z.x(yz)$

Here we will try something different, and perhaps more typical. We will provide ourselves with the type. We wish to check that function composition, written as $\lambda x.\lambda y.\lambda z.x(yz)$ has the type we expect it to: $(B \to C) \to (A \to B) \to (A \to C)$.

Example 7.7.1. We will show that for some types A, B and C, we have function composition $\Gamma \vdash \lambda x.\lambda y.\lambda z.x(yz) \Leftarrow (B \to C) \to (A \to B) \to (A \to C)$. A nice thing about being given the type is that type checking becomes much simpler. In fact, there is very little we actually need to do. Oberserve that in Appendix B.3 we have a derivation tree. We could write this all out as a proof but it would be pointless. As can be seen the derivation tree is much easier to read. This is the convention we will take for terms with a given type from now on: giving only a derivation tree.

7.8 Currying $\lambda x.\lambda y.\lambda z.x(y, z)$

Here is an interesting function that is quite useful. [[TODO discuss currying]] A full derivation is given in B.5.

7.9 Uncurrying $\lambda x.\lambda y.x(\text{fst}(y))(\text{snd}(y))$

[[TODO discuss un-doing functions]]

A full derivation is given in B.6.

7.10 Curry-Uncurry

We will now show that composing curry with uncurry gives us the identity. Unfortunately λ-terms will get very large so we will instead write them in a shorter way.

Example 7.10.1 (Curry-Uncurry)**.** We will assume the following:

- $\Gamma \vdash \mathbf{C} \equiv \lambda x.\lambda y.\lambda z.x(y, z) : (A \times B \to C) \to A \to B \to C$ denotes the function Curry.

- $\Gamma \vdash \mathbf{U} \equiv \lambda x.\lambda y.x(\mathrm{fst}(y))(\mathrm{snd}(y)) : (A \to B \to C) \to A \times B \to C$ denotes the function Uncurry.
- $\Gamma \vdash \mathbf{B} \equiv \lambda x.\lambda y.\lambda z.x(yz) : ((A \to B \to C) \to A \times B \to C) \to ((A \times B \to C) \to A \to B \to C) \to (A \times B \to C) \to (A \times B \to C)$ is the composition of two functions, conveniently with the correct types for curry and uncurry.

This means we want to derive the following:

$$\Gamma \vdash \mathbf{BUC} \equiv \lambda x.x : (A \times B \to C) \to (A \times B \to C)$$

Luckily we won't do this by hand and we will instead use a property of our type theory: Canonicity. This says that the normal form of a type is canonical. This means that reducing our terms to $\beta\eta$-normal form will be equal by reflexivity. Clearly $\lambda x.x$ is in normal form so we need to work on the left hand side.

First let us reduce $\mathbf{BU}$:

$$\begin{aligned}
\mathbf{BU} &= (\lambda x.\lambda y.\lambda z.x(yz))(\lambda x.\lambda y.x(\mathrm{fst}(y))(\mathrm{snd}(y)) \\
&= (\lambda a.\lambda b.\lambda c.a(bc))(\lambda x.\lambda y.x(\mathrm{fst}(y))(\mathrm{snd}(y)) \\
&\to_\beta \lambda b.\lambda c.(\lambda x.\lambda y.x(\mathrm{fst}(y))(\mathrm{snd}(y))(bc) \\
&\to_\beta \lambda b.\lambda c.\lambda y.bc(\mathrm{fst}(y))(\mathrm{snd}(y))
\end{aligned}$$

We are now in normal form for $\mathbf{BU}$ so we can reduce the whole of $\mathbf{BUC}$:

$$\begin{aligned}
\mathbf{BUC} = \mathbf{BU}&(\lambda i.\lambda j.\lambda k.i(j,k)) \\
&\twoheadrightarrow_\beta (\lambda b.\lambda c.\lambda y.bc(\mathrm{fst}(y))(\mathrm{snd}(y)))(\lambda i.\lambda j.\lambda k.i(j,k)) \\
&\to_\beta \lambda c.\lambda y.(\lambda i.\lambda j.\lambda k.i(j,k))c(\mathrm{fst}(y))(\mathrm{snd}(y)) \\
&\to_\beta \lambda c.\lambda y.(\lambda j.\lambda k.c(j,k))(\mathrm{fst}(y))(\mathrm{snd}(y)) \\
&\to_\beta \lambda c.\lambda y.(\lambda k.c(\mathrm{fst}(y),k))(\mathrm{snd}(y)) \\
&\to_\beta \lambda c.\lambda y.c(\mathrm{fst}(y),\mathrm{snd}(y)) \\
&\to_\eta \lambda c.\lambda y.cy \\
&\to_\eta \lambda c.c
\end{aligned}$$

