CURRY-HOWARD CORRESPONDENCE

AN INTUITIVE LANGUAGE FOR MATHEMATICS.

A thesis

submitted by

Xiao Tan

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Mathematics

TUFTS UNIVERSITY

May 2021

© Copyright 2021 by Xiao Tan

Adviser: David Smyth

Abstract

We all know how to write a proof, but what is exactly a proof? How do we know if a proof is correct or not? You may have learned a definition of "proof" in a first-order logic course, but it is cumbersome and not very close to the language of everyday mathematics. Consequently, this definition is not so useful e.g. for designing proof-checking software. In this thesis, we introduce an alternative proof system for mathematics called type theory, and explain the relationship between type theory and intuitionistic logic (the Curry-Howard Correspondence). This correspondence raises new problems like how to compare two proofs, which gives rise to the subject of Homotopy Type Theory.

Contents

List of Tables

List of Figures

Curry-Howard Correspondence

An intuitive language for mathematics.

Introduction

Traditionally, we seldom have a class called *introduction to proof* or so before we learn other mathematics. We always assume that proofs are intuitive enough for us to use. However, without an explicit and precise way to write a proof, our intuition is one of the biggest sources of confusion and wrongness*.

Though in a course of mathematical logic, we do give the strict definition of a proof such as first order language, this kind of definition is cumbersome and inconvenient, and is not the language we are using everyday. For example, it is very hard to prove that a group of order prime square is commutative in first order group theory. This strongly recommends a better way to formalize our mathematics.

Analytically, we make mathematics into a countable set of finite combinations of letters, like a free monoid, and then pick out helpful words and sentences. We filter a range of connectives such as conjunction (and) or disjunction (or), and organize them to form some grammatical sentences (often called well-formed formulas). Then we define the proof by a certain sequence of such sentences. Later we pick the concept 'set' formed by those sentences as the essences of mathematics. Finally, we define other mathematical objects like groups or topological spaces using sets.

But in this style, we have to treat our objects and language operations separately, making some inconvenience. For example, it is a bit hard to study category theory in set theory, because we have to define everything as a set, and thus it is hard to mention 'the category of all sets'. The situation is that the language for category theory does not rely on sets, but the operation of morphisms, while that meta-language operations is not transparent to category theory in set theory. So in some higher categorical theory, some other language

*Voevodsky was motivated by a mistake he had made to study formal proof in computer science, which led to the univalence axiom [5].

is prefered [15, 16].

And since we only restrict set operations, not the operations on our real targets, other problems also occur due to this lack of restrictions. For example, it is meaningful to speaking of the cardinality of a group, but is it still meaningful to talk about the cardinality of an element in a group, especially when this element is defined as a function in a cohomology group? Again, in calculus or differential geometry, it is quite common to apply a function to a 'variable'. If $u = f(x)$ and $y = g(u)$, then $g(u)$ should be $g \circ f$. This u can either be a function or a value, but the information about this is not carried by u, which is a common source of confusion for beginners.

Thus a new language is needed to help us handle these situations, which should formalize our intuitive proof together with how we construct an object as in human logic. A better language often helps you solve a problem more intuitively. I first noticed this in computer programming. A certain language is more suitable for a problem than another. Now it is the time to do the same thing to mathematics.

In this thesis, I am going to introduce you such a language called λ-calculus, which was proposed by Church [6] and is now used instead of set theory by computer scientists. The language was originally proposed as a model of computing, but a shocking fact is that it coincides with our logic structure. Roughly speaking, an implication $A \to B$ can be understood as a 'function' transporting the proof of A to the proof of B, and complicated implication can be viewed as function operations like composition or application.[†] We are then enabled to write proofs more algorithmically and many other problems like how to compare two proofs or how to cut the detours during a proof receive a clear answer. It might be a bit weird to think mathematics in this way because we all learned set theory in high school, but with some familiarity with it, you

[†]Originally, Brouwer, Heyting and Kolmogorov gave the basic idea about it. Then Kleene gave the realizability interpretation. Finally Curry [10] formalized it and Howard [14] extended it to full intuitionistic logic.

will find it is more natural to write a proof.

Since this is a relatively new way to illustrate mathematics, a lot of difficulties occurred during my own study. I hope I could fill in all the gaps you need to know in this thesis. You can then read other books like [21] without too much suffering.

In Chapter 1, I will first introduce the intuitionistic logic and discuss its relation with classical logic. Then I will introduce λ-calculus and prove the correspondence between it and intuitionistic logic in Chapter 2. Finally, I will show an interesting theory rising recently in Chapter 3, which uses this correspondence to define homotopical models.

Chapter 1

Propositional Logic

How do we make a judgement in mathematics? If we write down a sentence, how to know it is correct or not? And since we are going to study mathematics inside mathematics, how do we ensure a complete version of this mathematics. These questions tend to be paradoxical because we have to use a prior meta-logic to judge some facts about mathematics objects. At least you have to admit 'apple' is 'apple' as two combinations of letters. The idea is to build an identical but distinct copy of our language, a countable set of finite symbols, that represents the mathematics we are using, a bit like making a machine tool with an existing machine tool and making other machines with the new copy.

In this chapter, I am going to make a naive copy of our language, the *propositional logic*, where we only focus on logical connectives. This copy will be defined like a free monoid, and we will interpret it with some kind of morphism. Note that I have to use the words 'monoid' and 'morphism' to represent the new language and study it. This means the original 'machine tool', and the 'monoid' and its operations means the newly-made 'machine tool'. In general, the language we are using like the original machine tool is called the *meta-language* and the new copy we want to make is called the *target language*. What we are going to do is to model the meta-language into some grammatic structure in the target language.

1.1 Classical Logic

Let's think about a simple example. We know 'people need water' and 'birds can fly'. Then we know 'people need water and birds can fly'. Here the concept

'and' is used twice, one is in our meta logic showing we know two facts, the other is the correctness of the whole sentence composed by the connective 'and'. What we want to study is the latter 'and'.

In the most classical model of logic, our semantics is simply divided into 'true' and 'false', like 'people need water' is true and 'people can fly with wings' is false. And our target language is defined by breaking a complicated semantics into small pieces concatenated by those connectives such as 'and', 'or', e.g., 'people need water or people can fly' is true. We model the language with letters A, B, C for sentences like 'people need water' and then defined the connectives like 'or' as functions with domain $\{\mathsf{true}, \mathsf{false}\}^n$.

Definition 1.1.1 *(Formulas) Suppose $V = \{A, B, C, \cdots\}$ is a countable set. We recursively define the concept* **formula** *to be those we want to interpret as true or false.*

1. *The elements in the set V are formulas;*
2. *If φ is a formula, then so is the negation $(\neg\varphi)$;*
3. *If φ and ψ are formulas, then so is the conjunction (and) $(\varphi \wedge \psi)$;*
4. *If φ and ψ are formulas, then so is the disjunction (or) $(\varphi \vee \psi)$;*
5. *If φ and ψ are formulas, then so is the implication $(\varphi \to \psi)$.*

We denote the set of all formulas as $\mathscr{F}$.

Example 1.1.2 Examples of formulas:

1. A, B, C
2. $(A \to B)$
3. $(((A \vee B) \wedge B) \to (A \to B))$ ⋄

Remark 1.1.3 The recursive definition is similar to the definition of free group. We start from a 'generating set', like the set V. Then other rules show how to do the 'multiplication' as in a free group*. The elements in V are called *(propositional) variables* representing sentences that show a fact but cannot be further decomposed by a connective. The connectives $\neg, \wedge, \vee, \to$ are chosen differently from the words 'and', 'or', 'imply' so that the 'and' will be the meta one we use. ♢

Remark 1.1.4 Here I require the *variable set* V to be a countable set. Actually in real work, it is convenient to choose V as the free monoid generated by a finite set, like all English words generated by the set of 26 letters. I also tend to use upper case letters for a variable and lower case Greek letters for arbitrary formulas. ♢

Sometimes it is inconvenient to always write a lot of parentheses. We thus make some conventions. The outermost parenthesis can always be omit, and the precedence of the connectives are $\neg > \wedge > \vee > \to$. For example, $\neg A \vee B \wedge \neg C \to D$ is actually $(((\neg A) \vee (B \wedge (\neg C))) \to D)$.

Now let's think about the semantics for those formulas. When we mention $A \wedge B$ and if we know A is correct and B is not, then the formula $A \wedge B$ is incorrect. In this example, we first have a context that A is correct and B is not before we can evaluate $A \wedge B$. Roughly speaking, a *context* is a set of variables representing premises to our discussion. Thus it is natural to have the semantics of a formula related to the semantics defined on the variables.

Definition 1.1.5 *(Assignment) A function $v : V \to \{\mathsf{true}, \mathsf{false}\}$ is called a* **truth-value assignment**. *Then we extend it to a function $\bar{v} : \mathscr{F} \to \{\mathsf{true}, \mathsf{false}\}$.*

1. *$\bar{v}(A) = v(A)$, for all $A \in V$.*

*In fact this definition can be generalized by the Löwenheim–Skolem theorem and can be interpreted as the initiality of algebra [24].

2. $\bar{v}(\neg\varphi) = \begin{cases} \mathsf{true}, & \bar{v}(\varphi) = \mathsf{false} \\ \mathsf{false}, & \bar{v}(\varphi) = \mathsf{true} \end{cases}$.

3. $\bar{v}(\varphi \wedge \psi) = \mathsf{true}$ *if and only if* $\bar{v}(\varphi) = \mathsf{true}$ *and* $\bar{v}(\psi) = \mathsf{true}$.

4. $\bar{v}(\varphi \vee \psi) = \mathsf{false}$ *if and only if* $\bar{v}(\varphi) = \mathsf{false}$ *and* $\bar{v}(\psi) = \mathsf{false}$.

5. $\bar{v}(\varphi \to \psi) = \mathsf{false}$ *if and only if* $\bar{v}(\varphi) = \mathsf{true}$ *and* $\bar{v}(\psi) = \mathsf{false}$.

Without ambiguity, we may omit the bar above v.

Remark 1.1.6 Most definitions of this extension directly follow our intuition. The only exception is implication, which may not agree with our intuitive explanation. The semantics of it can be read as 'if and only if the premise is correct and the conclusion is wrong, then the implication is incorrect'. That means if you know the implication is correct and the premise is correct, then the conclusion must be correct. This is the only possible semantics of implication under the true or false explanation.

Another understanding is that you can derive anything from absurdity. This rule is called *ex falso (sequitur) quodlibet* (EFQ). In classical logic, we only meet this in the semantic part of our language. ♢

You may have noticed that the semantics of these connectives overlap. In fact you can think of each formula as a function $\{\mathsf{true}, \mathsf{false}\}^n \to \{\mathsf{true}, \mathsf{false}\}$ and the assignment v as an input context, and so are the connectives. Thus the formulas are only compositions of those connectives.

Definition 1.1.7 *(Functional Completeness) A set of connectives is* **functionally complete** *if and only if every function* $\{\mathsf{true}, \mathsf{false}\}^n \to \{\mathsf{true}, \mathsf{false}\}$ *can be defined by the composition of the connectives in this set.*

Lemma 1.1.8 *The set* $\{\neg, \wedge, \vee\}$ *is functionally complete.*

Proof: Given $\varphi : \{\mathsf{true}, \mathsf{false}\}^n \to \{\mathsf{true}, \mathsf{false}\}$, find all inputs $a_i = (t_i^1, \cdots, t_i^n)$ such that $\varphi(a_i) = \mathsf{true}$. Let m be the number of such a_i. For each a_i, define

$$\psi_i^j = \begin{cases} A_j, & t_i^j = \mathsf{true} \\ \neg A_j, & t_i^j = \mathsf{false} \end{cases}$$

Then let $\gamma_i = \psi_i^1 \wedge \psi_i^2 \wedge \cdots \wedge \psi_i^n$ and $\varphi' = \gamma_1 \vee \gamma_2 \vee \cdots \vee \gamma_m$. It is easy to observe that $\bar{v}(\varphi) = \bar{v}(\varphi')$ for any assignment v. The semantics of this φ' is that $\bar{v}(\varphi)$ is true if and only if $\bar{v}(A_1) = t_1^1$ and $\bar{v}(A_1) = t_1^2$ and ... and $\bar{v}(A_n) = t_1^n$, or $\bar{v}(A_1) = t_2^1$ and $\bar{v}(A_1) = t_2^2$ and ... and $\bar{v}(A_n) = t_2^n$. This φ' is called the Disjunctive Normal Form of φ. □

My current choice of connectives is certainly complete, but the overlapped choice is necessary for the intuitionistic logic because the semantics is no longer defined by true or false and certain connectives cannot be the composition of others. Some books choose to define all connective by $\neg$ and $\vee$, but I focus more on the semantics of each connective, so I give these five connectives, but mainly use the $\to$ and $\neg$.

Let's reconsider the meaning of context. In general, we may not only say variables A, B, C are correct and variables X, Y, Z are wrong, but we may also say that a formula $A \to B$ is correct, and we have to decide whether $B \vee C$ is correct under such a circumstance. Besides, we do not have to include every variable in the premises. We only want to focus on those used in our conclusion. Also, to say a variable A is assigned to false, we can instead say $\neg A$ is true. Thus we shall generalize the concept of context to a set of formulas.

Definition 1.1.9 *(Satisfiability) Let Γ be a set of formulas. An assignment v* **satisfies** *Γ if and only if for all $\varphi \in \Gamma$, $\bar{v}(\varphi) = \mathsf{true}$, written as $v \vDash \Gamma$.*

Definition 1.1.10 *(Logical Consequence) If Γ is a set of formulas and φ is a formula, then we say φ is the* **logical consequence** *or* **entailment** *of Γ if*

and only if for any assignment v such that $v \vDash \Gamma$, $v(\varphi) = \mathsf{true}$.

Example 1.1.11 $\{\varphi \to \psi, \varphi\} \vDash \psi$

For any assignment v satisfying $\{\varphi \to \psi, \varphi\}$, $\bar{v}(\varphi) = \mathsf{true}$. Since $\bar{v}(\varphi \to \psi) = \mathsf{true}$, we must have $\bar{v}(\psi) = \mathsf{true}$. ◇

Remark 1.1.12 Given an assignment v, you can collect all formulas it considers as true, i.e., a set $\Gamma = \{\varphi \in \mathscr{F} : \bar{v}(\varphi) = \mathsf{true}\}$. Since this set contains all variables, v is the unique assignment satisfying Γ. Thus for any formula ψ outside this set, $\Gamma \cup \{\psi\}$ cannot be satisfied by any assignment. In other words, such a context is *maximally satisfiable*. Thus an assignment can be considered as a maximal context.

Furthermore, this definition of Γ tells us that $\Gamma \vDash \varphi$ if and only if $\bar{v}(\varphi) = \mathsf{true}$ if and only if $\varphi \in \Gamma$. So it is also reasonable to think of a set of formulas as a generalization of assignment, i.e., a *partial assignment*. Then the entailment relation tells us how to judge whether a formula is correct or not with given truth values on formulas rather than variables.

This generalization also includes the situation when no assignment is valid for our context. For example, the set $\{A, \neg A\}$ cannot be satisfied by any assignment. Please note the 'for any' in the definition of logical concequence. If a context is not satisfiable, then it should entail any formula. For example $\{A, \neg A\} \vdash \varphi$. This is the EFQ rule (1.1.6) in our meta-logic. ◇

Here are some notation conventions for this relation $\vDash$. We write $\Gamma, \alpha \vDash \varphi$ for $\Gamma \cup \{\alpha\} \vDash \varphi$, and if Δ is another set of formulas, we can write $\Gamma, \Delta \vDash \varphi$ instead of $\Gamma \cup \Delta \vDash \varphi$. The empty set can be omitted, e.g. $\vDash \alpha \to \alpha$, $\alpha \vDash \alpha \vee \beta$.

Definition 1.1.13 *(Tautology) If $\vDash \varphi$, then φ is called a* **tautology**.

We have already known what the correctness of a proposition (formula) means under some context Γ. But a problem is how to check whether this is

correct or not. Because we have to check all the possible assignments, which is an infinite process. Since we have a limited life, a human way to verify the process is necessary.

The situation is that the semantics behind an entailment might be very complicated. What we are going to do is to break the semantics into steps, just like we break a function into connectives. But this time, we are going to compose formulas from formulas, and we have to ensure each step is correct. For example, if we know $\varphi \to \psi$ and φ are correct, then we must have ψ is correct. This semantics shows some way to compose ψ from $\{\varphi \to \psi, \varphi\}$. Instead of always verifying the semantics, we define all these as a syntactic rule. This process is often called a *proof*.

Definition 1.1.14 *(Proof) A* **proof** *from Γ to φ is a sequence $\psi_1, \psi_2, \cdots, \psi_n$ such that $\psi_n = \varphi$ and each ψ_i must satisfy one of the following rules.*

1. *ψ_i is an element in Γ.*
2. *ψ_i is in the form of one of the following:*
   - *(a)* $\alpha \to (\beta \to \alpha)$
   - *(b)* $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$
   - *(c)* $(\neg\beta \to \neg\alpha) \to ((\neg\beta \to \alpha) \to \beta)$
3. *There are ψ_j and ψ_k such that $j, k < i$ and $\psi_j = \psi_k \to \psi_i$. This rule is called Modus Ponens (MP).*

If there is a proof from Γ to φ, we write $\Gamma \vdash \varphi$.

The definition of proof is easily understood. Those given in the context to be true must be true. Some special kind of tautologies are correct, and modus ponens is always correct. They are the only methods we need to reveal the semantics behind an entailment.

Those formulas in the context Γ are called axioms. The tautologies are not all included in rule 2 because the semantics of them can also be composed in

this system. Though we can verify them within a finite process, their semantics can still be decomposed with the 3 tautologies in the definition. Actually in this system, we have no chance to decompose the three tautologies in rule 2 into smaller pieces. These special tautologies are called *logical axioms*. Finally modus ponens shows a way to make inference, which actually *cuts* a formula. In contrary, connectives can only lengthen a formula.

Remark 1.1.15 For formulas like $\alpha \vee \beta$ and $\alpha \wedge \beta$, they are thought as aliases to $\neg\alpha \to \beta$ and $\neg(\neg\alpha \vee \neg\beta)$. This is equivalent to introduce

1. $\alpha \vee \beta \to (\neg\alpha \to \beta)$,
2. $(\neg\alpha \to \beta) \to \alpha \vee \beta$,
3. $\alpha \wedge \beta \to \neg(\neg\alpha \vee \neg\beta)$,
4. $\neg(\neg\alpha \vee \neg\beta) \to \alpha \wedge \beta$

as logical axioms to our system. In fact if we can have $\Gamma \vdash \alpha \to \beta$ and $\Gamma \vdash \beta \to \alpha$, then we can use α and β alternatively. I did not do that because it would increase the techhnical complexity to prove properties about this definition of 'proof'. ⋄

This kind of proof is called **Hilbert system**. There are other ways to define proof (1.2) and there are other choices of logical axioms. My choice is Mendelssohn's variation [17]. In his system, the rule 2(a) and 2(b) are *intuitive*, while 2(c) is 'proof by contradiction', which is often not admitted by *intuitionists*. The idea of intuitionism will be immediately introduced in the next section(1.2). But you may want to ask why 2(c) is needed. Can't we just prove it from 2(a) and 2(b)? The answer is certainly negative, but the proof is a bit tricky and is not immediately necessary, so I put it in A.

A proof sequence is actually defined recursively. The recursion rule is the third one, so you can also perform induction on the proof. The sequence can

also be truncated or extended. Any ψ_i in the proof sequence can be the final ψ. You can also bring in any other valid formulas defined by the three rules. For example, to make a modus ponens, we only need to form the proofs $\Gamma \vdash \varphi$ and $\Gamma \vdash \varphi \to \psi$ and then conclude $\Gamma \vdash \psi$.

Example 1.1.16 Let's prove a formula with this definition of proof.

$\vdash \varphi \to \varphi$ ◇

Proof: 1. $\varphi \to (\psi \to \varphi)$ (Rule 2.a)

2. $\varphi \to ((\psi \to \varphi) \to \varphi)$ (Rule 2.a)

3. $(\varphi \to ((\psi \to \varphi) \to \varphi)) \to ((\varphi \to (\psi \to \varphi)) \to (\varphi \to \varphi))$ (Rule 2.b)

4. $(\varphi \to (\psi \to \varphi)) \to (\varphi \to \varphi)$ (Rule 3: 2,3+MP)

5. $\varphi \to \varphi$ (Rule 3: 1,4+MP) □

Here is an interesting lemma, which shows the implication '→' coincides with our intuition in this definition of proof $\vdash$. To prove an implication, you can bring the premise into the context.

Lemma 1.1.17 *(Deduction Lemma)* $\Gamma \vdash \varphi \to \psi$ *if and only if* $\Gamma, \varphi \vdash \psi$.

Proof: If $\Gamma \vdash \varphi \to \psi$, then $\Gamma, \varphi \vdash \varphi \to \psi$. Obviously, $\Gamma, \varphi \vdash \varphi$. Using MP, we can obtain ψ.

Conversely, suppose $\Gamma, \varphi \vdash \psi$. By definition, there is proof sequence $\psi_1, \psi_2, \cdots, \psi_n$ and $\psi_n = \psi$. Perform induction on n.

If $n = 1$, then ψ is in Γ, or $\psi = \varphi$, or it is a logical axiom. Since $\psi \to (\varphi \to \psi)$ and for any formula φ, we have proved $\vdash \varphi \to \varphi$, it is easy to show $\Gamma \vdash \varphi \to \psi$.

If for all $k < n$, $\Gamma, \varphi \vdash \psi_k$ implies $\Gamma \vdash \varphi \to \psi_k$. Then ψ_n is obtained as in the three cases when $n = 1$, or it is the result of MP. Thus it suffices to assume there are $i, j < n$ such that $\psi_i = \psi_j \to \psi_n$. By induction hypothesis, we have $\Gamma \vdash \varphi \to (\psi_j \to \psi_n)$ and $\Gamma \vdash \varphi \to \psi_j$. □

Remark 1.1.18 This lemma shows the definition of $\to$ in our target language reflects the implication $\vdash$ in our meta-language. In fact, you can also prove $\Gamma \vdash \varphi \wedge \psi$ if and only if $\Gamma \vdash \varphi$ and $\Gamma \vdash \psi$. We will bring more meta-language affairs into our formal target language. $\diamond$

As you may expect, we have to show this definition is correct and comprehensive, i.e., $\Gamma \vdash \varphi$ if and only if $\Gamma \vDash \varphi$. The idea of this proof is due to Lindenbaum. Given any context Γ, we can expand it so that it is induced by an assignment. For example, if we have $\Gamma = \{A, A \to B\}$, then we can expand it to $\Gamma \cup \{\neg C\}$. Here I include $\neg C$, but I had better not include $\neg B$, because that will cause the context is not satisfiable by an assignment. The correct extension is that if we can prove φ from Γ, then we have to enclose φ, not $\neg\varphi$ in Γ. But this C is special here, because it is not in the context. Thus we can choose either C or $\neg C$. To be more precise about this problem, I am going to build more tools about proof.

Definition 1.1.19 *(Consistency) A formula set Γ is* **inconsistent** *if there exists a formula φ such that $\Gamma \vdash \varphi$ and $\Gamma \vdash \neg\varphi$. If a set is not inconsistent, then it is consistent.*

Lemma 1.1.20 *Γ is inconsistent if and only if for any φ, $\Gamma \vdash \varphi$.*

Proof: ($\Rightarrow$) Suppose $\Gamma \vdash \psi$ and $\Gamma \vdash \neg\psi$. By rule 2(a), we can put $\psi \to (\neg\varphi \to \psi)$ in the proof sequence. By MP, we have $\neg\varphi \to \psi$ in the sequence. Similarly, we have $\neg\varphi \to \neg\psi$. Then we put into the sequence the logical axiom $(\neg\varphi \to \psi) \to ((\neg\varphi \to \neg\psi) \to \varphi)$. Using MP twice, we have $\Gamma \vdash \varphi$. ($\Leftarrow$) Choose any formula φ. We certainly have $\Gamma \vdash \varphi$ and $\Gamma \vdash \neg\varphi$. $\square$

Lemma 1.1.21 *$\Gamma \vdash \varphi$ if and only if $\Gamma \cup \{\neg\varphi\}$ is inconsistent.*

Proof: The 'only if' direction is trivial.

Suppose there is a ψ such that $\Gamma, \neg\varphi \vdash \psi$ and $\Gamma, \neg\varphi \vdash \neg\psi$. By 1.1.20, we know $\Gamma, \neg\varphi \vdash \varphi$. According to the deduction lemma (1.1.17), $\Gamma \vdash \neg\varphi \to \varphi$. We also put $\varphi \to \varphi$ and $(\varphi \to \neg\varphi) \to ((\varphi \to \varphi) \to \varphi)$ □

Finally, we can prove the following theorem.

Theorem 1.1.22 *For a formula φ and a set Γ of formulas,*

1. *(Soundness) $\Gamma \vdash \varphi$ implies $\Gamma \vDash \varphi$;*
2. *(Completeness) $\Gamma \vDash \varphi$ implies $\Gamma \vdash \varphi$.*

Proof: The soundness is easily verified by induction on the proof.

Now suppose $\Gamma \vDash \varphi$. If Γ is inconsistent, then certainly $\Gamma \vdash \varphi$. Otherwise, if $\Gamma \nvdash \varphi$, by 1.1.21, $\Gamma \cup \{\neg\varphi\}$ must be consistent. Let $\psi_1, \psi_2, \cdots$ be an enumeration of $\mathscr{F}$. Expand the set $\Gamma \cup \{\neg\varphi\}$ as follows.

$$\text{Let } \mathcal{L}_0 = \Gamma \cup \{\neg\varphi\},$$

$$\mathcal{L}_{n+1} = \begin{cases} \mathcal{L}_n \cup \{\psi_n\}, & \mathcal{L}_n \cup \{\psi_n\} \text{ is consistent} \\ \mathcal{L}_n \cup \{\neg\psi_n\}, & \mathcal{L}_n \cup \{\psi_n\} \text{ is not consistent.} \end{cases}$$

Let $\mathcal{L} = \bigcup_{n\in\mathbb{N}} \mathcal{L}_n$. It is easy to verify that $\mathcal{L}$ is consistent and we can define the assignment

$$v(A) = \mathsf{true} \text{ if and only if } A \in \mathcal{L}.$$

By induction on ψ, we have $\bar{v}(\psi) = \mathsf{true}$ if and only if $\psi \in \mathcal{L}$.

