Ludwig-Maximilians-Universität München
Faculty for Mathematical Logic

Bachelor's Thesis

Voevodsky's Axiom of Univalence

Armin Eghdami

Ludwig-Maximilians-Universität München
Fakultät für Mathematische Logik

Bachelorarbeit

Voevodskys Axiom der Univalenz

Vorgelegt von:
Armin Eghdami
geboren am 4. April 1997 in München

Betreuer:
Dr. Iosif Petrakis

München, den 29. Mai 2018

Contents

Abstract

Martin-Löf intensional type theory (MLITT) is a formal system for constructive mathematics that was developed by Martin-Löf in the 70's and the 80's. Univalent type theory is the extension of MLITT with Voevodsky's axiom of univalence (UA) and it is based on the homotopic interpretation of MLITT. Chapter 1 of this thesis is a short introduction to MLITT. Chapter 2 is an account of Voevodsky's UA, while chapter 3 includes some basic corollaries of UA. The final chapter of this thesis is devoted to the presentation of Voevodsky's proof of function extensionality from UA.

1 Introduction to Martin-Löf type theory

MLITT, see [7, 8, 9], was conceived by Martin-Löf as a formal system for Bishop's constructive mathematics that was developed in [2]. A special feature of MLITT is, that it is logic free, i.e. logic is built into the system, hence MLITT is not based on first order predicate logic. Another key feature of MLITT is the concept of propositional equality, which we will briefly touch on in the subsection on dependent function types of section 1.7.2 and then formally introduce in section 1.15. This new type of equality coexists alongside the familiar definitional or judgmental equality, which we will present in section 1.2. This first chapter is based on the formal treatment of type theory as found in the appendix of the HoTT-book [11]. Starting from section 1.16, which is based on chapter 2 of [11], we will allow ourselfs to treat MLITT more informally, as done in the main part of [11]. In contrast to [11], we will also give an inductive definition for the function type, as found in [10]. The distinction between the different possible definitions of the function type will be further explained in section 1.7.

The univalent interpretation of MLITT was originally based on Voevodsky's homotopical interpretation of MLITT in [12], as well as Awodey and Warren's work in [1]. It was motivated by Hofmann and Streicher's groupoid interpretation of intensional type theory, given in [4]. The classical model of univalent foundations in the category of simplicial sets, developed in [6], motivated the introduction of UA. Following [11], we will present UA in chapter 2 and give some corollaries of it in chapter 3.

Chapter 4 is devoted to the proof of function extensionality from UA. To this end, we prove the new lemmas 4.3.5, 4.3.6, 4.3.10 and give more detailed versions of proofs that can be found in [11] in lemmas 4.3.8 and 4.3.9. The main theorems of [11], that constitute the proof that UA implies function extensionality, are presented in a fully worked out and detailed version in section 4.4.

1.1 Basic structures of type theory

The type theory we use in this thesis will consist of two basic concepts: propositions and judgments. Propositions are internal statements which we can prove or disprove but also assume, negate, construct statements like "if ..., then ..." out of them, etc. In type theory, propositions will be represented by types. That means that each proposition in

our familiar mathematical language can be translated into a specific type and vice versa. Once we have introduced our basic types, we will explicitly give the correspondence between statements like "if A or B, then there exists ..." and their respective types in section 1.14.

There are three types of judgments which we will use in this thesis:

$$\Gamma \text{ ctx} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash a \equiv a' : A.$$

Judgments are on a higher level than propositions. They are part of the deductive system of MLITT, i.e. belong to its metatheory. We can't "prove" them and we also can't form statements like "if $a : A$, then $b : B$". In order to obtain a specific judgment we will use certain rules, so called inference rules to derive it step by step from other judgments which we have already derived. Judgments which we use in our deduction, but that were not derived themselves at some point, will be called axioms. A deduction will be of the form

$$\frac{J_1 \qquad \dots \qquad J_n}{J} \text{ Name}.$$

It derives a conclusion, the new judgment J, from a number of hypotheses, the already before derived judgments $J_1, \dots, J_n$. The name of the inference rule which allows us to conduct this derivation will always be written on the right side of the line. We can also chain such derivations, by replacing a hypotheses, by its own derivation. Through this process we get so called derivation trees.

1.2 Typing judgments and equality judgments

In the following sections we will explain the meanings of the three judgments listed above. Note that we will talk about $a : A$ and $a \equiv a' : A$ without "$\Gamma \vdash$" in front of them. Strictly speaking these aren't the "correct" judgments in a completely formal way, but since in the later sections of the thesis we will often abbreviate the formal judgments to this form, we will stick to the shorter version for now. $a : A$ expresses that "a has type A". It is called the typing judgment. When we have the judgment $a : A$, we will refer to a as "an element", "a witness", "an inhabitant", "a point" or "a proof" of A. We will always write uppercase letters for types and lowercase letters for elements.

The judgment $a \equiv a' : A$ can be read as "a and a' are judgmentally equal in the type A" and is called the equality judgment. We will sometimes abbreviate this and simply write $a \equiv a'$ instead. Note that our type system has the property that $a \equiv a' : A$ implies $a : A$ and $a' : A$.

In order to proof a proposition in MLITT, we need to "find" an element of the corresponding type in the following way. Let A be the type of the proposition we want to prove. We now have to move via our inference rules from one judgment to the next, until we arrive at a judgment $a : A$. We will refer to this process as "constructing", "forming" or "finding" an element of A, or as "inhabiting" A. This element a constitutes the proof of the proposition, i.e. in order to prove a proposition we must show that its type is inhabited, meaning that is has at least one element. Even more precisely, it is

not the element a itself which proves the proposition, but rather the fact that we can derive the judgment $a : A$.

1.3 Contexts

A context is an ordered list of the form

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n,$$

where each $x_i : A_i$ of the context can be read as the assumption that the variable of name x_i is of type A_i. Note that all variables $x_1, x_2, \ldots, x_n$ must be distinct from one another. We will often abbreviate this list of assumptions with uppercase greek letters like Γ or Δ. We will use the notation

$$\Gamma \vdash a : A \qquad \text{and} \qquad \Gamma \vdash a \equiv a' : A$$

to denote that under the assumptions of the context Γ we have the judgment $a : A$ or $a \equiv a' : A$ respectively. Instead of explicitly writing down the context or saying "under the assumptions of the context $a : A$", we will often simply use phrasings like "for a given $a : A$", "assume $a : A$" or "let $a : A$". Our context Γ may also be empty, i.e. contain no assumptions, in which case we write $\vdash a : A$ or $\cdot \vdash a : A$ (and similarly for the equality judgment).

The third and last judgment

$$(x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n) \text{ ctx}$$

expresses the fact that the context $x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$ is a so called well-formed context. The judgments $\Gamma \vdash a : A$ and $\Gamma \vdash a \equiv a' : A$ only make sense, if Γ is well-formed. For our current case well-formed means that each A_i is a type in the context of $x_1 : A_1, \ldots, x_{i-1} : A_{i-1}$, meaning that A_i may only contain the variables $x_1, \ldots, x_{i-1}$ and no others. This is the reason why the order in which the assumptions of a context are listed is important. In the same way

$$(x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n) \vdash a : A$$

together with

$$(x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n) \text{ ctx}$$

tells us that a and A must only contain the variables $x_1, \ldots, x_n$. A context therefore shows us the stock of all currently available variables, together with their respective types. For the context judgment there is the inference rule

$$\frac{(x_1 : A_1, \ldots, x_n : A_n) \text{ ctx}}{x_1 : A_1, \ldots, x_n : A_n \vdash x_i : A_i}\ \text{Var},$$

which expresses the fact, that under the assumption of $x_i : A_i$ we may derive exactly this typing judgment.

1.4 Universes

Every object in type theory must be of some type. A type must therefore itself also be an element of some higher level type. Such types whose elements are types are called universes. The fact that each universe itself must also be of some type, leads us to an infinite hierarchy of universes since we can always go "one level higher". We denote universes by the symbol $\mathcal{U}_i$ with the index i displaying its "level" regarding the other universes. Universes obey the inference rules

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}\ \mathcal{U}-\text{Intro}$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_{i+1}}\ \mathcal{U}-\text{Cumul},$$

which express the fact that each universe, as well as all its elements, is an element of the universe one level above. Our type system has the property that each type must automatically be part of some universe, i.e. from $\Gamma \vdash a : A$ it is implied that there is some level i so that $\Gamma \vdash A : \mathcal{U}_i$. Later on in this thesis we will stop explicitly indexing our universes, though one can always reintroduce an appropriate hierarchy to check the correctness of an expression.

1.5 Substitution

Given an element b or a type B, involving some variable $x : A$, we will for an element $a : A$ write $b[x/a]$ for the element or $B[x/a]$ for the type, in which all free occurrences of x were substituted by a. Note that until now all occurrences of variables were free. In section 1.7 we will encounter an instance where variables are bound to an expression, i.e. are no longer free. We will sometimes abuse notation and write $x \equiv a$, or "assume x is a" to denote this substitution. The formal inference rules of substitution for the typing and the equality judgment are as follows:

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A, \Delta \vdash b : B}{\Gamma, \Delta[x/a] \vdash b[x/a] : B[x/a]}\ \text{Subst}_1$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A, \Delta \vdash b \equiv c : B}{\Gamma, \Delta[x/a] \vdash b[x/a] \equiv c[x/a] : B[x/a]}\ \text{Subst}_2.$$

1.6 General procedure for type forming

When introducing a new fundamental type, we will always follow a series of steps with a fixed order. By proceeding in this fashion we can we be certain that everything is formally correct and consistent. First we must give a rule on how to form that type, i.e. an inference rule with the conclusion that our new type is the element of some universe. This rule will be called the **formation rule**. The next step is to give a rule on how to construct an element of this type, i.e. we must give an inference rule with a typing

judgment of our new type as the conclusion. This will be called the **introduction rule** and the elements constructed by it are called canonical elements. After this we will give a so called **elimination rule** which is an inference rule on how to "use" an element of that type and lastly we will give a **computation rule**, which will determine how the elimination rule acts on its canonical elements.

This series of steps appears to be the natural order, by first constructing the type, then the elements and then showing how to use the elements. In almost all cases though, instead of an elimination and a computation rule, we will have a so called **induction principle**, which, roughly explained, does the following: for any proposition that may depend on an element of our type, the induction principle gives us a proof of this proposition for any arbitrary element, given that we have a proof for every canonical element. In other words, if we have proved something for the canonical elements, the induction principle states that we have then already proved it for all possible elements of our type. When the proposition doesn't depend on our element, then the induction principle is called the recursion principle instead.

We may also have an additional rule, the so called **uniqueness principle**, which is a judgmental equality between an arbitrary element of the new type and a canonical element. In almost all cases though, we won't use a uniqueness principle, but rather prove a theorem of similar nature, without having to include any additional rule.

1.7 Function types ($\rightarrow$- and $\prod$-type)

We will give two different approaches on how to formally define the function type, whose elements are called functions. The first one is the ad hoc definition, in which an applied function is simply a completely new object by itself. The second one, the inductive definition, explicitly defines what an applied function is, using a recursion principle. This second approach is more in harmony with the pattern of type introduction that is usually used, but in turn requires us to take the primitive concept of transformations as already given.

1.7.1 The ad hoc definition

We will start by defining the more general dependent function type and then introduce the non-dependent function type as a special case.

Dependent function types

Giving the following inference rules by themselves without any further comments would be enough since all the information is already contained in them. Since this is the first time that we introduce a type following the scheme of section 1.6 though, there's a short explanation given for each rule. The $\prod-$Form rule

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x : A \vdash B : \mathcal{U}_i}{\Gamma \vdash \prod_{(x:A)} B : \mathcal{U}_i} \prod\text{-Form}$$

states that we write $\prod_{(x:A)} B$ for our dependent function type from A to B. As one can see from the hypotheses, B is a type which may depend on a variable $x : A$. Such "dependent" types will be called type families. Note that $\prod_{(x:A)}$ binds the variable x so that it is now no longer free in B, i.e. it is no longer susceptible to substitution. A will be referred to as the domain and B as the codomain.

The $\prod -$Intro rule

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : \prod_{(x:A)} B} \prod -\text{Intro}$$

states that for a variable $x : A$ and element $b : B$, where both b and B may depend on the variable x, we have the dependent function $\lambda(x : A).b$ as an element of our type $\prod_{(x:A)} B$. Elements of the form $\lambda(x : A).b$ are therefore our canonical elements. As above, $\lambda(x : A).b$ binds x, so that it is now no longer free in b. If the type is clear, we will sometimes omit it and simply write $\lambda x.b$ instead of $\lambda(x : A).b$.

The $\prod -$Elim rule

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[x/a]} \prod -\text{Elim}$$

states how one "applies" a dependent function by deriving from a dependent function f and an element $a : A$ the typing judgment for the element $f(a)$. This element can be thought of as the function f applied to a, giving back an element of $B[x/a]$, which is the type that comes from substituting the element a for the variable x in the type family B.

The $\prod -$Comp rule

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x : A).b)(a) \equiv b[x/a] : B[x/a]} \prod -\text{Comp}$$

states how one "applies" the canonical element. This is done by deriving the equality judgment which states that the dependent function applied to the element $a : A$ is equal to the element $b : B$ with the variable x substituted with a. We will also sometimes abuse notation a bit and instead of $\lambda(x : A).b$, directly refer to b as a function, writing $b : \prod_{(x:A)} B$ and also $b(a)$ instead of $(\lambda(x : A).b)(a)$ resulting in the equality $b(a) \equiv b[x/a]$.

The $\prod -$Uniq rule

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B}{\Gamma \vdash f \equiv (\lambda x.f(x)) : \prod_{(x:A)} B} \to -\text{Uniq or } \eta-\text{rule}$$

states that each element of the dependent function type is judgmentally equal to some canonical element by deriving the equality between the function f and its corresponding canonical element $(\lambda x.f(x))$. This special uniqueness principle is also called the $\eta-$rule.

Non-dependent function types

As described above, the type B may depend on a variable $x : A$. If this is not the case, i.e. if x doesn't occur in B at all or doesn't occur freely, we call the resulting type non-dependent function type. B is then no longer a type family with respect to x but simply an unchanging, i.e. non-dependent type. For such a B, we will write the non-dependent function type as $A \to B :\equiv \prod_{(x:A)} B$, where ":≡" denotes a definitional equality, i.e. we define a new object as being equal to an already known one. By omitting the x dependencies of B (note that b is still dependent on x) we can write down simplified inference rules for our non-dependent function type:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash A \to B : \mathcal{U}_i} \to -\text{Form}$$

$$\frac{\Gamma, B : \mathcal{U}_i, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : A \to B} \to -\text{Intro}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B} \to -\text{Elim}$$

$$\frac{\Gamma, B : \mathcal{U}_i, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x : A).b)(a) \equiv b[x/a] : B} \to -\text{Comp}$$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f \equiv (\lambda x.f(x)) : A \to B} \to -\text{Uniq or } \eta-\text{rule}.$$

We can use a type family B to give an example of a non-dependent function. Recall that the contexts at the beginning of each tree represent our assumptions. We start by deriving that for a given type A, we have the type $A \to \mathcal{U}_i$ of non-dependent functions:

$$\frac{\dfrac{\dfrac{(A : \mathcal{U}_i)\ \text{ctx}}{A : \mathcal{U}_i \vdash A : \mathcal{U}_i}\ \text{Var}}{A : \mathcal{U}_i \vdash A : \mathcal{U}_{i+1}}\ \mathcal{U}-\text{Cumul} \qquad \dfrac{(A : \mathcal{U}_i)\ \text{ctx}}{A : \mathcal{U}_i \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}\ \mathcal{U}-\text{Intro}}{A : \mathcal{U}_i \vdash A \to \mathcal{U}_i : \mathcal{U}_{i+1}} \to -\text{Form}.$$

Next we inhabit that type with

$$\frac{\dfrac{(B : \mathcal{U}_i, \mathcal{U}_i : \mathcal{U}_{i+1}, x : A)\ \text{ctx}}{B : \mathcal{U}_i, \mathcal{U}_i : \mathcal{U}_{i+1}, x : A \vdash B : \mathcal{U}_i}\ \text{Var}}{B : \mathcal{U}_i \vdash \lambda(x : A).B : A \to \mathcal{U}_i} \to -\text{Intro}$$

and deduce an equality about the application of the resulting function:

$$\frac{\dfrac{(a : A, B : \mathcal{U}_i, \mathcal{U}_i : \mathcal{U}_{i+1}, x : A)\ \text{ctx}}{a : A, B : \mathcal{U}_i, \mathcal{U}_i : \mathcal{U}_{i+1}, x : A \vdash B : \mathcal{U}_i}\ \text{Var} \qquad \dfrac{(a : A, B : \mathcal{U}_i)\ \text{ctx}}{a : A, B : \mathcal{U}_i \vdash a : A}\ \text{Var}}{a : A, B : \mathcal{U}_i \vdash (\lambda(x : A).B)(a) \equiv B[x/a] : \mathcal{U}_i} \to -\text{Comp}.$$

As mentioned above we will simply write $B : A \to \mathcal{U}_i$ instead of $\lambda(x : A).B : A \to \mathcal{U}_i$ and $B(a) : \mathcal{U}_i$ instead of $(\lambda(x : A).B)(a)$. Using this notation we get $B(a) \equiv B[x/a]$. Since B was a type family, we may say that type families are non-dependent functions of type $A \to \mathcal{U}_i$ for some type $A : \mathcal{U}_i$. We will often use the notation $\prod_{(x:A)} B(x)$ instead of $\prod_{(x:A)} B$ to highlight that B is a type family dependent on $x : A$.

1.7.2 The inductive definition

As mentioned in the beginning of this section, we will now introduce the function type in a more natural way, analogously to [10]. The conventions and general concepts mentioned in the ad hoc definition still hold, but the inference rules and the fundamental way in which function application is defined, change. We will first start with the non-dependent function type and then generalize everything for the dependent version.