Hence we clearly have

$$\Gamma \vdash \mathbf{BUC} \equiv \lambda x.x : (A \times B \to C) \to (A \times B \to C)$$

Remark 7.10.2. The previous example suggests an algorithm for deciding whether two terms are judgementally equal. Simply take the $\beta\eta$-normal form and compare terms. Since $\beta\eta$-reduction is strongly normalising, we see that equality of terms in simply typed lambda calculus is in fact decidable! [[LINK BACK TO NORMALISATION HERE]]

7.11 Swap $\lambda t.(\mathrm{snd}(t), \mathrm{fst}(t))$

This example demonstates a simple operation that manipualtes a data structure. We will later show that composing this function with itself is the identity.

Example 7.11.1. The type of $\lambda t.(\mathrm{snd}(t), \mathrm{fst}(t))$ is $A \times B \to B \times A$. Intuitively, this function simply swaps the order in an ordered pair. Here is a derivation tree showing that $\Gamma \vdash \lambda t.(\mathrm{snd}(t), \mathrm{fst}(t)) \Leftarrow B \times A \to A \times B$.

$$
\begin{prooftree}
\AxiomC{$(t : A \times B) \in \Gamma, t : A \times B$}
\LeftLabel{(var)}
\UnaryInfC{$\Gamma, t : A \times B \vdash t \Rightarrow A \times B$}
\LeftLabel{(cswitch)}
\UnaryInfC{$\Gamma, t : A \times B \vdash t \Leftarrow A \times B$}
\LeftLabel{($\times$-elim$_2$)}
\UnaryInfC{$\Gamma, t : A \times B \vdash \mathrm{snd}(t) \Rightarrow B$}
\LeftLabel{(cswitch)}
\UnaryInfC{$\Gamma, t : A \times B \vdash \mathrm{snd}(t) \Leftarrow B$}
\AxiomC{$(t : A \times B) \in \Gamma, t : A \times B$}
\RightLabel{(var)}
\UnaryInfC{$\Gamma, t : A \times B \vdash t \Rightarrow A \times B$}
\RightLabel{(cswitch)}
\UnaryInfC{$\Gamma, t : A \times B \vdash t \Leftarrow A \times B$}
\RightLabel{($\times$-elim$_1$)}
\UnaryInfC{$\Gamma, t : A \times B \vdash \mathrm{fst}(t) \Rightarrow A$}
\RightLabel{(cswitch)}
\UnaryInfC{$\Gamma, t : A \times B \vdash \mathrm{fst}(t) \Leftarrow A$}
\LeftLabel{($\times$-intro)}
\BinaryInfC{$\Gamma, t : A \times B \vdash (\mathrm{snd}(t), \mathrm{fst}(t)) \Rightarrow B \times A$}
\LeftLabel{(cswitch)}
\UnaryInfC{$\Gamma, t : A \times B \vdash (\mathrm{snd}(t), \mathrm{fst}(t)) \Leftarrow B \times A$}
\LeftLabel{($\to$-intro)}
\UnaryInfC{$\Gamma \vdash \lambda t.(\mathrm{snd}(t), \mathrm{fst}(t)) \Rightarrow B \times A$}
\LeftLabel{(cswitch)}
\UnaryInfC{$\Gamma \vdash \lambda t.(\mathrm{snd}(t), \mathrm{fst}(t)) \Leftarrow B \times A$}
\end{prooftree}
$$

7.12 Swap-Swap

We will now demonstrate that the swap function composes with itself to give the identity.

Example 7.12.1. We follow a similar argument to example 7.10.1.