Now $v \vDash \Gamma \cup \{\neg\varphi\} \subseteq \mathcal{L}$. That means $v \vDash \Gamma$ and $\bar{v}(\neg\varphi) = \mathsf{true}$. Since $\Gamma \vDash \varphi$, then $\bar{v}(\varphi) = \mathsf{true}$, but that gives $\bar{v}(\neg\varphi) = \mathsf{false}$, which is a contradiction. Thus $\Gamma \vdash \varphi$. □

1.2 Intuitionistic Logic

The classical logic works well. We have been able to tell whether a proof is correct or not, but one deficiency is that it is not constructive. One famous

example [23] is the Gelfond–Schneider constant, i.e., how to find two irrational numbers x, y such that x^y is rational. If $\sqrt{2}^{\sqrt{2}}$ is irrational, then we've found them; otherwise, set $x = \sqrt{2}^{\sqrt{2}}$, $y = \sqrt{2}$. But we don't know which combination is correct from this proof[†].

In the above proof, we used a tautology $(\varphi \to \psi) \wedge (\neg\varphi \to \psi) \to \psi$, or equivalently the proof $\varphi \to \psi, \neg\varphi \to \psi \vdash \psi$. But it does not tell you what fact about $\varphi \to \psi$ and $\neg\varphi \to \psi$ forces ψ. To avoid this situation, the best idea is not to define the semantics as true or false, but by how they are *constructed* in terms of their relations with other formulas. For example, to prove $\varphi \vee \psi$, we must prove either φ or ψ, instead of assuming $\neg\varphi$ and proving ψ. This is the idea of intuitionistic semantics of logic (called BHK-interpretation[‡]), which interprets how to construct the evidence of a formula as follows.

- An evidence of $\varphi \wedge \psi$ consists of both the evidence of φ and the evidence of ψ.
- An evidence of $\varphi \vee \psi$ is either the evidence of φ or the evidence of ψ.
- An evidence of $\varphi \to \psi$ is a function tranporting the evidence of φ to the evidence of ψ.
- The interpretation of negation is a bit different. We still admit if we have absurdity, then we can prove everything in intuitionistic logic. Now we bring the absurdity into the language by a nullary connective $\bot$. For example, $\bot$ is a formula as $A \to B$, and by the recursive definition, those like $\bot \to (A \vee C)$ are also formulas. Thus the meaning of the negation $\neg\varphi$ is that if we have φ, then we have the absurdity, i.e., $\neg\varphi$ is the abbreviation of $\varphi \to \bot$.

This interpretation is formalized by Heyting algebra.

[†]In fact, there is a Gelfond–Schneider Theorem that for any algebraic a, b with $a \neq 0, 1$, b irrational, a^b is transcendental.

[‡]This interpretation is named after Brouwer, Heyting and Kolmogorov.

Definition 1.2.1 *(Heyting Algebra) A* **Heyting algebra** *is a poset $(\mathcal{H}, \leq)$ as a category which admits finite products, coproducts and exponentials.*

Remark 1.2.2 Finite (co)products include the nullary ones. Thus we have the initial object $\mathbf{0}$ and the terminal object $\mathbf{1}$ in this category.

For objects A, B in such a category, the exponential A^B is an object such that there are natural bijectives between the hom-sets $\hom_{\mathcal{H}}[C, A^B]$ and $\hom_{\mathcal{H}}[C \times B, A]$. This coincides with the exponential rule in set theory. We also write $B \Rightarrow A$ for A^B.

To easily mention negation, we define $\sim A$ to be $A \Rightarrow \mathbf{0}$

In such a category, we have $\times$ and $+$ are commutative and associative. Also $(A + B) \times C$ is isomorphic to $(A \times C) + (B \times C)$ [4]. ◊

The semantics is defined similarly as in classical logic.

Definition 1.2.3 *Given a Heyting algebra $\mathcal{H}$, an assignment v is a function $V \to \mathcal{H}$. And we extend it to a function $\bar{v} : \mathscr{F} \to \mathcal{H}$ as follows.*

1. $\bar{v}(A) = v(A)$ *for any* $A \in V$.
2. $\bar{v}(\bot) = \mathbf{0}$.
3. $\bar{v}(\varphi \wedge \psi) = \bar{v}(\varphi) \times \bar{v}(\psi)$.
4. $\bar{v}(\varphi \vee \psi) = \bar{v}(\varphi) + \bar{v}(\psi)$.
5. $\bar{v}(\varphi \to \psi) = \bar{v}(\varphi) \Rightarrow \bar{v}(\psi)$.

Without ambiguity, we may omit the bar above v.

We also have to include the Heyting algebra when defining logical consequence.

Definition 1.2.4 *(Intuitionistic Logical Consequence) Suppose $\mathcal{H}$ is a Heyting algebra and $v : V \to \mathcal{H}$ is an assignment. For a formula set Γ and a formula φ, we define*

- $\mathcal{H}, v \vDash \varphi$ *if and only if* $v(\varphi) = 1$*;*
- $\mathcal{H} \vDash \varphi$ *if and only if for all assignment* v*,* $\mathcal{H}, v \vDash \varphi$*;*
- $\mathcal{H}, v \vDash \Gamma$ *if and only if for all* $\varphi \in \Gamma$*,* $\mathcal{H}, v \vDash \varphi$*;*
- $\mathcal{H} \vDash \Gamma$ *if and only if for all assignment* v*,* $\mathcal{H}, v \vDash \varphi$*;*
- $\Gamma \vDash \varphi$ *if and only if for all Heyting algebra* $\mathcal{H}$ *and assignment* v*,* $\mathcal{H}, v \vDash \Gamma$ *implies* $\mathcal{H}, v \vDash \varphi$*.*

Definition 1.2.5 *(Intuitionistic tautology) A formula* φ *is an intuitionistic tautology if and only if* $\vDash \varphi$*.*

This is the smallest way to define an algebraic semantics of intuitionistic logic. The Heyting algebra is the generalization of our Boolean semantics. If we require $A + (\sim A)$ to be isomorphic to $\mathbf{1}$, then the Heyting algebra is called Boolean algebra. In fact, that reflects the classical tautology $\varphi \vee \neg\varphi$, which is equivalent to rule 2(c) in the system with only 2(a) and 2(b) (1.2.13) . We can also interpret classical logic within Boolean algebra, and we will get the same definition of classical logical consequence.

Example 1.2.6 Here is an example that Heyting algebras are different from Boolean algebras. Let $\mathcal{O}(X)$ be the set of all open sets of a topological space X, which forms a poset under set inclusion $\subseteq$, where the product and coproduct are the set-theoretic intersection $\cap$ and union $\cup$. Of course, $\mathbf{0} = \varnothing$ and $\mathbf{1} = X$. The exponential is $A \Rightarrow B = \mathrm{Int}((X \setminus A) \cup B)$. Thus $\sim A = \mathrm{Int}(X \setminus A)$, which is in general not the complement of A.

If $X = \{0, 1\}$ and $\mathcal{O}(X) = \{\varnothing, \{0\}, \{0, 1\}\}$, and $v(A) = \{0\}$, $v(B) = \varnothing$, then $\bar{v}((\neg A \to \neg B) \to ((\neg A \to B) \to A)) = \{0\}$. Thus rule 2(c) is not a intuitionistic tautology. ♢

The morphisms in a Heyting algebra coorrespond to the provability of formluas. As in classical logic, the provability should be the rules to cut or

lengthen formulas to yield new formulas while keeping the correctness. Since I interpret proof as construction, it is natural to write the recursive construction as a tree. Recall 1.1.17 tells us $\Gamma, \varphi \vdash \psi$ if and only if $\Gamma \vdash \varphi \to \psi$. This gives us the idea to define a syntactic rule for the provability relation $\vdash$. For example, if we have constructed $\Gamma \vdash \varphi$ and $\Gamma \vdash \psi$, then we can construct $\Gamma \vdash \varphi \wedge \psi$. Instead of think of this as a proof sequence, I write this as

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} .$$

The horizontal line means if you can prove all the relation above it, then you can have the relation beneath it. Also, we have to recover the construction of φ from $\varphi \wedge \psi$. This rule is written as

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} .$$

This style of proof is called *natural deduction*, which is composed by such rules of construction of the relation $\vdash$. The rules generally tell you how to *introduce* a connective and how to *eliminate* it.

Definition 1.2.7 *(Intuitionistic Natural Deduction) In intuitionistic logic, we have the following rules of provability.*

- *Axiom:* $\Gamma, \varphi \vdash \varphi$

- $\wedge$*-Introduction:*
$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}$$

  $\wedge$*-Elimination:*
$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}$$
$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}$$

- $\vee$-*Introduction:*

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}$$

$$\frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}$$

$\vee$-*Elimination:*

$$\frac{\Gamma, \varphi \vdash \sigma \quad \Gamma, \psi \vdash \sigma \quad \Gamma \vdash \varphi \vee \psi}{\Gamma \vdash \sigma}$$

- $\rightarrow$-*Introduction:*

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \rightarrow \psi}$$

$\rightarrow$-*Elimination:*

$$\frac{\Gamma \vdash \varphi \rightarrow \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi}$$

- $\bot$-*Elimination:*

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}$$

Remark 1.2.8 Our definition of Γ is a set, but here I have to loosen the definition to a multiset, i.e., we admit repeated elements in a set. For example, $\varphi \rightarrow (\psi \rightarrow \varphi)$ is proved by

$$\frac{\dfrac{\varphi, \psi \vdash \varphi}{\varphi \vdash \psi \rightarrow \varphi}}{\vdash \varphi \rightarrow (\psi \rightarrow \varphi)}$$

But when $\psi = \varphi$, we have to put two φ in the premises.

$$\frac{\dfrac{\varphi, \varphi \vdash \varphi}{\varphi \vdash \varphi \rightarrow \varphi}}{\vdash \varphi \rightarrow (\varphi \rightarrow \varphi)}$$

So we use multiset for this situation.

The elimination rule of $\vee$ means that if we only know $\varphi \vee \psi$, but we do not know which one is correct, to eliminate the disjunction $\vee$, we have to ensure that ρ happens when either φ or ψ is held. This is exactly the coproduct rule in category theory, i.e., proof by cases.

The elimination rule of $\to$ is MP. And $\bot$ has no introduction rule, whose elimination rule is EFQ.

It is not hard to observe that $\Gamma \vdash \varphi \wedge \psi \to \tau$ if and only if $\Gamma, \varphi, \psi \vdash \tau$ if and only if $\Gamma \vdash \varphi \to (\psi \to \gamma)$. This is exactly the exponential rule. Thus it is reasonable to let '$\to$' be right associative, i.e., $\varphi \to \psi \to \tau = \varphi \to (\psi \to \tau)$ in order to save parentheses. E.g. we can write the rule $\alpha \to (\beta \to \alpha)$ as $\alpha \to \beta \to \alpha$, which can be understood as $\alpha, \beta \vdash \alpha$ by bringing all formulas before the last $\to$ to the context. $\diamond$

Example 1.2.9 $\varphi \vee \psi \vdash \neg\varphi \to \psi$. $\diamond$

Proof: Let $\Gamma = \{\varphi \vee \psi, \neg\varphi\}$.

$$\dfrac{\dfrac{\dfrac{\Gamma, \varphi \vdash \varphi \quad \Gamma, \varphi \vdash \neg\varphi}{\Gamma, \varphi \vdash \bot}}{\Gamma, \varphi \vdash \psi} \qquad \dfrac{\Gamma, \psi \vdash \psi \quad \Gamma \vdash \varphi \vee \psi}{\Gamma \vdash \psi}}{\varphi \vee \psi \vdash \neg\varphi \to \psi}$$

Example 1.2.10 $\neg(\varphi \vee \neg\varphi) \vdash \neg\neg\varphi$. $\diamond$

Proof:

$$\dfrac{\dfrac{\dfrac{\neg(\varphi \vee \neg\varphi), \neg\varphi \vdash \neg\varphi}{\neg(\varphi \vee \neg\varphi), \neg\varphi \vdash \varphi \vee \neg\varphi} \quad \neg(\varphi \vee \neg\varphi), \neg\varphi \vdash (\varphi \vee \neg\varphi) \to \bot}{\neg(\varphi \vee \neg\varphi), \neg\varphi \vdash \bot}}{\neg(\varphi \vee \neg\varphi) \vdash \neg\neg\varphi}$$

Remark 1.2.11 This proof is a strict one, but maybe too wordy. Sometimes we can just omit some trivial constructions like $\varphi \vdash \varphi \vee \neg\varphi$. Thus the above proof can be shortened and more intuitive.

$$\dfrac{\dfrac{\neg(\varphi \vee \neg\varphi), \neg\varphi \vdash \varphi \vee \neg\varphi}{\neg(\varphi \vee \neg\varphi), \neg\varphi \vdash \bot}}{\neg(\varphi \vee \neg\varphi) \vdash \neg\neg\varphi}$$

There are a lot of alternative choices for classical logical axioms, some are more convenient under certain circumstances. In fact, we can show many classical tautologies are equivalent to each other in the sense of intuitionism.

Lemma 1.2.12 *(Weakening) If* $\Gamma \vdash \varphi$*, then for any other context* Δ*,* $\Gamma \cup \Delta \vdash \varphi$*.*

Proof: Obvious. □

Proposition 1.2.13 *In intuitionistic logic, the following are equivalent.*

1. *Law of Excluded Middle(LEM):* $\varphi \vee \neg\varphi$
2. *Double Negation Elimination(DNE):* $\neg\neg\varphi \to \varphi$
3. *Proof by Contradiction:* $(\neg\alpha \to \neg\beta) \to (\neg\alpha \to \beta) \to \alpha$

More specifically, I want to show in intuitionistic logic that

1. $\varphi \vee \neg\varphi \vdash \neg\neg\varphi \to \varphi$*;*
2. $\neg\neg\alpha \to \alpha \vdash (\neg\alpha \to \neg\beta) \to (\neg\alpha \to \beta) \to \alpha$*;*
3. $(\neg(\varphi \vee \neg\varphi) \to \neg\neg\varphi) \to (\neg(\varphi \vee \neg\varphi) \to \neg\varphi) \to \varphi \vdash \varphi \vee \neg\varphi$ *(with* α *replaced by* $\varphi \vee \neg\varphi$ *and* β *replaced by* $\neg\varphi$*).*

Proof: (1 implies 2) Let $\Gamma_1 = \{(\varphi \to \bot) \to \bot, \varphi \vee \neg\varphi\}$. Note that $\neg\varphi = \varphi \to \bot$ and $\neg\neg\varphi = (\varphi \to \bot) \to \bot$.

$$\dfrac{\dfrac{\Gamma_1, \varphi \to \bot \vdash \varphi \to \bot \quad \Gamma_1, \varphi \to \bot \vdash (\varphi \to \bot) \to \bot}{\Gamma_1, \varphi \to \bot \vdash \bot} \quad \Gamma_1, \varphi \to \bot \vdash \varphi \qquad \Gamma_1 \vdash \varphi \vee \neg\varphi}{\dfrac{\Gamma_1 \vdash \varphi}{\varphi \vee \varphi \vdash \neg\neg\phi \to \varphi}}$$

The idea behind this proof is that to make use of $\varphi \vee \neg\varphi$, by the elimination rule of $\vee$ in 1.2.7, we want to construct φ under a context with φ and $\varphi \to \bot$ respectively. The latter gives us an excuse to obtain $\bot$ from Γ_1.

(2 implies 3). Let $\Gamma_2 = \{\neg\neg\alpha \to \alpha, \neg\alpha \to \neg\beta, \neg\alpha \to \beta\}$.

$$\dfrac{\dfrac{\dfrac{\Gamma_2, \neg\alpha \vdash \neg\alpha \quad \Gamma_2, \neg\alpha \vdash \neg\alpha \to \neg\beta}{\Gamma_2, \neg\alpha \vdash \neg\beta} \quad \dfrac{\ldots}{\Gamma_2, \neg\alpha \vdash \beta}}{\Gamma_2, \neg\alpha \vdash \bot}}{\dfrac{\Gamma_2 \vdash \neg\neg\alpha \qquad \Gamma_2 \vdash \neg\neg\alpha \to \alpha}{\Gamma_2 \vdash \alpha}}$$

If we have a function $\neg\alpha \to (\beta \to \bot)$ and a function $\neg\alpha \to \beta$, then it is natural to have a function $\neg\alpha \to \bot$, i.e., $\neg\neg\alpha$. Then we apply the double negation rule to obtain α. It suffices to show that $\Gamma_2 \vdash \alpha$.

(3 implies 2). $\Gamma_3 = \{(\neg(\varphi \vee \neg\varphi) \to \neg\neg\varphi) \to (\neg(\varphi \vee \neg\varphi) \to \neg\varphi) \to \varphi\}$. We want to construct a contradiction from $\neg(\varphi \vee \neg\varphi)$. The answer is quite clear. We must have $\neg\varphi$ and $\neg\neg\varphi$ from it. From 1.2.10, we know $\neg(\varphi \vee \neg\varphi) \vdash \neg\neg\varphi$. Thus $\Gamma_3 \vdash \neg(\varphi \vee \neg\varphi) \to \neg\neg\varphi$. Similarly, we have $\Gamma_3 \vdash \neg(\varphi \vee \neg\varphi) \to \neg\varphi$. By applying MP twice, we have $\Gamma_3 \vdash \varphi \vee \neg\varphi$. □

Remark 1.2.14 You may wonder whether all classical tautologies that are not intuitionistic tautologies are equivalent to each other in intuitionistic logic. Unfortunately, I do not know such a proof or counterexample. If this is correct, then by putting any non-intuitionistic classical tautology in the system, you can get the classical proof system. ♢

Remark 1.2.15 Recall we defined $\wedge$ and $\vee$ as abbreviations of $\neg$ and $\rightarrow$ when defining the proof system for classical logic. If you choose to introduce logical axioms for other connectives like $(\neg\alpha \rightarrow \beta) \rightarrow \alpha \vee \beta$ into that system, then you can find that they are equivalent to the above three formulas in intuitionistic logic. Thus we do not have to worry about the other choice f classical logic. $\diamond$

We still have the soundness and completeness in the intuitionistic system.

Theorem 1.2.16 *For a formula φ and a set Γ of formulas, in intuitionistic logic, we also have*

1. *(Soundness)* $\Gamma \vdash \varphi$ *implies* $\Gamma \vDash \varphi$*;*
2. *(Completeness)* $\Gamma \vDash \varphi$ *implies* $\Gamma \vdash \varphi$*.*

Proof: Soundness is obvious.

For completeness, the idea is still the same. We define a specific Heyting algebra based on the context Γ. Let $\sim$ be the equivalence relation of $\mathscr{F}$ such that

$$\theta \sim \psi \text{ if and only if } \Gamma \vdash \theta \rightarrow \psi \text{ and } \Gamma \vdash \psi \rightarrow \theta$$

Let $\mathcal{L}_\Gamma = \mathscr{F}/_\sim = \{[\theta] \mid \theta \in \mathscr{F}\}$ be a poset with order $[\theta] \leq [\psi]$ if and only if $\Gamma \vdash \theta \rightarrow \psi$, where $[\theta]$ denotes the equivalence class of θ. To make it a Heyting algebra, we define

- $[a] \cup [b] = [a \vee b]$;
- $[a] \cap [b] = [a \wedge b]$;
- $[a] \Rightarrow [b] = [a \rightarrow b]$;
- $0 = [\bot]$;
- $1 = [\bot \rightarrow \bot]$. $\square$

$\mathcal{L}_\Gamma$ is called the Lindenbaum algebra of Γ. It is easy to verify $\mathcal{L}_\Gamma$ is a well defined algebra. Definition the assignment $v(\psi) = [\psi]$. Clearly for all $\psi \in \Gamma$, $\Gamma \vdash \psi$. Thus $\mathcal{L}_\Gamma, v \vDash \Gamma$. Since $\Gamma \vDash \varphi$, we know $v(\varphi) = 1 = [\bot \to \bot]$, i.e., $\Gamma \vdash (\bot \to \bot) \to \varphi$. This forces $\Gamma \vdash \varphi$.

1.3 Relation between them

Though in intuitionistic logic we do not admit 'proof by contradiction', however, there are still an intuitionistic strategy based on contradiction. More specifically, a formula φ is a classical tautology if and only if its double negation $\neg\neg\varphi$ is an intuitionistic one, which means if we assume the negation of a classical tautology, we must find a contradiction $\bot$ constructively.

In 1.2.13, I showed that 'proof by contradiction' is equivalent to 'double negation elimination'. This double negation translation also tells us that in classical logic, you only have to prove the double negation in intuitionistic logic and use the double negation rule as the last step.

This relation was first revealed by Glivenko [12] in 1929. The proof I am going to show you is different from other proofs [23]. Actually my proof is inspired by a programming language called Haskell [2][§]. You may have noticed that I constructed some proofs in a computational sense like 1.3.2, 1.2.13. There exists a functional way to construct a proof. That is how I find this proof from computer programming. This idea will be formalized by *λ-calculus* in Chapter 2.

Lemma 1.3.1 *Rule 2(a)* $\alpha \to \beta \to \alpha$ *and 2(b)* $(\alpha \to (\beta \to \gamma)) \to (\alpha \to \beta) \to (\alpha \to \gamma)$ *in 1.1.14 are intuitionistic tautologies.*

Proof: If you understand $\to$ as a function, then the proof of 2(a) is to define a

[§]The Glivenko's theorem was used in computer science known as the continuation passing style transformation [3]. Haskell uses this idea to fulfill its concurrency model [8].

function ignoring an argument.

$$\dfrac{\dfrac{\alpha, \beta \vdash \alpha}{\alpha \vdash \beta \to \alpha}}{\vdash \alpha \to (\beta \to \alpha)}$$

Let $\Gamma = \{\alpha \to (\beta \to \gamma), \alpha \to \beta, \alpha\}$. It suffices to show $\Gamma \vdash \gamma$ because of the introduction rule of $\to$.

$$\dfrac{\dfrac{\Gamma \vdash \alpha \quad \Gamma \vdash \alpha \to \beta}{\Gamma \vdash \beta} \quad \dfrac{\Gamma \vdash \alpha \quad \Gamma \vdash \alpha \to (\beta \to \gamma)}{\Gamma \vdash \beta \to \gamma}}{\Gamma \vdash \gamma}$$

This idea of this proof is composition of functions. □

The first step is to prove the double negation of rule 2(c), the proof by contradiction axiom.

Lemma 1.3.2

$$\vdash \neg\neg((\neg\alpha \to \neg\beta) \to (\neg\alpha \to \beta) \to \alpha)$$

Proof: Let $\varphi = (\neg\alpha \to \neg\beta) \to (\neg\alpha \to \beta) \to \beta$. First observe that

$$\dfrac{\dfrac{\dfrac{\neg\varphi, \alpha, \neg\alpha \to \neg\beta, \neg\alpha \to \beta \vdash \alpha}{\neg\varphi, \alpha \vdash (\neg\alpha \to \neg\beta) \to (\neg\alpha \to \beta) \to \alpha}}{\neg\varphi, \alpha \vdash \bot}}{\neg\varphi \vdash \neg\alpha}$$

Thus by weakening,

$$\dfrac{\neg\varphi, \neg\alpha \to \neg\beta, \neg\alpha \to \beta \vdash \neg\alpha \quad \neg\varphi, \neg\alpha \to \neg\beta, \neg\alpha \to \beta \vdash \neg\alpha \to \neg\beta}{\neg\varphi, \neg\alpha \to \neg\beta, \neg\alpha \to \beta \vdash \neg\beta}$$

Similarly, $\neg\varphi, \neg\alpha \to \neg\beta, \neg\alpha \to \beta \vdash \beta$. Then

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg\varphi, \neg\alpha \to \neg\beta, \neg\alpha \to \beta \vdash \bot}{\neg\varphi, \neg\alpha \to \neg\beta, \neg\alpha \to \beta \vdash \alpha}}{\neg\varphi \vdash (\neg\alpha \to \neg\beta) \to (\neg\alpha \to \beta) \to \alpha}}{\neg\varphi \vdash \bot}}{\vdash \neg\neg((\neg\alpha \to \neg\beta) \to (\neg\alpha \to \beta) \to \alpha)}$$

Lemma 1.3.3 *In intuitionistic logic, we have* $\vdash \varphi \to \neg\neg\varphi$

Proof: $\varphi, \varphi \to \bot \vdash \bot$. □

Corollary 1.3.4 *We have double negation elimination for negation, namely* $\neg\neg\neg\varphi \vdash \neg\varphi$. *After we prove Glivenko's theorem, this shows if we want to prove a negation, there must be an intuitionistic proof.*

Proof:

$$\dfrac{\dfrac{\neg\neg\neg\varphi, \varphi \vdash \neg\neg\varphi}{\neg\neg\neg\varphi, \varphi \vdash \bot}}{\neg\neg\neg\varphi \vdash \neg\varphi}$$

Lemma 1.3.5 *In intuitionistic logic, if* $\Gamma \vdash \neg\neg\alpha$ *and* $\Gamma, \alpha \vdash \neg\neg\beta$, *then* $\Gamma \vdash \neg\neg\beta$. *Equivalently, this is a tautology* $\vdash (\alpha \to \neg\neg\beta) \to (\neg\neg\alpha \to \neg\neg\beta)$.

Proof: The idea is that if you have $\alpha \to \neg\neg\beta, \neg\beta$, then you are able to form a function $\alpha \to \neg\beta$ and then a function $\alpha \to \bot$. Together with $\neg\neg\alpha$, we can have the desired $\bot$. Let $\Sigma = \{\alpha \to \neg\neg\beta, \neg\neg\alpha, \neg\beta\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma, \alpha \vdash \alpha \quad \Sigma, \alpha \vdash \alpha \to \neg\neg\beta}{\Sigma, \alpha \vdash \neg\neg\beta} \quad \Sigma, \alpha \vdash \neg\beta}{\Sigma, \alpha \vdash \bot}}{\Sigma \vdash \neg\alpha} \quad \Sigma \vdash \neg\neg\alpha}{\Sigma \vdash \bot}}{\alpha \to \neg\neg\beta, \neg\neg\alpha \vdash \neg\neg\beta}}{\vdash (\alpha \to \neg\neg\beta) \to (\neg\neg\alpha \to \neg\neg\beta)}$$

Corollary 1.3.6 $\alpha \to \beta \vdash \neg\neg\alpha \to \neg\neg\beta$.