Non-dependent function types

We will often write $b(x)$ instead of b to indicate the term's dependency on the variable x and $b(a)$ instead of $b[x/a]$. We also assume the concept of so called transformations, which are objects that take certain inputs and transform them into an element of some type. The $\to -$Intro and $\to -$Form rules remain unchanged:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash A \to B : \mathcal{U}_i} \to -\text{Form}$$

$$\frac{\Gamma, B : \mathcal{U}_i, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : A \to B} \to -\text{Intro}.$$

Instead of the ad hoc elimination and computation rule, we use the following recursion and induction principle:

$$\frac{\Gamma, x : A, b(x) : B \vdash p(x, b(x)) : C \qquad \Gamma \vdash f : A \to B}{\Gamma \vdash \text{rec}_{A\to B}(p(x, b(x)), f) : C} \to -\text{rec}$$

$$\frac{\Gamma, x : A, b(x) : B \vdash p(x, b(x)) : P(\lambda(x : A).b(x)) \qquad \Gamma \vdash f : A \to B}{\Gamma \vdash \text{ind}_{A\to B}(p(x, b(x)), f) : P(f)} \to -\text{ind},$$

with p being a transformation. We will often refer to the result of the induction principle as the induction function and to that of the recursion principle as the recursion function or the recursor, since for all practical purposes, we can treat $\text{rec}_{A\to B}$ and $\text{ind}_{A\to B}$ just like functions. In the later sections we will even write them as elements of some function type, since they effectively behave just in the same way. For our canonical functions we compute $\text{rec}_{A\to B}$ and $\text{ind}_{A\to B}$ as follows:

$$\text{rec}_{A\to B}\Big(p(x, b(x)), \lambda(x : A).b(x)\Big) :\equiv p(x, b(x))$$
$$\text{ind}_{A\to B}\Big(p(x, b(x)), \lambda(x : A).b(x)\Big) :\equiv p(x, b(x)).$$

We now want to define function application via our recursor. Using the Var rule, we can derive $a : A, b(a) : B \vdash b(a) : B$. Therefore if we interpret $b(a)$ as our transformation, using the $\to -$rec rule, we get $\text{rec}_{A\to B}(b(a), f) : B$ for any arbitrary function $f : A \to B$. Since $\text{rec}_{A\to B}(b(a), f)$ is in B, which is the type of our codomain, we can define the applied function as

$$f(a) :\equiv \text{rec}_{A\to B}(b(a), f),$$

given that we have $a : A$ and $b(a) : B$. For our canonical elements we can simplify this by using the computational rule of the recursor, obtaining

$$\Big(\lambda(x : A).b(x)\Big)(a) :\equiv b(a).$$

Dependent function types

Now we generalize the above for dependent codomains, i.e. for B being a type family:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x : A \vdash B : \mathcal{U}_i}{\Gamma \vdash \prod_{(x:A)} B : \mathcal{U}_i}\ \prod\text{−Form}$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : \prod_{(x:A)} B}\ \prod\text{−Intro}$$

$$\frac{\Gamma, x : A, b(x) : B(x) \vdash p(x, b(x)) : C \qquad \Gamma \vdash f : \prod_{(x:A)} B(x)}{\Gamma \vdash \mathrm{rec}_{\prod_{(x:A)} B(x)}(p(x, b(x)), f) : C}\ \prod\text{−rec}$$

$$\frac{\Gamma, x : A, b(x) : B(x) \vdash p(x, b(x)) : P(\lambda(x : A).b(x)) \qquad \Gamma \vdash f : \prod_{(x:A)} B(x)}{\Gamma \vdash \mathrm{ind}_{\prod_{(x:A)} B(x)}(p(x, b(x)), f) : P(f)}\ \prod\text{−ind}.$$

As one can see, the first two rules are again identical to the ad hoc definition and the recursion and induction principles are analogous to the non-dependent function case. For canonical functions we compute $\mathrm{rec}_{\prod_{(x:A)} B(x)}$ and $\mathrm{ind}_{\prod_{(x:A)} B(x)}$ in the same way as before:

$$\mathrm{rec}_{\prod_{(x:A)} B(x)}\Big(p(x, b(x)), \lambda(x : A).b(x)\Big) :\equiv p(x, b(x))$$
$$\mathrm{ind}_{\prod_{(x:A)} B(x)}\Big(p(x, b(x)), \lambda(x : A).b(x)\Big) :\equiv p(x, b(x)).$$

Using the induction function, we can now again define function application

$$f(a) :\equiv \mathrm{ind}_{\prod_{(x:A)} B(x)}(b(a), f),$$

given that we have $a : A, b(a) : B(a)$ and $f : \prod_{(x:A)} B(x)$. Note that this time we need the induction function instead of the recursor, since the codomain of our function varies with our input variable. As before, the application of the canonical elements is defined via

$$\Big(\lambda(x : A).b(x)\Big)(a) :\equiv b(a).$$

Instead of the $\eta-$rule of the ad hoc definition, we can now prove a much more general proposition, namely, that for each element of the function type there is a canonical element which is propositionally equal to it. For this we must shortly touch on what propositional equality between objects, which we will denote by the sign "=", exactly is. The type corresponding to the proposition "a is equal to b" is the so called equality type $a =_A b$, where the subscript indicates the type of both a and b. Two elements $a : A$ and

$b : A$ are called propositionally equal in the type A, if we can inhabit the corresponding equality type. We will introduce the equality type in more detail later on in section 1.15, but for now, all we need to know is, that for any type $A : \mathcal{U}$ and element $a : A$ we have the element $\text{refl}_a : a =_A a$, i.e. we know that any element is propositionally equal to itself. With this knowledge we can now prove our desired proposition.

Theorem 1.7.1 *Let $A : \mathcal{U}$ and $B(x) : \mathcal{U}$ be a type family of A. For all functions $f : \prod_{(x:A)} B(x)$ the type*

$$f =_{\prod_{(x:A)} B(x)} \lambda(x : A).f(x)$$

is inhabited.

Proof: To prove this we need to find a function of type

$$\prod_{(f:\prod_{(x:A)} B(x))} \Big(f =_{\prod_{(x:A)} B(x)} \lambda(x : A).f(x)\Big).$$

This is done by using the induction principle of the dependent function type. We define P with $P(f) :\equiv f =_{\prod_{(x:A)} B(x)} \lambda(x : A).f(x)$ for any function $f : \prod_{(x:A)} B(x)$ and a transformation p with $p(x, b(x)) :\equiv \text{refl}_{\lambda(x:A).b(x)}$. We now know that

$$p(x, b(x)) : P\Big(\lambda(x : A).b(x)\Big),$$

since

$$P\Big(\lambda(x : A).b(x)\Big) \equiv \lambda(x : A).b(x) =_{\prod_{(x:A)} B(x)} \lambda(x : A).b(x),$$

because of the induction function's computation rule for canonical elements. The $\prod -$ind rule can then be used with p and P to get

$$\text{ind}_{\prod_{(x:A)} B(x)}\Big(p(x, b(x)), f\Big) : P(f).$$

If we now look at f as a variable and construct the function

$$\lambda(f : \prod_{(x:A)} B(x)).\text{ind}_{\prod_{(x:A)} B(x)}\Big(p(x, b(x)), f\Big) : \prod_{(f:\prod_{(x:A)} B(x))} \Big(f =_{\prod_{(x:A)} B(x)} \lambda(x : A).f(x)\Big),$$

we have our desired element, proving the theorem. □

Note that we will often write $a, b, c : A$ instead of $a : A, b : A, c : A$ if we have multiple elements of the same type. Since non-dependent functions are only a special case of dependent functions, with B being a constant type family, i.e. not dependent on x, this theorem is also applicable in the non-dependent case.

In this thesis we will use the inductive definition of function types, mainly because of the fact that we can really define what function application is, not just introduce applied functions as a completely new object. We also have the above theorem 1.7.1, instead of the ad hoc $\eta-$rule and as mentioned in this section's beginning. The inductive approach follows the natural way of type definition instead of being somewhat separate from the way we will do it for the other types.

1.8 Multi variable functions

Up to now we have only dealt with single variable functions. This already suffices though to build functions of multiple variables using a concept called currying. We will show how currying works by constructing a function of two variables. Currying of functions of more than two variables follows exactly the same pattern. The main idea is that we have a function, which, when applied to the first variable, gives back an element of a function type. This element, i.e. this function, is then applied to our second variable. For the non-dependent function version we will use a function like $f : A \to (B \to C)$. This function, when applied to $a : A$ gives back a function $f(a)$ of type $B \to C$ which will in turn give us an element of type C when applied to some $b : B$, namely $f(a)(b) : C$. Note that in every step we only have single variable functions. We will often write $f : A \to B \to C$, omitting the parenthesis and introducing right associativity as our convention. To avoid too many parenthesis we will also write $f(a, b)$ instead of $f(a)(b)$.

The difference to the dependent case is, that there is the possibility of the second domain depending upon the first one and the codomain even has the possibility to depend on both. For a type $A : \mathcal{U}$ and families $B : A \to \mathcal{U}$ and $C : \prod_{(x:A)} B(x) \to \mathcal{U}$ we have

$$\prod_{(x:A)} \prod_{(y:B(x))} C(x, y)$$

as the type of a curried two variable function, where we again use the notation $C(x, y)$ instead of $C(x)(y)$. Note that $\prod$ always binds over the remaining part of the expression, i.e. we write $\prod_{(x:A)} B(x) \to \mathcal{U}$, instead of $\prod_{(x:A)}(B(x) \to \mathcal{U})$. For dependent functions we will also keep the convention of writing $f(a, b)$ instead of $f(a)(b)$.

An example for this is the so called identity function $\mathrm{id} : \prod_{(A:\mathcal{U})} A \to A$, defined by $\mathrm{id} :\equiv \lambda(A : \mathcal{U}).\lambda(x : A).x$. For some type $A : \mathcal{U}$ and element $a : A$, we have the equality $\mathrm{id}(A, a) \equiv a$. Sometimes we will write the argument of a dependent function as a subscript. We will for example write $\mathrm{id}_A(x)$, instead of $\mathrm{id}(A, x)$ (or in this special case even just $\mathrm{id}(x)$, since the type is already implied by x).

1.9 Dependent pair type ($\sum$-type)

The next type will be the type theoretic version of ordered pairs, where the second component's type may depend on the first component. It is called the dependent pair type, or $\sum$-type, and has the following inference rules:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x : A \vdash B(x) : \mathcal{U}_i}{\Gamma \vdash \sum_{(x:A)} B(x) : \mathcal{U}_i} \sum\text{-Form}$$

$$\frac{\Gamma, x : A \vdash B(x) : \mathcal{U}_i \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B(a)}{\Gamma \vdash (a, b) : \sum_{(x:A)} B(x)} \sum\text{-Intro}$$

$$\frac{\Gamma, a : A, b : B(a) \vdash g(a, b) : C \qquad \Gamma \vdash z : \sum_{(x:A)} B(x)}{\Gamma \vdash \mathrm{rec}_{\sum_{(x:A)} B(x)}(C, g, z) : C} \sum\text{-rec}$$

$$\frac{\Gamma, a : A, b : B(a) \vdash g(a,b) : P((a,b)) \qquad \Gamma \vdash z : \sum_{(x:A)} B(x)}{\Gamma \vdash \mathrm{ind}_{\sum_{(x:A)} B(x)}(P, g, z) : P(z)} \sum -\mathrm{ind},$$

with $P : (\sum_{(x:A)} B(x)) \to \mathcal{U}_i$. Note that $\sum_{(x:A)}$ binds the variable x and also scopes over the whole expression to its right, just like $\prod$ does. We may interpret $\mathrm{rec}_{\sum_{(x:A)} B(x)}$ and $\mathrm{ind}_{\sum_{(x:A)} B(x)}$ as functions

$$\mathrm{rec}_{\sum_{(x:A)} B(x)} : \prod_{(C:\mathcal{U}_i)} \Big(\prod_{(x:A)} B(x) \to C \Big) \to \Big(\sum_{(x:A)} B(x) \Big) \to C$$

$$\mathrm{ind}_{\sum_{(x:A)} B(x)} : \prod_{(P:(\sum_{(x:A)} B(x)) \to \mathcal{U}_i)} \Big(\prod_{(a:A)} \prod_{(b:B(a))} P((a,b)) \Big) \to \prod_{(z:\sum_{(x:A)} B(x))} P(z),$$

with the following equalities for the canonical elements:

$$\mathrm{rec}_{\sum_{(x:A)} B(x)}(C, g, (a,b)) :\equiv g(a,b)$$
$$\mathrm{ind}_{\sum_{(x:A)} B(x)}(P, g, (a,b)) :\equiv g(a,b).$$

If we have a function $g : \prod_{(a:A)} \prod_{(b:B(a))} P((a,b))$, the induction function allows us to directly construct a function with domain $\sum_{(x:A)} B(x)$, namely

$$\lambda\Big(z : \sum_{(x:A)} B(x)\Big).\mathrm{ind}(P, g, z).$$

Using this, we also know, that if we have proved a proposition P for all ordered pairs, i.e. we have the function g, then we have automatically proven it for all elements of the $\sum_{(x:A)} B(x)$ type, since we can construct the above mentioned function on $\sum_{(x:A)} B(x)$.

As an example, we will construct the familiar projection functions. For the projection on the first component we need to construct a function

$$\mathrm{pr}_1 : \Big(\sum_{(x:A)} B(x) \Big) \to A.$$

We can do this by using a function g, defined by $g(a,b) :\equiv \lambda(a : A).\lambda(b : B(a)).a$, together with our recursor, obtaining

$$\mathrm{pr}_1(z) :\equiv \mathrm{rec}_{\sum_{(x:A)} B(x)}(A, g, z).$$

The second projection

$$\mathrm{pr}_2 : \prod_{(z:\sum_{(x:A)} B(x))} B(\mathrm{pr}_1(z))$$

can be constructed similarly, by using a family $P :\equiv \lambda(z : \sum_{(x:A)} B(x)).B(\mathrm{pr}_1(z))$, the function g, defined by $g(a,b) :\equiv \lambda(a : A).\lambda(b : B(a)).b$, and the induction function, to obtain

$$\mathrm{pr}_2(z) :\equiv \mathrm{ind}_{\sum_{(x:A)} B(x)}(P, g, z).$$

Just like the dependent function type is a generalization of the non-dependent function type, the $\sum$-type generalizes the so called product type, which is the type of cartesian products. The product type's elements are also ordered pairs, with the difference that the second component's type must now be independent. If this is the case, i.e. if B is independent of x, then we define the product type as $A \times B :\equiv \sum_{(x:A)} B(x)$. For the product type the recursor and induction function are

$$\begin{aligned}\mathrm{rec}_{A\times B} &: \prod_{C:\mathcal{U}_i} (A \to B \to C) \to A \times B \to C \\ \mathrm{ind}_{A\times B} &: \prod_{P:A\times B\to\mathcal{U}_i} \Big(\prod_{(a:A)} \prod_{(b:B)} P((a,b)) \Big) \to \prod_{z:A\times B} P(z),\end{aligned}$$

with similar equalities for the canonical elements as before:

$$\begin{aligned}\mathrm{rec}_{A\times B}(C, g, (a,b)) &:\equiv g(a,b) \\ \mathrm{ind}_{A\times B}(P, g, (a,b)) &:\equiv g(a,b).\end{aligned}$$

For the dependent pair type we can also derive a propositional equality between arbitrary elements and its canonical elements, just like we did in theorem 1.7.1 for the function type. For all $z : \sum_{(x:A)} B(x)$, we have

$$z = (\mathrm{pr}_1(z), \mathrm{pr}_2(z)).$$

We will formally prove this later on in corollary 2.3.6.

1.10 Coproduct type (+-type)

The coproduct type is the type theoretic version of the disjoint union in set theory. It has the following inference rules:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash A + B : \mathcal{U}_i}\ +\text{−Form}$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathrm{inl}(a) : A + B}\ +\text{−Intro}_1$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathrm{inr}(b) : A + B}\ +\text{−Intro}_2$$

$$\frac{\Gamma, a : A \vdash g_0(a) : C \qquad \Gamma, b : B \vdash g_1(b) : C \qquad \Gamma \vdash z : A + B}{\Gamma \vdash \mathrm{rec}_{A+B}(C, g_0, g_1, z) : C}\ +\text{−rec}$$

$$\frac{\Gamma, a : A \vdash g_0(a) : P(\mathrm{inl(a)}) \qquad \Gamma, b : B \vdash g_1(b) : P(\mathrm{inr(b)}) \qquad \Gamma \vdash z : A + B}{\Gamma \vdash \mathrm{ind}_{A+B}(P, g_0, g_1, z) : P(z)}\ +\text{−ind}.$$

Interpreted as functions, we have

$$\mathrm{rec}_{A+B} : \prod_{(C:\mathcal{U}_i)} (A \to C) \to (B \to C) \to A + B \to C$$
$$\mathrm{ind}_{A+B} : \prod_{(P:(A+B)\to\mathcal{U}_i)} \Big(\prod_{(a:A)} P(\mathrm{inl}(a)) \Big) \to \Big(\prod_{(b:B)} P(\mathrm{inr}(b)) \Big) \to \prod_{(z:A+B)} P(z).$$

Since we have two types of canonical elements, constructed either by $+-\mathrm{Intro}_1$ or $+-\mathrm{Intro}_2$, we also have two different computational equalities for both the recursor and the induction function:

$$\mathrm{rec}_{A+B}(C, g_0, g_1, \mathrm{inl}(a)) :\equiv g_0(a)$$
$$\mathrm{rec}_{A+B}(C, g_0, g_1, \mathrm{inr}(b)) :\equiv g_1(b)$$
$$\mathrm{ind}_{A+B}(P, g_0, g_1, \mathrm{inl}(a)) :\equiv g_0(a)$$
$$\mathrm{ind}_{A+B}(P, g_0, g_1, \mathrm{inr}(b)) :\equiv g_1(b).$$

1.11 The boolean type 2

The boolean type has the following inference rules:

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathbf{2} : \mathcal{U}_i}\ \mathbf{2}\text{−Form}$$

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash 0_{\mathbf{2}} : \mathcal{U}_i}\ \mathbf{2}\text{−Intro}_1$$

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash 1_{\mathbf{2}} : \mathcal{U}_i}\ \mathbf{2}\text{−Intro}_2$$

$$\frac{\Gamma \vdash c_0 : C \qquad \Gamma \vdash c_1 : C \qquad \Gamma \vdash z : \mathbf{2}}{\Gamma \vdash \mathrm{rec}_{\mathbf{2}}(C, c_0, c_1, z) : C}\ \mathbf{2}\text{−rec}$$

$$\frac{\Gamma \vdash c_0 : P(0_{\mathbf{2}}) \qquad \Gamma \vdash c_1 : P(1_{\mathbf{2}}) \qquad \Gamma \vdash z : \mathbf{2}}{\Gamma \vdash \mathrm{ind}_{\mathbf{2}}(P, c_0, c_1, z) : P(z)}\ \mathbf{2}\text{−ind}$$

For the recursor and induction function we have

$$\mathrm{rec}_{\mathbf{2}} : \prod_{C:\mathcal{U}_i} C \to C \to \mathbf{2} \to C$$
$$\mathrm{ind}_{\mathbf{2}} : \prod_{(P:\mathbf{2}\to\mathcal{U}_i)} P(0_{\mathbf{2}}) \to P(1_{\mathbf{2}}) \to \prod_{(z:\mathbf{2})} P(z),$$

with equalities

$$\mathrm{rec}_{\mathbf{2}}(C, c_0, c_1, 0_{\mathbf{2}}) :\equiv c_0$$
$$\mathrm{rec}_{\mathbf{2}}(C, c_0, c_1, 1_{\mathbf{2}}) :\equiv c_1$$
$$\mathrm{ind}_{\mathbf{2}}(P, c_0, c_1, 0_{\mathbf{2}}) :\equiv c_0$$
$$\mathrm{ind}_{\mathbf{2}}(P, c_0, c_1, 1_{\mathbf{2}}) :\equiv c_1.$$

Again, we want to have a propositional equality between arbitrary elements of $\mathbf{2}$ and its canonical elements. Since there are only two canonical elements though, we might also say, that we want to show that any element of $\mathbf{2}$ is equal to either $0_\mathbf{2}$ or $1_\mathbf{2}$. This proposition corresponds to the following type, as we will explain in section 1.14.