8 Curry-Howard correspondence

8.1 Mathematical logic

At the beginning of the 20th century, Whitehead and Russell published their *Principia Mathematica* [21], 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 influential mathematicians of the time was David Hilbert. Inspired by Whitehead and Russell's vision, Hilbert and his colleagues 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 [12], 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 definition of "effectively calculable" was required. So along came three proposed definitions of what it meant to be "effectively calculable": *lambda calculus*, published in 1936 by Alonzo Church [4]; *recursive functions*, proposed by Gödel in 1934 later published in 1936 by Stephen Kleene [19]; and finally *Turing machines* in 1937 by Alan Turing [26].

8.2 Lambda calculus

(Untyped) lambda calculus was discovered by Church at Princeton, originally as a way to define notations for logical formulas. It is a remarkably compact idea, with only three constructs: variables; lambda abstraction; and function application.z It is closely related to Curry's idea of combinatory logic [6, 7] 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." [1, 2]. 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. [15, 16]. 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.

8.3 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 [3] and Kleene later published it in detail [17]. This however did not have the intended effect on Gödel, whereby he then became convinced that his own definition was incorrect!

8.4 Turing machines

Alan Turing was at Cambridge 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 Turing's argument that later convinced Gödel that this was the correct notion of "effectively calculable".

8.5 Russell's paradox

[Talk about the origin of types and stuff]

8.6 The problem with lambda calculus as a logic

Church's students Kleene and Rosser quickly discovered that lambda calculus was inconsistent as a logic [18]. Curry later simplified the result which became known as Curry's paradox [8]. Curry's paradox was related to rissoles 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* [5]. 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. [26] [CITATION NEEDED] From this consistency of Church's STLC as a logic could be established.

8.7 Types to the rescue

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

8.8 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]

8.9 Curry and Howard

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

8.10 Propositions as types

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

8.11 Predicates [CHANGE] as types?

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

8.12 Dependent types

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

9 Simply typed lambda calculus with products, sums and natural numbers

9.1 Introduction

Historically the addition of a natural numbers type with a recursion principle $\mathbb{N}$ was done by Gödel in his *"System T"* of Higher-Order recursion. This is different than having *encoded* numbers in type theory. For example in $\lambda_{\to\times}$we have *Church numerals* [[CITE EXAMPLE]], and we have seen that it is possible to do basic arithmetic with. Church-Encodings are what are known as impredicative encodings, whereby the terms of the types are the same as the desired one but the eliminators are not present. This is demonstrated for Church-encodings of the natural numbers by the fact that it is *impossible* to define recursion over the natural numbers in $\lambda_{\to\times}$[**?**] [[CITATION NEEDED]]. This isn't the case for *untyped* lambda calculus however. It is well-known that untyped lambda calculus can have recursive definitions, but they come at a cost. Not every term in untyped lambda calculus is normalising. This corresponds to a computation which doesn't halt and is intimately related to the halting problem.[[CITE]] A natural numbers type can however be added to $\lambda_{\to\times}$leading to a type theory that is "equivalent" to Gödel's system T.

We will also look at some other types such as sums and 0 and eventually exhibit the properties of this type theory as a propositional logic, as the Curry-Howard corresondence suggests.

9.2 Natural numbers

We add natural numbers. This will be our first example of an *inductive type.* We will call the corresponding type theory $\lambda_{\to\times\mathbb{N}}$ and note that it enjoys *canonicity.* Meaning that not only do all terms *normalise* but they normalise to a canonical form. This means if we have a function that computes a natural number, we are guaranteed to get a numeral (an iterated number of successors to zero). If we had some rules in our type theory that broke canonicity, we may get a term that type checks as a natural number but isn't judgmentally equal to one.

9.3 Sum types

[[Sum types go by the name of unions in C, whereas product types correspond to structs.]]

They are like disjoint unions of sets.

Their induction principle is very simple, to build a function out of $A + B$ it suffices to give a function out of A and another out of B.

10 Dependent types

We have seen previously that the Curry-Howard correspondence is a deep parallel between logic and computation. We therefore will use it as a guiding principle for a type theory. This was originally sketched by Curry [[CITE]] and the project taken up by Per Martin-Löf [[CITE]]. In order to begin modifying our rules for the STLC we need to introduce the notion of a *universe*.