Remark 1.3.7 Since it is easy to prove $\alpha \to \beta, \beta \to \gamma \vdash \alpha \to \gamma$ and $\alpha \vdash \alpha$, we can think of both logics as categories, where objects are formulas and morphisms are provability $\vdash$ between them. The identities are represented by $\varphi \vdash \varphi$ and compositions are syllogism. For example, A and $A \vee B$ are two formulas. Since $A \vdash A \vee B$, there is a morphism from A to $A \vee B$.

This category is a preorder, where there exists at most one morphism from one object to another. This corollary tells us that $\neg\neg$ is an endofunctor. Together with 1.3.3 and 1.3.5, they give a *Kleisli tuple* [16], which is an equivalent form of adjunction (1.3.11). ◊

Theorem 1.3.8 *$\Gamma \vdash \varphi$ in classical logic if and only if $\Gamma \vdash \neg\neg\varphi$ in intuitionistic logic.*

Proof: If $\Gamma \vdash \neg\neg\varphi$ intuitionistically, then certainly $\Gamma \vdash \neg\neg\varphi$ classically. By 1.2.13, we can put $\neg\neg\varphi \to \varphi$ into the proof sequence. Using MP, we have $\Gamma \vdash \varphi$.

Conversely, we can assume a classical proof sequence $\varphi_1, \varphi_2, \cdots, \varphi_n = \varphi$ for $\Gamma \vdash \varphi$. For each φ_i, if it is obtained as a logical axiom, then we can freely move it to a former position in the sequence. That means we can move all φ_i obtained by 'proof by contradiction' to the beginning of the sequence. Thus we can assume the first k formulas $\varphi_1, \cdots, \varphi_k$ are 'proof by contradiction' and each of the rest is a logical axiom admitted by intuitionistic logic (rule 2(a) and 2(b)), or the result of MP, or a formula in Γ.

I am going to show $\Gamma, \varphi_1, \cdots, \varphi_k \vdash \varphi_i$ in intuitionistic logic by induction on i.

- If $i \leq k$, then certainly we have $\Gamma, \varphi_1, \cdots, \varphi_k \vdash \varphi_i$ in intuitionistic logic.
- Otherwise,
  - if φ_i is a logical axiom, by 1.3.1, $\Gamma, \varphi_1, \cdots, \varphi_k \vdash \varphi_i$;
  - if $\varphi \in \Gamma$, then $\Gamma, \varphi_1, \cdots, \varphi_k \vdash \varphi_i$

– if φ_i is obtained by MP, then we can find $\varphi_j = \varphi_l \to \varphi_i$ such that $j, l < i$. By inductive hypothesis, $\Gamma, \varphi_1, \cdots, \varphi_k \vdash \varphi_j$ and $\Gamma, \varphi_1, \cdots, \varphi_k \vdash \varphi_l$. Thus

$$\frac{\Gamma, \varphi_1, \cdots, \varphi_k \vdash \varphi_l \quad \Gamma, \varphi_1, \cdots, \varphi_k \vdash \varphi_l \to \varphi_i}{\Gamma, \varphi_1, \cdots, \varphi_k \vdash \varphi_i}$$

Now we have $\Gamma, \varphi_1, \cdots, \varphi_k \vdash \varphi$. By 1.3.3, $\Gamma, \varphi_1, \cdots, \varphi_k \vdash \neg\neg\varphi$, Since φ_k, is a 'proof by contradiction', by 1.3.2, $\Gamma, \varphi_1, \cdots, \varphi_{k-1} \vdash \neg\neg\varphi_k$. Finally, by 1.3.5, we have $\Gamma, \varphi_1, \cdots, \varphi_{k-1} \vdash \neg\neg\varphi$. Repeating this process, we have $\Gamma \vdash \neg\neg\varphi$. □

Corollary 1.3.9 *(Glivenko's Theorem) A formula is a classical tautology if and only if its double negation is an intuitionistic one.*

Remark 1.3.10 This theorem tells us that intuitionistic logic is not weaker than classical logic. We can give a constructive interpretation to the double negation of a classical tautology by stating why the negation will trigger a contradiction. For example, Brouwer's fixed point theorem [20] is proved by finding a counterexample($\perp$) constructively.

Though Glivenko's original theorem only states the tautologies, it motivates the generalization (1.3.8). So it is reasonable to also call it Glivenko's theorem.

The proof I used for Glivenko's theorem is a *monadic* one. The benefit of my proof is that monad raises adjoint functors [16]. This monadic proof also enables us to bring classical tautologies into our context directly. If you are trying to write a proof in a computational sense, this will helps you shorten your proof, which is also motivated by Haskell [1]. ⋄

According to 1.3.6, we can view intuitionistic logic and classical logic as categories. Then the two logics can be associated by an adjunction defined by double negation.

Corollary 1.3.11 *Let F be an object assignment from intuitionistic logic to classical logic sending each formula to itself, and G be an object assignment*

from classical logic to intuitionistic logic sending each φ to $\neg\neg\varphi$. Then F and G are functors and they are adjoint.

Proof: If $\alpha \vdash \beta$ classically, then by Glivenko's theorem 1.3.8, $\alpha \vdash \neg\neg\beta$ intuitionistically. By 1.3.5, $\neg\neg\alpha \vdash \neg\neg\beta$. This gives the morphism assignment of G. Since $\neg\neg\varphi \vdash \neg\neg\varphi$ and

$$\neg\neg\alpha \to \neg\neg\beta, \neg\neg\beta \to \neg\neg\gamma \vdash \neg\neg\alpha \to \neg\neg\gamma,$$

identities and compositions are preserved. Thus G is a functor. Similarly F is a functor.

By Glivenko's theorem 1.3.8, for fomulas α and β, $\alpha \vdash \beta$ classically if and only if $\alpha \vdash \neg\neg\beta$ intuitionistically. This means there is a morphism $F\alpha \vdash \beta$ if and only if there is a morphism $\alpha \vdash G\beta$, which exactly means they are ajoint to each other. □

Chapter 2

Curry-Howard Correspondence

In Chapter 1, we have already seen the proof of $A \to (B \to A)$ (1.3.1):

$$\dfrac{\dfrac{A, B \vdash A}{A \vdash B \to A}}{\vdash A \to (B \to A)}$$

The idea of this proof is that as long as we have the A in the premises, we can always bring it to the consequence, like a function $A \times B \to A$. As long as we have element $(a, b) \in A \times B$, we can use a as the returned value. You can also find the same idea in the proof of 1.2.13, 1.3.5, etc. This functional interpretation motivates us to think about whether we can really write a function as a proof.

Fortunately, there exists a language called λ-calculus* for this purpose. During the above proofs, we frequently use lots of function operations. Unlike the traditional functions, we do not want to always mention the names of these functions during the proof. Think about the function $f(x) = x^2$. Though we give it a name f, this function actually reflects a computation $x \mapsto x^2$. But how does this help with a proof? For example if you want to prove $A \to (B \to A)$, you can interpret A as the set of all proofs of A. Thus this formula is a function given $a \in A$ and $b \in B$ returning an element in A. So you can simply find a function $x \mapsto (y \mapsto x)$ as the evidence of $A \to (B \to A)$. λ-calculus enables us to define anonymous functions like $x \mapsto (y \mapsto x)$ easily, thus being a suitable language for intuitionistic logic.

*λ-calculus was purposed by Church [7] to represent computation in logic.

2.1 λ-calculus

Let's first take a look at the λ-calculus itself.

Definition 2.1.1 *(λ-terms) Let $U = \{a, b, c, \cdots\}$ be a set of variables like the variable set V we used in 1.1.1. We recursively define λ-terms with following rules.*

1. *Those in U are λ-terms.*
2. *If $x \in U$ and M is another λ-term, then so is $(\lambda x.M)$. This is the* **abstraction** *of M over x.*
3. *If M, N are λ-terms, then so is $(M\ N)$. This is the* **application** *of M to N.*

Let Λ be the set of all λ-terms.

Example 2.1.2 The following are λ-terms.

1. x

   A variable is a valid λ-term which may not have a functional interpretation, like the $1, 2, 3$ applied to a function $x \mapsto x^2$.

2. $(\lambda x.x)$

   This is the identity function $x \mapsto x$.

3. $((\lambda x.x)\ y)$

   We can apply a function to another term. Though we have not defined how to make a computation yet, by an intuitive substitution of variable, this application should yield y.

4. $(((\lambda x.(\lambda y.x))\ z)\ w)$ The term $(\lambda x.(\lambda y.x))$ is a bit like a function returning a function. If you apply it to z, then you should get a function $(\lambda y.z)$, and then if you apply it to w, it will give you z, i.e., $(((\lambda x.(\lambda y.x))\ z)\ w)$

should be calculated to z. This is how we define functions with multiple arguments. ◊

Intuitively, the abstraction is like the analytic form of a function. For example, the function $f(x) = x^2 + 1$ can be understood as abstraction $f = (\lambda x.(x^2+1))$. $(1, 2, +$ will be introduced into our theory later as λ-terms. Now you can just use them intuitively as real numbers.) The application is also analogous to function application, e.g., $(f\ 2) = ((\lambda x.(x^2 + 1))\ 2)$. As you may guess, we can 'compute' this λ-term with purely substitution: $(f\ 2) = ((\lambda x.(x^2 + 1))\ 2) = (2^2 + 1)$.

Let's make the following conventions to save parentheses.

1. The outermost parentheses are omitted without ambiguity.
2. $(\lambda x.\lambda y.M)$ is $(\lambda x.(\lambda y.M))$.
3. $(K\ L\ M)$ is $((K\ L)\ M)$.
4. $(\lambda x.M\ N)$ is $(\lambda x.(M\ N))$.
5. $(M\ \lambda x.N)$ is $(M\ (\lambda x.N)))$

Remark 2.1.3 There is a motivation for these conventions. Recall we let $\to$ be right associative because a function $f : A \times B \to C$ can be viewed as a function given an element in A returning a function from $B \to C$, i.e., $A \to (B \to C)$, where $(B \to C)$ means functions from B to C. If you apply such a function to $a \in A$, you will get a function $f(a) : B \to C$.

So for functions with multiple arguments, we prefer this style to define them as λ-terms, i.e., a function with two input variables is defined by a function outputing functions. Dually, if you want to feed two arguments to such a function, the application is naturally left associative to receive the arguments respectively. For example $f(a)(b)$ is written as $(f\ a)\ b$ in our theory, and that is $f\ a\ b$ as in our convention. This style is called *currying*[†]. ◊

[†]In memory of Haskell Curry.

Before we can define substitution, let's first handle some technical issues[‡]. In our definition of λ-terms, there are no restriction on the choice of variables in λ-calculus. That means we can reuse the variables after λ, e.g., $\lambda x.\lambda x.x$. This could cause some ambiguity, i.e., we can either view it as a function given an x, returning a function $\lambda x.x$, or a function given any value v, returning a function $\lambda x.v$. Intuitively, the two terms $\lambda x.x$ and $\lambda y.y$ should both be the same because they both represent the identity. Thus in most cases, we should avoid this ambiguity by writing $\lambda x.\lambda y.y$ to specify the function always returning $\lambda x.x$, and use $\lambda x.\lambda y.x$ to specify the other.

This also raises another issue. We have to define some equivalence for different terms like $\lambda x.x$ and $\lambda y.y$. Since we abstract a function by its computation, not by its graph as in set theory, we may use a different input variable when defining the computation, e.g., $f(y) = y^2 + 1$ should be the same function as $f(x) = x^2 + 1$, despite different variables.

Apparently, this equivalence relation is based on substitution of variables, i.e., we can freely change the variable we use behind λ. But we must be careful with substitution in order to avoid some weird results. For example, we can replace the z in term $\lambda y.z\ y$ with y, but after the substitution, we get $\lambda y.y\ y$, which certainly have a different computational meaning. Also, we can not abuse substitution. For example we think $\lambda y.z$ and $\lambda x.z$ are the same, but $\lambda y.z$ and $\lambda y.w$ should be different.

The key observation here is that the variable after the λ sign is bound to the term after it. Suppose we are writing a long passage. Throughout the passage we are using a symbol x, but in one paragraph, we use that symbol x in a lemma. The x in this lemma is certainly different from the x in the passage, and if we change it to y as well as other occurrences of x in the lemma, we will get the same lemma. Because the symbol x makes sense in the lemma, and it

[‡]These issues caused by naming conflict exist in many formal languages such as first order language, programming language.

will not cause trouble, we do not always change it to a cumbersome name. In this example, the symbol x is bound to the lemma, and other symbols in the lemma without such a binding have meaning associated to the context. Thus we have to distinguish the bound variables from the others.

Definition 2.1.4 *(Free Variable) For a λ-term M, we define a set $FV(M) \subseteq U$ of* **free variables** *recursively.*

$$FV(x) = \{x\}$$
$$FV(\lambda x.P) = FV(P) \setminus \{x\}$$
$$FV(P\ Q) = FV(P) \cup FV(Q)$$

Example 2.1.5

1. $\mathrm{FV}(a\ b) = \{a, b\}$
2. $\mathrm{FV}(\lambda x.x\ y) = \{y\}$ ◊

Our idea is very simple. We should only change the variable behind the λ, and all other free variables in a term should remain the same. You cannot substitute y for x in the term $\lambda y.x\ y$, because after the substitution, y at the occurrence of x will be bound by the λ, i.e. $\lambda y.y\ y$, which yields a different meaning. Since the choice of y is not important, we can choose a new variable z and do the substitution, i.e. $\lambda z.y\ z$, to capture the substitution y from the context.

Definition 2.1.6 *(Substitution of variables) For a term M and variables x, y, the* **substitution** *of y for x in M, wrtten as $M[x := y]$ is recursively defined as follows.*

1. $x[x := y] = y$.
2. $z[x := y] = z$, *where* $z \neq x$.

3. $(P\ Q)[x := y] = (P[x := y])\ (Q[x := y])$, *where* P, Q *are other two terms.*

4. $(\lambda x.P)[x := y] = \lambda x.P$.
   Nothing happens if you want to replace the x *after* λ.

5. $(\lambda z.P)[x := y] = \lambda z.P[x := y]$, $z \neq x$ *and* $y \neq z$.

6. $(\lambda y.P)[x := y] = \lambda z.(P[y := z][x := y])$, *where* z *is a variable such that* $z \neq x$ *and* $z \neq y$ *and* $z \notin FV(P)$.

Remark 2.1.7 In Rule 6, we always pick a brand new variable so that there will not be any conflict. Since we are going to define an equivalence based on substitution, you can choose any new variable. This is well defined only later. If you want a set-theoretical well-defined function, you can well-order the variable set V so that you can pick the minimal element among the unused variables. ◊

Now we can think about the equivalence based on variable substitution.

Definition 2.1.8 *(α-equivalence) We recursively define the relation* $=_\alpha$ *as follows.*

1. *For any* λ*-terms* M, $M =_\alpha M$.

2. *If* $y \notin FV(M)$, *then* $\lambda x.M =_\alpha \lambda y.(M[x := y])$.
   e.g. $\lambda x.x\ z =_\alpha \lambda y.y\ z \neq_\alpha \lambda z.z\ z$.

3. *If* $M =_\alpha M'$, *then* $\lambda x.M =_\alpha \lambda x.M'$.

4. *If* $M =_\alpha M'$, *then* $M\ N =_\alpha M'\ N$ *and* $N\ M =_\alpha N\ M'$.

5. $=_\alpha$ *is symmetric, i.e.,* $M =_\alpha M'$ *implies* $M' =_\alpha M$.

6. $=_\alpha$ *is transitive, i.e.,* $M =_\alpha N$ *and* $N =_\alpha P$ *implies* $M =_\alpha P$.

Example 2.1.9 1. $\lambda x.y =_\alpha \lambda z.y$

2. $\lambda x.x =_\alpha \lambda y.y$

3. $\lambda x.y \neq_\alpha \lambda x.z$

   By definition, if $\lambda x.y =_\alpha \lambda x.z$, then this should come from rule 3, i.e., $y =_\alpha z$, which is not included in the rules defining $=_\alpha$. ♢

Let's think about our intuition of computation. If you apply the identity $\lambda x.x$ to any term M, you should get the same term. The idea behind it is still substitution. Similarly, the free variables will still cause some problems. Consider the application $(\lambda x.\lambda y.x)\ y$. If you simply substitute y for x, the result is $\lambda y.y$, which has a different meaning. Thus we adopt the same pattern as in the definition of $=_\alpha$ to handle it.

Definition 2.1.10 *(Substitution of λ-terms) For $M, N \in \Lambda$ and variable x, the* **substitution** *of N for x in M, written as $M[x := N]$ is recursively defined as follows.*

1. $x[x := N] = N$
2. $y[x := N] = y,\ y \neq x$
3. $(P\ Q)[x := N] = (P[x := N]\ Q[x := N])$
4. $(\lambda x.P)[x := N] = \lambda x.P$
5. $(\lambda y.P)[x := N] = \lambda z.(P[y := z][x := N])$, *$y \neq x$ and z is chosen such that* $z \notin FV(N) \cup FV(P) \cup \{x, y\}$

Remark 2.1.11 It is easy to observe that this definition is compatible with our definition of variable substitution. The z in Rule 5 is to handle situations like $(\lambda x.\lambda y.x)\ y$. By this definition, $(\lambda y.x)[x := y] = \lambda z.y$, which is consistent with our intuition.

Rule 5 says that if we want to replace a term inside an abstraction, we always pick a new variable and do the variable substitution in order to avoid conflict. Since we have the α equivalence, this will not change the meaning of calculation. ♢

Finally, we define our computation as the relation of β-reduction. Intuitively, a computation happens if you apply a function to some value, e.g., $(x \mapsto x^2)\ 2$. The only thing we need to do is substitution, i.e., 2^2. But sometimes we have to do the substitution embedded in a longer expression like $1 + (f\ 2)$ where $f\ x = x^2$. The substitution only happens in some part of the whole expression. Thus we have to define it recursively as we recursively define the λ-term.

Definition 2.1.12 *(β-reduction) We recursively define a relation $\to_\beta$ (called* **β-reduction***) between λ-terms*

1. *$(\lambda x.M)\ N \to_\beta M[x := N]$. This is our intuition of computation.*
2. *If $M \to_\beta N$, then $\lambda x.M \to_\beta \lambda x.N$.*
3. *If $M \to_\beta N$, then $M\ P \to_\beta N\ P$ and $P\ M \to_\beta P\ N$.*

Example 2.1.13

$$\begin{aligned} & (\lambda x.\lambda y.x^2 + y)\ 2\ 3 \\ \to_\beta\ & (\lambda y.2^2 + y)\ 3 \\ \to_\beta\ & 2^2 + 3 = 7 \end{aligned}$$

If $M \to_\beta N$, then there must be exactly one substitution during this process. Thus this relation, just like the $=$ in basic arithmetic, enables us to perform a computation step by step. Hence this process is often called *λ-calculus*. It is then reasonable to define the transitive and reflexive closure $\twoheadrightarrow_\beta$ and the equivalent closure $=_\beta$.

Definition 2.1.14 *(multi-step β-reduction and β-equality) The* **multi-step β-reduction** *$\twoheadrightarrow_\beta$ is recursively generated under the rules:*

1. *If $M \to_\beta N$ then $M \twoheadrightarrow_\beta N$.*
2. *If $M \twoheadrightarrow_\beta N$ and $N \twoheadrightarrow_\beta P$, then $M \twoheadrightarrow_\beta P$.*
3. *$M \twoheadrightarrow_\beta M$.*

and the β**-equality** $=_\beta$ *is recursively generated under the rules:*

1. *If* $P \to_\beta Q$*, then* $P =_\beta Q$*.*
2. *If* $P =_\beta Q$ *and* $Q =_\beta R$*, then* $P =_\beta R$
3. $P =_\beta P$*.*
4. *If* $P =_\beta Q$*, then* $Q =_\beta P$*.*

Remark 2.1.15 Let's revisit the naming conflict problem. For example, $(\lambda z.\lambda x.x\ z)\ (x\ y)\ w$. Our definition of $\to_\beta$ and substitution should stop us from doing the substitution $(\lambda x.x\ z)[z := (x\ y)]$. But note the λ-terms are actually equivalence classes. Since $(\lambda z.\lambda x.x\ z) =_\alpha (\lambda a.\lambda b.b\ a)$, this term is equal to $(\lambda a.\lambda b.b\ a)\ (x\ y)\ w \to_\beta (\lambda b.b\ (x\ y))\ w \to_\beta w\ (x\ y)$. This renaming process is traditionally called α-conversion, which explains the name $=_\alpha$. $\diamond$

Up to now, we have defined a computational language which seems to be the evidence of certain formulas. However, this language is too flexible. We never put any restrict on the *domain* and *codomain* of a function. This means we can apply a term to itself, which could be paradoxical. For example, the term $Y = \lambda f.(\lambda x.f\ (x\ x))\ \lambda x.f\ (x\ x)$ will cause Curry's paradox [19] like Russell's paradox. We will develop a *typed* version of λ-calculus to avoid the paradox.

The idea is to assign a *type* to a λ-term in order to restrict the behavior of application. All the terms we defined above can be applied to any other term, but for a function $f : \mathbb{N} \to \mathbb{N}$, you will never apply it to $\sqrt{2}$ and you will not expect $f(3)$ is an element of the set $\{cat, dog, duck\}$. Thus we attach each λ-term with such an annotation.

Recall the set V of all propositional variables. We use it to generate our type annotation.

Definition 2.1.16 *(Simple Type) The* **simple types** *are recursively defined as follows.*

1. *Each element in V is a simple type.*
2. *If φ and ψ are simple types, then so is $\varphi \to \psi$.*

Example 2.1.17 The following are examples of simple types.

1. A, for some $A \in V$
2. $A \to B$
3. $A \to (A \to C)$ ◇

We make some conventions for λ-terms and simple types.

- We use x, y, z for variables in λ-terms (set U).
- We use M, N, P, Q for arbitrary λ-terms.
- We use A, B, C for simple types associated to variables (set V).
- We use σ, τ for arbitrary simple types.

We want to annotate each λ-term with a simple type, like the notation $f : \mathbb{R} \to \mathbb{C}$ in set theory to indicate its 'domain' and 'codomain'. For example, $\lambda x.x : \tau \to \tau$ for some simple type τ. Informally, the term $\lambda x.x$ can be viewed as given an x of type τ, you can obtain a result $x : \tau$. Here we have a naive idea that if you have $x : \tau$, then you can get $x : \tau$ in our meta-language, and the λ-term $\lambda x.x$ shows how we bring it to our target language, λ-calculus, a bit like you can obtain $\vdash \tau \to \tau$ from $\tau \vdash \tau$ in propositional logic. Thus we define the notation ':' together with a *context*, which, just like its counterpart in propositional logic, means what we have already had at hand. As in set theory, if you want to make a function $f : \mathbb{R} \to \mathbb{C}$ by $f(x) = x + x^2 i$, you first have an $x \in \mathbb{R}$ at hand and then you are allowed to define $x + x^2 i \in \mathbb{C}$. The

analytic form $f(x) = x + x^2 i$ is how we generalize the computation in our mind (meta-language) to a formal expression (target language). The notation ':' is analogous to the '$\in$' in set theory.

To formalize this for λ-calculus, we first make a context Γ containing the information like $x \in \mathbb{R}$, and then we decide whether we are allowed to annotation a term with a type under this context. We still write this with $\vdash$ symbol, e.g., $\Gamma \vdash M : \varphi$, but it is not related to the one we used in propositional logic. This relation $\vdash$ is defined for our meta-language, and the ':' notation is defined for λ-calculus. In 3, $\vdash$ will all be represented by λ-terms and in the final version, we no longer need the $\vdash$ notation. Thus we call $\Gamma \vdash M : \varphi$ a *typing (annotation) judgement* to indicate it happens in the meta-language and call $M : \varphi$ a *typing (annotation) statement* as a notation in the target language.

Definition 2.1.18 *(Context) A* **context** *is a set* $\Gamma = \{x_1 : \tau_1, x_2 : \tau_2, \cdots\}$ *of pairs with* $x_i \in U$ *(I use* U *for variables in* λ*-terms and* V *for simple types) and simple types* τ_i*. The* x_i *must be different from each other.*

As discussed above, we certainly have $\Gamma, x : \tau \vdash x : \tau$ and if we have $\Gamma, x : \tau \vdash M : \sigma$, then $\Gamma \vdash \lambda x.M : \tau \to \sigma$. As in propositional logic, to formally introduce this annotation rule, we write the latter as

$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x.M : \tau \to \sigma}$$

The horizontal bar still means if you have constructed the relations above the bar, then you can construct the relation beneath the bar.

Definition 2.1.19 *(Typability) We associate a term* M *with a simple type* τ *based on context* Γ*, written as* $\Gamma \vdash M : \tau$*. You can think of* $M : \tau$ *as a tuple* (M, τ) *and* $\Gamma \vdash M : \tau$ *is a relation defined on the tuple* $(\Gamma, (M, \tau))$*, or* $\vdash$*: is a relation defined on the triple* (Γ, M, τ) *of a context, a term and a simple type.*

1.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

This means without any premise (the blank on the line), you are allowed to define $\vdash$*: for triple* $(\Gamma \cup \{x : \tau\}, x, \tau)$ *The bar is often omitted.*

2.

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}$$

This means if you have defined the relation $\Gamma, x : \sigma \vdash M : \tau$*, then you can obtain a new relation* $\Gamma \vdash \lambda x.M : \sigma \to \tau$*. (introduction rule for* $\to$*)*

3.