Theorem 1.11.1 *The type*

$$\prod_{(x:\mathbf{2})} (x = 0_\mathbf{2}) + (x = 1_\mathbf{2})$$

is inhabited.

Proof: We prove this by using the induction principle of $\mathbf{2}$. For that, we define the function

$$P :\equiv \lambda(x : \mathbf{2}).(x = 0_\mathbf{2}) + (x = 1_\mathbf{2}),$$

which gives us

$$P(0_\mathbf{2}) \equiv (0_\mathbf{2} = 0_\mathbf{2}) + (0_\mathbf{2} = 1_\mathbf{2})$$
$$P(1_\mathbf{2}) \equiv (1_\mathbf{2} = 0_\mathbf{2}) + (1_\mathbf{2} = 1_\mathbf{2}).$$

Now we use the fact that each element is equal to itself, i.e. $\mathrm{refl}_{0_\mathbf{2}} : 0_\mathbf{2} = 0_\mathbf{2}$ and $\mathrm{refl}_{1_\mathbf{2}} : 1_\mathbf{2} = 1_\mathbf{2}$, to obtain

$$\mathrm{inl}(\mathrm{refl}_{0_\mathbf{2}}) : P(0_\mathbf{2})$$
$$\mathrm{inr}(\mathrm{refl}_{1_\mathbf{2}}) : P(1_\mathbf{2}).$$

Using these two elements, we get

$$\mathrm{ind}_\mathbf{2}(P, \mathrm{inl}(\mathrm{refl}_{0_\mathbf{2}}), \mathrm{inr}(\mathrm{refl}_{1_\mathbf{2}}), x) : P(x)$$

for any $x : \mathbf{2}$, and therefore we have

$$\lambda(x : \mathbf{2}).\mathrm{ind}_\mathbf{2}(P, \mathrm{inl}(\mathrm{refl}_{0_\mathbf{2}}), \mathrm{inr}(\mathrm{refl}_{1_\mathbf{2}}), x) : \prod_{(x:\mathbf{2})} (x = 0_\mathbf{2}) + (x = 1_\mathbf{2}),$$

proving our theorem. Note that if we interpret $\mathrm{ind}_\mathbf{2}$ as a function of the above given type, we may even leave out the element $x : \mathbf{2}$ in the induction function and get the simpler expression

$$\mathrm{ind}_\mathbf{2}(P, \mathrm{inl}(\mathrm{refl}_{0_\mathbf{2}}), \mathrm{inr}(\mathrm{refl}_{1_\mathbf{2}})) : \prod_{(x:\mathbf{2})} (x = 0_\mathbf{2}) + (x = 1_\mathbf{2}).$$

From here on out we will use this interpretation as functions for the recursors and induction functions of all of our types without further mention. □

1.12 The empty type 0

The empty type has the following inference rules:

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathbf{0} : \mathcal{U}_i}\ \mathbf{0}\text{−Form}$$

$$\frac{\Gamma \vdash C : \mathcal{U}_i \qquad \Gamma \vdash z : \mathbf{0}}{\Gamma \vdash \mathrm{rec}_{\mathbf{0}}(C, z) : C}\ \mathbf{0}\text{−rec}$$

$$\frac{\Gamma, a : \mathbf{0} \vdash P(a) : \mathcal{U}_i \qquad \Gamma \vdash z : \mathbf{0}}{\Gamma \vdash \mathrm{ind}_{\mathbf{0}}(P, z) : P(z)}\ \mathbf{0}\text{−ind}.$$

Note that there is no introduction rule, i.e. we can not construct canonical elements of the empty type. The recursor and the induction function are

$$\mathrm{rec}_{\mathbf{0}} : \prod_{(C:\mathcal{U}_i)} \mathbf{0} \to C$$

$$\mathrm{ind}_{\mathbf{0}} : \prod_{(P:\mathbf{0}\to\mathcal{U}_i)} \prod_{(z:\mathbf{0})} P(z).$$

There are no equalities for computing their application on the canonical elements, since there are none.

If we interpret the empty type as falsum, then the recursor can be interpreted as the ex-falso quod libet principle, since if we have proven falsum, it gives us a proof of any desired proposition.

1.13 The unit type 1

The unit type has the following inference rules:

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathbf{1} : \mathcal{U}_i}\ \mathbf{1}\text{−Form}$$

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \star : \mathbf{1}}\ \mathbf{1}\text{−Intro}$$

$$\frac{\Gamma \vdash c : C \qquad \Gamma \vdash z : \mathbf{1}}{\Gamma \vdash \mathrm{rec}_{\mathbf{1}}(C, c, z) : C}\ \mathbf{1}\text{−rec}$$

$$\frac{\Gamma \vdash c : P(\star) \qquad \Gamma \vdash z : \mathbf{1}}{\Gamma \vdash \mathrm{ind}_{\mathbf{1}}(P, c, z) : P(z)}\ \mathbf{1}\text{−ind}.$$

The recursor and the induction function are

$$\mathrm{rec}_{\mathbf{1}} : \prod_{C:\mathcal{U}_i} C \to \mathbf{1} \to C$$

$$\mathrm{ind}_{\mathbf{1}} : \prod_{P:\mathbf{1}\to\mathcal{U}_i} P(\star) \to \prod_{(z:\mathbf{1})} P(z),$$

with equalities

$$\mathrm{rec}_{\mathbf{1}}(C, c, \star) :\equiv c$$
$$\mathrm{ind}_{\mathbf{1}}(P, c, \star) :\equiv c.$$

Theorem 1.13.1 *For all $z : \mathbf{1}$, the type $z = \star$ is inhabited.*

Proof: This proof is conducted analogously to that of theorem 1.11.1. We start with a function $P :\equiv \lambda(z : \mathbf{1}).z = \star$, so that we have $\mathrm{refl}_\star : P(\star)$. From this we get

$$\mathrm{ind}_{\mathbf{1}}(P, \mathrm{refl}_\star) : \prod_{z:\mathbf{1}} z = \star,$$

proving our theorem. □

1.14 Propositions as types

As mentioned in section 1.1 there exists a translation between propositions in our common mathematical language and the types of type theory. This concept is known as propositions as types:

Natural language	Type theory
True	$\mathbf{1}$
False	$\mathbf{0}$
A and B	$A \times B$
A or B	$A + B$
If A then B	$A \to B$
A if and only if B	$(A \to B) \times (B \to A)$
Not A	$A \to \mathbf{0}$
For all $x : A$, we know $P(x)$	$\prod_{(x:A)} P(x)$
There exists $x : A$, such that $P(x)$	$\sum_{(x:A)} P(x)$

We also alternatively write $\neg A :\equiv A \to \mathbf{0}$ for the type of "Not A", often using the phrasing "the type A is not inhabited" instead of "$\neg A$ is inhabited". For example, we may say that the type $\mathbf{0}$ is not inhabited, since we can simply use our identity function $\mathrm{id}_{\mathbf{0}} : \mathbf{0} \to \mathbf{0} \equiv \neg\mathbf{0}$ from section 1.8 to prove $\neg\mathbf{0}$.

1.15 Equality type

The last type we will introduce in this thesis is the equality type or sometimes also called identity type. As already mentioned in section 1.7, it is the type corresponding to propositional equality between two elements. Its inference rules are

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a =_A b : \mathcal{U}_i} = -\mathrm{Form}$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathrm{refl}_a : a =_A a} = -\mathrm{Intro}$$

$$\frac{\Gamma, x : A \vdash g(x) : C \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : a =_A b}{\Gamma \vdash \mathrm{rec}(C, g, a, b, p) : C} = -\mathrm{rec}$$

$$\frac{\Gamma, x : A \vdash g(x) : P(x, x, \mathrm{refl}_x) \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : a =_A b}{\Gamma \vdash \mathrm{ind}(P, g, a, b, p) : P(a, b, p)} = -\mathrm{ind}.$$

If we have proven $a =_A b$ we will say that a and b are propositionally equal. Elements of the equality type $a =_A b$ will be called paths in A with starting point a and endpoint b. For the sake of better readability we will often leave the subscript implicit, writing simply $a = b$. We will also use the notation $(a \neq_A b) :\equiv \neg(a =_A b)$ for disequality, calling a and b not equal or unequal. The "refl" in the canonical elements is short for "reflexivity" and expresses the fact that judgmentally equal elements are propositionally equal. The recursor and the induction function are

$$\mathrm{rec}_{=A} : \prod_{(C:\mathcal{U}_i)} \Big(\prod_{(x:A)} C \Big) \to \prod_{(x,y:A)} \prod_{(p:x=_A y)} C$$

$$\mathrm{ind}_{=A} : \prod_{(P:\prod_{(x,y:A)}(x=_A y)\to\mathcal{U}_i)} \Big(\prod_{(x:A)} P(x, x, \mathrm{refl}_x) \Big) \to \prod_{(x,y:A)} \prod_{(p:x=_A y)} P(x, y, p),$$

with equalities

$$\mathrm{rec}_{=A}(C, g, x, x, \mathrm{refl}_x) :\equiv g(x)$$
$$\mathrm{ind}_{=A}(P, g, x, x, \mathrm{refl}_x) :\equiv g(x).$$

Note that we will often write $\prod_{(x,y:A)}$ instead of $\prod_{(x:A)} \prod_{(y:A)}$ if we have two or more variables of the same type.

In Voevodsky's intuitive homotopic interpretation of MLITT in [12], types are thought of as spaces, elements as points in those spaces and equalities as paths between those points. In regular MLITT, the induction principle is called the J-rule, but the above interpretation lead to the alternative name of path induction. Roughly speaking, path induction expresses the fact, that in order to prove a proposition for all paths $p : x = y$, it suffices to prove it for all constant paths $\mathrm{refl}_x : x = x$. In some cases though, it is useful to fix one of the variables. In this case, in order to prove a proposition for all paths of the form $p : a = x$, with x being the variable, it suffices to prove it for the specific constant path $\mathrm{refl}_a : a = a$. This modified way of proving a proposition is valid by the alternative induction principle

$$\mathrm{ind}'_{=A} : \prod_{(a:A)} \prod_{(P:\prod_{(x:A)}(a=_A x)\to\mathcal{U})} P(a, \mathrm{refl}_a) \to \prod_{(x:A)} \prod_{(p:a=_A x)} P(x, p),$$

with defining equation

$$\mathrm{ind}'_{=A}(a, P, c, a, \mathrm{refl}_a) :\equiv c.$$

This alternative induction principle is called based path induction.

Theorem 1.15.1 *Path induction is equivalent to based path induction.*

Due to the above theorem, we will generally not discern between the two possible induction principles and simply say path induction in both cases.

1.16 Corollaries of path induction

As mentioned above, we will often refer to an element $p : x =_A y$ of the equality type as a path from x to y. Of course we can now also form paths between paths, for example for two elements $p, q : x =_A y$, we could have some element $r : p =_{x=_A y} q$. Going one step further, for $r, s : p =_{x=_A y} q$ we could form the type $r =_{p=_{x=_A y} q} s$. Elements like r will be called 2-dimensional paths and paths between such 2-dimensional paths are called 3-dimensional paths. In this section we will introduce and prove numerous different lemmas using path induction. The proofs for those lemmas for which no proof is given in this thesis, can be found in the HoTT-book [11].

When proving theorems or lemmas we want to follow the principle of "data suffice". This means that before conducting the actual proof we want to first think about what results would actually be possible with our currently available data. If we want to prove an equation for example, we would first look at the left hand side and think about what object we could construct from the available data that may equate the already existent object on the left. Only then will we go on to the actual proof. This method prohibits results from just appearing without any motivation on how one would come up with it.

Lemma 1.16.1 (Path inversion) *For all $A : \mathcal{U}$ and $x, y : A$, there exists a function of type*

$$(x = y) \to (y = x).$$

We will write the result of applying this function to a path $p : x = y$ as $p^{-1} : y = x$, with the equality $\mathrm{refl}_x^{-1} \equiv \mathrm{refl}_x$ for all $x : A$. p^{-1} is called the inverse of p.

Lemma 1.16.2 (Path concatenation) *For all $A : \mathcal{U}$ and $x, y, z : A$, there exists a function of type*

$$(x = y) \to (y = z) \to (x = z).$$

For elements $p : x = y$ and $q : y = z$ we will write the application of this function as $p \cdot q$ with the equality $\mathrm{refl}_x \cdot \mathrm{refl}_x \equiv \mathrm{refl}_x$ for all $x : A$. $p \cdot q$ is called the concatenation or the composite of p and q.

Lemma 1.16.3 *For all $A : \mathcal{U}$, $p : x =_A y$, $q : y =_A z$ and $r : z =_A w$, we know:*

(i) $p = p \cdot \mathrm{refl}_y$ *and* $p = \mathrm{refl}_x \cdot p$.
(ii) $p^{-1} \cdot p = \mathrm{refl}_y$ *and* $p \cdot p^{-1} = \mathrm{refl}_x$.
(iii) $(p^{-1})^{-1} = p$.
(iv) $p \cdot (q \cdot r) = (p \cdot q) \cdot r$.

We will now look at how a function $f : A \to B$ acts on a path $p : x =_A y$. The following lemma can be interpreted, in MLITT, as the fact that all functions respect equality or from Voevodsky's viewpoint, that they preserve paths.

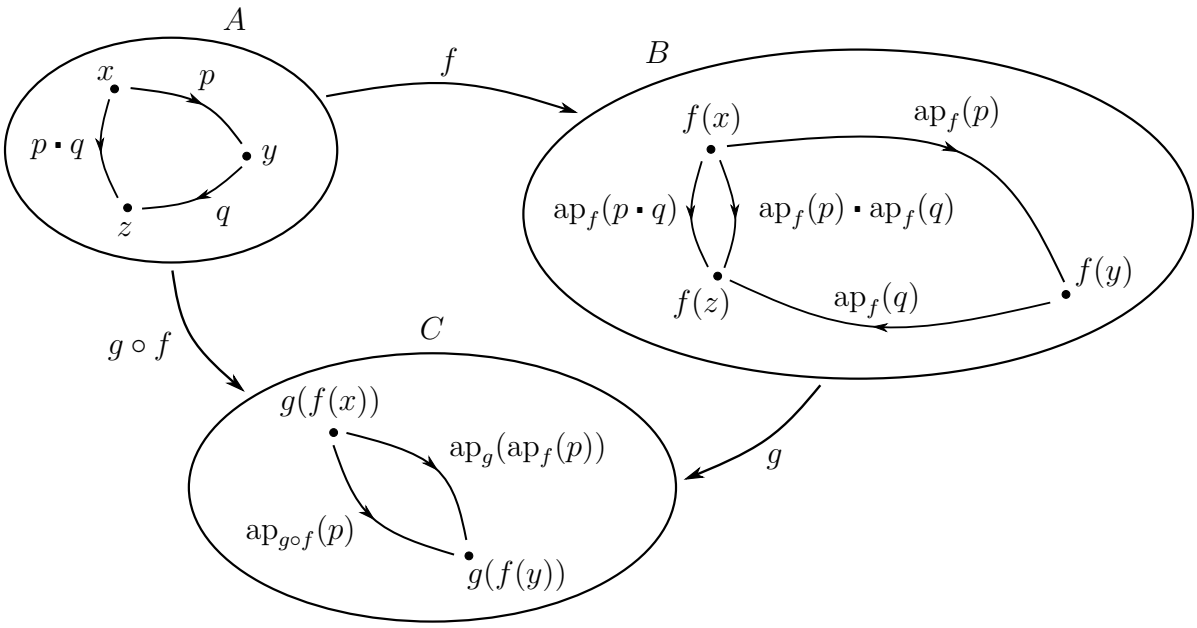
Lemma 1.16.4 (Function application) *Let $f : A \to B$. For all $x, y : A$, there exists a function*

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

with the equality $\mathrm{ap}_f(\mathrm{refl}_x) \equiv \mathrm{refl}_{f(x)}$ for all $x : A$. $\mathrm{ap}_f(p)$ is called the application of f to path p. We sometimes also use the notation $f(p) :\equiv \mathrm{ap}_f(p)$.

Lemma 1.16.5 *Let $f : A \to B$, $g : B \to C$, $p : x =_A y$ and $q : y =_A z$. The following holds:*

(i) $\mathrm{ap}_f(p \cdot q) = \mathrm{ap}_f(p) \cdot \mathrm{ap}_f(q)$.
(ii) $\mathrm{ap}_f(p^{-1}) = \mathrm{ap}_f(p)^{-1}$.
(iii) $\mathrm{ap}_g(\mathrm{ap}_f(p)) = \mathrm{ap}_{g \circ f}(p)$.
(iv) $\mathrm{ap}_{\mathrm{id}_A}(p) = p$.



Proof:

(i) $\mathrm{ap}_f(p \cdot q)$ is of the type $f(x) =_B f(z)$. This already implies that we might get a concatenation of the paths $\mathrm{ap}_f(p) : f(x) =_B f(y)$ and $\mathrm{ap}_f(q) : f(y) =_B f(z)$ as the right side of the equation, i.e. instead of applying the function to the concatenated paths, we try to concatenate the results of individually applying f to them. We now have to find an element of the type $\mathrm{ap}_f(p \cdot q) = \mathrm{ap}_f(p) \cdot \mathrm{ap}_f(q)$ to show that these two methods are actually equal.