11 Universes

11.1 Introduction

Originally Martin-Löf had added a type of all types. But this, unsurprisingly, led to Russellian paradoxes. This is known as Girard's paradox [11]. There is a simple resolution to this, which is inspired to a similar technique in set theory known as *Grothendieck universes*, though the type theoretic counterpart is much simpler to state [23].

There are two approaches to universes. Universes a la Russel and universes a la Tarski. The former is much simpler to state but loses unicity of typing. The latter keeps unicity of typing and corresponds closely with the semantic models, however unfortunately has many annotations and extra congruence rules. It is generally believed that the latter can be compressed into the former, and the former annotated to give the latter. [[CITE]]

Of course we don't actually *need* universes to discuss dependent types, but we will soon see that there aren't many interesting dependent types we can write down if we have no way of letting types vary over terms. In order to do this we need to be able to write down a *type family*, which is a function $F : A \to \mathcal{U}$ from a type A to some universe U, giving us each $F(a)$ as a type, i.e. $F(a)$ varies with $a : A$.

11.2 Universes a la Tarski

We add a type $\mathcal{U}$ which has terms for each type former in our type theory. Universes are a general concept which can be added to many type theories but we will restrict ourselves and add it to $\lambda_{\to\times}$the simply typed lambda calculus.

Definition 11.2.1 (Universes a la Tarski)**.** For every $i \in \mathbb{N}$ we have a universe type:

$$\frac{}{\Gamma \vdash \mathcal{U}_i \ \mathsf{type}} \ (\mathcal{U}_i\text{-form})$$

This type has a special property in that all it's terms are types:

$$\frac{\Gamma \vdash a \Leftarrow \mathcal{U}_i}{\Gamma \vdash \mathsf{El}_i(a)\ \mathsf{type}}\ (\text{Universe}_1)$$

In particular there is a term u_i in $\mathcal{U}_{i+1}$:

$$\frac{}{\Gamma \vdash u_i : \mathcal{U}_{i+1}}\ (\text{Universe}_2)$$

This little universe is in fact the corresponding universe as a type:

$$\frac{}{\Gamma \vdash \mathsf{El}_{i+1}(u_i) \equiv \mathcal{U}_i\ \mathsf{type}}\ (\text{Universe}_3$$

Now for each type former of $\lambda_{\to\times}$we add an introduction rule and add some congruence rules.

Remark 11.2.2. It quickly turns out that we essentially double the number of rules that we have by adding universes a la Tarski since we have to have a "mini"-version of each type, add rules for a type El which ensures terms are types, and finally make sure El is congruent with respect to the formers we gave before.

Remark 11.2.3. A *metalogic* is the logic in which all these formal statments take place. We have not discussed this notion much since we would like our ideas to be mostly invariant of the metalogic. For technical satisfaction suppose we are working in ZFC with as many inaccessible cardinals as needed. This should be sufficiently strong enough to prove what we discussing. Further discussion on the foundational issues can be attained from [23, 22, 14] but is ultimately irrelevant in this context.

Appendices

A Simply typed lambda calculus $\lambda_{\to\times}$

This is the full-presentation of the simply typed lambda calculus $\lambda_{\to\times}$. It has function types, product types and a unit type.

A.1 Syntax

Written in BNF:

$$\text{Term} ::= x \mid \lambda x.a \mid (a, b) \mid ab \mid c$$

$$\text{Type} ::= \mathbf{1} \mid A \times B \mid A \to B$$

Or listed as operators:

Op	Sort	Vars	Type args	Term args	Scoping	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

A.2 Judgements

Judgement	Meaning
$\Gamma \vdash A$ 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 judgmentally equal types in context Γ.
$\Gamma \vdash S \equiv T : A$	S and T are judgmentally equal terms of type A in context Γ.

A.3 Structural rules

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ (\text{var}) \qquad \frac{\Gamma \vdash t \Rightarrow A \qquad \Gamma \vdash A \equiv B \ \mathsf{type}}{\Gamma \vdash t \Leftarrow B}\ (\text{switch})$$

$$\frac{\Gamma \vdash t \Rightarrow A}{\Gamma \vdash t \Leftarrow A}\ (\text{cswitch})$$

[[TODO: Include admissible rules?]]