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}$$

This means you can eliminate the arrow $\to$ *in the type by application.*

Example 2.1.20 1. For any simple type σ, $\vdash \lambda x.x : \sigma \to \sigma$

$$\frac{x : \sigma \vdash x : \sigma}{\vdash \lambda x.x : \sigma \to \sigma}$$

2. $\vdash \lambda x.\lambda y.x : \sigma \to \tau \to \sigma$

(according to currying, $\sigma \to \tau \to \sigma$ should be $\sigma \to (\tau \to \sigma)$)

$$\frac{\dfrac{x : \sigma, y : \tau \vdash x : \sigma}{x : \sigma \vdash \lambda y.x : \tau \to \sigma}}{\vdash \lambda x.\lambda y.x : \sigma \to \tau \to \sigma}$$

3. $\vdash \lambda f.\lambda g.\lambda a.f\ a\ (g\ a) : (A \to (B \to C)) \to (A \to B) \to (A \to C)$

Let $\Gamma = \{f : A \to B \to C, g : A \to B, a : A\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash f : A \to B \to C \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B \to C} \quad \dfrac{\Gamma \vdash g : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash g\ a : B}}{\Gamma \vdash f\ a\ (g\ a) : C}}{f : A \to B \to C, g : A \to B \vdash \lambda a.f\ a\ (g\ a) : A \to C}}{f : A \to B \to C \vdash \lambda g.\lambda a.f\ a\ (g\ a) : (A \to B) \to A \to C}}{\vdash \lambda f.\lambda g.\lambda a.f\ a\ (g\ a) : (A \to (B \to C)) \to (A \to B) \to (A \to C)}$$

Remark 2.1.21 Recall we define the context Γ to be a multiset in intuitionistic logic, but we do not need it here because we can use different variables before ':', e.g. $\{x : \varphi, y : \varphi\}$, but we require those variables be mutually distinct. ◊

You can see how the implicational tautologies are proved and interpreted in λ-calculus.

Lemma 2.1.22 *(Weakening) Suppose* $\Gamma = \{x_1 : \tau_1, x_2 : \tau_2, \cdots\}$ *and* $\Delta = \{y_1 : \sigma_1, y_2 : \sigma_2, \cdots\}$*, where* x_i*,* x_j*,* y_m*,* y_n *are mutually distinct. If* $\Gamma \vdash M : \varphi$*, then* $\Gamma \cup \Delta \vdash M : \varphi$*.*

Proof: By introduction on the construction of $\Gamma \vdash M : \varphi$. □

Remark 2.1.23 This type system is called Curry typing, where we only determine whether a λ-term is typable without assigning to a specific type. However, in the construction of $\vdash \lambda x.x : \sigma \to \sigma$, the last step must be

$$\dfrac{x : \sigma \vdash x : \sigma}{\vdash \lambda x.x : \sigma \to \sigma}$$

This strongly hints us to give type σ to the x after λ sign. In fact, there is a different but frequently used type system called Church typing where you have to specify a concrete type to a term you define. For example,

$$\dfrac{\Gamma, x : \sigma \vdash M : \sigma}{\Gamma \vdash \lambda x : \sigma.x : \sigma \to \sigma}$$

The term $\lambda x : \sigma.x$ specifies the type σ directly. Thus you are unable to say that it can has type $\tau \to \tau$ even if they have the same computational nature.

The benefit of Curry typing is that it gives you a lot of flexibility. For example, the only types that can be assigned to $\lambda x.x$ are those of the form $\varphi \to \varphi$ for some formula φ. So if you want to apply $\lambda x : x$ to a term $M : \sigma$, then $\lambda x.x$ must be typed $\sigma \to \sigma$.

On the other hand, Church typing is helpful if you want to find a term with a give type. If your aim is to find a term with type $(A \to B \to C) \to (A \to B) \to A \to C$, then you automatically defines a context $\{f : A \to B \to C, g : A \to B, x : A\}$. We tend to use Church typing because that will force us to find suitable λ-term constructions based on their types, e.g., you will not try to apply g to f.

Our target is to annotate a term with a type in order to avoid those problematic situations like $\lambda x.x\ x$. In fact, it will be a part of our language as in Chapter 3. So each term should always occur with a type notation. But do not let this restrict our expressibility. You can type $\lambda x.x$ with $\varphi \to \varphi$ as well as $\psi \to \psi$. That is why Curry typing is preferred in this thesis and why we defined another judgement relation $\vdash$ in meta-language. The problem is that we do not have free variable in the term $\lambda x.x$, but we have somehow a 'free variable' φ in the type. The arbitrariness of φ is in our meta language, not in the theory itself, i.e., we do not have quantifier over type variables. To solve this problem, we have to lift the order of our language. You will see this in Chapter 3.

Based on this discussion, we do not have to always find a context and write down a typing tree explicitly. Instead we can vaguely say 'I want to apply $f : \varphi \to \psi$ to a term $\lambda x.x$'. Since the term $\lambda x.x$ must be given a type of form $\sigma \to \sigma$ for some σ, then the φ in the type of f should be $\sigma \to \sigma$. As long as this process will not cause any ambiguity, we can free ourselves from a too long proof. Sometimes, to emphasize the type of an element, we also write a

Church style term like $\lambda x : \varphi.M$. $\diamond$

Yet there is one more problem, whether $\rightarrow_\beta$ will cause inconsistency with typed terms. The answer is certainly negative. The reduction always preserves typing.

Lemma 2.1.24 *If* $\Gamma, x : \varphi \vdash P : \psi$ *and* $\Gamma \vdash Q : \varphi$, *then* $\Gamma \vdash P[x := Q] : \psi$.

Proof: Recall the 3 construction rules for typability (2.1.19). Let's do induction on the construction of $\Gamma, x : \varphi \vdash P : \psi$.

1. If P is simply a variable x, then it must be obtained by

$$\frac{}{\Gamma, x : \varphi \vdash x : \varphi}$$

   Thus $P[x := Q] = x[x := Q] = Q$ and certainly $\Gamma \vdash P[x := Q] : \psi$.

2. If $P = \lambda y.R$ for some R, then the the last step of $\Gamma, x : \varphi \vdash P : \psi$ must be

$$\frac{\Gamma, y : \sigma, x : \varphi \vdash R : \tau}{\Gamma, x : \varphi \vdash \lambda y.R : \sigma \rightarrow \tau}$$

   for some $\psi = \sigma \rightarrow \tau$. Note that if $\Gamma, y : \sigma, x : \varphi$ is used as a context, then $y : \delta$ is not an element in Γ for any type δ. So we can do the weakening that $\Gamma \vdash Q : \varphi$ implies $\Gamma, y : \sigma \vdash Q : \varphi$. Also note that $y \in \mathrm{FV}(Q)$ means $P[x := Q] = (\lambda y.R)[x := Q] = \lambda y.(R[x := Q])$. Thus by inductive hypothesis, $\Gamma, y : \sigma \vdash R[x := Q] : \tau$. So $\Gamma \vdash \lambda y.(R[x := Q]) : \sigma \rightarrow \tau$, which exactly $\Gamma \vdash P[x := Q]$.

3. Suppose $P = R\ S$ for some R and S. Then the last construction of $\Gamma, x : \varphi \vdash P : \psi$ must be

$$\frac{\Gamma, x : \varphi \vdash R : \sigma \rightarrow \psi \quad \Gamma, x : \varphi \vdash S : \sigma}{\Gamma, x : \varphi \vdash R\ S : \psi}$$

for some type σ, where $P = R\ S$. By inductive hypothesis, $\Gamma \vdash R[x := Q] : \sigma \to \psi$ and $\Gamma \vdash S[x := Q] : \sigma$. Thus

$$\frac{\Gamma \vdash R[x := Q] : \sigma \to \psi \quad \Gamma \vdash S[x := Q] : \sigma}{\Gamma \vdash (R[x := Q])\ (S[x := Q]) : \psi}$$

Since $P[x := Q] = (R\ S)[x := Q] = (R[x := Q])\ (S[x := Q])$, we can conclude $\Gamma \vdash P[x := Q] : \psi$.

Theorem 2.1.25 *If $M \to_\beta N$ and $\Gamma \vdash M : \varphi$, then $\Gamma \vdash N : \varphi$.*

Proof: Let's do induction on the construction of $\to_\beta$.

- **Base Step** If $(\lambda x.P)\ Q \to_\beta P[x := Q]$, then we do induction on the construction of $\Gamma \vdash (\lambda x.P)\ Q : \varphi$. The last several steps of it should be

$$\frac{\dfrac{\Gamma, x : \psi \vdash P : \varphi}{\Gamma \vdash \lambda x.P : \psi \to \varphi} \quad \Gamma \vdash Q : \psi}{\Gamma \vdash (\lambda x.P)\ Q : \varphi}$$

  for some simple type ψ. By preceding lemma, $\Gamma, x : \varphi \vdash P[x := Q] : \varphi$. Since x is no longer used in $P[x := Q]$, we can conclude $\Gamma \vdash P[x := Q] : \psi$

- **Inductive Step**

  1. $M \to_\beta N$ is obtained by 'if $P \to_\beta Q$, then $\lambda x.P \to_\beta \lambda x.Q$', where $M = \lambda x.P$ and $N = \lambda x.Q$. The last step of $\Gamma \vdash M : \varphi$ is

$$\frac{\Gamma, x : \sigma \vdash P : \tau}{\Gamma \vdash \lambda x.P : \sigma \to \tau}$$

     where $\varphi = \sigma \to \tau$. By inductive hypothesis, $\Gamma, x : \sigma \vdash Q : \tau$. Thus $\Gamma \vdash \lambda x.Q : \sigma \to \tau$.

  2. $M \to_\beta N$ is obtained by 'if $P \to_\beta Q$ then $L\ P \to_\beta L\ Q$', where

$L\ P = M$ and $L\ Q = N$. The last step of $\Gamma \vdash M : \varphi$ is

$$\frac{\Gamma \vdash L : \sigma \to \varphi \quad \Gamma \vdash P : \sigma}{\Gamma \vdash (L\ P) : \varphi}$$

for some type σ. By inductive hypothesis, $\Gamma \vdash Q : \sigma$. Thus $\Gamma \vdash (L\ Q) : \varphi$.

3. The case that $M \to_\beta N$ is obtained by 'if $P \to_\beta Q$ then $P\ L \to_\beta Q\ L$' is similar. □

2.2 Curry-Howard Correspondence

We have developed a formal language with computational meaning hoping to use it as the evidence of an intuitionistic proof as mentioned before. Naturally, an implication $A \to B$ is analogous to a function from A to B. From the definition of the $\vdash$: relation, you can find this analogy is described by the ':' notation. For a context $\Gamma = \{x_1 : \tau_1, x_2 : \tau_2, \cdots\}$, if we can prove $\Gamma \vdash M : \sigma$, then $\{\tau_1, \tau_2, \cdots\} \vdash \sigma$ in intuitionistic logic, where each simply type (τ_i and σ) is understood as implication(formula) in logic.

Conversely, it is not hard to find that if $\{\tau_1, \tau_2, \cdots\} \vdash \sigma$, then we can find suitable $x_1, x_2, \cdots$ and M such that $\{x_1 : \tau_1, x_2 : \tau_2, \cdots\} \vdash M : \sigma$,

This is the original Curry-Howard Correspondence, which only works for implicational fragment of intuitionistic logic. To fully support intuitionistic logic, we have to expand our language.

Definition 2.2.1 *(Simple Types) We use logic formulas as* **simple types**, *i.e.,*

$$\mathscr{F} ::\equiv V \mid \bot \mid \mathscr{F} \to \mathscr{F} \mid \mathscr{F} \wedge \mathscr{F} \mid \mathscr{F} \vee \mathscr{F}$$

The above notation is called Backus normal form (BNF). It is a convenient tool to define a concept recursively. As before, this BNF tells us that the simple types $\mathscr{F}$ are recursively defined as follows.

1. *All elements in V are simple types. Note I use letter V for variables in simple types and U for variables in λ-terms.*

2. *The absurdity $\perp$ is a simple type.*

3. *If φ and ψ are simple types, then so is $\varphi \to \psi$.*

4. *If φ and ψ are simple types, then so is $\varphi \wedge \psi$.*

5. *If φ and ψ are simple types, then so is $\varphi \vee \psi$.*

And $\mathscr{F}$ is the set of all simple types, equivalently, formulas. BNF helps us to write a recursive definition easily. Thus I adopt this pattern for this section.

Then let's think about how the inference rules of intuitionistic logic correspond to the constructions of λ-terms. Recall we defined the implication in intuitionistic logic by manipulating formulas, i.e., by introduction and elimination rules (1.2.7). In intuitionistic logic, we have the introduction rule

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}$$

Now it is similar to the abstract

$$\frac{\Gamma, x : \varphi \vdash M : \psi}{\Gamma \vdash \lambda x.M : \varphi \to \psi}$$

Also in intuitionistic logic, we have the elimination rule

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \varphi \to \psi}{\Gamma \vdash \psi}$$

This is analogous to application

$$\frac{\Gamma \vdash M : \varphi \to \psi \quad \Gamma \vdash N : \varphi}{\Gamma \vdash M\ N : \psi}$$

Besides, we also define a computation $\to_\beta$ if an abstraction is applied to another

term, i.e., $(\lambda x.M)\ N \to_\beta M[x := N]$. This sometimes means a detour in the proof. For example, let $\Gamma = \{A \to B, B \to C, D \to A, D\}$. We can prove $\Gamma \vdash C$ by

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, A \vdash A \quad \Gamma, A \vdash A \to B}{\Gamma, A \vdash B} \qquad \Gamma, A \vdash B \to C}{\Gamma, A \vdash C}}{\Gamma \vdash A \to C} \qquad \dfrac{\Gamma \vdash D \quad \Gamma \vdash D \to A}{\Gamma \vdash A}}{\Gamma \vdash C}$$

In this proof, we first introduce a premise A and prove C from it, so this means $\Gamma \vdash A \to C$. Then we prove $\Gamma \vdash A$, so $\Gamma \vdash B$. Obviously, we do not have to introduce the A in the premise since we can prove it directly.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash D \to A \quad \Gamma \vdash D}{\Gamma \vdash A} \qquad \Gamma \vdash A \to B}{\Gamma \vdash B} \qquad \Gamma \vdash B \to C}{\Gamma \vdash C}$$

Let's discuss the same situation in λ-calculus. Let $\Gamma = \{f : A \to B, g : B \to C, h : D \to A, d : D\}$ as in simply typed Λ-calculus. The detour can be considered as follows (with some steps skipped).

$$\dfrac{\dfrac{\dfrac{\Gamma, x : A \vdash f\ x : B \quad \Gamma, x : A \vdash g : B \to C}{\Gamma, x : A \vdash g\ (f\ x) : C}}{\Gamma \vdash \lambda x.g\ (f\ x) : A \to C} \qquad \dfrac{\Gamma \vdash d : D \quad \Gamma \vdash h : D \to A}{\Gamma \vdash h\ d : A}}{\Gamma \vdash (\lambda x.g\ (f\ x))\ (h\ d) : C}$$

Note that $\lambda x.g\ (f\ x))\ (h\ d) \to_\beta g\ (f\ (h\ d))$. Thus we can directly prove $\Gamma \vdash g\ (f\ (h\ d)) : C$.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash h : D \to A \quad \Gamma \vdash d : D}{\Gamma \vdash h\ d : A} \qquad \Gamma \vdash f : A \to B}{\Gamma \vdash f\ (h\ d) : B} \qquad \Gamma \vdash g : B \to C}{\Gamma \vdash g\ (f\ (h\ d)) : C}$$

If we know $\Gamma \vdash \lambda x.M : \varphi \to \psi$, then by our definition of the $\vdash$: relation, we must prove that $\Gamma, x : \varphi \vdash M : \psi$. Further, if $\Gamma \vdash N : \varphi$, then we can substitute

N for each occurrence of x during the proof. This is how the β-reduction cuts the detour in a proof.

Analogously, we informally introduce terms for other logic connectives.

- For conjunction $\wedge$(and), we use pairs $\langle , \rangle$ for introduction rule,

$$\frac{\Gamma \vdash M : \varphi \quad \Gamma \vdash N : \psi}{\langle M, N \rangle : \varphi \wedge \psi}$$

  and we use projections $\mathsf{pr}_1, \mathsf{pr}_2$ for elimination rules in λ-calculus

$$\frac{\Gamma \vdash M : \varphi \wedge \psi}{\Gamma \vdash \mathsf{pr}_1(M) : \varphi}$$

$$\frac{\Gamma \vdash M : \varphi \wedge \psi}{\Gamma \vdash \mathsf{pr}_2(M) : \psi}$$

  with computation $\rightarrow_\beta$

$$\mathsf{pr}_1(\langle M, N \rangle) \rightarrow_\beta M$$

$$\mathsf{pr}_2(\langle M, N \rangle) \rightarrow_\beta N$$

  for any λ-term M,N.

- The disjunction $\vee$ (or) behavior like a coproduct (disjoint union). The coproduct is a bit like 'natural numbers consist of odd numbers and even numbers'. This seems to be a union of the two sets. But the two sets are equinumerous, so if we want to use a set to describe 'all odd numbers and even numbers', we have to use an explicit labelling for those elements. As in set theory, the disjoint union of A and B is $A + B = \{(a, 0) : a \in A\} \cup \{(b, 1) : b \in B\}$. The 0 and 1 automatically distinguish elements in $A + B$ even if the two sets are identical.

  Since every elements in $A + B$ is either an element in A or an element in B, a function $f : A + B \rightarrow C$ is always determined by a function $A \rightarrow C$

and a function $B \to C$. For example, the identity function on $A + B$ is determined by the two *natural injections*:

$$\begin{aligned} \mathsf{inj}_1 : A &\to A + B \\ a &\mapsto (a, 0) \end{aligned}$$

$$\begin{aligned} \mathsf{inj}_2 : B &\to A + B \\ b &\mapsto (b, 1) \end{aligned}$$

On the other hand, given the two functions $f_1 : A \to C$ and $f_2 : B \to C$, you automatically has a function $f : A + B \to C$:

$$f(x) = \begin{cases} f_1(a), & x = (a, 0) \\ f_2(b), & x = (b, 1) \end{cases}$$

Equivalently, you can describe f by inj_1, inj_2

$$f(\mathsf{inj}_1(a)) = f_1(a)$$

$$f(\mathsf{inj}_2(b)) = f_2(b)$$

As in category theory, we can use inj_1 and inj_2 as the labelling. We adopt the same pattern for disjunction in our language. An evidence of $\varphi \vee \psi$ should be either an evidence of φ or an evidence of ψ. If it comes from the left, we use this rule:

$$\frac{\Gamma \vdash M : \varphi}{\mathsf{inj}_1(M) : \varphi \vee \psi}$$

If it comes from the right, we use

$$\frac{\Gamma \vdash N : \psi}{\mathsf{inj}_2(N) : \varphi \vee \psi}$$

That defines the introduction rule of $\vee$.

The elimination rule is quite similar to the category theoretical universal property. To prove σ assuming $\varphi \vee \psi$, we use *proof by case analysis*. we assert σ given φ, and then assert σ given ψ. Thus we define a new syntax:

$$\frac{\Gamma \vdash P : \varphi \to \sigma \quad \Gamma \vdash Q : \psi \to \sigma}{\Gamma \vdash \mathsf{case}(P, Q) : \varphi \vee \psi \to \sigma}$$

Then if we apply the term $\mathsf{case}(P, Q)$ to $\mathsf{inj}_1(L)$ and $\mathsf{inj}_2(N)$, computations should take place:

$$\mathsf{case}(P, Q)\ \mathsf{inj}_1(L) \to_\beta P\ L$$

$$\mathsf{case}(P, Q)\ \mathsf{inj}_2(N) \to_\beta Q\ N$$

Note this is a bit different from our elimination rule of $\vee$ in intuitionistic logic. Actually we did not 'eliminate' the $\vee$ by this syntax, so in [23], the author chooses to apply this rule only if M, P, Q are all given:

$$\frac{\Gamma \vdash M : \varphi \vee \psi \quad \Gamma \vdash P : \varphi \to \sigma \quad \Gamma \vdash Q : \psi \to \sigma}{\Gamma \vdash \mathsf{case}(M, P, Q) : \sigma}$$

In practice, we often write an $f : \varphi \vee \psi \to \sigma$ defined by $\mathsf{case}(\lambda x.R, \lambda y.S)$ as

$$f\ \mathsf{inj}_1(x) = R$$

$$f\ \mathsf{inj}_2(y) = S$$

where R, S are terms of type σ. This style, called (*pattern matching*), is

more intuitional. So in [23], the author actually uses a more complicated system (note R, S may depend on x, y):

$$\frac{\Gamma \vdash M : \varphi \vee \psi \quad \Gamma, x : \varphi \vdash R : \sigma \quad \Gamma, y : \psi \vdash S : \sigma}{\Gamma \vdash \mathsf{case}(M, x.R, y.S) : \sigma}$$

This keeps compliance with the elimination rule in intuitionistic logic.

- For absurdity $\perp$, there is no way to find a proof. We only have the elimination rule. For arbitrary φ, we define a new syntax:

$$\frac{\Gamma \vdash M : \perp}{\Gamma \vdash \epsilon(M) : \varphi}$$

Based on the above discussion, the formal definition is the following.

Definition 2.2.2 *(Extended λ-terms)*

$$\begin{array}{rcl} \Lambda & ::= & U \\ & | & \lambda U.\Lambda \mid \Lambda\ \Lambda \\ & | & \langle \Lambda, \Lambda \rangle \mid \mathsf{pr}_1(\Lambda) \mid \mathsf{pr}_2(\Lambda) \\ & | & \mathsf{inj}_1(\Lambda) \mid \mathsf{inj}_2(\Lambda) \mid \mathsf{case}(\Lambda, \Lambda) \\ & | & \epsilon(\Lambda) \end{array}$$

Definition 2.2.3 *(Extended Typability)*

- *Axiom:*

$$\frac{}{\Gamma, x : \varphi \vdash x : \varphi}$$

- *$\rightarrow$-Introduction:*

$$\frac{\Gamma, x : \varphi \vdash M : \psi}{\Gamma \vdash \lambda x.M : \varphi \rightarrow \psi}$$

$\to$-*Elimination:*

$$\frac{\Gamma \vdash M : \varphi \to \psi \quad \Gamma \vdash N : \varphi}{\Gamma \vdash M\ N : \psi}$$

- $\wedge$-*Introduction:*

$$\frac{\Gamma \vdash M : \varphi \quad \Gamma \vdash N : \psi}{\Gamma \vdash \langle M, N \rangle : \varphi \wedge \psi}$$

$\wedge$-*Elimination:*

$$\frac{\Gamma \vdash M : \varphi \wedge \psi}{\Gamma \vdash \mathsf{pr}_1(M) : \varphi}$$

$$\frac{\Gamma \vdash M : \varphi \wedge \psi}{\Gamma \vdash \mathsf{pr}_2(M) : \psi}$$

- $\vee$-*Introduction:*

$$\frac{\Gamma \vdash M : \varphi}{\Gamma \vdash \mathsf{inj}_1(M) : \varphi \vee \psi}$$

$$\frac{\Gamma \vdash N : \psi}{\Gamma \vdash \mathsf{inj}_2(N) : \varphi \vee \psi}$$

$\vee$-*Elimination:*

$$\frac{\Gamma \vdash P : \varphi \to \sigma \quad \Gamma \vdash Q : \psi \to \sigma}{\Gamma \vdash \mathsf{case}(P, Q) : \varphi \vee \psi \to \sigma}$$

- $\bot$-*Elimination:*

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash \epsilon(M) : \varphi}$$

Definition 2.2.4 *(Extended β-reduction) As before, the relation is defined by*

$$\textit{base steps}\begin{cases} (\lambda x.M)\ N \to_\beta M[x := N] \\ \\ \mathsf{pr}_1(\langle M, N\rangle) \to_\beta M \\ \mathsf{pr}_2(\langle M, N\rangle) \to_\beta N \\ \\ \mathsf{case}(P, Q)\ \mathsf{inj}_1(L) \to_\beta P\ L \\ \mathsf{case}(P, Q)\ \mathsf{inj}_2(N) \to_\beta Q\ N \end{cases}$$

and

$$
\textit{recursive steps}\begin{cases}
\textit{If } M \to_\beta N, \textit{ then } \lambda x.M \to_\beta \lambda x.N. \\
\textit{If } M \to_\beta N, \textit{ then } P\ M \to_\beta P\ N. \\
\textit{If } M \to_\beta N, \textit{ then } M\ Q \to_\beta N\ Q. \\
\\
\textit{If } M \to_\beta N, \textit{ then } \langle P, M\rangle \to_\beta \langle P, N\rangle. \\
\textit{If } M \to_\beta N, \textit{ then } \langle M, Q\rangle \to_\beta \langle N, Q\rangle. \\
\textit{If } M \to_\beta N, \textit{ then } \mathsf{pr}_1(M) \to_\beta \mathsf{pr}_1(N). \\
\textit{If } M \to_\beta N, \textit{ then } \mathsf{pr}_2(M) \to_\beta \mathsf{pr}_2(N). \\
\\
\textit{If } M \to_\beta N, \textit{ then } \mathsf{inj}_1(M) \to_\beta \mathsf{inj}_1(N). \\
\textit{If } M \to_\beta N, \textit{ then } \mathsf{inj}_2(M) \to_\beta \mathsf{inj}_2(N). \\
\textit{If } M \to_\beta N, \textit{ then } \mathsf{case}(P, M) \to_\beta \mathsf{case}(P, N). \\
\textit{If } M \to_\beta N, \textit{ then } \mathsf{case}(M, Q) \to_\beta \mathsf{case}(N, Q). \\
\\
\textit{If } M \to_\beta N, \textit{ then } \epsilon(M) \to_\beta \epsilon(N).
\end{cases}
$$

$\twoheadrightarrow_\beta$ *and* $=_\beta$ *are extended as before.*

Theorem 2.2.5 *In the extended* λ*-calculus, if* $M \to_\beta N$ *and* $\Gamma \vdash M : \varphi$*, then* $\Gamma \vdash N : \varphi$.