For the proof we first use path induction on p, so it suffices to look at the case that $y \equiv x$ and $p \equiv \mathrm{refl}_x$. Then by path induction on q, we only need to look at the case that $z \equiv x$ and $q \equiv \mathrm{refl}_x$. We get

$$\begin{aligned}\mathrm{ap}_f(p \cdot q) &\equiv \mathrm{ap}_f(\mathrm{refl}_x \cdot \mathrm{refl}_x)\\ &\equiv \mathrm{ap}_f(\mathrm{refl}_x)\\ &\equiv \mathrm{refl}_{f(x)}\\ &\equiv \mathrm{refl}_{f(x)} \cdot \mathrm{refl}_{f(x)}\\ &\equiv \mathrm{ap}_f(\mathrm{refl}_x) \cdot \mathrm{ap}_f(\mathrm{refl}_x)\\ &\equiv \mathrm{ap}_f(p) \cdot \mathrm{ap}_f(q).\end{aligned}$$

We therefore have $\mathrm{refl}_{\mathrm{refl}_{f(x)}}$ as an inhabitant of the type

$$\mathrm{ap}_f(p \cdot q) = \mathrm{ap}_f(p) \cdot \mathrm{ap}_f(q),$$

proving the lemma. As mentioned in section 1.15, we know that judgmentally equal elements are also propositionally equal. Therefore, once we have shown judgmental equality we will end our proofs, without explicitly denoting the reflexivity element every time.

(ii) $\mathrm{ap}_f(p^{-1}) : f(y) =_B f(x)$, so the obvious choice for a path from $f(y)$ to $f(x)$ in the space B would be the inverse of $\mathrm{ap}_f(p)$, i.e. instead of applying the function on the inverse of a path, we can first apply the function and then take the inverse of the result.
We use path induction on p and therefore only look at the case that $y \equiv x$ and $p \equiv \mathrm{refl}_x$:

$$\begin{aligned}\mathrm{ap}_f(p^{-1}) &\equiv \mathrm{ap}_f(\mathrm{refl}_x^{-1})\\ &\equiv \mathrm{ap}_f(\mathrm{refl}_x)\\ &\equiv \mathrm{refl}_{f(x)}\\ &\equiv \mathrm{refl}_{f(x)}^{-1}\\ &\equiv \mathrm{ap}_f(\mathrm{refl}_x)^{-1}\\ &\equiv \mathrm{ap}_f(p)^{-1}.\end{aligned}$$

(iii) Since $\mathrm{ap}_f(p)$ is of type $f(x) =_B f(y)$ we know that $\mathrm{ap}_g(\mathrm{ap}_f(p))$ is of type $g(f(x)) =_C g(f(y))$, which we will equivalently write as $g \circ f(x) =_C g \circ f(y)$ using the common notation for function concatenation. To obtain such a type from our available data, we may apply $g \circ f$ to our path p, i.e. instead of first applying f on the path and then g on the result, we directly apply $g \circ f$ to our path p, skipping the intermediate step in B.

We use induction on p, looking only at the case $y \equiv x$ with $p \equiv \mathrm{refl}_x$:

$$\begin{aligned}\mathrm{ap}_g(\mathrm{ap}_f(p)) &\equiv \mathrm{ap}_g(\mathrm{ap}_f(\mathrm{refl}_x))\\ &\equiv \mathrm{ap}_g(\mathrm{refl}_{f(x)})\\ &\equiv \mathrm{refl}_{g(f(x))}\\ &\equiv \mathrm{refl}_{g\circ f(x)}\\ &\equiv \mathrm{ap}_{g\circ f}(\mathrm{refl}_x)\\ &\equiv \mathrm{ap}_{g\circ f}(p).\end{aligned}$$

(iv) $\mathrm{ap}_{\mathrm{id}_A}(p)$ is of type $\mathrm{id}_\mathrm{A}(x) =_A \mathrm{id}_\mathrm{A}(y)$ which is equivalent to $x =_A y$. Since our only available data satisfying this type is simply p, we assume that the application of the identity function to a path is equal to the path itself.
With induction on p, we only need to look at the case where $y \equiv x$ and $p \equiv \mathrm{refl}_x$:

$$\begin{aligned}\mathrm{ap}_{\mathrm{id}_A}(p) &\equiv \mathrm{ap}_{\mathrm{id}_A}(\mathrm{refl}_x)\\ &\equiv \mathrm{refl}_{\mathrm{id}_\mathrm{A}(x)}\\ &\equiv \mathrm{refl}_x\\ &\equiv p.\end{aligned}$$

□

We now want to generalize the ap function to also work with dependent functions $f : \prod_{(x:A)} B(x)$. The obvious difficulty here is, that the two points $f(x) : B(x)$ and $f(y) : B(y)$ may not lie in the same type any more, which prohibits us from having a path between them. In order to still have a relation between $B(x)$ and $B(y)$, we use the so called transport operation.

Lemma 1.16.6 (Transport) *Let $A : \mathcal{U}$, $P : A \to \mathcal{U}$, $x, y : A$ and $p : x =_A y$. There exists a function*

$$p^P_* : P(x) \to P(y),$$

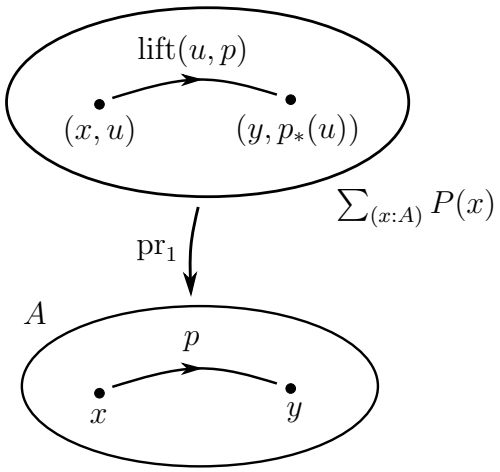
with the equality $(\mathrm{refl}_x)^P_* \equiv \mathrm{id}_{P(x)}$ for all $x : A$. Since p^P_* is called the transport function, we sometimes also write $\mathrm{transport}^P(p, x)$ instead of $p^P_*(x)$. If the type family P is clear from the context, we may also leave it implicit, writing only p_*.

The transport lemma also has a topological interpretation, if we think of our type A as the base space of the total space $\sum_{(x:A)} P(x)$. It allows us to "lift" a path $p : x =_A y$ from the base space up into the total space, where it will be a path starting at a point $u : P(x)$ "lying over" x and ending at $p_*(u)$. The following lemma gives us exactly such a path.

Lemma 1.16.7 *Let $P : A \to \mathcal{U}$, $x, y : A$, $u : P(x)$ and $p : x =_A y$. There exists an element*

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

in the equality type of $\sum_{(x:A)} P(x)$, with projection $\mathrm{pr}_1(\mathrm{lift}(u, p)) = p$.



PROOF: We conduct the proof by induction on p. We therefore only look at the case where $y \equiv x$ and $p \equiv \mathrm{refl}_x$. Since for this path, our transport function becomes the identity function, we automatically have $(y, p_*(u)) \equiv (x, u)$. Therefore we can define $\mathrm{lift}(u, p)$ as $\mathrm{refl}_{(x,u)}$, resulting in

$$\begin{aligned}\mathrm{pr}_1(\mathrm{lift}(u,p)) &\equiv \mathrm{pr}_1(\mathrm{lift}(u, \mathrm{refl}_x)) \\ &\equiv \mathrm{pr}_1(\mathrm{refl}_{(x,u)}) \\ &\equiv \mathrm{refl}_{\mathrm{pr}_1(x,u)} \\ &\equiv \mathrm{refl}_x \\ &\equiv p.\end{aligned}$$

□

We can apply a dependent function $f : \prod_{(x:A)} P(x)$ to a path $p : x =_A y$, by defining a function $f' : A \to \sum_{(x:A)} P(x)$, via $f'(x) :\equiv (x, f(x))$, which is now non-dependent, and applying it to p, obtaining $f'(p) : f'(x) = f'(y)$. With this, we have lifted the path p into the total space $\sum_{(x:A)} P(x)$. We can easily see that $f'(p)$ actually lies over p by applying the first projection, obtaining $\mathrm{pr}_1(f'(p)) = p$, by the fact that $\mathrm{pr}_1 \circ f' \equiv \mathrm{id}_A$ together with lemma 1.16.5(iv).

Using $\mathrm{lift}(u, p)$ we have a path from (x, u) to $(y, p_*(u))$, which lies over p. We now look at a path with a different endpoint $v : P(y)$. A path from u to v should run through $\mathrm{lift}(u, p)$, with an additional section from $p_*(u)$ to v, lying in $P(y)$. The next lemma shows that dependent functions produce exactly such paths.

Lemma 1.16.8 *Let $f : \prod_{(x:A)} P(x)$. There exists a function*

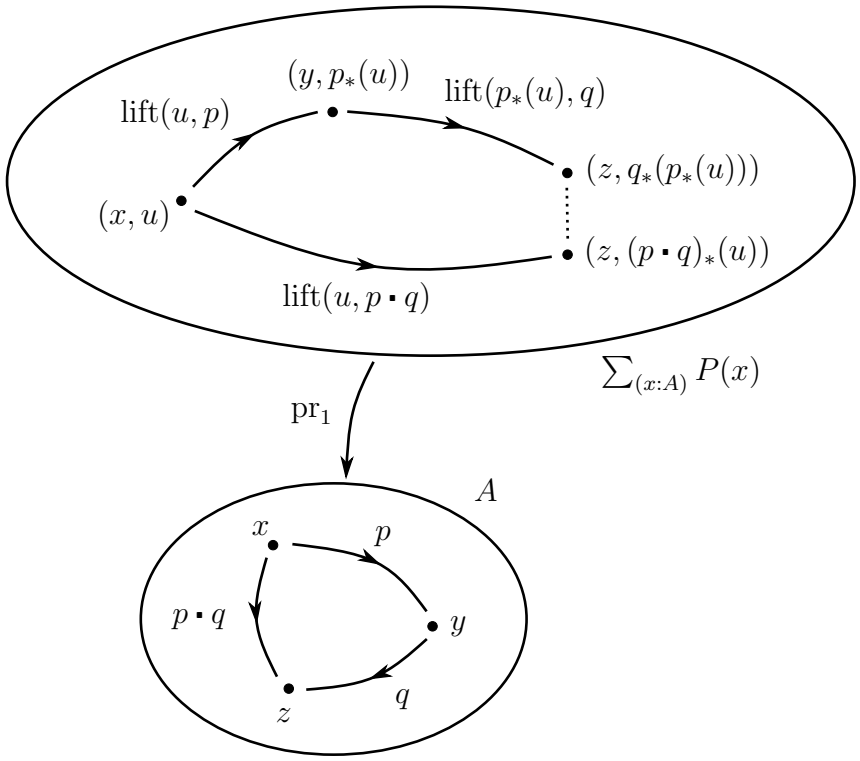
$$\mathrm{apd}_f : \prod_{p:x=y} (p_*(f(x)) =_{P(y)} f(y)).$$

We end our first chapter by proving some additional lemmas about the transport function.

Lemma 1.16.9 *Let $P : A \to \mathcal{U}$, $p : x =_A y$, $q : y =_A z$ and $u : P(x)$. The type*

$$q_*(p_*(u)) = (p \cdot q)_*(u)$$

is inhabited.



Proof: We first look at each piece of the left side of the equation and analyse the types: $u : P(x)$, $p_* : P(x) \to P(y)$ and $q_* : P(y) \to P(z)$. Therefore $q_*(p_*(u))$ is of type $P(z)$. Since u is the only variable available as an input, we need to construct a function which takes u as its input and gives us back an element of type $P(z)$. This can obviously be achieved through a transport function of a path from x to z, which we can obtain by concatenating p and q. All together, instead of applying the transport functions one after another, we first concatenate both paths and then transport in one step along the new path.

We use induction on p and q, now only looking at the case that $y \equiv x$, $z \equiv x$ and $p \equiv q \equiv \mathrm{refl}_x$. We again use the fact that $(\mathrm{refl}_x)_*$ is the identity function:

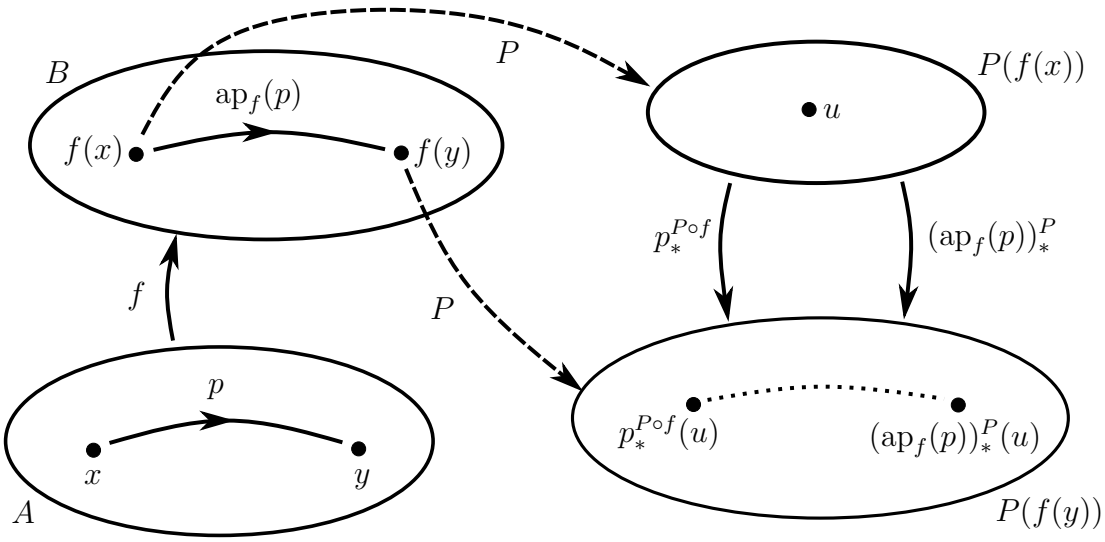
$$\begin{aligned} q_*(p_*(u)) &\equiv (\mathrm{refl}_x)_*((\mathrm{refl}_x)_*(u)) \\ &\equiv (\mathrm{refl}_x)_*(\mathrm{id}(u)) \\ &\equiv (\mathrm{refl}_x)_*(u) \\ &\equiv (\mathrm{refl}_x \cdot \mathrm{refl}_x)_*(u) \\ &\equiv (p \cdot q)_*(u). \end{aligned}$$

□

Lemma 1.16.10 *Let $f : A \to B$, $P : B \to \mathcal{U}$, $p : x =_A y$ and $u : P(f(x))$. The type*

$$p_*^{P \circ f}(u) = (\mathrm{ap}_f(p))_*^P(u)$$

is inhabited.



Proof: The type of $p_*^{P \circ f}$ is $P \circ f(x) \to P \circ f(y)$, i.e. $P(f(x)) \to P(f(y))$. This implies that we could also use an element of $f(x) = f(y)$ instead of p as our path, and only P instead of $P \circ f$ as our type family when conducting the transport. To obtain such a path, we can use $\mathrm{ap}_f(p)$ to get $(\mathrm{ap}_f(p))_*^P : P(f(x)) \to P(f(y))$. We therefore predict, that it has the same result, whether we first apply the function to our path and then transport it, or transport the path directly with a different family that includes the function.

We prove this by induction on p, looking at the case where $y \equiv x$ and $p \equiv \mathrm{refl}_x$:

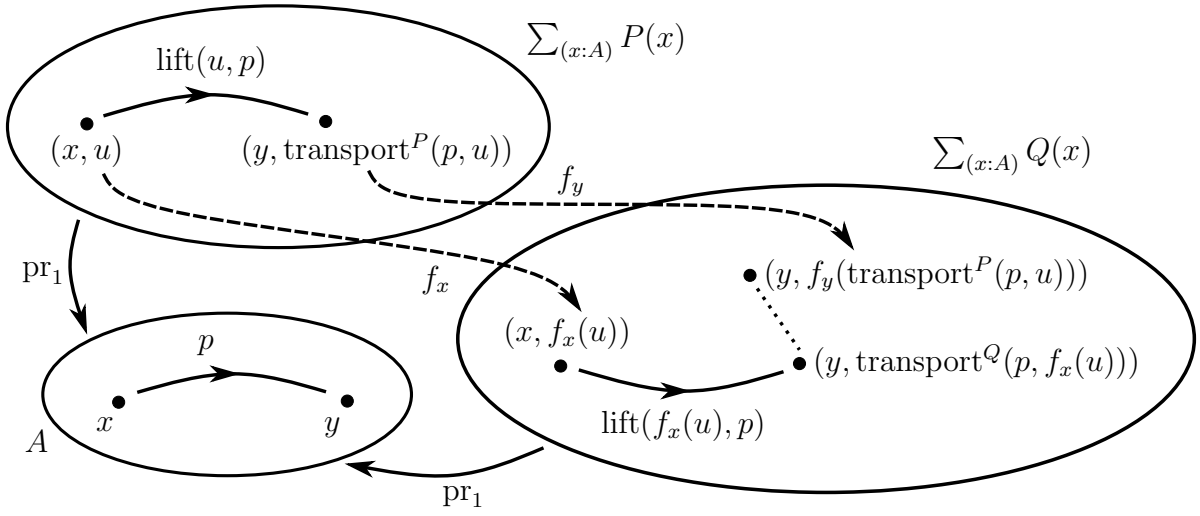
$$\begin{aligned} p_*^{P \circ f}(u) &\equiv (\mathrm{refl}_x)_*^{P \circ f}(u) \\ &\equiv \mathrm{id}_{P \circ f(x)}(u) \\ &\equiv u \\ &\equiv \mathrm{id}_{P(f(x))}(u) \\ &\equiv (\mathrm{refl}_{f(x)})_*^P(u) \\ &\equiv (\mathrm{ap}_f(\mathrm{refl}_x))_*^P(u) \\ &\equiv (\mathrm{ap}_f(p))_*^P(u). \end{aligned}$$

□

Lemma 1.16.11 *Let* $P, Q : A \to \mathcal{U}$, $f : \prod_{(x:A)} P(x) \to Q(x)$, $x, y : A$, $u : P(x)$ *and* $p : x =_A y$. *The type*

$$\mathrm{transport}^Q(p, f_x(u)) = f_y(\mathrm{transport}^P(p, u))$$

is inhabited.