A.4 Equality rules

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

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

$$\frac{\Gamma \vdash t \Leftarrow A}{\Gamma \vdash t \equiv t : A}\ (\equiv_{\mathsf{term}}\text{-refl}) \qquad \frac{\Gamma \vdash s \equiv t : A}{\Gamma \vdash t \equiv s : A}\ (\equiv_{\mathsf{term}}\text{-symm})$$

$$\frac{\Gamma \vdash t \Leftarrow A \qquad \Gamma \vdash s \equiv t : A \qquad \Gamma \vdash t \equiv r : A}{\Gamma \vdash s \equiv r : A}\ (\equiv_{\mathsf{term}}\text{-tran})$$

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash s \equiv t : A \qquad \Gamma \vdash A \equiv B \text{ type}}{\Gamma \vdash s \equiv t : B}\ (\equiv_{\mathsf{term}}\text{-}\equiv_{\mathsf{type}}\text{-cong})$$

A.5 Function type

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

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

$$\frac{\Gamma, x : A \vdash y \Leftarrow B \qquad \Gamma \vdash t \Leftarrow A}{\Gamma \vdash (\lambda x.y)t \equiv y[t/x] : B}\ (\to\text{-}\beta) \qquad \frac{\Gamma, y : A \vdash My \equiv M'y : B}{\Gamma \vdash M \equiv M' : A \to B}\ (\to\text{-}\eta)$$

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

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

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

A.6 Product type

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

$$\frac{\Gamma \vdash t \Leftarrow A \times B}{\Gamma \vdash \mathrm{fst}(t) \Rightarrow A}\ (\times\text{-elim}_1) \qquad \frac{\Gamma \vdash t \Leftarrow A \times B}{\Gamma \vdash \mathrm{snd}(t) \Rightarrow B}\ (\times\text{-elim}_2)$$

$$\frac{\Gamma \vdash x \Leftarrow A \qquad \Gamma \vdash y \Leftarrow B}{\Gamma \vdash \mathrm{fst}(x, y) \equiv x : A}\ (\times\text{-}\beta_1) \qquad \frac{\Gamma \vdash x \Leftarrow A \qquad \Gamma \vdash y \Leftarrow B}{\Gamma \vdash \mathrm{snd}(x, y) \equiv y : B}\ (\times\text{-}\beta_2)$$

$$\frac{\Gamma \vdash \mathrm{fst}(t) \equiv \mathrm{fst}(t') : A \qquad \Gamma \vdash \mathrm{snd}(t) \equiv \mathrm{snd}(t') : B}{\Gamma \vdash t \equiv t' : A \times B}\ (\times\text{-}\eta)$$

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

$$\frac{\Gamma \vdash a \equiv a' : A \qquad \Gamma \vdash b \equiv b' : B}{\Gamma \vdash (a, b) \equiv (a', b') : A \times B}\ (\times\text{-}\equiv_{\mathsf{term}}\text{-cong})$$

$$\frac{\Gamma \vdash t \equiv t' : A \times B}{\Gamma \vdash \mathrm{fst}(t) \equiv \mathrm{fst}(t') : A}\ (\times\text{-elim}_1\text{-cong})$$

$$\frac{\Gamma \vdash t \equiv t' : A \times B}{\Gamma \vdash \mathrm{snd}(t) \equiv \mathrm{snd}(t') : B}\ (\times\text{-elim}_2\text{-cong})$$

A.7 Unit type

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

B Examples

B.1 Identity function

$$\dfrac{(\to\text{-intro})\ \dfrac{\dfrac{(\text{var})\ \dfrac{(x:A)\in\Gamma,x:A}{\Gamma,x:A\vdash x\Rightarrow A} \qquad \dfrac{\Gamma,x:A\vdash A\ \mathsf{type}}{\Gamma,x:A\vdash A\equiv A\ \mathsf{type}}\ (\equiv\text{-refl})}{\Gamma,x:A\vdash x\Leftarrow A}\ (\text{switch})}{\Gamma\vdash\lambda x.x\Rightarrow A\to A} \quad \dfrac{\Gamma}{\Gamma\vdash T\equiv A\to A\ \mathsf{type}}}{\Gamma\vdash\lambda x.x\Leftarrow T}\ (\text{switch})$$