Proof: Left to the readers. □

Example 2.2.6 Recall in 1.2.13, we proved that double negation elimination ($\neg\neg\varphi \to \varphi$) is equivalent to law of excluded middle ($\varphi \vee \neg\varphi$). Let's see how to prove $\neg\neg\varphi \to \varphi$ from $lem : \varphi \vee \neg\varphi$ with λ-terms. ⋄

Proof: Recall that $\neg\tau$ is alias to $\tau \to \bot$. Let $\Gamma = \{lem : \varphi \vee \neg\varphi, x : \neg\neg\varphi\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, y : \varphi \vdash y : \varphi}}{\Gamma \vdash \lambda y.y : \varphi \to \varphi} \quad \dfrac{\dfrac{\dfrac{}{\Gamma, z : \neg\varphi \vdash x\ z : \bot}}{\Gamma, z : \neg\varphi \vdash \epsilon(x\ z) : \varphi}}{\Gamma \vdash \lambda z.\epsilon(x\ z) : \neg\varphi \to \varphi}}{\Gamma \vdash \mathsf{case}(\lambda y.y, \lambda z.\epsilon(x\ z)) : \varphi \vee \neg\varphi \to \varphi} \quad \Gamma \vdash lem : \varphi \vee \neg\varphi}{\Gamma \vdash \mathsf{case}(\lambda y.y, \lambda z.\epsilon(x\ z))\ lem : \varphi}}{lem : \varphi \to \neg\varphi \vdash \lambda x.\mathsf{case}(\lambda y.y, \lambda z.\epsilon(x\ z))\ lem : \neg\neg\varphi \to \varphi}$$

Remark 2.2.7 Let's think about

$$lem : \varphi \vee \neg\varphi, x : \neg\neg\varphi \vdash \mathsf{case}(\lambda y.y, \lambda z.\epsilon(x\ z))\ lem : \varphi$$

We obtain φ by applying $\mathsf{case}(\lambda y.y, \lambda z.\epsilon(x\ z))$ to $lem : \varphi \vee \neg\varphi$. The case notation tells you that when $y : \varphi$, we can prove $y : \varphi$ directly. Otherwise, when $z : \neg\varphi$, we apply $x : \neg\varphi \to \bot$ to obtain absurdity $\bot$. Then we apply ex falso quodlibet(ϵ) to prove φ. ♢

Example 2.2.8 There exists a term M such that $\vdash M : \neg\neg(\varphi \vee \neg\varphi)$ for any simple type φ. ♢

Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f : \neg(\varphi \vee \neg\varphi), x : \varphi \vdash \mathsf{inj}_1(x) : \varphi \vee \neg\varphi}}{f : \neg(\varphi \vee \neg\varphi), x : \varphi \vdash f\ \mathsf{inj}_1(x) : \bot}}{f : \neg(\varphi \vee \neg\varphi) \vdash \lambda x.f\ \mathsf{inj}_1(x) : \varphi \to \bot}}{f : \neg(\varphi \vee \neg\varphi) \vdash \mathsf{inj}_2(\lambda x.f\ \mathsf{inj}_1(x)) : \varphi \vee \neg\varphi}}{f : \neg(\varphi \vee \neg\varphi) \vdash f\ \mathsf{inj}_2(\lambda x.f\ \mathsf{inj}_1(x)) : \bot}}{\vdash \lambda f.f\ \mathsf{inj}_2(\lambda x.f\ \mathsf{inj}_1(x)) : \neg\neg(\varphi \vee \neg\varphi)}$$

Remark 2.2.9 I skip some constructions as before, but the whole history of the proof tree is preserved by the λ-term. Think about the following typability relation:

$$f : \neg(\varphi \vee \neg\varphi), x : \varphi \vdash f\ \mathsf{inj}_1(x) : \bot$$

From the λ-term, you know the last step must be an elimination of implication

(MP), and the premise is obtained by affirming the left of disjunction, i.e.,

$$\dfrac{\neg(\varphi \vee \neg\varphi), \varphi \vdash (\varphi \vee \neg\varphi) \to \bot \quad \dfrac{\varphi \vee \neg\varphi \vdash \varphi}{\neg(\varphi \vee \neg\varphi), \varphi \vdash \varphi \vee \neg\varphi}}{\neg(\varphi \vee \neg\varphi), \varphi \vdash \bot}$$

Thus in practice, in order to communicate the result, it suffices for us to just write down a typable λ-term instead of writing the whole tree. $\diamond$

Remark 2.2.10 Let $\mathsf{lem} = \lambda f.f\ \mathsf{inj}_2(\lambda x.f\ \mathsf{inj}_1(x))$. In example 2.2.8, we have seen that $\vdash \mathsf{lem} : \neg\neg(\varphi \vee \neg\varphi)$ for any simple type φ. Note that there is no free variable in lem, so we can give it a different type as mentioned in 2.1.23. If $\mathsf{lem} = \lambda f.f\ \mathsf{inj}_2(\lambda x.f\ \mathsf{inj}_1(x))$ is give a simple type a, then a must be $b \to c$ for some b, c, and the last construction must be

$$\dfrac{f : b \vdash f\ \mathsf{inj}_2(\lambda x.f\ \mathsf{inj}_1(x)) : c}{\vdash \lambda f.f\ \mathsf{inj}_2(\lambda x.f\ \mathsf{inj}_1(x)) : b \to c}$$

Since $f\ \mathsf{inj}_2(\lambda x.f\ \mathsf{inj}_1(x))$ is an application, f must be typed $d \to c$ for some type d and $d \to c$ and the last construction of $f : b \vdash f\ \mathsf{inj}_2(\lambda x.f\ \mathsf{inj}_1(x)) : c$ must be

$$\dfrac{f : d \to c \vdash f : d \to c \quad f : d \to c \vdash \mathsf{inj}_2(\lambda x.f\ \mathsf{inj}_1(x)) : d}{f : d \to c \vdash f\ \mathsf{inj}_2(\lambda x.f\ \mathsf{inj}_1(x)) : c}$$

Repeatedly, the typing of $\mathsf{inj}_2(\lambda x.f\ \mathsf{inj}_1(x))$ forces d to be $m \vee n$ for some m,n and $f : m \vee n \to c$ and $\lambda x.f\ \mathsf{inj}_1(x) : n$. Again n should be $p \to q$ for some p, q and $\lambda x.f\ \mathsf{inj}_1(x) : n$ is obtained from

$$\dfrac{f : m \vee n \to c, x : p \vdash f\ \mathsf{inj}_1(x) : q}{\lambda x.f\ \mathsf{inj}_1(x) : p \to q}$$

The typing $f : m \vee n \to c$ forces $\mathsf{inj}_1(x) : m \vee n$ and $q = c$. Thus we should have $p = m$. Reordering the above process, the type assigned to lem must be of the

form

$$\begin{aligned} a &= b \to c \\ &= (d \to c) \to c \\ &= ((m \vee n) \to c) \to c \\ &= ((m \vee (p \to q)) \to c) \to c \\ &= ((m \vee (m \to c)) \to c) \to c \end{aligned}$$

When $m = \varphi$ and $c = \bot$, this is $\neg\neg(\varphi \vee \neg\varphi)$. ♢

Remark 2.2.11 You may think pr_1 itself as a function of type $\varphi \wedge \psi \to \varphi$. For example, we can add a typability rule

$$\frac{}{\Gamma \vdash \mathsf{pr}_1 : \varphi \wedge \psi \to \varphi}$$

Note this means you have to define pr_1 as a λ-term (not $\mathsf{pr}_1(\Lambda)$). This is only a choice of taste. You can do the same for pr_2, $\langle , \rangle$, inj_1,inj_2,case. I choose the current style only to keep pace with our definition of intuitionistic logic. In 3, you will see this style is more convenient.

Or you can just think of the usage of pr_1 as a λ-term as an abbreviation of $\lambda x.\mathsf{pr}_1(x)$ because

$$\frac{x : \varphi \wedge \psi \vdash \mathsf{pr}_1(x) : \varphi}{\vdash \lambda x.\mathsf{pr}_1(x) : \varphi \wedge \psi \to \varphi}$$

This raises one more question: whether $\lambda x.f\ x = f$. Up to now we only define the computation if an elimination rule is applied to an introduction rule, e.g., $(\lambda x.M)\ x =_\beta M$. The above question asks what if we apply an introduction rule to an elimination rule. This kind of relation is called η-equality $=_\eta$. It is more natural for us to think this problem of other connectives. For example, $\langle \mathsf{pr}_1(M), \mathsf{pr}_2(M) \rangle = M$ as in set theory. However, in my theory, I choose to

define the equality only for $\lambda x.f\ x =_\eta f$ if $x \notin \mathrm{FV}(f)$, because the equality for other connectives raise some homotopical structures [21].

You can choose whether to define the η-equality based on your need. Sometimes it is convenient to define the equality for conjunction and disjunction if you decide not to step too much into the intensional structure of a λ-term as a proof [23].

Finally, we come to our aim of this section.

Definition 2.2.12 *(Range of a context) Suppose* $\Gamma = \{x_1 : \tau_1, x_2 : \tau_2, \cdots\}$. *The range of it is*

$$|\Gamma| = \{\tau_1, \tau_2, \cdots\}.$$

Theorem 2.2.13 *(Curry-Howard Correspondence)*

1. *If* $\Gamma \vdash M : \sigma$ *in* λ*-calculus, then* $|\Gamma| \vdash \sigma$ *in intuitionistic logic.*
2. *If* $\{\tau_1, \tau_2, \cdots\} \vdash \sigma$, *then there exists a* λ*-term* M *such that* $\{x_1 : \tau_1, x_2 : \tau_2, \cdots\} \vdash M : \sigma$

Proof: Part 1 is obvious according to our above discussion except case. We perform induction on the construction of typability relation. Suppose the last step is

$$\frac{\Gamma \vdash P : \varphi \to \sigma \quad \Gamma \vdash Q : \psi \to \sigma}{\Gamma \vdash \mathsf{case}(P, Q) : \varphi \vee \psi \to \sigma}$$

By inductive hypothesis, we have $|\Gamma| \vdash \varphi \to \sigma$, and $|\Gamma| \vdash \psi \to \sigma$. Thus we certainly have

$$\dfrac{\dfrac{|\Gamma|, \varphi \vee \psi \vdash \varphi \vee \psi \quad \dfrac{|\Gamma|, \varphi \vee \psi \vdash \varphi \to \sigma}{|\Gamma|, \varphi \vee \psi, \varphi \vdash \sigma} \quad \dfrac{|\Gamma|, \varphi \vee \psi \vdash \psi \to \sigma}{|\Gamma|, \varphi \vee \psi, \psi \vdash \sigma}}{|\Gamma|, \varphi \vee \psi \vdash \sigma}}{|\Gamma| \vdash \varphi \vee \psi \to \sigma}$$

For Part 2, let $\Gamma = \{x_1 : \tau_1, x_2 : \tau_2, \cdots\}$. Then we perform introduction on the construction of $|\Gamma| \vdash \sigma$ (the height of the proof tree).

- Axiom:

  In this case, $\sigma = \tau_i$ for some τ_i Thus $\Gamma \vdash x_i : \tau_i$

- Implication($\to$):

  Suppose the last step of $|\Gamma| \vdash \sigma$ is an introduction

$$\frac{|\Gamma|, \varphi \vdash \psi}{|\Gamma| \vdash \varphi \to \psi}$$

  where $\sigma = \varphi \to \psi$. By inductive hypothesis, we can find an M such that $\Gamma, x : \varphi \vdash M : \psi$. Thus

$$\frac{\Gamma, x : \varphi \vdash M : \psi}{\Gamma \vdash \lambda x.M : \varphi \to \psi}$$

  If the last step is an elimination

$$\frac{|\Gamma| \vdash \varphi \to \sigma \quad |\Gamma| \vdash \varphi}{|\Gamma| \vdash \sigma},$$

  then find M, N such that $\Gamma \vdash M : \varphi \to \sigma$ and $\Gamma \vdash N : \varphi$. Thus

$$\frac{\Gamma \vdash M : \varphi \to \sigma \quad \Gamma \vdash N : \varphi}{\Gamma \vdash M\ N : \sigma}$$

- Conjunction($\wedge$):

  Suppose

$$\frac{|\Gamma| \vdash \varphi \quad |\Gamma| \vdash \psi}{|\Gamma| \vdash \varphi \wedge \psi}$$

  Find M, N such that $\Gamma \vdash M : \varphi$, $\Gamma \vdash N : \varphi$. Then

$$\frac{\Gamma \vdash M : \varphi \quad \Gamma \vdash N : \psi}{\Gamma \vdash \langle M, N \rangle : \varphi \wedge \psi}$$

  Suppose

$$\frac{|\Gamma| \vdash \varphi \wedge \psi}{|\Gamma| \vdash \varphi}$$

Find M such that $\Gamma \vdash M : \varphi \wedge \psi$. Then

$$\frac{\Gamma \vdash M : \varphi \wedge \psi}{\Gamma \vdash \mathsf{pr}_1(M) : \varphi}$$

- Disjunction($\vee$):

  Suppose

  $$\frac{|\Gamma| \vdash \varphi}{|\Gamma| \vdash \varphi \vee \psi}$$

  Find M such that $\Gamma \vdash M : \varphi$. Then

  $$\frac{\Gamma \vdash M : \varphi}{\Gamma \vdash \mathsf{inj}_1(M) : \varphi \vee \psi}$$

  Suppose

  $$\frac{|\Gamma| \vdash \varphi \vee \psi \quad |\Gamma|, \varphi \vdash \sigma \quad |\Gamma|, \psi \vdash \sigma}{|\Gamma| \vdash \sigma}$$

  Find M, R, S such that $\Gamma \vdash M : \varphi \vee \psi$, $\Gamma, x : \varphi \vdash R : \sigma$ and $\Gamma, y : \psi \vdash S : \sigma$.

  Then

  $$\frac{\Gamma \vdash M : \varphi \vee \psi \qquad \dfrac{\dfrac{\Gamma, x : \varphi \vdash R : \sigma}{\Gamma \vdash \lambda x.R : \varphi \to \sigma} \quad \dfrac{\Gamma, y : \psi \vdash S : \sigma}{\Gamma \vdash \lambda y.S : \psi \to \sigma}}{\Gamma \vdash \mathsf{case}(\lambda x.R, \lambda y.S) : \varphi \vee \psi \to \sigma}}{\Gamma \vdash \mathsf{case}(\lambda x.R, \lambda y.S)\ M : \sigma}$$

- Absurdity($\perp$):

  Suppose

  $$\frac{|\Gamma| \vdash \perp}{|\Gamma| \vdash \sigma}$$

  Find an M such that $\Gamma \vdash M : \perp$. Then

  $$\frac{\Gamma \vdash M : \perp}{\Gamma \vdash \epsilon(M) : \sigma}$$

2.3 Examples of Curry Howard Correspondence

We have developed a type system for λ-calculus as a language for intuitionistic logic. But this system seems to be cheating, because we only write down what we need for the constructions of intuitionistic logic. Let's think about this very trival example, $\vdash \varphi \to \varphi \to \varphi$ (recall this is $\varphi \to (\varphi \to \varphi)$ due to currying). We have two 'different' proof trees:

$$\dfrac{\dfrac{\varphi, \varphi \vdash \varphi}{\varphi \vdash \varphi \to \varphi}}{\vdash \varphi \to \varphi \to \varphi}$$

and

$$\dfrac{\dfrac{\varphi, \varphi \vdash \varphi}{\varphi \vdash \varphi \to \varphi}}{\vdash \varphi \to \varphi \to \varphi}$$

They seem identical but are different in the sense that my proof ideas are different, which is revealed by the Curry-Howard Correspondence.

$$\dfrac{\dfrac{x : \varphi, y : \varphi \vdash x : \varphi}{x : \varphi \vdash \lambda y.x : \varphi \to \varphi}}{\vdash \lambda x.\lambda y.x : \varphi \to \varphi \to \varphi}$$

$$\dfrac{\dfrac{x : \varphi, y : \varphi \vdash y : \varphi}{x : \varphi \vdash \lambda y.y : \varphi \to \varphi}}{\vdash \lambda x.\lambda y.y : \varphi \to \varphi \to \varphi}$$

In other words, the proof of $\varphi \to \varphi \to \varphi$ may come from $\varphi \to \psi \to \varphi$ or from $\psi \to \varphi \to \varphi$. When ψ is φ, you do not know the mentality behind the proof tree in intuitionistic logic. Thus Curry-Howard Correspondence is helpful in the sense of comparing proofs.

Another interesting example is that a syntax can be helpful to shorten our proof. Let's review the proof of Glivenko's theorem (1.3.8) with Curry-Howard Correspondence. The sketch of that proof consists of three parts,

1. We can prove the double negation of non-intuitionistic axioms, e.g., $\vdash$

$\neg\neg(\varphi \vee \neg\varphi)$

2. For any formulas φ, ψ, we can prove in intuitionistic logic that $\vdash \varphi \to \neg\neg\varphi$ and $\vdash (\varphi \to \neg\neg\psi) \to \neg\neg\varphi \to \neg\neg\psi$.

3. Every proof of classical tautology φ can be written in intuitionistic logic by introducing the non-intuitionistic axioms to the context, i.e., $\psi_1 \vee \neg\psi_1, \cdots, \psi_n \vee \neg\psi_n \vdash \varphi$. Then we eliminate $\psi_n \vee \neg\psi_n$ by observating that $\psi_1 \vee \neg\psi_1, \cdots, \psi_{n-1} \vee \neg\psi_{n-1} \vdash \psi_n \vee \neg\psi_n \to \neg\neg\varphi$ and $\psi_1 \vee \neg\psi_1, \cdots, \psi_{n-1} \vee \neg\psi_{n-1} \vdash \neg\neg(\psi_n \vee \neg\psi_n)$.

We have already seen lem can be given $((\varphi \vee (\varphi \to r)) \to r) \to r$ for any φ and r.

Example 2.3.1 Let $\mathsf{unit} = \lambda x.\lambda f.f\ x$. Then for any simple types φ and r, we have $\vdash \mathsf{unit} : \varphi \to (\varphi \to r) \to r$. Specially, $\vdash \mathsf{unit} : \varphi \to \neg\neg\varphi$. ◊

Proof:

$$\frac{x : \varphi, f : \varphi \to r \vdash f\ x : r}{\vdash \lambda x.\lambda f.f\ x : \varphi \to (\varphi \to r) \to r}$$

Setting $r = \bot$, we have $(\varphi \to r) \to r = (\varphi \to \bot) \to \bot = \neg\neg\varphi$. □

Example 2.3.2 Let $\mathsf{bind} = \lambda x.\lambda f.\lambda k.x\ (\lambda z.f\ z\ k)$. Then for any simple types φ, ψ, r, we have $\vdash \mathsf{bind} : ((\varphi \to r) \to r) \to (\varphi \to (\psi \to r) \to r) \to (\psi \to r) \to r$. Specially, $\vdash \mathsf{bind} : \neg\neg\varphi \to (\varphi \to \neg\neg\psi) \to \neg\neg\psi$. ◊

Proof: Let $\Gamma = \{f : \varphi \to (\psi \to r) \to r, x : (\varphi \to r) \to r, k : \psi \to r\}$

$$\frac{\dfrac{\Gamma \vdash x : (\varphi \to r) \to r \qquad \dfrac{\dfrac{\Gamma, z : \varphi \vdash f\ z : (\psi \to r) \to r}{\Gamma, z : \varphi \vdash f\ z\ k : r}}{\Gamma \vdash \lambda z.f\ z\ k) : \varphi \to r}}{\Gamma \vdash x\ (\lambda z.f\ z\ k) : r}}{\vdash \mathsf{bind} : ((\varphi \to r) \to r) \to (\varphi \to (\psi \to r) \to r) \to (\psi \to r) \to r}$$

Let's think about the classical tautology $(A \wedge B) \vee \neg A \vee \neg B$. The classical meaning of it is that if we must have either $A \wedge B$ is correct or $\neg(A \wedge B)$ is

correct. If the later case is correct, we must have either A is wrong or B is wrong. To prove it, we can divide the situations into 3 cases:

1. A and B are correct;
2. $\neg A$ is correct;
3. $\neg B$ is correct.

Since our disjunction(or) is defined with 2 cases, we reorder it as

- $\neg A$ is correct;
- A is correct and
  - B is correct;
  - $\neg B$ is correct.

Let $\Gamma \vdash= \{A \vee \neg A, B \vee \neg B\}$. We can prove the following.

- $$\Gamma, \neg A \vdash (A \wedge B) \vee \neg A \vee \neg B$$
- $$\frac{\Gamma, A, B \vdash A \wedge B}{\Gamma, A, B \vdash (A \wedge B) \vee \neg B}$$
- $$\frac{\Gamma, A, \neg B \vdash \neg B}{\Gamma, A, \neg B \vdash (A \wedge B) \vee \neg B}$$

By the elimination rule of $\vee$, we have

$$\frac{\Gamma, A \vdash B \vee \neg B \quad \Gamma, A, B \vdash (A \wedge B) \vee \neg B \quad \Gamma, A, \neg B \vdash (A \wedge B) \vee \neg B}{\Gamma, A \vdash (A \wedge B) \vee \neg B}$$

$$\frac{\Gamma \vdash A \vee \neg A \quad \dfrac{\Gamma, A \vdash (A \wedge B) \vee \neg B}{\Gamma, A \vdash (A \wedge B) \vee \neg A \vee \neg B} \quad \Gamma, \neg A \vdash (A \wedge B) \vee \neg A \vee \neg B}{\Gamma \vdash (A \wedge B) \vee \neg A \vee \neg B}$$

The process can be naturally written in λ-terms. Suppose we have $l_1 : A \vee \neg A$ and $l_2 : B \vee \neg B$. l_1 should be used with a case with two terms of type $(A \to (A \wedge B) \vee \neg A) \vee \neg B$ and $\neg A \to ((A \wedge B) \vee \neg A) \vee \neg B$. For the latter, obviously we can choose $\lambda x.\mathsf{inj}_1(\mathsf{inj}_2(x))$. For the former, we can further assume $a : A$ and we use case analysis for l_2, i.e. two terms of type $B \to (A \wedge B) \vee \neg A \vee \neg B$ and $\neg B \to (A \wedge B) \vee \neg A \vee \neg B$. They are defined by $\lambda b.\mathsf{inj}_1(\mathsf{inj}_1(\langle a, b\rangle))$ and $\lambda y.\mathsf{inj}_2(y)$. In a word, we have

$$
\begin{aligned}
l_1 : A \vee \neg A, l_2 : B \vee \neg B \vdash \quad & \mathsf{case}(\\
& \quad \lambda a.\mathsf{case}(\\
& \qquad \lambda b.\mathsf{inj}_1(\mathsf{inj}_1(\langle a, b\rangle))), \\
& \qquad \lambda y.\mathsf{inj}_2(y) \\
& \quad)\ l_2, \\
& \quad \lambda x.\mathsf{inj}_1(\mathsf{inj}_2(x)) \\
&)\ l_1 : (A \wedge B) \vee \neg A \vee \neg B
\end{aligned}
$$

According to Glivenko's theorem, we should be able to prove the double negation of $(A \wedge B) \vee \neg A \vee \neg B$, which means we must be able to find a term of type $\neg\neg((A \wedge B) \vee \neg A \vee \neg B)$. Let $\varphi = (A \wedge B) \vee \neg A \vee \neg B$ and $M =$

$$
\mathsf{case}(\lambda a.\mathsf{case}(\lambda b.\mathsf{inj}_1(\mathsf{inj}_1(\langle a, b\rangle))), \lambda y.\mathsf{inj}_2(y))\ l_2, \lambda x.\mathsf{inj}_1(\mathsf{inj}_2(x)))\ l_1
$$

For any τ, we have

$$
\frac{\vdash \mathsf{bind} : \neg\neg(\tau \vee \neg\tau) \to (\tau \vee \neg\tau \to \neg\neg\varphi) \to \neg\neg\varphi \quad \vdash \mathsf{lem} : \neg\neg(\tau \vee \neg\tau)}{\vdash \mathsf{bind}\ \mathsf{lem} : (\tau \vee \neg\tau \to \neg\neg\varphi) \to \neg\neg\varphi}
$$

The above shows that

$$
\frac{\dfrac{l_1 : A \vee \neg A, l_2 : B \vee \neg B \vdash M : \varphi \quad \vdash \mathsf{unit} : \varphi \to \neg\neg\varphi}{l_1 : A \vee \neg A, l_2 : B \vee \neg B \vdash \mathsf{unit}\ M : \neg\neg\varphi}}{l_1 : A \vee \neg A \vdash \lambda l_2.\mathsf{unit}\ M : B \vee \neg B \to \neg\neg\varphi}
$$

Thus

$$\dfrac{\dfrac{\dfrac{\vdash \mathsf{bind}_2\ \mathsf{lem}_2 : (B \vee \neg B \to \neg\neg\varphi) \to \varphi}{l_1 : A \vee \neg A \vdash \mathsf{bind}_2\ \mathsf{lem}_2\ (\lambda l_2.\mathsf{unit}\ M) : \neg\neg\varphi}}{\vdash \lambda l_1 : \mathsf{bind}_2\ (\lambda l_2.\mathsf{unit}\ M)\ \mathsf{lem}_2 : A \vee \neg A \to \neg\neg\varphi}}{\vdash \mathsf{bind}_1\ \mathsf{lem}_1\ (\lambda l_1 : \mathsf{bind}_2\ \mathsf{lem}_2\ (\lambda l_2.\mathsf{unit}\ M)) : \neg\neg\varphi}$$

Let's look at the type $\neg\neg\varphi \to (\varphi \to \neg\neg\psi) \to \neg\neg\psi$ for bind. To derive $\neg\neg\psi$, we need two things, a term of type $\neg\neg\varphi$ and a term $\varphi \to \neg\neg\psi$. The latter is often defined by a λ-abstraction $\lambda x.P$. Note P should be given the type $\neg\neg\psi$, which could also be derived from an application of bind, as the example above. Thus let's written $\mathsf{bind}\ m\ \lambda x.P$ as

$$\begin{aligned} &x \leftarrow m \\ &P \end{aligned}$$

If $P = \mathsf{bind}\ n\ \lambda y.Q$, we can further write this as

$$\begin{aligned} &x \leftarrow m \\ &y \leftarrow n \\ &Q \end{aligned}$$

For example, $\mathsf{bind}_1\ \mathsf{lem}_1\ (\lambda l_1 : \mathsf{bind}_2\ \mathsf{lem}_2\ (\lambda l_2.\mathsf{unit}\ M)) : \neg\neg\varphi$ can be written as

$$\begin{aligned} &l_1 \leftarrow \mathsf{lem}_1 \\ &l_2 \leftarrow \mathsf{lem}_2 \\ &\mathsf{unit}\ M \end{aligned}$$

For clarity, we can add types to the above notation.

$$\begin{aligned} &l_1 : A \vee \neg A \leftarrow \mathsf{lem}_1 : \neg\neg(A \vee \neg A) \\ &l_2 : B \vee \neg B \leftarrow \mathsf{lem}_2 : \neg\neg(B \vee \neg B) \\ &\mathsf{unit}\ M \end{aligned}$$

Let's also define a more readable notation for case. If we have typed terms

$M : \varphi \vee \psi$, $\lambda x.P : \varphi \to \sigma$ and $\lambda y.Q : \psi \to \sigma$ under some context, then we write $\mathsf{case}(\lambda x.P, \lambda y.Q)\ M$ as

$$\begin{array}{l} \mathsf{case}\ M\ \mathsf{of} \\ \quad x \mapsto P \\ \quad y \mapsto Q \end{array}$$

For example, the above term can be written as

$$\begin{array}{l} l_1 \leftarrow \mathsf{lem}_1 : \neg\neg(A \vee \neg A) \\ l_2 \leftarrow \mathsf{lem}_2 : \neg\neg(B \vee \neg B) \\ \mathsf{unit}(\\ \quad \mathsf{case}\ l_1\ \mathsf{of} \\ \quad\quad a : A \mapsto \mathsf{case}\ l_2\ \mathsf{of} \\ \quad\quad\quad b : B \mapsto \mathsf{inj}_1(\mathsf{inj}_1(\langle a, b\rangle)) : ((A \wedge B) \vee \neg A) \vee \neg B \\ \quad\quad\quad y : \neg B \mapsto \mathsf{inj}_2(y) : ((A \wedge B) \vee \neg A) \vee \neg B \\ \quad\quad x : \neg A \mapsto \mathsf{inj}_1(\mathsf{inj}_2(x)) : ((A \wedge B) \vee \neg A) \vee \neg B \\ ) \end{array}$$

The above is a totally classical proof, but it is automatically translated to the double negation in intuitionistic logic. This shows how Curry-Howard Correspondence helps us build a proof more intuitively than a proof in intuitionistic logic.