Proof: The type of $\mathrm{transport}^Q(p, f_x(u))$ is $Q(y)$. It is obtained by first applying f_x to u, giving back an element of $Q(x)$ and then transporting this element, ending up in $Q(y)$. Another way of obtaining an element of $Q(y)$ would be to use the function f_y on a variable of type $P(y)$. Using our data the obvious way to get such an element of $P(y)$ is to transport u. We therefore assume, that first transporting and then applying f gives us an equal result to first applying f and then transporting, like it is done on the left side of our equality.

We prove this by induction on p, only looking at the case where $y \equiv x$ and $p \equiv \mathrm{refl}_x$:

$$\begin{aligned}
\mathrm{transport}^Q(p, f_x(u)) &\equiv \mathrm{transport}^Q(\mathrm{refl}_x, f_x(u)) \\
&\equiv \mathrm{id}_{Q(x)}(f_x(u)) \\
&\equiv f_x(u) \\
&\equiv f_x(\mathrm{id}_{P(x)}(u)) \\
&\equiv f_x(\mathrm{transport}^P(\mathrm{refl}_x, u)) \\
&\equiv f_y(\mathrm{transport}^P(p, u)).
\end{aligned}$$

□

Theorem 1.16.12 *The type* $0_{\mathbf{2}} \neq 1_{\mathbf{2}}$ *is inhabited.*

PROOF: We need to construct a function of type $(0_{\mathbf{2}} = 1_{\mathbf{2}}) \to \mathbf{0}$. For this, we define a type family $P : \mathbf{2} \to \mathcal{U}$ by

$$P :\equiv \mathrm{rec}_{\mathbf{2}}(\mathcal{U}, \mathbf{1}, \mathbf{0}) : \mathbf{2} \to \mathcal{U},$$

so that we have $P(0_{\mathbf{2}}) \equiv \mathbf{1}$ and $P(1_{\mathbf{2}}) \equiv \mathbf{0}$. By using this family, together with an element $p : 0_{\mathbf{2}} = 1_{\mathbf{2}}$, we get the transport function $p^P_* : \mathbf{1} \to \mathbf{0}$. Since we know that $\star : \mathbf{1}$, we get $p^P_*(\star) : \mathbf{0}$ and therefore

$$\lambda(p : 0_{\mathbf{2}} = 1_{\mathbf{2}}).p^P_*(\star) : (0_{\mathbf{2}} = 1_{\mathbf{2}}) \to \mathbf{0} \equiv 0_{\mathbf{2}} \neq 1_{\mathbf{2}}.$$

□

2 Voevodsky's axiom of univalence

In the previous chapter we talked about identifying two elements of the same type with one another in a very general way, by using the equality type. We now want to introduce more specialized notions for functions and even for types themselves.

2.1 Function homotopy

We start by introducing a formal way of identifying two functions f and g. Intuitively one might say that two functions are equal if their values are equal for all possible inputs. The following definition formalizes this.

Definition 2.1.1 (Homotopy) *Let $f, g : \prod_{(x:A)} P(x)$. A homotopy from f to g is an element of type*

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

Homotopies are equivalence relations on the dependent functions, i.e. they fulfil reflexivity, symmetry and transitivity. This is formalized by the next lemma.

Lemma 2.1.2 *The following types are inhabited:*

(i)

$$\prod_{f:\prod_{(x:A)} P(x)} (f \sim f).$$

(ii)

$$\prod_{f,g:\prod_{(x:A)} P(x)} (f \sim g) \to (g \sim f).$$

(iii)

$$\prod_{f,g,h:\prod_{(x:A)} P(x)} (f \sim g) \to (g \sim h) \to (f \sim h).$$

Proof: In the following we will explicitly give an element of each type, using the ability to reverse, and concatenate paths in (ii) and (iii) respectively:

(i)

$$\lambda\Big(f : \prod_{(x:A)} P(x)\Big).\Big(\lambda(x : A).\mathrm{refl}_{f(x)}\Big).$$

(ii)

$$\lambda\Big(f : \prod_{(x:A)} P(x)\Big).\lambda\Big(g : \prod_{(x:A)} P(x)\Big).\lambda\Big(\alpha : \prod_{(x:A)} f(x) = g(x)\Big).\Big(\lambda x.\alpha(x)^{-1}\Big).$$

(iii)

$$\lambda\Big(f : \prod_{(x:A)} P(x)\Big).\lambda\Big(g : \prod_{(x:A)} P(x)\Big).\lambda\Big(h : \prod_{(x:A)} P(x)\Big)$$
$$.\lambda\Big(\alpha : \prod_{(x:A)} f(x) = g(x)\Big).\lambda\Big(\beta : \prod_{(x:A)} g(x) = h(x)\Big).\Big(\lambda x.\alpha(x) \cdot \beta(x)\Big).$$

□

Lemma 2.1.3 *Let $f, g : A \to B$, $H : f \sim g$ and $p : x =_A y$. The type*

$$H(x) \cdot g(p) = f(p) \cdot H(y)$$

is inhabited.

Corollary 2.1.4 *Let $f : A \to A$, $H : f \sim \mathrm{id}_A$ and $x : A$. The type*

$$H(f(x)) = f(H(x))$$

is inhabited.

2.2 Equivalence of types

Before we can define our notion of type identification, we first need to formalize the concept of inverse functions.

Definition 2.2.1 (Quasi-inverse) *A quasi-inverse of a function $f : A \to B$, is a function $g : B \to A$, together with elements of $f \circ g \sim \mathrm{id}_B$ and $g \circ f \sim \mathrm{id}_A$. The type of all quasi-inverses of f is written as*

$$\mathrm{qinv}(f) :\equiv \sum_{g:B\to A} ((f \circ g \sim \mathrm{id}_B) \times (g \circ f \sim \mathrm{id}_A)).$$

The idea behind this definition is, that we have an inverse of f, if there is a function g that gives us, when concatenated with f from the left and from the right, functions that are point wise equal to the identity functions.

Now we can introduce the type that corresponds to a non-dependent function f being a so called equivalence:

$$\mathrm{isequiv}(f) :\equiv \Big(\sum_{g:B\to A} (f \circ g \sim \mathrm{id}_B)\Big) \times \Big(\sum_{h:B\to A} (h \circ f \sim \mathrm{id}_A)\Big).$$

Lemma 2.2.2 *Let $f : A \to B$. There exists a function of type*

$$\mathrm{qinv(f)} \to \mathrm{isequiv(f)}.$$

PROOF: We can simply construct a function that takes an element $(g, (\alpha, \beta)) : \text{qinv}(f)$ to $((g, \alpha), (g, \beta)) : \text{isequiv}(f)$, by using our projection functions:

$$\lambda(x : \text{qinv}(f)).\Big((\text{pr}_1(x), \text{pr}_1(\text{pr}_2(x))), (\text{pr}_1(x), \text{pr}_2(\text{pr}_2(x)))\Big).$$

□

This lemma gives us a way of proving that a function f is an equivalence by simply giving an element of $\text{qinv}(f)$. We also have the reverse direction of lemma 2.2.2.

Lemma 2.2.3 *Let $f : A \to B$. There exists a function of type*

$$\text{isequiv(f)} \to \text{qinv(f)}.$$

Definition 2.2.4 (Equivalence of types) *The type of equivalences between two types $A, B : \mathcal{U}$ is defined as*

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

Therefore an equivalence between A and B is defined as a function $f : A \to B$, together with a proof of it being an equivalence, i.e. an inhabitant of the type $\text{isequiv}(f)$. We will often be somewhat inaccurate with notation, not clearly discerning between equivalences and their functions. For example, given elements $f : A \to B$, $e : \text{isequiv}(f)$, $g : A \simeq B$ and $a : A$, we may write $f : A \simeq B$ instead of $(f, e) : A \simeq B$ or $g(a)$ instead of $(\text{pr}_1(g))(a)$, just for the sake of simplicity.

Lemma 2.2.5 *For all $x, y : \mathbf{1}$, the type $(x = y) \simeq \mathbf{1}$ is inhabited.*

Lemma 2.2.6 *Type equivalence fulfils reflexivity, symmetry and transitivity, i.e. it is an equivalence relation on types:*

(i) For all types A, the identity function forms the equivalence $A \simeq A$.
(ii) For all $f : A \simeq B$, there exists a function $f^{-1} : B \simeq A$, called the inverse of f.
(iii) For all $f : A \simeq B$ and $g : B \simeq C$, the concatenation also forms an equivalence $g \circ f : A \simeq C$.

2.3 Function extensionality axiom

We now want to investigate whether there exists an equivalence between the types of two functions being equal and two functions having equal values for all points. Let $B : A \to \mathcal{U}$ and $f, g : \prod_{(x:A)} B(x)$. The first direction of our desired equivalence is given by the function

$$\text{happly} : (f = g) \to \prod_{x:A} (f(x) =_{B(x)} g(x)).$$

This function exists due to path induction. Looking only at the case that $g \equiv f$ and our input variable being refl_f, we can define it by

$$\mathrm{happly}(\mathrm{refl}_f) :\equiv \lambda(x : A).\mathrm{refl}_{f(x)}.$$

It turns out though, that by only using the type theory we introduced in chapter 1, we can't prove the existence of a function in the other direction. To still obtain the equivalence we will therefore introduce the following axiom.

Axiom 2.3.1 (Function extensionality axiom) *Let $A : \mathcal{U}$ and $B : A \to \mathcal{U}$. For two functions $f, g : \prod_{(x:A)} B(x)$ the function* happly*, as defined above, forms an equivalence*

$$(f = g) \simeq \Big(\prod_{(x:A)} (f(x) =_{B(x)} g(x)) \Big).$$

If we assume this axiom, we know that there exists a quasi-inverse function of happly, namely

$$\mathrm{funext} : \Big(\prod_{(x:A)} (f(x) =_{B(x)} g(x)) \Big) \to (f = g),$$

which we will call function extensionality.

Corollary 2.3.2 *Let $h : \prod_{(x:A)}(f(x) = g(x))$ and $p : f = g$. The following types are inhabited:*

(i)
$$h = \mathrm{happly}(\mathrm{funext}(h)).$$

(ii)
$$p = \mathrm{funext}(\mathrm{happly}(p)).$$

Proof: Since the axiom states the existence of the equivalence, we also know that there exist two homotopies, i.e. point wise equalities to the identity function when forming the compositions of happly and funext. □

Lemma 2.3.3 *Let $f, g, h : \prod_{(x:A)} B(x), \alpha : f = g, \beta : g = h$. The following types are inhabited:*

(i)
$$\mathrm{refl}_f = \mathrm{funext}(\lambda(x : A).\mathrm{refl}_{f(x)}).$$

(ii)
$$\alpha^{-1} = \mathrm{funext}(\lambda(x : A).\mathrm{happly}(\alpha, x)^{-1}).$$

(iii)
$$\alpha \cdot \beta = \mathrm{funext}(\lambda(x : A).\mathrm{happly}(\alpha, x) \cdot \mathrm{happly}(\beta, x)).$$

PROOF:

(i) refl_f is of type $f = f$. To obtain such a function we need to plug a function of type $\prod_{x:A}(f(x) =_{B(x)} f(x))$ into funext. $\mathrm{refl}_{f(x)}$ is an inhabitant of $f(x) =_{B(x)} f(x)$ and therefore we only need to plug $\lambda(x : A).\mathrm{refl}_{f(x)}$ into funext to get a candidate for the right side of our equation.
As a proof we can simply use corollary 2.3.2(ii) together with the defining equality of happly, namely $\mathrm{happly}(\mathrm{refl}_f) \equiv \lambda(x : A).\mathrm{refl}_{f(x)}$.

(ii) Since α^{-1} is of type $g = f$, we need to find the right input function of type $\prod_{(x:A)}(g(x) =_{B(x)} f(x))$, in order to apply funext to it. Such a function can be constructed by assigning x to an inverted $\mathrm{happly}(\alpha, x)$. We therefore predict that there is an equality between the inverted path, and funext applied to the function of inverted point wise equalities obtained by the original path.
We use path induction, with $\alpha \equiv \mathrm{refl}_f$:

$$\begin{aligned}
\alpha^{-1} &\equiv \mathrm{refl}_f^{-1} \\
&\equiv \mathrm{refl}_f \\
&= \mathrm{funext}(\mathrm{happly}(\mathrm{refl}_f)) \\
&\equiv \mathrm{funext}(\lambda(x : A).\mathrm{refl}_{f(x)}) \\
&\equiv \mathrm{funext}(\lambda(x : A).\mathrm{refl}_{f(x)}^{-1}) \\
&\equiv \mathrm{funext}(\lambda(x : A).\mathrm{happly}(\mathrm{refl}_f, x)^{-1}) \\
&\equiv \mathrm{funext}(\lambda(x : A).\mathrm{happly}(\alpha, x)^{-1}).
\end{aligned}$$

(iii) Using the same arguments as in (ii), it makes sense to predict an equality between the composition and funext applied to the function of the composition of point wise equalities obtained via happly by the original paths.
The proof is conducted by path induction first on α and then on β, looking only at the case that $g \equiv f$ and $h \equiv f$ and both $\alpha \equiv \beta \equiv \mathrm{refl}_f$:

$$\begin{aligned}
\alpha \cdot \beta &\equiv \mathrm{refl}_f \cdot \mathrm{refl}_f \\
&\equiv \mathrm{refl}_f \\
&= \mathrm{funext}(\mathrm{happly}(\mathrm{refl}_f)) \\
&\equiv \mathrm{funext}(\lambda(x : A).\mathrm{refl}_{f(x)}) \\
&\equiv \mathrm{funext}(\lambda(x : A).\mathrm{refl}_{f(x)} \cdot \mathrm{refl}_{f(x)}) \\
&\equiv \mathrm{funext}(\lambda(x : A).\mathrm{happly}(\mathrm{refl}_f, x) \cdot \mathrm{happly}(\mathrm{refl}_f, x)) \\
&\equiv \mathrm{funext}(\lambda(x : A).\mathrm{happly}(\alpha, x) \cdot \mathrm{happly}(\beta, x)).
\end{aligned}$$

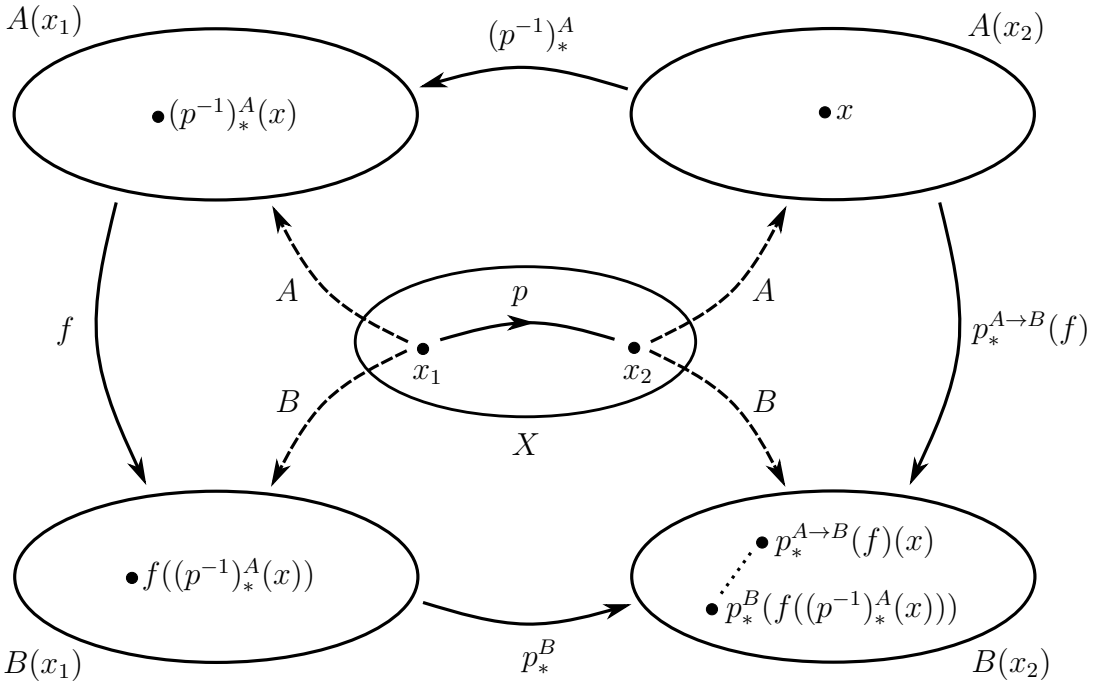
□

We now want to prove an equality that relates the transport of a function with the regular transport of elements. Since the dependent function case is rather complicated, we will first explain and prove the equality for non-dependent functions an then give the generalization afterwards.

Lemma 2.3.4 *Let $X : \mathcal{U}$, $x_1, x_2 : X$, $p : x_1 =_X x_2$ and $A, B : X \to \mathcal{U}$. For any function $f : A(x_1) \to B(x_1)$ the type*

$$p_*^{A\to B}(f) = \Big(\lambda(x : A(x_2)).p_*^B(f((p^{-1})_*^A(x)))\Big),$$

with $A \to B$ denoting the type family over X defined by $(A \to B)(x) :\equiv (A(x) \to B(x))$, is inhabited.



Proof: The function $p_*^{A\to B}$ is of type $(A \to B)(x_1) \to (A \to B)(x_2)$ and therefore the left side of the equation is of type $A(x_2) \to B(x_2)$. To construct such a function we must find an object of type $B(x_2)$ that depends on some variable $x : A(x_2)$. Using the $\lambda(x : A(x_2))$ notation, we would then have a function of the desired type. The obvious way to obtain an element of type $B(x_2)$ from our available data, is to use the transport function p_*^B with an argument of type $B(x_1)$. To obtain such an element we use the function f, with input of type $A(x_1)$. If we can now find an element of $A(x_1)$, depending on $x : A(x_2)$, we are done. To obtain $A(x_1)$ from $A(x_2)$ though is an easy task, if we simply use the $(p^{-1})_*^A$ function with input x. Chaining all these steps together, we get exactly the object on the right side of the equation. In summary, we predict that a function transported along a certain path is equal to the function that first transports its argument backwards along that path, then lets the original function act on the transported variable and finally transports the result back along the original direction of the path.