B.2 Function application $\lambda x.\lambda y.xy$

$$\dfrac{(\to\text{-intro})\ \dfrac{\dfrac{(\to\text{-intro})\ \dfrac{\dfrac{(\text{switch})\ \dfrac{(\to\text{-elim})\ \dfrac{(\text{var})\ \dfrac{x:A\in\Gamma,x:A,y:C}{\Gamma,x:A,y:C\vdash x\Rightarrow A} \qquad \dfrac{}{\Gamma,x:A,y:C\vdash C\to D\equiv A\ \mathsf{type}}\ (***)}{\Gamma,x:A,y:C\vdash x\Leftarrow C\to D} \qquad \dfrac{}{\Gamma,x:A,y:C\vdash y\Leftarrow C}\ (\dagger)}{\Gamma,x:A,y:C\vdash xy\Rightarrow D} \qquad \dfrac{}{\Gamma,x:A,y:C\vdash D\equiv D\ \mathsf{type}}\ (\equiv_{\mathsf{type}}\text{-refl})}{\Gamma,x:A,y:C\vdash xy\Leftarrow D}}{\Gamma,x:A\vdash\lambda y.xy\Rightarrow C\to D} \ (\text{switch}) \qquad \dfrac{}{\Gamma,x:A\vdash B\equiv C\to D\ \mathsf{type}}\ (**)}{\Gamma,x:A\vdash\lambda y.xy\Leftarrow B}}{\Gamma\vdash\lambda x.\lambda y.xy\Rightarrow A\to B} \ (\text{switch}) \qquad \dfrac{}{\Gamma\vdash T\equiv A\to B\ \mathsf{type}}\ (*)}{\Gamma\vdash\lambda x.\lambda y.xy\Leftarrow T}$$

B.3 Function composition $\lambda x.\lambda y.\lambda z.x(yz)$

$$\dfrac{\text{(cswitch)}\,\dfrac{\text{(var)}\,\dfrac{(z:A)\in\Gamma,x:B\to C,y:A\to B,z:A}{\Gamma,x:B\to C,y:A\to B,z:A\vdash z\Rightarrow A}}{\Gamma,x:B\to C,y:A\to B,z:A\vdash z\Leftarrow A} \qquad \dfrac{\dfrac{(y:A\to B)\in\Gamma,x:B\to C,y:A\to B,z:A}{\Gamma,x:B\to C,y:A\to B,z:A\vdash y\Rightarrow A\to B}\,\text{(var)}}{\Gamma,x:B\to C,y:A\to B,z:A\vdash y\Leftarrow A\to B}\,\text{(cswitch)}}{\dfrac{\Gamma,x:B\to C,y:A\to B,z:A\vdash yz\Rightarrow B}{\Gamma,x:B\to C,y:A\to B,z:A\vdash yz\Leftarrow B}\,\text{(cswitch)}}\,(\to\text{-elim})$$

$$\text{(cswitch)}\,\dfrac{(\to\text{-intro})\,\dfrac{\text{(cswitch)}\,\dfrac{(\to\text{-intro})\,\dfrac{\text{(cswitch)}\,\dfrac{(\to\text{-intro})\,\dfrac{\text{(cswitch)}\,\dfrac{(\to\text{-elim})\,\dfrac{\text{(cswitch)}\,\dfrac{\text{(var)}\,\dfrac{(x:B\to C)\in\Gamma,x:B\to C,y:A\to B,z:A}{\Gamma,x:B\to C,y:A\to B,z:A\vdash x\Rightarrow B\to C}}{\Gamma,x:B\to C,y:A\to B,z:A\vdash x\Leftarrow B\to C} \qquad \vdots}{\Gamma,x:B\to C,y:A\to B,z:A\vdash x(yz)\Rightarrow C}}{\Gamma,x:B\to C,y:A\to B,z:A\vdash x(yz)\Leftarrow C}}{\Gamma,x:B\to C,y:A\to B\vdash \lambda z.x(yz)\Rightarrow A\to C}}{\Gamma,x:B\to C,y:A\to B\vdash \lambda z.x(yz)\Leftarrow A\to C}}{\Gamma,x:B\to C\vdash \lambda y.\lambda z.x(yz)\Rightarrow (A\to B)\to(A\to C)}}{\Gamma,x:B\to C\vdash \lambda y.\lambda z.x(yz)\Leftarrow (A\to B)\to(A\to C)}}{\Gamma\vdash \lambda x.\lambda y.\lambda z.x(yz)\Rightarrow (B\to C)\to(A\to B)\to(A\to C)}}{\Gamma\vdash \lambda x.\lambda y.\lambda z.x(yz)\Leftarrow (B\to C)\to(A\to B)\to(A\to C)}$$