Chapter 3

Homotopical Interpretation of Logic

We have already done a very naive kind of logic, which is far from the one we use in daily mathematics, where we have to talk about properties about some objects, like the parity of natural numbers. In propositional logic, we only focus on the atomic meaning of a 'proposition' by using a propositional variable for it. This means you cannot first assert a number n is even and then derive a conclusion $\sin(n\pi) = 0$ in propositional logic, because the relation between this two propositions are not captured by logical connectives.

To solve this issue, we split our language into two parts, objects like n, 0, $\sin(n\pi)$ and statements about them, e.g., $\sin(n\pi) = 0$. Instead of using a variable A for 'a number n is even' as we did in 1, we use a notation with the information about n, e.g., $E(n)$ to mean 'n is an even enumber'. Then you can describe $\sin(n\pi) = 0$ with n as the bridge between the semantics of the two propositions. These symbols for properties on objects like n are called *predicates*, and we use variables for the objects as the role of n. Objects also include constants like π or 0, and we have operations on objects, like the sin function. Note that sin is considered as a part of our language, not a real function. Also, we have a word '=' to represent the equality, like a predicate. Now the atoms of our language are broken into predicates and those objects formed by variables, constants and operations. We then concatenate them by logical connectives, e.g., $E(n) \to \sin(n\pi) = 0$. The semantics of such a formula is defined by finding a set U called the *universe*, an assignment assigning each variable like n an element $v(n) \in U$. The predicates are interpreted as relations on U and operations like sin are functions on this U. In this way, we define the truth value of the predicates and then determine whether the whole sentence is correct by the semantics of logical connectives. Besides, we use quantifiers $\forall$

to range over all elements in U, representing each iteration by a variable after $\forall$ and ignoring the present assignment for this variable and we also use $\exists$ to mean the existence of an element satisfying certain properties. For example, $\forall n, E(n) \to \sin(n\pi) = 0$.

This style is called first-order language in the sense that we require all n-ary predicates must be applied to n objects and n-ary functions must be fed with n arguments. In other words, we can only assert on the properties about the elements, not the subsets of the universe U, or the predicates we are using. Though we say that we use first order language for all mathematical theory, but in fact we are using first order set theory. For example, all groups of order prime square are commutative, but the proof of it requires subgroups as in Lagrange's theory. In first order language, we only need a binary function $\cdot$ and formulas $\forall x, y, z, x\cdot(y\cdot z) = (x\cdot y)\cdot z$, $\exists e, \forall x, e\cdot x = x\cdot e = x \wedge \exists y, x\cdot y = y\cdot x = e$ to characterize a group, i.e., a universe satisfying these formulas. If the group only has n elements, we can use the formula $\exists x_1, \cdots, x_n, x_1 \neq x_2 \wedge \cdots \wedge x_{n-1} \neq x_n \wedge \forall y, y = x_1 \vee \cdots \vee y = x_n$ to ensure the cardinality of a group. But it is not intuitive why these formulas can result in $\forall x, y, x \cdot y = y \cdot x$ when $n = 25$.

Despite the difficulty to prove the commutativity directly, the commutativity is easy to prove if you interpret a group as a set. In fact, we only use the first order language, and implement other theory as second order concepts in the first order set theory. Our only objects are sets with special properties, not a brand new logic system. If you consider the set theory as the meta language used for the some-order logic system like the group theory above, then we can figure out that *propositions* formed inside a formal language are different from the *judgements* we make in meta language. For example, in the fundamental group of some topological space, the equality of two paths is based on homotopy. In first order group theory, you can only speak of the equality, neglecting all higher order structures in this space raised by homotopies.

Clearly, we have to study two parts of our language: the first order formalization of our meta language like set theory, and the implementation of other theories like group theory in this language. My aim is to use λ-calculus as such a language. Up to now, we have studied λ-calculus as our first order target, like the axiomized set theory. We build the relation $\Gamma \vdash a : A$ as the deduction rules in this system. Since the relation is built by a finite process, we know only finitely many elements in Γ are necessary. According to the deduction rules, $\Gamma, x : \varphi \vdash M : \psi$ if and only if $\Gamma \vdash \lambda x.M : \varphi \to \psi$, so we can always make our typing process end in the form of $\vdash M : \varphi$. Thus in our system, we only have to use one kind of typability statements, i.e. some $\vdash M : \varphi$. In other words, you can omit the $\vdash$ sign and we shall assume every term is given a type based on those rules. But note this typing process happens in our meta language, as in propositional logic, where $p \vdash p$ is different from $\vdash p \to p$.

Later I will introduce more deduction rules so that we have new terms and types like $n : \mathbb{N}$, which informally means n is a natural number, and $p : E(n)$, which means p proves n is a natural number, to describe predicates as in first order language. The proposition $E(n)$ is in the the final formal language, but the typability $p : E(n)$ is the deduction in our meta language. To distinguish them, we shall call those types like $E(n)$ *propositions* and those relate to meta language *judgements*.

Another notable judgemental property is equality. As proved before, if $M \to_\beta N$ and $\Gamma \vdash M : \varphi$, then $\Gamma \vdash N : \varphi$. This shows we can consider the $=_\beta$ as a judgemental affair. For example, if we have $(\lambda x.x)\ a : \varphi$, then there is no need to distinguish a and $(\lambda x.x)\ a$. This equality is different from the equalities we can prove, e.g., $p : n + 0 = n$. We then use $\equiv$ for the one we use in meta language, like $=_\beta, =_\alpha, =_\eta$, and sometimes to emphasize a definition, we use '$:\equiv$' for that equality.

According to Curry-Howard Correspondence, we can use λ-calculus as a proof system, while it is intrinsically a computation model. $\to \wedge, \vee$ are mapped

to functions, pairs and disjoint unions like ordinary mathematical objects. We do not have to separate objects and properties about them. The only thing we need is the typability judgement.

3.1 Universe of types

In our previous development of logic and λ-calculus, I frequently used the expressions like 'for a formula of form $\varphi \to \varphi$'. The arbitrariness of this φ happens in our meta language. We hope to bring it to our formal language, a bit like the relation between $p \vdash p$ and $\vdash p \to p$, where the provability $\vdash$ is captured by the connective '$\to$' in the formal language.

Recall in 2.1.23, we have seen that we can assign any type of the form $\varphi \to \varphi$ to the term $\lambda x.x$. The problem here is that the type $\varphi \to \varphi$ has an unbounded variable φ in our meta language, but the term assigned to it has no information about that. A natural way to solve it is to make a λ-abstraction over types, i.e., $\lambda\varphi.\lambda(x : \varphi).x$. Here I use Church typing to hint and enforce the type bound to $\lambda x.x$.

In order to talk about types in the term, let's loosen our typing system by allowing one to mention that a variable is a type variable. For a type variable A, we allow terms like $A : *$ in the context, where $*$ is a special notation informally meaning 'all types'. For example, a guess of the proof of the above example might be

$$\dfrac{\dfrac{A : *, x : A \vdash x : A}{A : * \vdash \lambda x.x : A \to A}}{\vdash \lambda A.\lambda(x : A).x}$$

Here are two problems. The first is that we define our context to be an unordered set, but now, the element $x : A$ is dependent on $A : *$. That may cause some term like $\lambda x.\lambda A.\lambda x$, which is certainly unwanted. Thus we technically require the context to be a finite ordered sequence, where later elements may dependent on the previous ones.

The second is what type should be given to the term $\lambda A.\lambda(x : A).x$. You may want to use $* \to A \to A$ for the type according to our previous discussion. But the problem of that is the A in the type is still free, and it should be bounded by some notation. Since the term $\lambda A.\lambda x.x$ is still assigned to a specific type $* \to A \to A$. If we want to prove $\lambda x.x : B \to B$, how should we apply $\lambda A.\lambda x.x : * \to A \to A$ to obtain $\lambda x.x : B \to B$? Thus we want to introduce a notation to bind it. We write $\Pi_{A:*}(A \to A)$ for the term $\lambda A.\lambda x.x$ as quantifiers over types, i.e., the following rule.

$$\frac{\Gamma, A : * \vdash M : \varphi}{\Gamma \vdash \lambda A.M : \Pi_{A:*}\varphi}$$

For example, the term $\lambda A.\lambda x.x$ is typed by

$$\frac{\dfrac{A : *, x : A \vdash x : A}{A : * \vdash \lambda x.x : A \to A}}{\vdash \lambda A.\lambda x.x : \Pi_{A:*}(A \to A)}$$

Our types (previously defined as formulas) are then extended by

$$\mathscr{F} ::= V \mid \mathscr{F} \to \mathscr{F} \mid \cdots \mid \Pi_{V:*}\mathscr{F}$$

Note the notation $*$ is not included in this set. In fact, in such a second order λ-calculus, we are only allowed to speak of the types defined in this way, thus nothing like $* \to *$ can be formed. So it seems that the subscript $V : *$ after Π is unnecessary, but later we have to distinguish a more general Π-type, so here I define it explicitly. Then if φ is a type, we have the rule

$$\Gamma \vdash \varphi : *$$

without any premises if all free variables in φ are included in Γ. The application

is also obvious.

$$\frac{\Gamma \vdash M : \Pi_{A:*}\varphi \quad \Gamma \vdash \psi : *}{\Gamma \vdash M\ \psi : \varphi[A \coloneqq \psi]}$$

where the substitution is defined similarly.

Let $\mathsf{id} \coloneqq \lambda A.\lambda x.x : \Pi_{A:*}A \to A$. We have $\mathsf{id}\ B : B \to B$. Note now we have quantifiers over type variables, so it is necessary and possible to always give a term a type. Previously, I avoid this to enhance the flexibility, but now the Π-types enable us to write it in the formal language. With every term assigned a type, our λ-terms become more human-readable.

Another benefit of this notation is that we can allow subscript as a partial application of terms on types. For example, $\mathsf{id}\ B : B \to B$ can be written as $\mathsf{id}_B : B \to B$. And we abuse our language by making the convention that the subscript can be omitted if the subscript can be inferred from the context, e.g. for $b : B$, $\mathsf{id}\ b : B$. This will not weaken the rigorousness of our language because $\mathsf{id} : \Pi_{A:*}A \to A$ must be first applied to a type and then some $a : A$. Thus if you find id is only apply to a single term, or is used for some place where a term of type $A \to A$ is required, then we know it should be an id_A. But if this causes ambiguity, then we have to specify the type A.

The introduction of $*$ also enables us to bring the syntax of type in our meta language to the formal language. For example

$$\frac{A : *, B : * \vdash A : * \quad A : *, B : * \vdash B : *}{A : *, B : * \vdash A \to B : *}$$

This seems unnecessary, but we can use this style to extend the types $\mathscr{F}$, and we shall then think whether we can make abstraction based on it, e.g., shall we admit $\lambda A.\lambda B.A \to B : * \to * \to *$? There is no reason why we cannot do this. This abstraction will not break the consistency of our language because it only generalizes the rule in our meta language. But not we cannot consider $* \to * \to *$ of type $*$, because self-referring will cause Russell's paradox

[9]. Traditionally, we lift the order of our language by introducing one more notation $\Box$ for this situation, e.g., $* \to * \to * : \Box$. More formally, we add the following rules.

$$\Gamma \vdash * : \Box$$

$$\frac{\Gamma \vdash \alpha : \Box \quad \Gamma \vdash \beta : \Box}{\Gamma \vdash \alpha \to \beta : \Box}$$

Now we have a really complicated but expressive system. We should verify whether $\lambda x.x : \varphi \to \varphi$ is correct in our language, whether $\varphi \to \varphi : *$ is well formed, and even whether $* : \Box$ is valid. But we now can express polymorphic terms with different types easily.

In general, if we want to introduce a type like $A \wedge B$, we have to specify forming rule(s),

$$\frac{\Gamma \vdash A : * \quad \Gamma \vdash B : *}{\Gamma \vdash A \wedge B : *}$$

construction rule(s),

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash \langle a, b \rangle : A \wedge B}$$

elimination rule(s),

$$\frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \mathsf{pr}_1(M) : A}$$

$$\frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \mathsf{pr}_2(M) : B}$$

computation rules $\mathsf{pr}_1(\langle a, b \rangle) \to_\beta a$, $\mathsf{pr}_2(\langle a, b \rangle) \to_\beta b$, and $=_\alpha$,$=_\eta$ for this type.

So far we have defined terms, terms depending on types, and even types depending on types. What about types depending on terms? To motivate such a kind of type, let's consider the equality between two elements $a, b : A$. In the sense of Curry-Howard, if we want to proof two elements are equal, we have to define a type judgement rule to allow some annotated term like $p : a = b$. Intuitively, if $\Gamma \vdash A : *$, then for each possible $a, b : A$, there should be

a type $a = b : *$ so that we can later decide how to find terms with this type. This should be written as

$$\frac{\Gamma \vdash A : * \quad \Gamma \vdash a : A \quad \Gamma \vdash b : A}{\Gamma \vdash a =_A b : *}$$

Here $a =_A b$ is such a type depending on terms. We then want to generalize this kind of arbitrariness by $\lambda a.\lambda b.a =_A b$ with type $A \to A \to *$, which means a family of types indexed by $A \times A$. If you view each $a =_A b$ as a proposition, then $p : a =_A b$ means this proposition is provable. Thus the type family tells us that we actually define a proposition for each pair (a, b), which is exactly the meaning of predicate. This behavior is a bit like a function defined on infinitely many arguments. For example, if a function has countably many real arguments, we do not define it as $\mathbb{R}_1 \times \mathbb{R}_2 \times \cdots \to \mathbb{R}$, but as $(\mathbb{N} \to \mathbb{R}) \to \mathbb{R}$, where $(\mathbb{N} \to \mathbb{R})$ means all real arrays, i.e., arguments indexed by $\mathbb{N}$.

But this type $A \to A \to *$ does not fit in any previous cases like $*$ or $\Box$. To fully find a suitable scope like $\Box$ for $A \to A \to *$, we introduce a hierarchical universe. Now we do not think $*$ is a type because that will trigger Russell's paradox. However it can be roughly thought as of type $\Box$, if we introduce $\Box$ as a higher order type. This is the idea of von Neumann's universe [22] for set theory. We introduce $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \cdots$ to our language. The typing $\mathcal{U}_i : \mathcal{U}_{i+1}$ is a bit like $* : \Box$. For example we have the following rules:

$$\frac{\Gamma \vdash A : U_0}{\Gamma \vdash A \to A : \mathcal{U}_0}$$

and

$$\frac{\Gamma \vdash \mathcal{U}_0 : \mathcal{U}_1}{\Gamma \vdash \mathcal{U}_0 \to \mathcal{U}_0 : \mathcal{U}_1}$$

Thus we can freely use $\mathcal{U}_i$ as a normal type.

$$\frac{\Gamma \vdash A : \mathcal{U}_{i+1} \quad \Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}{\Gamma \vdash A \to \mathcal{U}_i : \mathcal{U}_{i+1}}$$

We can then speak of a type family easily by $A \to \mathcal{U}_i$. For example, = can be give the type $\Pi_{A:\mathcal{U}_i} A \to A \to \mathcal{U}_i$.

Further, we require $\mathcal{U}_i$ to be cumulative, i.e., if $A : \mathcal{U}_i$, then $A : \mathcal{U}_{i+1}$. This is very convenient but may cause other troubles since the type of a term is no longer unique. We also add a convention that the subscript i can be omitted if the typing of a term does not rely on the subscript, e.g. $\lambda A.\lambda x.x : \Pi_{A:\mathcal{U}} A \to A$. Be sure that the type of each term or its type should always be some $\mathcal{U}_n$. For example, $\lambda i : \mathbb{N}.\mathcal{U}_i$ cannot be put in a suitable universe $\mathcal{U}_n$.

By introducing the type universe, we dim the distinction between terms and types. As in set theory, everything in our theory can be called a type, though 'terms' are still used to emphasize those before ':'. And our theory is formally called *type theory*.

Now I can give the most general definition of a function.

Definition 3.1.1 *(Dependent functions). Suppose $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, then we can form $\Pi_{a:A} B(a)$, called* **dependent function type** *or* **Π-type**.

1. *The formation rule is that*

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma, a : A \vdash B(a) : \mathcal{U}}{\Gamma \vdash \Pi_{a:A} B(a) : \mathcal{U}}$$

2. *The construction rule for it is still the λ-abstraction.*

$$\frac{\Gamma, x : A \vdash M : B(x)}{\Gamma \vdash \lambda x.M : \Pi_{a:A} B(a)}$$

3. *The elimination rule is still the application.*

$$\frac{\Gamma \vdash f : \Pi_{a:A} B(a) \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B(a)}$$

4. *Computation rule is $\to_\beta$ defined by substitution.*

5. *We define the η-equality for functions, i.e. $\lambda x.f\ x =_\eta f$. Note this is a judgemental equality, so $\lambda x.f\ x \equiv f$.*

6. *When $B : A \to \mathcal{U}$ is a constant family, e.g., $B \equiv \lambda(a : A).C$ for some $C : \mathcal{U}$, the type $\Pi_{a:A}B(a)$ is just the normal function type $A \to C$.*

Here $B(a)$ should have been $(B\ a)$ as an application. Though we should define all functions by λ-abstraction, there is no reason to refuse traditional notations for functions, e.g., the identity function can be written as $\mathsf{id}_A(x) :\equiv x$. If B is $\lambda x.M$, then we think $B(a)$ as $M[x := a]$, the result after substitution rather than the term before a β-reduction. This convention can give us more convenience. For example, if you let $B :\equiv \lambda x.x =_A x$, then $\Pi_{a:A}B(a)$ is $\Pi_{a:A}a =_A a$ rather than $\Pi_{a:A}((\lambda x.x =_A x)\ a)$, though they are judgementally equal by $=_\beta$.

For a function type $\Pi_{x:A}(\Pi_{y:B}C(x,y))$, we also admit the currying convention, i.e., we write $\Pi_{x:A}\Pi_{y:B}C(x,y)$ omitting the parentheses. Also for $\Pi_{x:A}\Pi_{y:A}C(x,y)$, we can write $\Pi_{x,y:A}C(x,y)$ to save one Π.

In traditional mathematics, the proposition $\forall n, E(n) \to \sin(n \cdot \pi) = 0$ is a family of propositions indexed by $\mathbb{N}$, i.e., it contains the information of $E(1) \to \sin(1 \cdot \pi) = 0$, $E(2) \to \sin(2 \cdot \pi) = 0$, $\cdots$. So with dependent type, we can define $P(n) :\equiv E(n) \to \sin(n \cdot \pi) = 0$, and $P : \mathbb{N} \to \mathcal{U}$. If we have a function $f : \Pi_{n:\mathbb{N}}P(n)$, then for each $n : \mathbb{N}$, we have $f\ n : P(n)$. Thus Π-types act as the universal quantifier $\forall$.

Before we continue to other types, let me introduce some other simple types we need later.

Definition 3.1.2 *(**1**-type) We define $\mathbf{1} : \mathcal{U}$ with construction rule*

$$\vdash \star : \mathbf{1}$$

and elimination rule

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{elim}_{\mathbf{1}}(A, a) : \mathbf{1} \to A}$$

such that $\mathsf{elim}_{\mathbf{1}}(A, a)\ \star :\equiv a$.

Here the symbol ':≡' is actually $\to_\beta$*, but to emphasize it is an equality judgementally, I use the equality sign* $\equiv$ *to show that not only do we define the equality, but also the equality closure of it.*

The elimination $\mathsf{elim}_{\mathbf{1}}$ *can either be defined as an extension of* λ*-terms as we have done for* $\wedge$*, or as a dependent function* $\mathsf{elim}_{\mathbf{1}} : \Pi_{A:\mathcal{U}} A \to \mathbf{1} \to A$*. The later is more easy to be implemented by a computer program because we only have to check the typing process when an application happens. Since it is easy to observe that this two definitions are equivalent to each other, I will use them alternatively.*

Definition 3.1.3 *(*$\mathbf{0}$*-type) We define* $\mathbf{0} : \mathcal{U}$ *with empty construction rule and elimination rule*

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash z : \mathbf{0}}{\Gamma \vdash \mathsf{elim}_{\mathbf{0}}(A, z) : A}$$

This is an alias to absurdity $\bot$*.*

Definition 3.1.4 *(*$\mathbf{2}$*-type) We define* $\mathbf{2} : \mathcal{U}$ *with 2 construction rules*

$$\vdash 0_{\mathbf{2}} : \mathbf{2}$$

$$\vdash 1_{\mathbf{2}} : \mathbf{2}$$

and elimination rule

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash a_0 : A \quad \Gamma \vdash a_1 : A}{\Gamma \vdash \mathsf{elim}_{\mathbf{2}}(A, a_0, a_1) : \mathbf{2} \to A}$$

such that $\mathsf{elim}_{\mathbf{2}}(A, a_0, a_1)\ 0_{\mathbf{2}} :\equiv a_0$ *and* $\mathsf{elim}_{\mathbf{2}}(A, a_0, a_1)\ 1_{\mathbf{2}} :\equiv a_1$.

3.2 Inductively defined types

Now let's think about the recursively defined natural numbers $\mathbb{N}$. Curry-Howard correspondence enables us to write proof as terms, but do not forget

that λ-terms are also our objects for computation. We do not have to separate objects and proofs as in set theory. So we also have to discuss how to introduce natural numbers in type theory. Though we have introduced a principle for new types, we cannot abuse it. We only add one rule to define new types, i.e., via induction (recursion).

Let's first define natural numbers.

Definition 3.2.1 *(Natural Numbers)*

1. *The formation rule is just* $\vdash \mathbb{N} : \mathcal{U}$.

2. *Natural numbers are composed by two parts, a zero* $\mathsf{Z} : \mathbb{N}$ *and a successor* $\mathsf{S} : \mathbb{N} \to \mathbb{N}$*, they are the construction rules.*

$$\vdash \mathsf{Z} : \mathbb{N}$$

$$\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{S}(n) : \mathbb{N}}$$

3. *If you want to define a function* $f : \mathbb{N} \to C$ *out of* $\mathbb{N}$*, you ought to define it recursively, i.e., we only admit those functions that are defined by specifying a value on* Z *and how* $f(\mathsf{S}(n)) : C$ *is defined by* $n : \mathbb{N}$ *and* $f(n) : C$*. i.e.*

$$\frac{\Gamma \vdash C : \mathcal{U} \quad \Gamma \vdash c_z : C \quad \Gamma \vdash c_s : \mathbb{N} \to C \to C}{\Gamma \vdash \mathsf{rec}(C, c_z, c_s) : \mathbb{N} \to C}$$

such that

$$\mathsf{rec}(C, c_z, c_s)\ \mathsf{Z} :\equiv c_z$$

$$\mathsf{rec}(C, c_z, c_s)\ \mathsf{S}(n) :\equiv c_s\ n\ (\mathsf{rec}(C, c_z, c_s)\ n)$$

Remark 3.2.2 As mentioned in 2.2.11, we can think S as a function of type $\mathbb{N} \to \mathbb{N}$ and $\mathsf{rec} : \Pi_{C:\mathcal{U}} C \to (\mathbb{N} \to C \to C) \to \mathbb{N} \to C$. In this style, we do not have to extend the syntax for terms. We only need to introduce new constant terms

like S and rec and apply them to other terms. Though they are given function types and behavior like functions, they act more like axioms, not just ordinary functions. So we call $Z : \mathbb{N}$ (a nullary function) and $\mathsf{S} : \mathbb{N} \to \mathbb{N}$ *constructors* to emphasize this particularity. Similarly, we call rec the *eliminator*. ◇

Example 3.2.3 (Addition on Natural Numbers) We all know that addition $m + n$ on Peano numbers are defined recursively by

$$0 + n = n$$

and

$$\mathsf{S}(m) + n = \mathsf{S}(m + n)$$

which should be considered as a function $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that $(\mathsf{Z}+) : \mathbb{N} \to \mathbb{N}$ is the identity function and $(\mathsf{S}(m)+) : \mathbb{N} \to \mathbb{N}$ is the successor of the result of $m+$. Based on this observation, we can define addition in our theory by $+ :\equiv \mathsf{rec}(\mathbb{N} \to \mathbb{N}, \lambda n.n, \lambda m.\lambda h.\lambda n.\mathsf{S}(h\ n))$. Clearly

$$\begin{aligned}
1 + 2 &\equiv \mathsf{S}(\mathsf{Z}) + 2 \\
&\equiv \mathsf{rec}(\mathbb{N} \to \mathbb{N}, \lambda n.n, \lambda m.\lambda h.\lambda n.\mathsf{S}(h\ n))\ \mathsf{S}(\mathsf{Z})\ 2 \\
&\equiv (\lambda m.\lambda h.\lambda n.\mathsf{S}(h\ n))\ \mathsf{Z}\ (\mathsf{Z}+)\ 2 \\
&\equiv \lambda h.(\lambda n.\mathsf{S}(h\ n))\ (\mathsf{Z}+)\ 2 \\
&\equiv \mathsf{S}(Z + 2) \\
&\equiv \mathsf{S}(\mathsf{rec}(\mathbb{N} \to \mathbb{N}, \lambda n.n, \lambda m.\lambda h.\lambda n.\mathsf{S}(h\ n))\ \mathsf{Z}\ 2) \\
&\equiv \mathsf{S}((\lambda n.n)\ 2) \\
&\equiv \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))) \\
&\equiv 3
\end{aligned}$$ ◇

Recall in the first two chapters, we frequently used the technique that to infer over a recursively defined function (concept), we use induction over

the recursive definition. If we want to prove some properties about natural numbers, then we should also do induction. Now we have defined $\mathsf{Z} + n \equiv n$. This does not mean $n + \mathsf{Z} \equiv n$, and we cannot even write down '$\equiv$' because there is not any β-reduction since n is only a letter, a bit like $f\ x$ cannot be reduced any more. We can only prove $\Pi_{n:\mathbb{N}}(n + \mathsf{Z} =_{\mathbb{N}} n)$ in the propositional sense.