The proof is conducted by path induction on p. We look at the case that $x_2 \equiv x_1$ and $p \equiv \mathrm{refl}_{x_1}$:

$$\begin{aligned}
p_*^{A\to B}(f) &\equiv (\mathrm{refl}_{x_1})_*^{A\to B}(f)\\
&\equiv \mathrm{id}_{(A\to B)(x_1)}(f)\\
&\equiv f\\
&= \Big(\lambda(x : A(x_1)).f(x)\Big)\\
&\equiv \Big(\lambda(x : A(x_1)).\mathrm{id}_{B(x_1)}(f(x))\Big)\\
&\equiv \Big(\lambda(x : A(x_1)).(\mathrm{refl}_{x_1})_*^{B}(f(x))\Big)\\
&\equiv \Big(\lambda(x : A(x_1)).(\mathrm{refl}_{x_1})_*^{B}(f(\mathrm{id}_{A(x_1)}(x)))\Big)\\
&\equiv \Big(\lambda(x : A(x_1)).(\mathrm{refl}_{x_1})_*^{B}(f((\mathrm{refl}_{x_1})_*^{A}(x)))\Big)\\
&\equiv \Big(\lambda(x : A(x_1)).(\mathrm{refl}_{x_1})_*^{B}(f((\mathrm{refl}_{x_1}^{-1})_*^{A}(x)))\Big)\\
&\equiv \Big(\lambda(x : A(x_2)).p_*^{B}(f((p^{-1})_*^{A}(x)))\Big).
\end{aligned}$$

□

Before we can go on to give the generalized version of the above lemma, we first require a theorem regarding the dependent pair type.

Theorem 2.3.5 *Let $A : \mathcal{U}$, $P : A \to \mathcal{U}$ and $w, w' : \sum_{(x:A)} P(x)$. The type*

$$(w = w') \simeq \sum_{(p:\mathrm{pr}_1(w)=\mathrm{pr}_1(w'))} p_*(\mathrm{pr}_2(w)) = \mathrm{pr}_2(w')$$

is inhabited. The corresponding inverse function will be called

$$\mathrm{pair}^{=} : \sum_{(p:\mathrm{pr}_1(w)=\mathrm{pr}_1(w'))} p_*(\mathrm{pr}_2(w)) = \mathrm{pr}_2(w') \to (w = w'),$$

with

$$\mathrm{pair}^{=}(\mathrm{refl}_x, \mathrm{refl}_y) \equiv \mathrm{refl}_{(x,y)}$$

for $x : A$ and $y : P(x)$.

Recall lemma 1.16.7, which stated that a lifted path is of type $(x, u) = (y, p_*(u))$. For $w \equiv (x, u)$ and $w' \equiv (y, p_*(u))$, we get

$$\mathrm{pair}^{=} : \sum_{(p:x=y)} p_*(u) = p_*(u) \to (x, u) = (y, p_*(u)),$$

and therefore the counterpart of the lift function is

$$\text{pair}^{=}(p, \text{refl}_{p_*(u)}) : (x, u) = (y, p_*(u)).$$

From the above theorem we also obtain our propositional uniqueness principle for the $\sum$-type.

Corollary 2.3.6 *Let $z : \sum_{(x:A)} P(x)$. The type*

$$z = (\text{pr}_1(z), \text{pr}_2(z))$$

is inhabited.

PROOF: For $w \equiv z$ and $w' \equiv (\text{pr}_1(z), \text{pr}_2(z))$, we get

$$\text{pair}^{=} : \sum_{(p:\text{pr}_1(z)=\text{pr}_1(z))} p_*(\text{pr}_2(z)) = \text{pr}_2(z) \to z = (\text{pr}_1(z), \text{pr}_2(z)).$$

Therefore if we take $\text{refl}_{\text{pr}_1(z)}$ as our p, turning $p_*(\text{pr}_2(z)) = \text{pr}_2(z)$ into $\text{pr}_2(z) = \text{pr}_2(z)$, we can apply our $\text{pair}^{=}$ function, obtaining

$$\text{pair}^{=}(\text{refl}_{\text{pr}_1(z)}, \text{refl}_{\text{pr}_2(z)}) : z = (\text{pr}_1(z), \text{pr}_2(z)),$$

which proves our corollary. □

We are now finally ready to generalize lemma 2.3.4.

Lemma 2.3.7 *Let $X : \mathcal{U}$, $x_1, x_2 : X$, $p : x_1 =_X x_2$, $A : X \to \mathcal{U}$, $B : \prod_{(x:X)}(A(x) \to \mathcal{U})$ and $f : \prod_{(a:A(x_1))} B(x_1, a)$. For $a : A(x_2)$, the type*

$$p_*^{\prod_A(B)}(f)(a) = \text{transport}^{\widehat{B}}\Big((\text{pair}^{=}(p^{-1}, \text{refl}_{(p^{-1})_*(a)}))^{-1}, f(\text{transport}^{A}(p^{-1}, a))\Big)$$

is inhabited. We use the notation

$$\prod_A(B) :\equiv \lambda(x : X). \prod_{(a:A(x))} B(x, a)$$
$$\widehat{B} :\equiv \lambda(w : \sum_{(x:X)} A(x)).B(\text{pr}_1(w), \text{pr}_2(w)).$$

PROOF: Even though this equation is a bit more complicated than the one in lemma 2.3.4, the general idea is the same. We know that

$$p_*^{\prod_A(B)} : \prod_A(B)(x_1) \to \prod_A(B)(x_2),$$

which leads to $p_*^{\prod_A(B)}(f)$ being of type $\prod_A(B)(x_2)$, since $f : \prod_A(B)(x_1)$. Using this and the fact that $a : A(x_2)$, we get that

$$p_*^{\prod_A(B)}(f)(a) : B(x_2, a).$$

Now we need to construct an element of this type, using only p, f and a. The first thought that comes to mind, is to again use some sort of transport over B. This time though, B is not only dependent on a variable $x : X$, but also on some variable of $A(x)$. As one can see, A itself is a type family over X and therefore "x_1's $a : A(x_1)$" needn't be the same as "x_2's $a : A(x_2)$". They aren't even necessarily of the same type for different x_1 and x_2. We therefore can't simply transport over x like in the non-dependent function case, while keeping the same a. To solve this, we use a type family $\widehat{B}$, which is now dependent on pairs of form (x, a). This resolves the problem since we can have different a for different x. Our plan therefore is to use a function $\text{transport}^{\widehat{B}}(\dots, \dots)$. If we can now fill up the gaps, we have our candidate for the right side of the equation. For the first gap, we will need a path from (x_1, a') to (x_2, a) with some $a' : A(x_1)$. To this end, we recall the result we got from theorem 2.3.5 above:

$$\text{pair}^{=}(p, \text{refl}_{p_*(u)}) : (x, u) = (y, p_*(u)).$$

We know both x_1 and x_2 as well as "the A variable of x_2", namely a. Since we don't explicitly know our a', we can use $(p^{-1})_*$ to obtain it and then apply our $\text{pair}^{=}$ function to get an element of $(x_2, a) = (x_1, (p^{-1})_*(a))$. Now we just need to invert this path to get

$$(\text{pair}^{=}(p^{-1}, \text{refl}_{(p^{-1})_*(a)}))^{-1} : (x_1, (p^{-1})_*(a)) = (x_2, a).$$

For the second gap we need some element of $B(x_1, (p^{-1})_*(a))$. As in the non-dependent function case, we can use f to obtain it. A look at the type of f quickly shows us, that we only need to plug in $(p^{-1})_*(a)$ into f and we get our desired result

$$f((p^{-1})_*(a)) : B(x_1, (p^{-1})_*(a)).$$

Putting everything together, we get exactly the right side of the lemma's equation. The intuitive idea behind it is the same as in the non-dependent case: we backwards transport the argument a, then apply f to the result and finally transport everything forwards again.

We use path induction on p looking only at the case that $x_2 \equiv x_1$ and $p \equiv \mathrm{refl}_{x_1}$:

$$\begin{aligned}
p_*^{\prod_A(B)}(f)(a) &\equiv (\mathrm{refl}_{x_1})_*^{\prod_A(B)}(f)(a)\\
&\equiv \mathrm{id}_{\prod_A(B)(x_1)}(f)(a)\\
&\equiv f(a)\\
&\equiv \mathrm{id}_{\widehat{B}(x_1,a)}(f)(a)\\
&\equiv \mathrm{transport}^{\widehat{B}}\Big(\mathrm{refl}_{(x_1,a)}, f(a)\Big)\\
&\equiv \mathrm{transport}^{\widehat{B}}\Big(\mathrm{refl}^{-1}_{(x_1,a)}, f(a)\Big)\\
&\equiv \mathrm{transport}^{\widehat{B}}\Big((\mathrm{pair}^{=}(\mathrm{refl}_{x_1}, \mathrm{refl}_a))^{-1}, f(a)\Big)\\
&\equiv \mathrm{transport}^{\widehat{B}}\Big((\mathrm{pair}^{=}(\mathrm{refl}^{-1}_{x_1}, \mathrm{refl}_{\mathrm{id}_{A(x_1)}(a)}))^{-1}, f(\mathrm{id}_{A(x_1)}(a))\Big)\\
&\equiv \mathrm{transport}^{\widehat{B}}\Big((\mathrm{pair}^{=}(\mathrm{refl}^{-1}_{x_1}, \mathrm{refl}_{(\mathrm{refl}_{x_1})_*(a)}))^{-1}, f(\mathrm{transport}^A(\mathrm{refl}_{x_1}, a))\Big)\\
&\equiv \mathrm{transport}^{\widehat{B}}\Big((\mathrm{pair}^{=}(\mathrm{refl}^{-1}_{x_1}, \mathrm{refl}_{(\mathrm{refl}^{-1}_{x_1})_*(a)}))^{-1}, f(\mathrm{transport}^A(\mathrm{refl}^{-1}_{x_1}, a))\Big)\\
&\equiv \mathrm{transport}^{\widehat{B}}\Big((\mathrm{pair}^{=}(p^{-1}, \mathrm{refl}_{(p^{-1})_*(a)}))^{-1}, f(\mathrm{transport}^A(p^{-1}, a))\Big).
\end{aligned}$$

□

Before we finally arrive at the univalence axiom, we end our current section by proving another equivalence. As before we will first show the non-dependent function version and then the general case.

Lemma 2.3.8 *Let $X : \mathcal{U}$, $A, B : X \to \mathcal{U}$ and $p : x =_X y$. For $f : A(x) \to B(x)$ and $g : A(y) \to B(y)$ the type*

$$(p_*(f) = g) \simeq \prod_{a:A(x)} (p_*(f(a)) = g(p_*(a))$$

is inhabited.

Proof: Intuitively this equivalence says that the equality between the transported function f and g is equivalent to the function that gives point wise equalities between the transport of the value of f at that point and the value of g at the transported point.

By using path induction on p we may only look at the case that $p \equiv \mathrm{refl}_x$, which turns the equivalence into

$$(f = g) \simeq \prod_{a:A(x)} (f(a) = g(a)),$$

which is true by axiom 2.3.1. □

Lemma 2.3.9 *Let $X : \mathcal{U}$, $A : X \to \mathcal{U}$, $B : \prod_{(x:X)} A(x) \to \mathcal{U}$ and $p : x =_X y$. For $f : \prod_{(a:A(x))} B(x,a)$ and $g : \prod_{(a:A(y))} B(y,a)$ the type*

$$(p_*(f) = g) \simeq \Big(\prod_{a:A(x)} \mathrm{transport}^{\widehat{B}}(\mathrm{pair}^{=}(p, \mathrm{refl}_{p_*(a)}), f(a)) = g(p_*(a)) \Big),$$

with $\widehat{B}$ defined as in lemma 2.3.7, is inhabited.

Proof: The left part of the equivalence stays the same as in the non-dependent function case. The $g(p_*(a))$ on the right side also doesn't have to change. The transport of $f(a)$ though now requires modifications, since the type of $f(a)$ also has an A dependency now. Therefore as before in lemma 2.3.7, we transport over the family of pairs, namely $\widehat{B}$. The required path is also obtained analogously to lemma 2.3.7, but now we know "x's a", so we can simply use p. Together with our $\mathrm{pair}^{=}$ function we get the right path, namely $\mathrm{pair}^{=}(p, \mathrm{refl}_{p_*(a)})$, to transport $f(a)$ over the family $\widehat{B}$.

We use path induction on p, looking only at the case that $p \equiv \mathrm{refl}_x$, obtaining

$$\begin{aligned}
(p_*(f) = g) &\simeq \Big(\prod_{a:A(x)} \mathrm{transport}^{\widehat{B}}(\mathrm{pair}^{=}(p, \mathrm{refl}_{p_*(a)}), f(a)) = g(p_*(a)) \Big) \\
&\equiv (f = g) \simeq \Big(\prod_{a:A(x)} \mathrm{transport}^{\widehat{B}}(\mathrm{pair}^{=}(\mathrm{refl}_x, \mathrm{refl}_a), f(a)) = g(a) \Big) \\
&\equiv (f = g) \simeq \Big(\prod_{a:A(x)} \mathrm{transport}^{\widehat{B}}(\mathrm{refl}_{(x,a)}, f(a)) = g(a) \Big) \\
&\equiv (f = g) \simeq \Big(\prod_{a:A(x)} f(a) = g(a) \Big),
\end{aligned}$$

with the last equivalence being true by axiom 2.3.1. □

2.4 Univalence axiom

Just like we did for functions in the last section, we now want to find an equivalence regarding types, more specifically we want to have an equivalence between two types being equal and them being equivalent. The first direction of our equivalence is given by the following lemma.

Lemma 2.4.1 *Let $A, B : \mathcal{U}$. There exists a function*

$$\mathrm{idtoeqv} : (A =_{\mathcal{U}} B) \to (A \simeq B),$$

with the equivalence $\mathrm{idtoeqv}(p)$*, or rather its underlying function, being defined by* $\mathrm{idtoeqv}(p) :\equiv p_*^{\mathrm{id}_{\mathcal{U}}}$*, for all all $p : A =_{\mathcal{U}} B$.*

We would now like to extend this function into an equivalence between $(A =_{\mathcal{U}} B)$ and $(A \simeq B)$, but just like with our happly function in the last section, such an equivalence can not be shown with the type theory we introduced in chapter 1. Because of this, we need to introduce another axiom, namely Voevodsky's axiom of univalence.

Axiom 2.4.2 (Voevodsky's axiom of univalence) *Let $A, B : \mathcal{U}$. The function* idtoeqv*, defined in lemma 2.4.1, forms an equivalence*

$$(A =_{\mathcal{U}} B) \simeq (A \simeq B).$$

Universes with this property are called univalent. If we assume this axiom, we know that there exists a quasi-inverse function of idtoeqv, namely

$$\mathrm{ua} : (A \simeq B) \to (A =_{\mathcal{U}} B).$$

Corollary 2.4.3 *Let $f : A \simeq B$ and $p : A =_{\mathcal{U}} B$. The following types are inhabited:*

(i)
$$f = \mathrm{idtoeqv}(\mathrm{ua}(f)).$$

(ii)
$$p = \mathrm{ua}(\mathrm{idtoeqv}(p)).$$

Proof: Since UA states the existence of the equivalence, we also know that there exist two homotopies, i.e. point wise equalities to the identity function, when forming the compositions of idtoeqv and ua. □

Lemma 2.4.4 *Let $A, B, C : \mathcal{U}$, $f : A \simeq B$ and $g : B \simeq C$. The following types are inhabited:*

(i)
$$\mathrm{refl}_A = \mathrm{ua}(\mathrm{id}_A).$$

(ii)
$$\mathrm{ua}(f) \cdot \mathrm{ua}(g) = \mathrm{ua}(g \circ f).$$

(iii)
$$\mathrm{ua}(f)^{-1} = \mathrm{ua}(f^{-1}).$$

Proof:

(i) On the left side of the equation we have refl_A, which is of type $A =_{\mathcal{U}} A$. The obvious object for creating an element of $A =_{\mathcal{U}} A$ is our new function ua, with an input of type $A \simeq A$. From lemma 2.2.6(i) we know that the identity function id_A gives us exactly that type. Intuitively the equality says, that the identity function of a type is the equivalence that produces reflexivity of that type.
We can prove this by using corollary 2.4.3(ii) and the fact, that by definition we have $\mathrm{idtoeqv}(\mathrm{refl}_A) \equiv (\mathrm{refl}_A)^{\mathrm{id}_{\mathcal{U}}}_* \equiv \mathrm{id}_A$, obtaining

$$\begin{aligned} \mathrm{refl}_A &= \mathrm{ua}(\mathrm{idtoeqv}(\mathrm{refl}_A)) \\ &\equiv \mathrm{ua}(\mathrm{id}_A). \end{aligned}$$

(ii) Since $\mathrm{ua}(f) : A =_{\mathcal{U}} B$ and $\mathrm{ua}(g) : B =_{\mathcal{U}} C$, the left side has type $A =_{\mathcal{U}} C$. To get an element of this type, we need to use our function ua with an input of type $A \simeq C$, which we obtain, as stated in 2.2.6(iii), by forming the composition of f and g. We therefore predict that the path obtained by a composition of two equivalences is equal to the concatenation of the separately obtained paths.
For the proof recall lemma 1.16.9, which states that for two paths p and q, with matching end and start point, we have a point wise equality between $q_* \circ p_*$ and $(p \centerdot q)_*$ and therefore, using function extensionality, even $q_* \circ p_* = (p \centerdot q)_*$. To use this lemma, we can now define the paths $p :\equiv \mathrm{ua}(f)$ and $q :\equiv \mathrm{ua}(g)$, and together with corollary 2.4.3 we get

$$\begin{aligned}
\mathrm{ua}(g \circ f) &= \mathrm{ua}(\mathrm{idtoeqv}(\mathrm{ua}(g)) \circ \mathrm{idtoeqv}(\mathrm{ua}(f))) \\
&\equiv \mathrm{ua}(\mathrm{idtoeqv}(q) \circ \mathrm{idtoeqv}(p)) \\
&\equiv \mathrm{ua}(q_* \circ p_*) \\
&= \mathrm{ua}((p \centerdot q)_*) \\
&\equiv \mathrm{ua}(\mathrm{idtoeqv}(p \centerdot q)) \\
&= p \centerdot q \\
&\equiv \mathrm{ua}(f) \centerdot \mathrm{ua}(g).
\end{aligned}$$

(iii) Since $f : A \simeq B$, we have $\mathrm{ua}(f)^{-1} : B =_{\mathcal{U}} A$. We can construct an element of that type by plugging an equivalence of type $B \simeq A$ into ua. We obtain exactly such an equivalence by f^{-1} of lemma 2.2.6(ii). We therefore predict that the inverse of the path constructed by some equivalence, is equal to the path constructed by the equivalence's inverse.
In order to prove this, we first need to show that for a path p, we have $\mathrm{idtoeqv}(p)^{-1} = \mathrm{idtoeqv}(p^{-1})$, which we can quickly prove by path induction over p, assuming that $p \equiv \mathrm{refl}_x$ for an element x of some type:

$$\begin{aligned}
\mathrm{idtoeqv}(p)^{-1} &\equiv (p_*)^{-1} \\
&\equiv ((\mathrm{refl}_x)_*)^{-1} \\
&\equiv \mathrm{id}^{-1} \\
&\equiv \mathrm{id} \\
&\equiv (\mathrm{refl}_x)_* \\
&\equiv (\mathrm{refl}_x^{-1})_* \\
&\equiv (p^{-1})_* \\
&\equiv \mathrm{idtoeqv}(p^{-1}).
\end{aligned}$$

Note that we used the fact that the identity function is its own quasi-inverse. Using

this equality, and the definition of a path $p :\equiv \mathrm{ua}(f)$, we get

$$\begin{aligned}\mathrm{ua}(f^{-1}) &= \mathrm{ua}(\mathrm{idtoeqv}(\mathrm{ua}(f))^{-1}) \\ &\equiv \mathrm{ua}(\mathrm{idtoeqv}(p)^{-1}) \\ &= \mathrm{ua}(\mathrm{idtoeqv}(p^{-1})) \\ &= p^{-1} \\ &\equiv \mathrm{ua}(f)^{-1}.\end{aligned}$$

□

Lemma 2.4.5 *Let $C : A \to \mathcal{U}$, $p : x =_A y$ and $u : C(x)$. The type*

$$\begin{aligned}\mathrm{transport}^C(p, u) &= \mathrm{transport}^{\mathrm{id}_{\mathcal{U}}}(\mathrm{ap}_C(p), u) \\ &= \mathrm{idtoeqv}(\mathrm{ap}_C(p))(u)\end{aligned}$$

is inhabited.