B.4 Owl combinator $\lambda x.\lambda y.\lambda z.y(xy)$

B.5 Currying $\lambda x.\lambda y.\lambda z.x(y, z)$

$$\text{(var)}\ \frac{(y : A) \in \Gamma, x : A \times B \to C, y : A, z : B}{\text{(cswitch)}\ \dfrac{\Gamma, x : A \times B \to C, y : A, z : B \vdash y \Rightarrow A}{\Gamma, x : A \times B \to C, y : A, z : B \vdash y \Leftarrow A}} \qquad \frac{(z : B) \in \Gamma, x : A \times B \to C, y : A, z : B}{\dfrac{\Gamma, x : A \times B \to C, y : A, z : B \vdash z \Rightarrow B}{\Gamma, x : A \times B \to C, y : A, z : B \vdash z \Leftarrow B}\ \text{(cswitch)}}\ \text{(var)}$$

$$\text{(×-intro)}\ \frac{\Gamma, x : A \times B \to C, y : A, z : B \vdash y \Leftarrow A \qquad \Gamma, x : A \times B \to C, y : A, z : B \vdash z \Leftarrow B}{\Gamma, x : A \times B \to C, y : A, z : B \vdash (y, z) \Rightarrow A \times B}$$

$$\text{(cswitch)}\ \frac{\Gamma, x : A \times B \to C, y : A, z : B \vdash (y, z) \Rightarrow A \times B}{\Gamma, x : A \times B \to C, y : A, z : B \vdash (y, z) \Leftarrow A \times B}$$

$$\text{(}{\to}\text{-elim)}\ \frac{\Gamma, x : A \times B \to C, y : A, z : B \vdash (y, z) \Leftarrow A \times B}{\Gamma, x : A \times B \to C, y : A, z : B \vdash x(y, z) \Rightarrow C}$$

$$\text{(cswitch)}\ \frac{\Gamma, x : A \times B \to C, y : A, z : B \vdash x(y, z) \Rightarrow C}{\Gamma, x : A \times B \to C, y : A, z : B \vdash x(y, z) \Leftarrow C}$$

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

$$\text{(cswitch)}\ \frac{\Gamma, x : A \times B \to C, y : A \vdash \lambda z.x(y, z) \Rightarrow B \to C}{\Gamma, x : A \times B \to C, y : A \vdash \lambda z.x(y, z) \Leftarrow B \to C}$$

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

$$\text{(cswitch)}\ \frac{\Gamma, x : A \times B \to C \vdash \lambda y.\lambda z.x(y, z) \Rightarrow A \to B \to C}{\Gamma, x : A \times B \to C \vdash \lambda y.\lambda z.x(y, z) \Leftarrow A \to B \to C}$$

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

$$\text{(cswitch)}\ \frac{\Gamma \vdash \lambda x.\lambda y.\lambda z.x(y, z) \Rightarrow (A \times B \to C) \to A \to B \to C}{\Gamma \vdash \lambda x.\lambda y.\lambda z.x(y, z) \Leftarrow (A \times B \to C) \to A \to B \to C}$$