To fully show you how to prove this property about natural numbers, here I introduce two rules for the equality type $=$, which will be derived later from general rules of recursion. First we have a construction rule (reflexivity)

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash \mathsf{refl}(x) : x =_A x}$$

and a rule to make use of equality (called Leibniz's rule)

$$\frac{\Gamma \vdash C : A \to \mathcal{U} \quad \Gamma \vdash p : x =_A y}{\mathsf{leibniz}(C, p) : C(y) \to C(x)}$$

This means if C is a predicate on A, then if $C(y)$ and $x = y$, we must have $C(x)$.

Think about how we prove $n + 0 = n$ in human mathematics. We first have a base step that when $n = 0$, $0 + 0 = 0$. Then if $n = S(m)$, we have a inductive hypothesis $m + 0 = m$, so $n + 0 = S(m) + 0 = S(m + 0) = S(m) = n$. This can be formalized as a more general elimination rule for $\mathbb{N}$:

$$\frac{\Gamma \vdash C : \mathbb{N} \to \mathcal{U} \quad \Gamma \vdash c_z : C(0) \quad \Gamma \vdash c_s : \Pi_{n:\mathbb{N}} C(n) \to C(\mathsf{S}(n))}{\Gamma \vdash \mathsf{ind}(C, c_z, c_s) : \Pi_{n:\mathbb{N}} C(n)}$$

with

$$\mathsf{ind}(C, c_z, c_s)\ \mathsf{Z} :\equiv c_z$$

$$\mathsf{ind}(C, c_z, c_s)\ \mathsf{S}(n) :\equiv c_s\ n\ (\mathsf{ind}(C, c_z, c_s)\ n)$$

So to proof $\Pi_{n:\mathbb{N}}(n + \mathsf{Z} = n)$, let $C(n) :\equiv n + \mathsf{Z} = n$. We have to find a

$c_z : \mathsf{Z} + \mathsf{Z} = \mathsf{Z}$ and $c_s : \Pi_{n:\mathbb{N}}(n + \mathsf{Z} = n) \to (\mathsf{S}(n) + \mathsf{Z} = \mathsf{S}(n))$. The former is just the reflexivity $\mathsf{refl}(\mathsf{Z})$. Let $C(x) :\equiv \mathsf{S}(x) = \mathsf{S}(n)$ and the latter is proved by

$$\frac{n : \mathbb{N}, H : n + Z = n \vdash \mathsf{leibniz}(C, H) : \mathsf{S}(n) = \mathsf{S}(n) \to \mathsf{S}(n + \mathsf{Z}) = \mathsf{S}(n)}{n : \mathbb{N}, H : n + Z = n \vdash \mathsf{leibniz}(C, H)\ \mathsf{refl}(\mathsf{S}(n)) : \mathsf{S}(n + Z) = \mathsf{S}(n)}$$

Note that $\mathsf{S}(n) + Z :\equiv \mathsf{S}(n + Z)$. This finishes the proof.

If the type family $C : \mathbb{N} \to \mathcal{U}$ is a constant family, then ind is identical to rec. In this sense, induction and recursion are the same in our discussion.

But we have to be careful with induction! Let's consider the following type D, how a naive intuition gives a contradiction.

Definition 3.2.4 *1. The formation rule is*

$$\vdash D : \mathcal{U}$$

2. The construction rule is

$$\frac{\Gamma \vdash p : D \to \bot}{\Gamma \vdash d(p) : D}$$

This means type D contains the information of a witness of $D \to \bot$.

3. Thus the elimination rule is that to define a function $h : D \to C$, you only have to define how to obtain C given the information $D \to \bot$.

$$\frac{\Gamma \vdash x : D \quad \Gamma \vdash q : (D \to \bot) \to C}{\Gamma \vdash e(q, x) : C}$$

4. So the computation rule is simply

$$e(q, d(p)) \to_\beta q\ p$$

Though there is not any meaningful explanation for such a type, it seems all right, and is actually admitted by untyped λ-calculus and even some other

typed λ-calculus [2]. But you can derive the absurdity $\perp$ from it, which is the construction of Curry's paradox.

1. First note that $\mathsf{id}_{D\to\perp} : (D \to \perp) \to (D \to \perp)$. Thus we have an $\mathsf{unpack} \equiv \lambda x.e(\mathsf{id}_{D\to\perp}, x) : D \to (D \to \perp)$.

2. Next, we can find a term with type D.

$$\dfrac{\dfrac{\dfrac{x : D \vdash x : D \quad x : D \vdash \mathsf{unpack}(x) : D \to \perp}{x : D \vdash \mathsf{unpack}(x)\ x : D \to \perp}}{\vdash \lambda x.\mathsf{unpack}(x)\ x : D \to \perp}}{\vdash d(\lambda x.\mathsf{unpack}(x)\ x) : D}$$

3. Since now we have a term $h \equiv d(\lambda x.\mathsf{unpack}(x)\ x) : D$, we can derive absurdity from it.

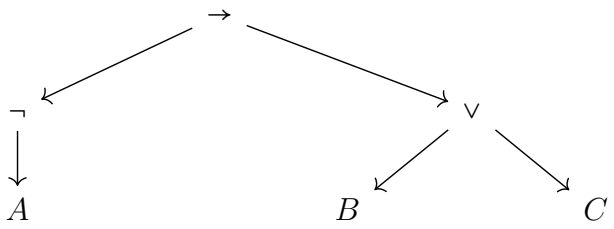
$$\dfrac{\dfrac{\vdash h : D}{\vdash \mathsf{unpack}(h) : D \to \perp} \quad \vdash h : D}{\vdash \mathsf{unpack}(h)\ h : \perp}$$

This process has a very concrete computational meaning of '*infinite loop*', which should be excluded by our theory [23]. Clearly, this problem is caused by the recursion $d : (D \to \perp) \to D$. But we know recursively defined natural numbers will not cause us any trouble. So we have to discuss what can be defined inductively.

Our requirement is very simple, the recursively defined type should form a *tree structure* to avoid the unhealthy way to define infinite loop. For example, our formulas can be defined recursively by

$$\mathscr{F} ::\equiv V \mid \neg\mathscr{F} \mid \mathscr{F} \to \mathscr{F} \mid \mathscr{F} \wedge \mathscr{F} \mid \mathscr{F} \vee \mathscr{F}$$

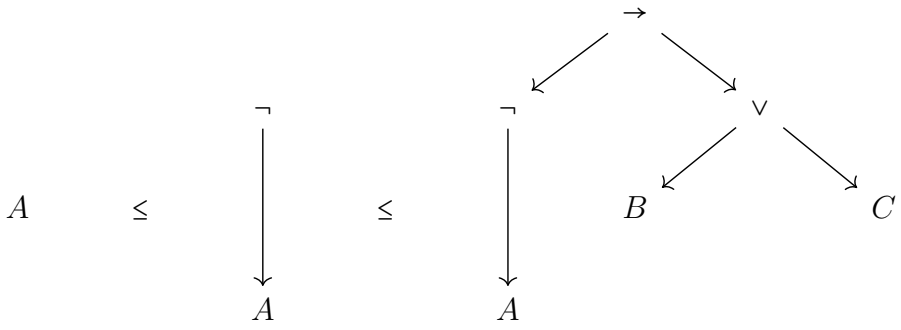
The formula $\neg A \to (B \vee C)$ is a tree by drawing it as



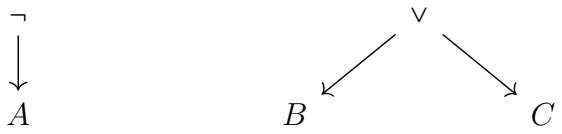
In such a tree, each node comes from 5 possibilities.

1. A node labelled by a variable with no further branch.
2. A node labelled by $\neg$, with one branch.
3. A node labelled by $\to$, with two branches.
4. A node labelled by $\wedge$, with two branches.
5. A node labelled by $\vee$, with two branches.

Each branch of a node is called a *subtree* of this node, and let's write this relation as $\leq$. For example, A is the subtree of the node at $\neg$, which is the subtree of the whole tree.



For a specific label $\to$ and two fixed trees $\neg A$ and $B \vee C$,



We say $\neg A \to (B \vee C)$ is the supremum of $\to$, $\neg A$ and $B \vee C$, because it is the

smallest (with respect to $\leq$) way to compose a tree with this label and this two branches.

Besides, we require all trees to be finite, like a freely generated structure. Thus by this way, we will not meet with an infinite loop caused by the new type and consistent with our current system [18].

The idea is generalized by the W-types. A W-type consists of a labelling $A : \mathcal{U}$, and a branch description $B : A \to \mathcal{U}$ for each label $a : A$ describing how many branches is needed for this label. For example, the natural numbers is a tree has two kind of labels, one is the situation Z, the other is S. The former has no more subtrees, while the latter has one subtree. So we use $\mathbf{2}$ as the labelling system of $\mathbb{N}$, and for label $0_{\mathbf{2}}$, we need no more subtrees, while for label $1_{\mathbf{2}}$, we need one subtree. Thus we define $B :\equiv \mathbf{2} \to \mathcal{U}$ as $\mathsf{elim}_{\mathbf{2}}(\mathcal{U}, \mathbf{0}, \mathbf{1})$. And now we write $\mathsf{W}_{a:\mathbf{2}} B(x)$ to represent natural numbers $\mathbb{N}$.

To find an element of W-type, we specific a label $a : A$ and subtrees to it indexed by $f : B(a) \to \mathsf{W}_{x:A} B(x)$, i.e.,

$$\frac{\Gamma \vdash a : A \quad \Gamma, x : A \vdash f : B(x) \to \mathsf{W}_{x:A} B(x)}{\Gamma \vdash \sup(a, f) : \mathsf{W}_{x:A} B(x)}$$

For example, $0 : \mathbb{N}$ is defined by label $0_{\mathbf{2}}$, to gather with a function $\mathbf{0} \to \mathbb{N}$ by assigning the zero branch to subtrees (which is certainly empty, since $\mathbf{0} \to \mathbb{N}$ is always definable), i.e., $0 :\equiv \sup(0_{\mathbf{2}}, \lambda x.\mathsf{elim}_{\mathbf{0}}(\mathbb{N}, x))$. Similarly, $1 : \mathbb{N}$ is the tree with label $1_{\mathbf{2}}$ and use 0 as the subtree, i.e., $1 = \sup(1_{\mathbf{2}}, \lambda x.0)$. Thus the successor of n is a node labelled $1_{\mathbf{2}}$ and use n as subtree, i.e., $\mathsf{S} :\equiv \lambda n. \sup(1_{\mathbf{2}}, \lambda x.n)$.

Induction rules over W-types are thus very obvious. Let $W = \mathsf{W}_{x:A} B(x)$ and $E : W \to \mathcal{U}$ be a predicate over W. For each element $\sup(a, f)$, we have inductive hypothesis $g : \Pi_{b:B(a)} E(f\ b)$ for each subtree $f\ b$ of a node labelled $a : A$, then we should prove the predicate on the supremum, $E(\sup(a, f))$.

This gives the elimination rule for W-type.

$$\frac{\Gamma \vdash E : W \to \mathcal{U} \quad \Gamma \vdash e : \Pi_{a:A}\Pi_{f:B(a)\to W}\Pi_{g:\Pi_{b:B(a)}E(f\ b)}E(\sup(a,f))}{\Gamma \vdash \mathsf{ind}_W(E,e) : \Pi_{w:W}E(w)}$$

Definition 3.2.5 *(W-type) For $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, we define a new type $\mathsf{W}_{x:A}B(x)$.*

1. *Formation rule:*

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash B : A \to \mathcal{U}}{\Gamma \vdash \mathsf{W}_{x:A}B(x) : \mathcal{U}}$$

2. *Construction Rule:*

$$\frac{\Gamma \vdash a : A \quad \Gamma, x : A \vdash f : B(x) \to \mathsf{W}_{x:A}B(x)}{\Gamma \vdash \sup(a,f) : \mathsf{W}_{x:A}B(x)}$$

3. *Elimination rule:*

$$\frac{\Gamma \vdash E : W \to \mathcal{U} \quad \Gamma \vdash e : \Pi_{a:A}\Pi_{f:B(a)\to W}\Pi_{g:\Pi_{b:B(a)}E(f\ b)}E(\sup(a,f))}{\Gamma \vdash \mathsf{ind}_W(E,e) : \Pi_{w:W}E(w)}$$

such that

$$\mathsf{ind}_W(E,e)\ \ \sup(a,f) :\equiv e\ a\ f\ (\lambda b.\mathsf{ind}_W(E,e)\ (f\ b))$$

W-type then enables us to give a general way to introduce new types. A type W is defined by many constructors, like Z and S, which are function types given the type W in the codomain. We also allow the type W to occur in the parameters to the constructors recursively, but they must be defined like a subtree as discuss above. So we only allow constructors with types like $W \to W \to W$. If you need infinitely many constructors, you are allowed to index them with another type like $(\mathbb{N} \to W) \to W$, since this can be captured by the constructor sup of W-type. The induction rule ind_W of it is defined by assigning each constructors a derivation of sup from inductive hypothesis. For

example, if W is defined by 2 constructors $p : W$ and $q : (\mathbb{N} \to W) \to W$, then the induction rule is

$$\frac{\Gamma \vdash E : W \to \mathcal{U} \quad \Gamma \vdash b : E(p) \quad \Gamma \vdash h : \Pi_{t:\mathbb{N}\to W}(\Pi_{n:\mathbb{N}} E(t\ n)) \to E(q\ t)}{\Gamma \vdash \mathsf{ind}_W(E, b, h) : \Pi_{w:W} E(w)}$$

The premise $h : \Pi_{t:\mathbb{N}\to W}(\Pi_{n:\mathbb{N}} E(t\ n)) \to E(q\ t)$ means you are given arbitrary subtrees index by $\mathbb{N}$ and you know E is correct on each $t\ n$, then you know they are correct on the supremum $q\ t$. And the computation rules are

$$\mathsf{ind}_W(E, b, h)\ p :\equiv b$$

$$\mathsf{ind}_W(E, b, h)\ (q\ t) :\equiv h\ t\ \lambda n.\mathsf{ind}_W(E, b, h)\ (t\ n)$$

We also extend the method to define induction without further explanation. Constructor are allowed to be dependent functions and so is every parameter to it. We are also allowed to define mutually recursive types and inductive type families. The strict definition [11] is quite technical and tedious, so I suggest you try some real computer proof assistants based on type theory like Coq or Agda to get a feeling of induction. As a quick example, let's take a look at the definition of evenness.

Definition 3.2.6 *(Evenness)*

1. *Formation rule:*

$$\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{even}(n) : \mathcal{U}}$$

2. *Construction rules:*

$$\frac{\Gamma \vdash \mathsf{Z} : \mathbb{N}}{\Gamma \vdash \mathsf{even\_Z} : \mathsf{even}(Z)}$$

$$\frac{\Gamma \vdash n : \mathbb{N} \quad \Gamma \vdash p : \mathsf{even}(n)}{\Gamma \vdash \mathsf{even\_SS}(n, p) : \mathsf{even}(\mathsf{S}(\mathsf{S}(n)))}$$

3. Induction principle: If we have

$$\Gamma \vdash P : \Pi_{n:\mathbb{N}}(\mathsf{even}(n) \to \mathcal{U}),$$

$$\Gamma \vdash z : P\ Z\ (\mathsf{even}\ Z)$$

and

$$\Gamma \vdash ss : \Pi_{n:\mathbb{N}}\Pi_{p:\mathsf{even}(n)} P\ n\ p \to P\ \mathsf{S}(\mathsf{S}(n))\ (\mathsf{even\_SS}(n,p)),$$

then

$$\Gamma \vdash \mathsf{ind}_{\mathsf{even}}(P,z,ss) : \Pi_{n:\mathbb{N}}\Pi_{p:\mathsf{even}\ n} P\ n\ p$$

such that

$$\mathsf{ind}_{\mathsf{even}}(P,z,ss)\ \mathsf{Z}\ \mathsf{even\_Z} :\equiv z$$

$$\mathsf{ind}_{\mathsf{even}}(P,z,ss)\ \mathsf{S}(\mathsf{S}(n))\ \mathsf{even\_SS}(n,p) :\equiv ss\ n\ p\ (\mathsf{ind}_{\mathsf{even}}(P,z,ss)\ n\ p)$$

This type even is defined as a family $\mathbb{N} \to \mathcal{U}$. Constructors of it is defined in the same pattern, a tree-like structure, where we allow each subtree to be some $\mathsf{even}(n) : \mathcal{U}$. You can think of this process as we first define $\mathsf{even\_Z} : \mathsf{even}(Z)$, and then $\mathsf{even\_SS}(Z, \mathsf{even\_Z}) : \mathsf{even}(2)$, and then $\mathsf{even\_SS}(2, \mathsf{even\_SS}(Z, \mathsf{even\_Z})) : \mathsf{even}(4)$, $\cdots$. To prove a proposition $P : \Pi_{n:\mathbb{N}}(\mathsf{even}(n) \to \mathcal{U})$ out of this type, we prove the base step and inductive step as before, but this process is indexed by $\mathbb{N}$ (the $\Pi_{n:\mathbb{N}}$).

Now we can discover the definition of Equality.

Definition 3.2.7 *(Equality Type). For $A : \mathcal{U}$ and $a, b : A$, we define* **equality**

type $\mathsf{Eq}(A, a, b) : \mathcal{U}$ *or* $a =_A b$*, with the only constructor*

$$\mathsf{refl} : \Pi_{x:A}\mathsf{Eq}(A, x, x)$$

Without ambiguity, we can omit the subscript $_A$ *of* $=_A$*, and we also write* $\mathsf{refl}_x : x = x$ *for* $(\mathsf{refl}\ x)$*. The elimination rule is*

$$\frac{\Gamma \vdash P : \Pi_{x,y:A}(x = y) \to \mathcal{U} \quad \Gamma \vdash c : \Pi_{x:A} P(x, x, \mathsf{refl}_x)}{\Gamma \vdash \mathsf{ind}_{=_A}(P, c) : \Pi_{x,y:A}\Pi_{p:x=y} P(x, y, p)}$$

such that $\mathsf{ind}_{=_A}(P, c)\ x\ x\ \mathsf{refl}_x :\equiv c\ x$*.*

In some materials, this type is called **identity type** *and is written as* $\mathsf{Id}(A, x, y)$*. To avoid confusion with the identity fucntion* id*, I prefer to use* Eq *instead of* Id*, though I still use the name 'identity' for some widely recognized name such as 'uniquess of identity proofs' (3.3.1).*

The equality type $=_A : A \to A \to \mathcal{U}$ is inductively defined with index $A \times A$. The induction rule means that if you want to prove $P(x, y, p)$ about an indexed equality $p : x = y$, then you only have to prove the situation when $x \equiv y$ and p is refl. Thus if $C : A \to \mathcal{U}$ and $p : x = y$, to prove $C(x) \to C(y)$ out of the type $p : x = y$, it suffices to assume $x \equiv y$ and trivially, $C(x) \to C(x)$, which is exactly the $\mathsf{leibniz}$ rule.

Induction is so powerful that we can even define (co)product of two types from it. A product $A \times B$ is defined by an only constructor $\langle , \rangle : A \to B \to A \times B$.

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash A \times B : \mathcal{U}} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash \langle A, B \rangle : A \times B}$$

We can make use of Π-types to write this as $\vdash \times : \mathcal{U} \to \mathcal{U} \to \mathcal{U}$ and $\vdash \langle , \rangle : \Pi_{A:\mathcal{U}}\Pi_{B:\mathcal{U}} A \to B \to A \times B$

This is equivalent to the conjunction $A \wedge B$ before. The only difference is we are able to describe an extra 'induction rule' as an elimination rule though

there is no induction involved here.

$$\mathsf{ind}_{A\times B} : \Pi_{P:A\times B\to\mathcal{U}}(\Pi_{a:A}\Pi_{b:B}P(\langle a,b\rangle)) \to \Pi_{t:A\times B}P(t)$$

Actually the elimination rules pr_i are equivalent to the *recursor*

$$\mathsf{rec}_{A\times B} : \Pi_{C:\mathcal{U}}(A\to B\to C) \to A\times B\to C$$

Similarly we can define $\vee$ as a coproduct $A+B$ by inductively defining two constructors $\mathsf{inj}_1 : A\to A+B$ and $\mathsf{inj}_2 : B\to A+B$. Other types like $\mathbf{1}$, $\mathbf{0}$, $\mathbf{2}$ can all be defined by the same pattern. In a word, in this theory, we only need dependent functions and induction to define what we need. Thus we no longer need so many deduction rules.

The last example is how existential quantifier is defined. Suppose $A:\mathcal{U}$ and $B : A\to U$, then we have a type $\Sigma_{x:A}B(x)$ called dependent pair. The only constructor is still the pair

$$\langle , \rangle : \Pi_{a:A}(B(a)\to\Sigma_{x:A}B(x))$$

This means the information of $\Sigma_{x:A}B(x)$ contains an element $a:A$ and a proof $p:B(a)$ about the property B. So it is the intuitionistic way to define existential quantifier. This type is called dependent pair because it is like the product type, with the only different that the type of the second element of the pair dependent on the first element. The induction rule (elimination rule) is that

$$\mathsf{ind}_{\Sigma_{x:A}B(x)} : \Pi_{P:\Sigma_{x:A}B(x)\to\mathcal{U}}(\Pi_{a:A}\Pi_{b:B(a)}P(\langle a,b\rangle)) \to \Pi_{t:\Sigma_{x:A}B(x)}P(t)$$

such that

$$\mathsf{ind}_{\Sigma_{x:A}B(x)}\ C\ g\ \langle a,b\rangle :\equiv g\ a\ b$$

3.3 Homotopy Type theory

The type theory we have been developing is only a formal language without any interpretation like groups interpreted as sets. A very naive idea is to interpret each type as a set, and we interpret constructors as elements in this set. Then functions are normal functions between sets; type families are defined like fiber bundle; dependent functions are functions from the base space to the fiber bundle, and inductively defined types are well-founded trees as in descriptive set theory. This seems work, but it raises some difficulty to understand propositional equality.

For example, let's interpret $\mathbf{2}$ as $\mathbb{Z}_2$, with $0_{\mathbf{2}}$ as $0 \in \mathbb{Z}_2$ and $1_{\mathbf{2}}$ as $1 \in \mathbb{Z}_2$. We have four equalities: $0_{\mathbf{2}} = 0_{\mathbf{2}}$, $0_{\mathbf{2}} = 1_{\mathbf{2}}$, $1_{\mathbf{2}} = 0_{\mathbf{2}}$, $1_{\mathbf{2}} = 1_{\mathbf{2}}$. Clearly, only $0_{\mathbf{2}} = 0_{\mathbf{2}}$ and $1_{\mathbf{2}} = 1_{\mathbf{2}}$ should be provable. Thus we interpret them as two singleton sets $\mathbb{Z}_1$, i.e., the only possible way to prove them is the reflexivity $\mathsf{refl}_{0_{\mathbf{2}}}$, $\mathsf{refl}_{1_{\mathbf{2}}}$ and the other two as empty sets. We thus interpret $\mathsf{refl}_{0_{\mathbf{2}}}$ as $0 \in \mathbb{Z}_1$ and $\mathsf{refl}_{1_{\mathbf{2}}}$ as $0 \in \mathbb{Z}_1$. Since we only have refl in the interpretation of $0_{\mathbf{2}} = 0_{\mathbf{2}}$, it strongly recommends us to prove the following.

Definition 3.3.1 *(Uniquess of Identity Proofs) We define*

$$\mathsf{UIP} :\equiv \Pi_{A:\mathcal{U}}\Pi_{x,y:A}\Pi_{p,q:x=_A y} p =_{x=_A y} q$$

This is to say if p, q are proofs of equality $x =_A y$, then they should be equal as terms of type $x =_A y$. However, it is impossible to prove it from the induction rule $\mathsf{ind}_{=_A}$ for all A. If you interpret the universe $\mathcal{U}$ as a set containing $\mathbb{Z}_2$, then it is impossible to prove that $\Pi_{p,q:\mathbf{2}=_{\mathcal{U}}\mathbf{2}} p = q$.