Proof: This lemma is a special case of lemma 1.16.10: if we use that lemma with $P :\equiv \mathrm{id}_{\mathcal{U}}$ and $f :\equiv C$, we get exactly the above. Technically in order to use lemma 1.16.10, we would have to use a function of type $\mathcal{U}_i \to \mathcal{U}_{i+1}$ as our P, but since all elements of $\mathcal{U}_i$ are also in $\mathcal{U}_{i+1}$, we could simply use a modified version of our identity function with type $\mathcal{U}_i \to \mathcal{U}_{i+1}$. Therefore this doesn't pose a problem and we just keep our notation $\mathrm{id}_{\mathcal{U}}$. □

3 Corollaries of the univalence axiom

Assuming the univalence axiom leads to some propositions that contradict familiar set theoretic results. In this chapter we will investigate this problem and introduce the new notion of so called mere proposition in order to resolve it.

3.1 n-types

One of the discerning characteristics between MLITT and regular set theory is, that we have the concept of identity types, whose elements can carry additional information. In set theory an equality between two objects simply states that they are equal, nothing more. This leads us to the following definition of a set in MLITT.

Definition 3.1.1 *A type $A : \mathcal{U}$ is called a set or a 0-type, if the type*

$$\mathrm{isSet}(A) :\equiv \prod_{(x,y:A)} \prod_{(p,q:x=y)} (p = q)$$

is inhabited.

This definition expresses the fact that sets don't carry any extra information in their paths, i.e. all paths between two arbitrary points are equal. Two examples of sets are the type $\mathbf{1}$ and $\mathbf{0}$.

Lemma 3.1.2 *The types* $\mathrm{isSet}(\mathbf{1})$ *and* $\mathrm{isSet}(\mathbf{0})$ *are inhabited.*

Proof: For the $\mathbf{1}$ type, we know by lemma 2.2.5, that for all $x, y : \mathbf{1}$ there exists a function $f : \mathbf{1} \to (x = y)$, which is an equivalence. Using lemma 4.3.5, which we will prove later on, this gives us an element of type

$$\prod_{(p:x=y)} \sum_{(a:\mathbf{1})} (f(a) = p).$$

Therefore, for all $p, q : x = y$, there exist elements $a, b : \mathbf{1}$ with $f(a) = p$ and $f(b) = q$. By theorem 1.13.1, we know that both a and b are equal to $\star$ and therefore equal to each other, i.e. we have some $r : a = b$. We can then apply the function f to this r to obtain $f(r) : f(a) = f(b)$. The concatenation of $f(r)$ with the paths of type $f(a) = p$ and $f(b) = q$ then gives us an element of $p = q$, proving $\mathbf{1}$ to be a set.

For any given $x, y : \mathbf{0}$, the induction function $\mathrm{ind}_{\mathbf{0}}$ allows us to construct an element of any desired type, and therefore also one of $\prod_{(p,q:x=y)}(p = q)$. □

The alternative name 0-type stems from the fact, that we can take this definition one step further and introduce so called 1-types, for which all paths between two given paths are equal.

Definition 3.1.3 *A type $A : \mathcal{U}$ is called a 1-type, if the type*

$$\prod_{(x,y:A)} \prod_{(p,q:x=y)} \prod_{r,s:p=q} (r = s)$$

is inhabited.

In the same fashion we can proceed as far as we wish, defining n-types, with n expressing the last level at which the paths can carry additional information. 0-types for example carry no extra information from the start, i.e. already their first level of paths, p and q, must be equal. The 1-types have their last non-trivial paths at the first level, i.e. paths between two points can be unequal, but paths between paths can not. The fact that n is really the last level of additional information and that there mustn't be any higher level for which paths become unequal again is stated by the following lemma.

Lemma 3.1.4 *If $A : \mathcal{U}$ is an n-type then it also is an $n+1$-type.*

Lemma 3.1.5 *Let $\mathcal{U}$ be a universe. $\mathcal{U}$ is not a set.*

Proof: We will use the type $\mathbf{2}$ to construct a path of type $\mathbf{2} = \mathbf{2}$, for which we will then show, that it is not equal to $\mathrm{refl}_{\mathbf{2}} : \mathbf{2} = \mathbf{2}$. We define a function $f : \mathbf{2} = \mathbf{2}$ by

$$f :\equiv \mathrm{rec}_{\mathbf{2}}(\mathbf{2}, 1_{\mathbf{2}}, 0_{\mathbf{2}}),$$

so that $f(0_{\mathbf{2}}) \equiv 1_{\mathbf{2}}$ and $f(1_{\mathbf{2}}) \equiv 0_{\mathbf{2}}$. Since, by using theorem 1.11.1, one can see that $f(f(x)) = x$, we know that f is an equivalence and therefore univalence gives us a path

$$p :\equiv \mathrm{ua}(f) : \mathbf{2} = \mathbf{2}.$$

We now assume that there is an element of $p = \mathrm{refl}_{\mathbf{2}}$. Corollary 2.4.4(i) states that $\mathrm{refl}_{\mathbf{2}} = \mathrm{ua}(\mathrm{id}_{\mathbf{2}})$, so path concatenation gives us an element of $\mathrm{ua}(f) = \mathrm{ua}(\mathrm{id}_{\mathbf{2}})$. If we now apply the function idtoeqv to this element and use corollary 2.4.3(i), we get an element of $f = \mathrm{id}_{\mathbf{2}}$. The function happly then gives us an element of $f(1_{\mathbf{2}}) = \mathrm{id}_{\mathbf{2}}(1_{\mathbf{2}}) \equiv 0_{\mathbf{2}} = 1_{\mathbf{2}}$, but with theorem 1.16.12, this results in falsum, proving $p \neq \mathrm{refl}_{\mathbf{2}}$. □

Using univalence one can show that for any n, there exists a type, which is not an n-type.

3.2 Limitations of propositions as types

In section 1.14 we introduced a translation between propositions in our natural mathematical language and types in type theory. We will now see that under our assumption of UA, there are certain problems with our current propositions as types approach, since it will produce results that aren't compatible with classical reasoning. The following theorem and corollary show, that double negation and the law of excluded middle are both not true when assuming UA.

Theorem 3.2.1 (Coquand) *It is not true that for all $A : \mathcal{U}$ we have $\neg(\neg A) \to A$.*

Proof: We will assume there exists a function $f : \prod_{(A:\mathcal{U})}(\neg\neg A \to A)$ and construct an element of the $\mathbf{0}$ type. We define the type family $P :\equiv \lambda(A : \mathcal{U}).(\neg\neg A \to A)$. Let $e : \mathbf{2} \simeq \mathbf{2}$ be an equivalence, defined analogously to that of lemma 3.1.5, i.e. $e(0_{\mathbf{2}}) \equiv 1_{\mathbf{2}}$, $e(1_{\mathbf{2}}) \equiv 0_{\mathbf{2}}$ and $p :\equiv \mathrm{ua}(e) : \mathbf{2} = \mathbf{2}$. By lemma 1.16.8 we know that

$$\mathrm{apd}_f : \prod_{(p:A=B)} (p_*^P(f(A)) = f(B)),$$

and therefore

$$\mathrm{apd}_f(p) : p_*^P(f(\mathbf{2})) = f(\mathbf{2}).$$

By using happly we get

$$\mathrm{happly}(\mathrm{apd}_f(p), u) : p_*^P(f(\mathbf{2}))(u) = f(\mathbf{2})(u)$$

for any $u : \neg\neg\mathbf{2}$. If we now use lemma 2.3.4 with $A :\equiv \lambda(C : \mathcal{U}).(\neg\neg C)$ and $B :\equiv \mathrm{id}_{\mathcal{U}}$, the type family $A \to B$ of that lemma becomes P and we get

$$p_*^P(f(\mathbf{2})) = \lambda(x : \neg\neg\mathbf{2}).p_*^{\mathrm{id}_{\mathcal{U}}}\Big(f(\mathbf{2})((p^{-1})_*^{\lambda(C:\mathcal{U}).(\neg\neg C)}(x))\Big)$$

which, by using happly, turns into

$$p_*^P(f(\mathbf{2}))(u) = p_*^{\mathrm{id}_{\mathcal{U}}}\Big(f(\mathbf{2})((p^{-1})_*^{\lambda(C:\mathcal{U}).(\neg\neg C)}(u))\Big).$$

We now observe that all $u, v : \neg\neg\mathbf{2}$ are equal. This is due to the fact, that for any $x : \neg\mathbf{2}$, we get $u(x) : \mathbf{0}$, which allows us to inhabit any type we want, by using $\mathrm{ind}_{\mathbf{0}}$. Therefore we can also inhabit $u(x) = v(x)$, which gives us $u = v$ by function extensionality. If we now use

$$(p^{-1})_*^{\lambda(C:\mathcal{U}).(\neg\neg C)}(u) : \neg\neg\mathbf{2}$$

as our v, we can plug this into the above equality derived from lemma 2.3.4 and get

$$p_*^P(f(\mathbf{2}))(u) = p_*^{\mathrm{id}_{\mathcal{U}}}(f(\mathbf{2})(u)).$$

Together with the equality inhabited by $\mathrm{happly}(\mathrm{apd}_f(p), u)$, we then get

$$f(\mathbf{2})(u) = p_*^{\mathrm{id}_{\mathcal{U}}}(f(\mathbf{2})(u)).$$

From the definition of idtoeqv in lemma 2.4.1, together with lemma 2.4.3(i), we know that

$$p_*^{\mathrm{id}_{\mathcal{U}}} \equiv \mathrm{idtoeqv}(p) \equiv \mathrm{idtoeqv}(\mathrm{ua}(e)) = e$$

and therefore by happly

$$p_*^{\mathrm{id}_{\mathcal{U}}}(f(\mathbf{2})(u)) = e(f(\mathbf{2})(u)).$$

Combining this with the equality above gives us

$$e(f(\mathbf{2})(u)) = f(\mathbf{2})(u).$$

As our final step we show that

$$\prod_{(x:\mathbf{2})} \neg(e(x) = x),$$

which gives us the desired element of the $\mathbf{0}$ type when combined with our derived equality. To prove that this dependent function type is inhabited, we observe that by theorem 1.11.1, all elements of $\mathbf{2}$ are either equal to $0_\mathbf{2}$ or $1_\mathbf{2}$. This gives us $e(x) = e(0_\mathbf{2})$ for those $x : \mathbf{2}$ that are equal to $0_\mathbf{2}$ and $e(x) = e(1_\mathbf{2})$ for those that are equal to $1_\mathbf{2}$. Because $e(0_\mathbf{2}) \equiv 1_\mathbf{2}$ and $e(1_\mathbf{2}) \equiv 0_\mathbf{2}$ we therefore know, since $0_\mathbf{2} \neq 1_\mathbf{2}$ by theorem 1.16.12, that for all possible $x : \mathbf{2}$ the equality $e(x) = x$ results in falsum. Therefore our desired dependent function type is inhabited. □

Corollary 3.2.2 *It is not true that for all $A : \mathcal{U}$ we have $A + (\neg A)$.*

Since we neither want to abandon UA, nor our familiar classical laws, we need to modify how we interpret our types, i.e. we need to rethink our notion of propositions as types.

3.3 Mere propositions

We have already seen many examples of how elements of types can carry more information than simply acting as a proof of the type's corresponding proposition. The fact that we don't merely know if the proposition is true or not is exactly what causes the problems mentioned in the last section. If we have a proof of "A or B" then we will know if the element of $A + B$ which proves this proposition came from A or if it came from B. Similarly if we have a proof of "there exists $x : A$ such that $P(x)$" then we will know exactly which x was the witness of our proof by forming the first projection of the element of $\sum_{(x:A)} P(x)$. If we prove the $\mathbf{1}$ type though, we gain no additional knowledge since all its elements are equal to one another. If we restrict our propositions in MLITT to types like this, that, like set theoretic propositions, carry no extra information, then we can reproduce our classical reasoning without problems.

Definition 3.3.1 *A type $P : \mathcal{U}$ is called a mere proposition if all its elements are equal, i.e if the type*

$$\mathrm{isProp}(P) :\equiv \prod_{(x,y:P)} (x = y)$$

is inhabited.

The word "mere" in the name stems from the fact that we merely know if the proposition is true or not, but have no additional information, since no matter what inhabitant is used to prove it, they are all equal. These mere propositions can now be without restrictions translated into regular mathematical language since the problem of additional information doesn't occur any more.

The correct formulation of the law of excluded middle and double negation therefore are as follows:

$$\text{LEM} :\equiv \prod_{(A:\mathcal{U})} (\text{isProp}(A) \to (A + \neg A))$$

$$\text{DN} :\equiv \prod_{(A:\mathcal{U})} (\text{isProp}(A) \to (\neg\neg A \to A)).$$

Lemma 3.3.2 *Let $P, Q : \mathcal{U}$ be mere propositions and $f : P \to Q$, $g : Q \to P$. The type $P \simeq Q$ is inhabited.*

Proof: Since P is a proposition, we know that $g(f(x))$ for an arbitrary $x : P$ is equal to any element of P. Therefore we also have $g(f(x)) = x$. In the same way we get $f(g(y)) = y$ for any $y : Q$. Therefore f and g are the quasi inverses of each other connecting P and Q and therefore we have $P \simeq Q$. □

Lemma 3.3.3 *If $A : \mathcal{U}$ is a mere proposition, it also is a set.*

Lemma 3.3.4 *Let $A : \mathcal{U}$.* isProp(A) *and* isSet(A) *are mere propositions.*

Proof: Let $f, g :$ isProp(A) and $x, y : A$. Since A is a mere proposition and therefore by lemma 3.3.3 also a set, the paths $f(x, y)$ and $g(x, y)$ are equal. Using function extensionality, this gives us $f = g$ and therefore isProp(A) is a mere proposition.

Let $f, g :$ isSet(A), $x, y : A$ and $p, q : x = y$. Since A is a set, we know that $f(x, y, p, q) = g(x, y, p, q)$ and therefore by function extensionality also $f = g$, proving isSet(A) to be a mere proposition. □

4 Univalence implies function extensionality

Our ultimate goal in this chapter is to show that UA implies function extensionality, so that we can drop axiom 2.3.1 for univalent universes. We will approach this proof analogously to the HoTT-book [11], but also prove additional lemmas 4.3.5, 4.3.6 and 4.3.10. This will enable us to work out the proofs of lemmas 4.3.8 and 4.3.9, as well as the theorems in section 4.4, in a much more precise manner than it is done in [11], providing in this way a complete presentation of the main proof.

4.1 Half adjoint equivalence

Recall our definition

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

from section 2.2, which we used to define type equivalence in definition 2.2.4.

We now want to introduce an alternative concept of equivalence, called half adjoint equivalence.

Definition 4.1.1 *An function $f : A \to B$ is called a half adjoint equivalence, if the type*

$$\mathrm{ishae}(f) :\equiv \sum_{(g:B\to A)} \sum_{(\eta:g\circ f\sim \mathrm{id}_A)} \sum_{(\epsilon:f\circ g\sim \mathrm{id}_B)} \prod_{(x:A)} f(\eta(x)) = \epsilon(f(x))$$

is inhabited.

Theorem 4.1.2 *Let $f : A \to B$. The type* $\mathrm{ishae}(f)$ *is a mere proposition, i.e.*

$$\mathrm{isProp}(\mathrm{ishae}(f))$$

is inhabited.

Theorem 4.1.3 *Let $f : A \to B$. We have an element of* $\mathrm{qinv}(f)$*, if and only if, we have an element of* $\mathrm{ishae}(f)$*.*

By lemmas 2.2.2 and 2.2.3 we also have functions between $\mathrm{qinv}(f)$ and $\mathrm{isequiv}(f)$, which allows us to replace our old notion of equivalence with the new one of half adjoint equivalence. From now on we therefore redefine what we mean by isequiv:

$$\mathrm{isequiv}(f) :\equiv \mathrm{ishae}(f).$$

4.2 Preparatory definitions

Definition 4.2.1 (Fibers) *Let $f : A \to B$ and $y : B$. The type*

$$\mathrm{fib}_f(y) :\equiv \sum_{(x:A)} (f(x) = y)$$

is called the fiber of f over y.

If there is a function

$$g : \prod_{(x:C)} (P(x) \to Q(x))$$

for families $P, Q : C \to \mathcal{U}$, we will call g a fiberwise transformation.

If we know that

$$g(x) : P(x) \to Q(x)$$

is an equivalence for all $x : C$, we call g a fiberwise equivalence.