B.6 Uncurry

$$
\dfrac{
\text{(var)}\dfrac{(x : A \to B \to C) \in \Gamma, x : A \to B \to C, y : A \times B}{\text{(cswitch)}\dfrac{\Gamma, x : A \to B \to C, y : A \times B \vdash x \Rightarrow A \to B \to C}{\Gamma, x : A \to B \to C, y : A \times B \vdash x \Leftarrow A \to B \to C}}
\qquad
\dfrac{\dfrac{\dfrac{\dfrac{(y : A \times B) \in \Gamma, x : A \to B \to C, y : A \times B}{\Gamma, x : A \to B \to C, y : A \times B \vdash y \Rightarrow A \times B}\text{(var)}}{\Gamma, x : A \to B \to C, y : A \times B \vdash y \Leftarrow A \times B}\text{(cswitch)}}{\Gamma, x : A \to B \to C, y : A \times B \vdash \mathrm{fst}(y) \Rightarrow A}\text{(}\times\text{-elim}_1}{\Gamma, x : A \to B \to C, y : A \times B \vdash \mathrm{fst}(y) \Leftarrow A}\text{(cswitch)}
}{
\text{(cswitch)}\dfrac{\Gamma, x : A \to B \to C, y : A \times B \vdash x(\mathrm{fst}(y)) \Rightarrow A}{\Gamma, x : A \to B \to C, y : A \times B \vdash x(\mathrm{fst}(y)) \Leftarrow A}
}\text{(}\to\text{-elim)}
$$

$$
\text{(cswitch)}\dfrac{\text{(}\to\text{-intro)}\dfrac{\text{(cswitch)}\dfrac{\text{(}\to\text{-intro)}\dfrac{\text{(cswitch)}\dfrac{\text{(}\to\text{-elim)}\dfrac{\begin{matrix}\vdots\\ \vdots\end{matrix}
\qquad
\dfrac{\dfrac{\dfrac{\dfrac{(y : A \times B) \in \Gamma, x : A \to B \to C, y : A \times B}{\Gamma, x : A \to B \to C, y : A \times B \vdash y \Rightarrow A \times B}\text{(var)}}{\Gamma, x : A \to B \to C, y : A \times B \vdash \mathrm{snd}(y) \Leftarrow B}\text{(cswitch)}}{\Gamma, x : A \to B \to C, y : A \times B \vdash \mathrm{snd}(y) \Rightarrow B}\text{(}\times\text{-elim}_2}{\Gamma, x : A \to B \to C, y : A \times B \vdash \mathrm{snd}(y) \Leftarrow B}\text{(cswitch)}
}{\Gamma, x : A \to B \to C, y : A \times B \vdash x(\mathrm{fst}(y))(\mathrm{snd}(y)) \Rightarrow C}}{\Gamma, x : A \to B \to C, y : A \times B \vdash x(\mathrm{fst}(y))(\mathrm{snd}(y)) \Leftarrow C}}{\Gamma, x : A \to B \to C \vdash \lambda y. \vdash x(\mathrm{fst}(y))(\mathrm{snd}(y)) \Rightarrow A \times B \to C}}{\Gamma, x : A \to B \to C \vdash \lambda y. \vdash x(\mathrm{fst}(y))(\mathrm{snd}(y)) \Leftarrow A \times B \to C}}{\Gamma \vdash \lambda x.\lambda y. \vdash x(\mathrm{fst}(y))(\mathrm{snd}(y)) \Rightarrow (A \to B \to C) \to A \times B \to C}}{\Gamma \vdash \lambda x.\lambda y. \vdash x(\mathrm{fst}(y))(\mathrm{snd}(y)) \Leftarrow (A \to B \to C) \to A \times B \to C}
$$

B.7 Swap $\lambda t.(\mathrm{snd}(t), \mathrm{fst}(t))$

12 Bibliography

[1] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

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

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

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

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

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

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

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

[9] Steven Fortune, Daniel Leivant, and Michael O'Donnell. The expressiveness of simple and second-order type structures. *J. ACM*, 30(1):151–185, January 1983.

[10] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[11] J.Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. 1972.

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

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

[14] Chris Kapulkin and Peter LeFanu Lumsdaine. The Simplicial Model of Univalent Foundations (after Voevodsky). *arXiv e-prints*, page arXiv:1211.2851, Nov 2012.

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

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

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

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

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

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

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

[22] Michael Shulman. All \$(,1)\$-toposes have strict univalent universes. *arXiv e-prints*, page arXiv:1904.07004, Apr 2019.

[23] Michael A. Shulman. Set theory for category theory. *arXiv e-prints*, page

arXiv:0810.1279, October 2008.

[24] R.M. Smullyan. *To Mock a Mocking Bird*. Knopf Doubleday Publishing Group, 2012.

[25] 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.

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