Let's define a toy formal language to illustrate this issue with familiar concepts.

Definition 3.3.2 *We recursively define the language $\mathscr{G}$ by*

$$\mathscr{G} ::= a \mid b \mid e \mid (\mathscr{G} \cdot \mathscr{G}) \mid \mathscr{G}^{-1}$$

This is intended to be a free group generated by $\{a, b\}$. e is the identity element, and all other elements are like $((a \cdot a) \cdot b)$. $\cdot$ is the concatenation, i.e., the binary operation required by a group.

To fully make it a group, we add the following reduction rules. For any words x, y, z in $\mathscr{G}$, we define a relation $\to$ generated by the following rules.

1. $x \cdot (y \cdot z) \to (x \cdot y) \cdot z$
2. $x \cdot x^{-1} \to e$ *and* $x^{-1} \cdot x \to e$
3. $e \cdot x \to x$ *and* $x \cdot e \to x$
4. *If* $y \to z$, *then* $x \cdot y \to x \cdot z$, $y \cdot x \to z \cdot x$ *and* $y^{-1} \to z^{-1}$.

We further define $=$ to be the transitive, reflexive and symmetric closure of $\to$. Thus we have $x \cdot (y \cdot z) = (x \cdot y) \cdot z$, so we can omit the parentheses.

To interpret this formal language, you can find an abelian group G, and interpret a, b as elements in G. e is the identity element; $\cdot$ is the group operation of G; $^{-1}$ is the inverse in G. Clearly, in any interpretation G, we must admit $x \cdot y = y \cdot x$, but it is not derivation from the definition of $\mathscr{G}$.

Similarly in type theory, we should not interpret the equality in such a naive way. Martin Hofmann and Thomas Streicher gave a groupoid explanation to types [13]. A groupoid is a category where each morphism is invertible. Since a group can be understood as a category with only one object with all morphisms invertible, groupoid is a generalization of group. The idea of the groupoid interpretation is that each type is a groupoid and terms of this type are object in this groupoid, and the propositional equalities between them

are the morphisms. If two terms are judgementally equal, then they must be interpreted as the same object in the groupoid.

In the example of $\mathbf{2} =_{\mathcal{U}} \mathbf{2}$, the $\mathbf{2}$ is interpreted as a very trivial groupoid $\mathbb{Z}_2$ with two objects $0, 1$ and the only morphisms are identities representing reflexivities over the two terms $0_{\mathbf{2}}$, $1_{\mathbf{2}}$, while $\mathcal{U}$, interpreted as category of all groupoids with invertible functors as morphisms, has a non-identity isomorphism: $p : \mathbb{Z}_2 \to \mathbb{Z}_2$, $p(1) = 0, p(0) = 1$, $p(\mathsf{id}_1) = \mathsf{id}_0$, $p(\mathsf{id}_0) = \mathsf{id}_1$. Clearly $p \circ p$ is the identity functor, which shows the failure of UIP.

To make this interpretation reasonable, we have to prove the $=$ really behaviors like a groupoid.

Lemma 3.3.3 *For $x, y : A$ and $p : x = y$, we have $p^{-1} : y = x$, i.e. a function $(\cdot)^{-1} : x = y \to y = x$ such that $\mathsf{refl}_x^{-1} = \mathsf{refl}_x$. p^{-1} is the inverse of p.*

Proof: Let $D(x, y, p) :\equiv (y = x)$, $D : \Pi_{x,y:A}(x = y) \to \mathcal{U}$ and $d :\equiv (\lambda x.\mathsf{refl}_x) : \Pi_{x:A} D(x, x, \mathsf{refl}_x)$. Applying the induction principle, we get $p :\equiv \mathsf{ind}_{=_A}(D, d)\ x\ y\ p : (y = x)$. According to the computation rule, $\mathsf{refl}_x^{-1} :\equiv \mathsf{ind}_{=_A}(D, d)\ x\ x\ \mathsf{refl}_x \equiv d(x) \equiv \mathsf{refl}_x$. □

Lemma 3.3.4 *For $x, y, z : A$, $(\cdot) : (x = y) \to (y = z) \to (x = z)$ such that $\mathsf{refl}_x \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x$.*

Lemma 3.3.5 *Suppose $x, y, z, w : A$ and $p : x = y$, $q : y = z$, $r : z = w$. Then*

1. *$p = p \cdot \mathsf{refl}$, and $p = \mathsf{refl} \cdot p$.*
2. *$p^{-1} \cdot p = \mathsf{refl}$, $p \cdot p^{-1} = \mathsf{refl}$.*
3. *$(p^{-1})^{-1} = p$.*
4. *$p \cdot (q \cdot r) = (p \cdot q) \cdot r$*

Clearly refl, $(\cdot)$ and $(\cdot)^{-1}$ provide the necessary ingredients of a groupoid. However, in the above lemma, $p \cdot (q \cdot r) =_{x=z} (p \cdot q) \cdot r$ should also be interpreted

as a morphism in the groupoid $x = z$. This means they are not judgementally equal. So the groupoid interpretation is a weak ∞ one, which means the equality between morphisms is defined by an isomorphism in a higher order category, which again forms a groupoid.

The most natural example of such a weak ∞-groupoid is the fundamental groupoid of a topological space X. Traditionally, we define the fundamental groupoid of X with objects being all points and morphisms being equivalence classes of paths between points, where the equivalence is defined by *homotopy* $\simeq$. The necessarity of homotopy is the groupoid requirement such as the concatenate of a path and its inverse should be the identity at a point. In fact, we can replace the requirement $p \cdot p^{-1} = \mathsf{id}$ with a weaker $p \cdot p^{-1} \simeq \mathsf{id}$. Everything we have in category theory can be inherited by this weaker structure, while the higher dimensional structures can be captured by the homotopy between homotopies like $\pi_2(X)$.

This raises the homotopy interpretation of type theory. We interpret each type X as a space, terms as points, and equalities as paths between them. Note we do not really define a topological space and define the type as the weak ∞-groupoid of this space, but call it a space by specifying all morphisms in the groupoid. This is like the *final topology* of a space, where the open sets are defined such that a family of functions on this spaces are continuous. In other words, we first specify continuous functions (morphisms) and then decide the topological structures. Since we do not care about the point-set topological structures at all, we can use the morphisms to describe topological properties we needed.

For example, we can define some homotopical concepts as follows.

Definition 3.3.6 *For $f, g : \Pi_{x:A} P(x)$, the* **homotopy relation** *between f and g, written as $f \sim g$, is a dependent function*

$$f \sim g :\equiv \prod_{x:A} f(x) = g(x)$$

Definition 3.3.7 *If $f : A \to B$ is a function, we say that f is an* **equivalence** *provided there exist a $g : B \to A$ such that $f \circ g \sim \mathsf{id}_B$ and an $h : B \to A$ such that $h \circ f \sim \mathsf{id}_A$, i.e.,*

$$\mathsf{isequiv}(f) :\equiv \left(\sum_{g:B\to A} f \circ g \sim \mathsf{id}_B\right) \times \left(\sum_{h:B\to A} h \circ f \sim \mathsf{id}_A\right)$$

And we define

$$A \simeq B :\equiv \sum_{f:A\to B} \mathsf{isequiv}(f)$$

Remark 3.3.8 Note we do not define h and g to be the same to avoid some technical inconvenience [21]. ◊

Definition 3.3.9 *(Pointed Type) We call (A, a) a* **pointed type** *for $A : \mathcal{U}$, $a : A$. a is called the* **basepoint** *and we write $\mathcal{U}_\bullet :\equiv \sum_{(A:\mathcal{U})} A$ for the type of pointed types.*

Definition 3.3.10 *(Loop Space) Given any pair (A, a), we define the* **loop space** *as*

$$\Omega(A, a) :\equiv ((a =_A a), \mathsf{refl}_a).$$

*For $n : \mathbb{N}$, we define the n-***fold iterated loop space** *as*

$$\Omega^0(A, a) :\equiv (A, a)$$

$$\Omega^{n+1}(A, a) :\equiv \Omega^n(\Omega(A, a))$$

e.g.

$$\Omega^2(A, a) \equiv \Omega(\Omega((A, a))) \equiv \Omega(a =_A a, \mathsf{refl}_a) \equiv (\mathsf{refl}_a =_{a=_A a} \mathsf{refl}_a, \mathsf{refl}_{\mathsf{refl}_a}).$$

The function $f : A \to B$ and type family $P : A \to \mathcal{U}$ also get their homotopical interpretations.

Lemma 3.3.11 *For $f : A \to B$ and $p : x =_A y$, we have*

$$\mathsf{ap}_f : (x =_A y) \to (f(x) =_B f(y)).$$

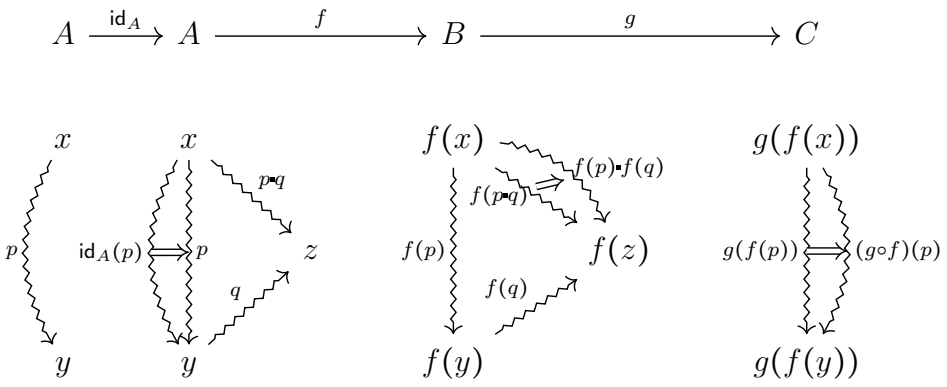
Furthermore, given $g : B \to C$ and $q : y =_A z$, we have

1. $\mathsf{ap}_f(p \cdot q) = \mathsf{ap}_f(p) \cdot \mathsf{ap}_f(q)$
2. $\mathsf{ap}_{\mathsf{id}_A}(p) = p$
3. $\mathsf{ap}_f(p^{-1}) = \mathsf{ap}_f(p)^{-1}$
4. $\mathsf{ap}_g(\mathsf{ap}_f(p)) = \mathsf{ap}_{g \circ f}(p)$

Proof: By leibniz rule. □

Remark 3.3.12 Sometimes we just write $f(p)$ for $\mathsf{ap}_f(p)$ for convenience. This behavior is called *type coercion* or *type casting*. Here the application of $f : A \to B$ to a term $p : x = y$ is not allowed by our deduction rules. Thus as long as you see such an 'error', you shall automatically apply the convention that $f(p)$ represents $\mathsf{ap}_f(p)$ to preserve the rigorousness of our formal language.

A common situation of type casting in traditional mathematics is using the underlying set of a group when the group is used for some set operations. ♢



This shows the functoriality of $f : A \to B$, but for dependent functions, the functoriality is not easy. For an $f : \Pi_{x:A} B(x)$ and $p : x =_A y$, $f(x) : B(x)$

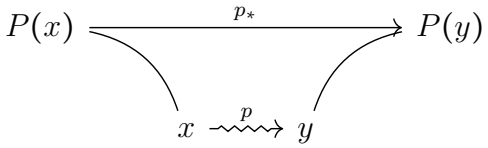
and $f(y) : B(y)$ lie in different types, so we cannot even define the equality between them. This difficulty raises the fibration of a type.

Recall the definition of tangent bundle or fiber bundle. For a space X, and $p : X$, we specify a tangent space $T_p(X)$ for each point. Thus the tangent bundle is the collection of all pairs (p, v) where $p \in X$ and $v \in T_p(X)$. Clearly, this is a dependent pair*. For a type family $P : A \to \mathcal{U}$, we may consider $P(a) : \mathcal{U}$ as the fiber over $a : A$. Thus $\Sigma_{a:A} P(a)$ is the bundle gathering all the fibers. Besides, the projection $\mathsf{pr}_1 : \Sigma_{a:A} P(a) \to A$ acts as the bundle projection, which suggests that we should call $\Sigma_{a:A} P(a)$ the **total space** and A the **base space**. We also call $P : A \to \mathcal{U}$ the fibration directly.

Thus to compare two elements in different fibers, we can analogously define the *transport* between them.

Lemma 3.3.13 *For $P : A \to \mathcal{U}$ and $p : x =_A y$, we have $p_* : P(x) \to P(y)$. Sometimes if we need to indicate the P explicitly, we write it as $\mathsf{transport}^P(p, -) : P(x) \to P(y)$*

Proof: Let $D(x, y, p) :\equiv P(x) \to P(y)$. We only have to prove $D(x, x, \mathsf{refl}_x) \equiv P(x) \to P(x)$ for all $x : A$, which is certainly proved by $\mathsf{id}_{P(x)}$. □



Lemma 3.3.14 *Properties of* transport.

- *For $P : A \to \mathcal{U}$, $p : x = y$, $q : y = z$, $u : P(x)$, we have $q_*(p_*(u)) = (p \cdot q)_*(u)$.*
- *For $f : A \to B$, $P : B \to \mathcal{U}$,$p : x = y$, $u : P(f(x))$, we have $\mathsf{transport}^{P \circ f}(p, u) = \mathsf{transport}^P(\mathsf{ap}_f(p), u)$.*

*If you think the p as a labelling to distinguish those v, then $\Sigma_{p:X} T_p(X)$ is the disjoint union of those $T_p(X)$ indexed by p, which explains the name Σ. Similarly, Π-types is analogous to the product of a type family.

- *For* $P, Q : A \to \mathcal{U}$, $f : \Pi_{x:A} P(x) \to Q(x)$, $p : x = y$, $u : P(x)$, *we have* $\mathsf{transport}^{Q}(p, f_x(u)) = f_y(\mathsf{transport}^{P}(p, u))$.

Remark 3.3.15 If we can prove $p : A =_{\mathcal{U}} B$, then $\mathsf{transport}^{\mathsf{id}_{\mathcal{U}}}(p) : A \to B$ gives us an equivalence $A \simeq B$. This shows our definition of equivalence is consistent with the groupoid interpretation. ♢

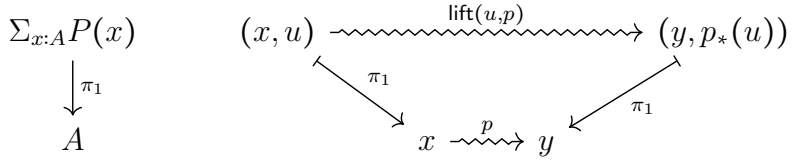
Now we can handle the dependent version of functoriality. First let's discuss the lift of a path in the total space.

Lemma 3.3.16 *For a fibration* $P : A \to U$ *and a path* $p : x = y$ *in the base space, given a starting point* $u : P(x)$, *we can define a lift*

$$\mathsf{lift}(u, p) : (x, u) = (y, p_*(u))$$

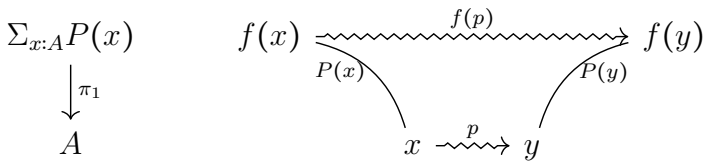
in the total space $\Sigma_{x:A} P(x)$ *such that* $\mathsf{pr}_1(\mathsf{lift}(u, p)) = p$.

Remark 3.3.17 In $\Sigma_{x:A} P(x)$, we should think (x, u) as the starting point, but since $u : P(x)$, giving u is equivalent to giving the tuple (x, u). $\mathsf{lift}(u, p) : (x, u) = (y, p_*(u))$ is a path in the total space. According to the functoriality, $\mathsf{pr}_1 : \Sigma_{x:A} P(x) \to A$ can be extended to $\mathsf{ap}_{\mathsf{pr}_1} : (=_{\Sigma_{x:A} P(x) \to A}) \to (=_A)$, so $\mathsf{pr}_1(\mathsf{lift}(u, p))$ is actually $\mathsf{ap}_{\mathsf{pr}_1}(\mathsf{lift}(u, p)) : \mathsf{pr}_1(x, u) = \mathsf{pr}_1(y, p_*(u))$. ♢



Proof: By setting $D(x, y, p) :\equiv (x, u) = (y, p_*(u))$ □

Second, note that by setting $f'(x) :\equiv (x, f(x)) : A \to \Sigma_{x:A} P(x)$, we can apply the functoriality for $f : \Pi_{x:A} P(x)$ as in the following diagram, where $f(x)$ means $(x, f(x))$ and $f(p)$ is $\mathsf{ap}_{f'}(p)$.



In this solution, we abandoned the information about the $p : x = y$, which may be important sometimes. Since we also have a $\mathsf{lift}(f(x), p) : (x, f(x)) = (y, p_*(f(y)))$, it is very natural for us to think about the equality $(y, f(y)) = (y, p_*(f(y)))$.

Lemma 3.3.18 *For $f : \Pi_{a:A}P(x)$, we have*

$$\mathsf{apd}_f : \prod_{p:x=y} p_*(f(x)) =_{P(y)} f(y)$$

Proof: Let $D(x, y, p) :\equiv p_*(f(x)) = f(y)$. □

More generally, for $u : P(x)$ and $v : P(y)$, we can call the equality $p_*(u) =_{P(y)} v$ *a path from u to v lying over p.*

The homotopical interpretation gives us the power to described a topological structures as types. For example, the homotopical circle $\mathbb{S}$ is defined by a base point and a loop (path, i.e. the equality) between the basepoint:

$$\mathbb{S} :\equiv \begin{cases} \mathsf{base} : \mathbb{S} \\ \mathsf{loop} : \mathsf{base} = \mathsf{base} \end{cases}$$

This further shows the failure of UIP. In fact, the type $=_A$ is defined as a family, not a single type in $\mathcal{U}$. We use the induction rule for the family $C : \Pi_{x,y:A} x = y \to \mathcal{U}$, not $\Pi_{x:A} x = x \to \mathcal{U}$. So we interpret $x = x$ as a path with fixed endpoints like the loop in $\mathbb{S}$.

To calculate its fundamental group, we find the universal covering space of it, namely a family $\mathsf{cov} : \mathbb{S} \to \mathcal{U}$ by fibration. The path loop will then raises a lift in the total space $\sum_{x:\mathbb{S}} \mathsf{cov}(x)$. According to our knowledge in algebraic

topology, it is reasonable to define $\mathsf{cov}(\mathsf{base}) :\equiv \mathbb{Z}$. Since $\mathbb{S}$ is inductively defined by a base point and a path, we have to find an evidence of $\mathsf{cov}(\mathsf{loop}) : \mathbb{Z} = \mathbb{Z}$.

There is a difficulty to define the equality $\mathbb{Z} = \mathbb{Z}$. After we move along the path loop, the covering space should also shift a unit length. So this equality should be raised by the $(+1)$ function, a bit like the non-identity equality $\mathbf{2} =_{\mathcal{U}} \mathbf{2}$.

According to the groupoid interpretation, each $=_{\mathcal{U}}$ will raise an isomorphism between two groupoids, which is formalized by the equivalence $\simeq$ in our theory. What about the converse? In pure type theory, the converse is certainly not provable. But in traditional mathematics, if two objects are equivalence, i.e., if there is an isomorphism between them, we shall think them as identitcal. Like in group theory, we often use two totally different but isomorphic groups alternatively. Thus we may consider introduce the converse as an axiom to use the equivalence as equality conveniently as in traditional mathematics. This was introduced by Voevodsky as the univalence axiom.

Definition 3.3.19 *(Univalence Axiom).*

$$\mathsf{univalence} : (A \simeq B) \simeq (A =_{\mathcal{U}} B)$$

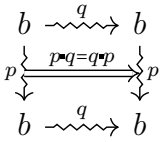
Since each isomorphism is a kind of equality in the groupoid interpretation, this axiom forces every type to be defined in such a pattern.

According to the univalence axioms, we can use the equivalence like the $(+1)$ or $0_{\mathbf{2}} \mapsto 1_{\mathbf{2}}, 1_{\mathbf{2}} \mapsto 0_{\mathbf{2}}$ as equality. Clearly, we use $(+1)$ and (-1) as the equivalence defining the equality $\mathbb{Z} = \mathbb{Z}$.

Now the space $\sum_{x:\mathbb{S}} \mathsf{cov}(x)$ does form a covering space of the homotopical sphere $\mathbb{S}$. The the transport raised by a path $p : x =_{\mathbb{S}} y$ in this covering space will be the winding process. Thus we define $\mathsf{wind} : \Omega(\mathbb{S}, \mathsf{base}) \to \mathbb{Z}$ such that for any $p : \mathsf{base} = \mathsf{base}$, $\mathsf{wind}(p) = p_*(0)$, which is the inverse of $n \mapsto \mathsf{loop}^n : \mathbb{Z} \to \Omega(\mathbb{S}, \mathsf{base})$. So we can use the univalence axiom again to

conclude $\Omega(\mathbb{S}, \mathsf{base}) = \mathbb{Z}$.

Similarly, a torus is defined by a base point b with two paths $p, q : b = b$, together with a homotopy $H : p \cdot q = q \cdot p$.



And a projective plane is a base point b with one path $p : b = b$ and one homotopy $H : p = p^{-1}$.

Appendix A

Independence of Logical Axioms

We have introduce three logical axioms for classical propositional logic.

1. $\alpha \to (\beta \to \alpha)$
2. $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$
3. $(\neg\beta \to \neg\alpha) \to ((\neg\beta \to \alpha) \to \beta)$

I have already shown that the third is not an intuitionistic tautology by Heyting algebra. So now I am going to prove the first one cannot be proved by the proof system in 1.1.14 with only the other two as logical axioms.

Suppose we can prove the first one from the other two, i.e., we have a proof sequence $\varphi_1, \varphi_2, \cdots, \varphi_n$ where φ_n is $A \to (B \to A)$ for variables A and B, and other φ_i are of the form of the second or the third one, or derived from MP.

Let v be an assigment $v : V \to \{0, 1, 2\}$ and $v(A) = 0$, $v(B) = 1$. Here we define a 'triple-value' semantics for the formulas, which does not have any concrete meaning. Similarly, we define $\neg$ and $\to$ as functions $\{0, 1, 2\}^n \to \{0, 1, 2\}$ by listing all possibilities of input (A.1).

We also extend v to all formulas. Thus $v(A \to (B \to A)) = 2$ according to this semantics. It can be verified that for any input, other two axioms can only be evaluated as 0 or 1 according to the assignment v. That means $v(\varphi_1) \neq 2$ and $v(\varphi_2) \neq 2$. Besides, if φ_k is derived by MP, i.e., there are $i, j < k$ such that $\varphi_j = \varphi_i \to \varphi_k$, and $v(\varphi_i) \neq 2$ and $v(\varphi_j) \neq 2$, then $v(\varphi_k)$ cannot be 2. Thus the whole proving sequence is evaluated to 0 or 1, which is a contradiction.

Table A.1: Independence of Logical Axioms

$\neg$	a
0	0
0	1
1	2

a	$\rightarrow$	b
0	0	0
0	0	1
0	2	2
1	2	0
1	0	1
1	2	2
2	0	0
2	0	1
2	0	2

Bibliography

[1] *All about monads*. https://wiki.haskell.org/All_About_Monads.

[2] *Haskell*. https://www.haskell.org/.

[3] Andrew W Appel, *Compiling with continuations*, Cambridge university press, 2007.

[4] Steve Awodey, *Category theory*, Oxford university press, 2010.

[5] Steve Awodey, *Univalence as a principle of logic*, Indagationes Mathematicae, Volume 29 (2018), pp. 1497–1510.

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

[7] Alonzo Church, *The calculi of lambda-conversion*, Princeton University Press, 1985.

[8] Koen Claessen, *A poor man's concurrency monad*, Journal of Functional Programming, 9 (1999), pp. 313–323.

[9] Thierry Coquand, *The paradox of trees in type theory*, BIT, 32 (1992), pp. 10–14.

[10] Haskell B Curry and Feys Robert, *Combinatory logic, vol. 1*, 1958.

[11] Peter Dybjer, *Inductive families*, Formal aspects of computing, 6 (1994), pp. 440–465.

[12] Valery Glivenko, *Sur quelques points de la logique de m. brouwer*, Bulletins de la classe des sciences, 15 (1929), pp. 183–188.

[13] Martin Hofmann and Thomas Streicher, *The groupoid interpretation of type theory*, Twenty-five years of constructive type theory (Venice, 1995), 36 (1998), pp. 83–111.

[14] William A Howard, *The formulae-as-types notion of construction*, To HB Curry: essays on combinatory logic, lambda calculus and formalism, 44 (1980), pp. 479–490.

[15] Bart Jacobs, *Categorical Logic and Type Theory*, vol. 141 of Studies in Logic and the Foundations of Mathematics, Elsevier Science, 1998.

[16] Joachim Lambek and P. J Scott, *Introduction to higher order categorical logic*, vol. 7 of Cambridge studies in advanced mathematics, Cambridge Cambridgeshire, England; New York:Cambridge University Press, c1986.

[17] Elliott Mendelson, *Introduction to mathematical logic*, CRC press, 2009.

[18] Paul Francis Mendler, *Inductive definition in type theory*, tech. report, Cornell University, 1987.

[19] Robert K Meyer, Richard Routley, and J Michael Dunn, *Curry's paradox*, Analysis, 39 (1979), pp. 124–128.

[20] J.R. Munkres, *Topology*, Topology, Prentice-Hall, 2000.

[21] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, 2013. (This is a live updating book written by a bunch of people https://github.com/HoTT/book and is freely available at https://homotopytypetheory.org/book/).

[22] D. Singh and J. N. Singh, *von neumann universe: A perspective*, International Journal of Contemporary Mathematical Sciences, Vol. 2 (2007), pp. 475 – 478.

[23] Morten Heine B. Sørensen and Pawel Urzyczyn, *Lectures on the Curry-Howard Isomorphism*, vol. 149 of Studies in Logic and the Foundations of Mathematics, Elsevier Science, 1 ed., July 2006.

[24] Varmo Vene, *Categorical programming with inductive and coinductive types*, Citeseer, 2000.