Definition 4.2.2 (Surjectivity) *An non-dependent function $f : A \to B$ is called surjective, if the type*

$$\mathrm{isSurj}(f) :\equiv \prod_{(y:B)} \mathrm{fib}_f(y)$$

is inhabited.

Definition 4.2.3 *Let $P, Q : A \to \mathcal{U}$ and $f : \prod_{(x:A)} P(x) \to Q(x)$. We define the function* $\mathrm{total}(f)$ *by*

$$\mathrm{total}(f) :\equiv \lambda w.\Big(\mathrm{pr}_1(w), f(\mathrm{pr}_1(w), \mathrm{pr}_2(w))\Big) : \Big(\sum_{(x:A)} P(x)\Big) \to \Big(\sum_{(x:A)} Q(x)\Big).$$

Definition 4.2.4 (Contractibility) *A type $A : \mathcal{U}$ is called contractible, if it is inhabited by some $a : A$, called center point, so that we have $x =_A a$ for all elements $x : A$. We will write*

$$\mathrm{isContr}(A) :\equiv \sum_{(a:A)} \prod_{(x:A)} (a = x)$$

for type corresponding to the proposition that A is contractible.

If the type

$$\prod_{(x:B)} \mathrm{isContr}(P(x)),$$

for a family $P : B \to \mathcal{U}$, is inhabited, we will call P a family of contractible types.

If the type

$$\mathrm{isContr}(f) :\equiv \prod_{(y:B)} \mathrm{isContr}(\mathrm{fib}_f(y)),$$

for a function $f : A \to B$, is inhabited, we will call f contractible.

Definition 4.2.5 *Let $A, B : \mathcal{U}$. B is called a retract of A, if we have a function $r : A \to B$, called retraction, a function $s : B \to A$, called the retraction's section, and a homotopy $\epsilon : \prod_{(y:B)}(r(s(y)) = y)$.*

4.3 Preparatory lemmas

Lemma 4.3.1 *Let $X : \mathcal{U}$, $A : X \to \mathcal{U}$ and $P : \prod_{(x:X)} A(x) \to \mathcal{U}$. There exists a function of type*

$$\Big(\prod_{(x:X)} \sum_{(a:A(x))} P(x,a)\Big) \to \Big(\sum_{(g:\prod_{(x:X)} A(x))} \prod_{(x:X)} P(x, g(x))\Big),$$

which forms an equivalence.

Lemma 4.3.2 *Let $f : A \to B$. The type*

$$\mathrm{isContr}(f) \simeq \mathrm{isequiv}(f)$$

is inhabited.

Lemma 4.3.3 *Let $P, Q : A \to \mathcal{U}$, $x : A$, $v : Q(x)$, and $f : \prod_{(x:A)}(P(x) \to Q(x))$ a fiberwise transformation. The type*

$$\mathrm{fib}_{\mathrm{total}(f)}((x,v)) \simeq \mathrm{fib}_{f(x)}(v)$$

is inhabited.

Lemma 4.3.4 *Let $P : A \to \mathcal{U}$, $\mathrm{pr}_1 : \sum_{(x:A)} P(x)$ and $a : A$. The type*

$$\mathrm{fib}_{\mathrm{pr}_1}(a) \simeq P(a)$$

is inhabited.

Lemma 4.3.5 *Let $f : A \to B$. There exists a function of type*

$$\mathrm{isequiv}(f) \to \mathrm{isSurj}(f).$$

Proof: Assume we have an element of $\mathrm{isequiv}(f)$. Lemma 4.3.2 then gives us some element g of $\mathrm{isContr}(f)$, i.e. we have

$$g : \prod_{(y:B)} \mathrm{isContr}(\mathrm{fib}_f(y)).$$

By applying g to any $y : B$, we obtain an inhabitant $g(y) : \mathrm{isContr}(\mathrm{fib}_f(y))$, which in turn gives us, when forming the first projection $\mathrm{pr}_1(g(y))$, an element of $\mathrm{fib}_f(y)$. We can therefore construct the function

$$\lambda(y : B).\mathrm{pr}_1(g(y)) : \prod_{(y:B)} \mathrm{fib}_f(y),$$

proving our lemma. □

Lemma 4.3.6 *Let $A, B : \mathcal{U}$. If $A \simeq B$ is inhabited, then we have an element of* $\mathrm{isContr}(A)$ *if and only if we have an element of* $\mathrm{isContr}(B)$.

Proof: We will only show the direction from $\mathrm{isContr}(A)$ to $\mathrm{isContr}(B)$. The reverse is completely analogous. Assume we have

$$\gamma : \mathrm{isContr}(A) \equiv \sum_{(a:A)} \prod_{(x:A)} (a = x).$$

By the induction principle of the $\sum$-type, we may assume that $\gamma \equiv (a, P)$, for some $a : A$ and $P : \prod_{(x:A)}(a = x)$. If we can now construct an element

$$\epsilon : \mathrm{isContr}(B) \equiv \sum_{(b:B)} \prod_{(y:B)} (b = y),$$

we are done with the proof.

Since we have some element of $A \simeq B$, we also have a function $f : A \to B$ and an element of $\mathrm{isequiv}(f)$. The first component of ϵ will be $f(a)$, i.e. we define

$$\epsilon :\equiv (f(a), \dots),$$

with a of course being obtained from γ by the pr_1 function. By using the second projection of γ and applying it to some $x : A$, we get

$$P(x) : a = x.$$

If we now let f act on this path, we obtain

$$\beta :\equiv f(P(x)) : f(a) = f(x).$$

Since we have an element of $\mathrm{isequiv}(f)$, lemma 4.3.5 tells us that we also have some element

$$\alpha : \prod_{(y:B)} \sum_{(x:A)} (f(x) = y),$$

which gives us

$$\mathrm{pr}_2(\alpha(y)) : f(x) = y,$$

for any $y : B$. By concatenating β and the above path, we get

$$\beta \cdot \mathrm{pr}_2(\alpha(y)) : f(a) = y,$$

so we can construct the function

$$\lambda(y : B).\Big(\beta \cdot \mathrm{pr}_2(\alpha(y))\Big) : \prod_{(y:B)} (f(a) = y).$$

With this we can now fully define our ϵ as

$$\epsilon :\equiv \Big(f(a), \lambda(y : B).(\beta \cdot \mathrm{pr}_2(\alpha(y)))\Big).$$

□

Lemma 4.3.7 *Let $A, B : \mathcal{U}$ and let B be a retract of A. There exists a function*

$$\text{isContr}(A) \to \text{isContr}(B).$$

PROOF: The proof can be done analogously to that of lemma 4.3.6, because all we used there, was the fact that we had a surjective function from A to B. Since B is a retract of A, we have such a function $r : A \to B$, together with function $s : B \to A$ and homotopy $\epsilon : \prod_{(y:B)}(r(s(y)) = y)$, which give us surjectivity by

$$\lambda(y : B).(s(y), \epsilon(y)) : \text{isSurj}(r).$$

□

Lemma 4.3.8 *Let $A : \mathcal{U}$ and $a : A$. The type*

$$\text{isContr}\Big(\sum_{(x:A)}(a = x)\Big)$$

is inhabited.

PROOF: We will use the element (a, refl_a) as our center point. Let $(x, p) : \sum_{(x:A)}(a = x)$. We now have to show the equality $(a, \text{refl}_a) = (x, p)$. If we use the result

$$\text{pair}^{=}(p, \text{refl}_{p_*(u)}) : (x, u) = (y, p_*(u)),$$

from section 2.3, we get

$$\text{pair}^{=}(p, \text{refl}_{p_*(\text{refl}_a)}) : (a, \text{refl}_a) = (x, p_*(\text{refl}_a)).$$

If we can now show that $p_*(\text{refl}_a) = p$, we are finished, since by simple function application we then obtain $(x, q_*(\text{refl}_a)) = (x, p)$ and therefore our desired equality. We can prove $p_*(\text{refl}_a) = p$ by path induction on $p : a = x$, looking only at the case that $p \equiv \text{refl}_a$:

$$\begin{aligned} p_*(\text{refl}_a) &\equiv (\text{refl}_a)_*(\text{refl}_a) \\ &\equiv \text{id}_{a=a}(\text{refl}_a) \\ &\equiv \text{refl}_a \\ &\equiv p. \end{aligned}$$

□

Lemma 4.3.9 *Let $P, Q : A \to \mathcal{U}$, $x : A$, $v : Q(x)$, and $f : \prod_{(x:A)}(P(x) \to Q(x))$ a fiberwise transformation. f forms a fiberwise equivalence if and only if* $\text{total}(f)$ *forms an equivalence.*

PROOF: We need to show

$$\prod_{(x:A)} \text{isequiv}(f(x)) \text{ if and only if isequiv}(\text{total}(f)).$$

Using lemma 4.3.2, we can prove this, by instead showing $\prod_{(x:A)} \text{isContr}(f(x))$ if and only if isContr(total(f)), i.e.

$$\prod_{(x:A)} \prod_{(y:Q(x))} \text{isContr}(\text{fib}_{f(x)}(y)) \text{ if and only if } \prod_{w:\sum_{(x:A)} Q(x)} \text{isContr}(\text{fib}_{\text{total}(f)}(w)).$$

By the induction principle of the $\sum$-type, we may assume that w is an ordered pair (x, y), writing $\prod_{(x:A)} \prod_{(y:Q(x))} \text{isContr}\Big(\text{fib}_{\text{total}(f)}((x, y))\Big)$ instead of the above. Lemma 4.3.3, together with lemma 4.3.6, gives us

$$\text{isContr}(\text{fib}_{f(x)}(v)) \text{ if and only if isContr}\Big(\text{fib}_{\text{total}(f)}((x, v))\Big)$$

for arbitrary $x : A$ and $v : Q(x)$. Using this, we can obviously construct the above, proving the lemma. □

Lemma 4.3.10 *Let $f : A \to B$ and A and B be contractible. The type* isContr(f) *is inhabited.*

PROOF: We need to show

$$\prod_{(y:B)} \text{isContr}(\text{fib}_f(y)).$$

We know that A is inhabited by $a :\equiv \text{pr}_1(\text{isContr}(A)) : A$. Since B is contractible we have $b :\equiv \text{pr}_1(\text{isContr}(B)) : B$ with $b = f(a)$, but also $b = y$, for any $y : B$. Therefore we have for all $y : B$ that $f(a) = y$. We can now fix $y : B$ and define the element $c :\equiv (a, q) : \text{fib}_f(y)$, with

$$q :\equiv \Big(\text{pr}_2(\text{isContr}(B))(f(a))\Big)^{-1} \cdot \Big(\text{pr}_2(\text{isContr}(B))(y)\Big) : f(a) = y.$$

We now have to show that for any element $w : \text{fib}_f(y)$, we know that $c = w$ is inhabited. Using the induction principle of the $\sum$-type we may assume that $w \equiv (x, p)$ for some $x : A$ and $p : f(x) = y$. Using theorem 2.3.5 we can prove $c = w$, by showing that

$$\sum_{(k:\text{pr}_1(c)=\text{pr}_1(w))} k_*^P(\text{pr}_2(c)) = \text{pr}_2(w),$$

with $P :\equiv \lambda(x : A).(f(x) = y) : A \to \mathcal{U}$, is inhabited. We have the element

$$k :\equiv \text{pr}_2(\text{isContr}(A))(x) : a = x$$

as the first component. For the second one we use path induction, first on k, looking at the case that $x \equiv a$, i.e. $k \equiv \mathrm{refl}_a$, $p : f(a) = y$, and then on q, looking at the case that $y \equiv f(a)$, i.e. $q \equiv \mathrm{refl}_{f(a)}$, $p \equiv \mathrm{refl}_{f(a)}$:

$$\begin{aligned} k^P_*(q) = p &\equiv \mathrm{id}_{P(a)}(q) = p \\ &\equiv q = p \\ &\equiv \mathrm{refl}_{f(a)} = \mathrm{refl}_{f(a)}. \end{aligned}$$

We therefore have

$$(k, \mathrm{refl}_{\mathrm{refl}_{f(a)}}) : \sum_{(k:\mathrm{pr}_1(c)=\mathrm{pr}_1(w))} k^P_*(\mathrm{pr}_2(c)) = \mathrm{pr}_2(w),$$

proving our lemma. □

4.4 The main proof

We will conduct the proof in two steps. We will first show that UA implies so called weak function extensionality, and then prove that this weak version implies the regular one of axiom 2.3.1. Of course we won't assume function extensionality as an axiom for this section and universes will only be univalent if explicitly denoted so.

Definition 4.4.1 *Let $A : \mathcal{U}$ and $P : A \to \mathcal{U}$. Weak function extensionality states that there exists a function of type*

$$\Big(\prod_{(x:A)} \mathrm{isContr}(P(x))\Big) \to \mathrm{isContr}\Big(\prod_{(x:A)} P(x)\Big).$$

Lemma 4.4.2 *Let $\mathcal{U}$ be univalent, $A, B, X : \mathcal{U}$ and $(f, e) : A \simeq B$. The type*

$$(X \to A) \simeq (X \to B),$$

with underlying function $\lambda(a : X \to A).\lambda(x : X).f(a(x))$, is inhabited.

Proof: By UA we know that the function idtoeqv : $(A = B) \to (A \simeq B)$ forms an equivalence and is therefore, by lemma 4.3.5, surjective. Consequently there exists a $p : A = B$, such that $(f, e) = \mathrm{idtoeqv}(p)$. We can now use path induction, assuming that $p \equiv \mathrm{refl}_A$, and therefore $f \equiv p^{\mathrm{id}_{\mathcal{U}}}_* \equiv \mathrm{id}_A$. The function given in the lemma then becomes the identity function on $X \to A$, which, as we know by lemma 2.2.6(i), constitutes an equivalence. □

Corollary 4.4.3 *Let $\mathrm{pr}_1 : (\sum_{(x:A)} P(x)) \to A$ and $P : A \to \mathcal{U}$ be a family of contractible types. There exists an element of $\mathrm{isequiv}(\mathrm{pr}_1)$, and if $\mathcal{U}$ is univalent, there also exists an element*

$$\alpha : \Big(A \to \sum_{(x:A)} P(x)\Big) \simeq (A \to A),$$

with underlying function $\lambda(a : A \to \sum_{(x:A)} P(x)).\lambda(x : A).\mathrm{pr}_1(a(x))$.

PROOF: Let $x : A$. Using lemma 4.3.4, we know

$$\mathrm{fib}_{\mathrm{pr}_1}(x) \simeq P(x),$$

which, together with lemma 4.3.6, gives us

$$\mathrm{isContr}\Big(\mathrm{fib}_{\mathrm{pr}_1}(x)\Big) \text{ if and only if } \mathrm{isContr}(P(x)).$$

Since P is a family of contractible types, we therefore know

$$\prod_{(x:A)} \mathrm{isContr}(\mathrm{fib}_{\mathrm{pr}_1}(x)) \equiv \mathrm{isContr}(\mathrm{pr}_1)$$

and by lemma 4.3.2, we then get $\mathrm{isequiv}(\mathrm{pr}_1)$. Since we now have an inhabitant of

$$\Big(\sum_{(x:A)} P(x)\Big) \simeq A,$$

with underlying function pr_1, lemma 4.4.2 directly gives us the equivalence and underlying function as written in our current lemma. □

Theorem 4.4.4 *Univalence implies weak function extensionality.*

PROOF: We need to show that if $P : A \to \mathcal{U}$ is a family of contractible types, with $\mathcal{U}$ being univalent, then $\mathrm{isContr}(\prod_{(x:A)} P(x))$ is inhabited. We assume that P is a family of contractible types, i.e.

$$\prod_{(x:A)} \mathrm{isContr}(P(x)),$$

which lets us use corollary 4.4.3, giving us a function

$$\alpha :\equiv \lambda(a : A \to \sum_{(x:A)} P(x)).\lambda(x : A).\mathrm{pr}_1(a(x)) : \Big(A \to \sum_{(x:A)} P(x)\Big) \to (A \to A).$$

From the corollary we also know $\mathrm{isequiv}(\alpha)$, which together with lemma 4.3.2 gives us $\mathrm{isContr}(\alpha)$ and therefore $\mathrm{isContr}(\mathrm{fib}_\alpha(\mathrm{id}_A))$. If we now show that $\prod_{(x:A)} P(x)$ is a retract of $\mathrm{fib}_\alpha(\mathrm{id}_A)$, we can use lemma 4.3.7 to obtain $\mathrm{isContr}(\prod_{(x:A)} P(x))$, proving our lemma.

To show that $\prod_{(x:A)} P(x)$ is a retract of

$$\mathrm{fib}_\alpha(\mathrm{id}_A) \equiv \sum_{(g:A\to\sum_{(x:A)} P(x))} (\alpha(g) = \mathrm{id}_A),$$

we must construct the functions

$$r : \mathrm{fib}_\alpha(\mathrm{id}_A) \to \prod_{(x:A)} P(x) \text{ and } s : \Big(\prod_{(x:A)} P(x)\Big) \to \mathrm{fib}_\alpha(\mathrm{id}_A),$$

together with a homotopy

$$\prod_{(f:\prod_{(x:A)} P(x))} (r(s(f)) = f.$$

Using the induction principle of the $\sum$-type, we can define the first function r, by defining it on the ordered pairs of form (g, p), with $g : A \to \sum_{(x:A)} P(x)$ and $p : \alpha(g) = \mathrm{id}_A$. We use the function

$$\mathrm{happly} : (\alpha(g) = \mathrm{id}_A) \to \prod_{(x:A)} \Big(\alpha(g, x) = \mathrm{id}_A(x)\Big),$$

which gives us the transport function

$$\mathrm{happly}(p, x)^P_* : P(\alpha(g, x)) \to P(x)$$

for any $x : A$. We now need to find an element of $P(\alpha(g, x))$. By looking at the definition of α, we see that $\alpha(g, x) \equiv \mathrm{pr}_1(g(x))$, and that an element of the matching type $P(\alpha(g, x))$ is obtained by simply forming the second projection instead of the first:

$$\mathrm{pr}_2(g(x)) : P(\mathrm{pr}_1(g(x))).$$

We can therefore define our function r by

$$r(g, p) :\equiv \lambda(x : A).\mathrm{happly}(p, x)^P_*\Big(\mathrm{pr}_2(g(x))\Big).$$

For the second function, we define $s(f)$, for any $f : \prod_{(x:A)} P(x)$, as an ordered pair. Since

$$\alpha(g) \equiv \lambda(x : A).\mathrm{pr}_1(g(x)),$$

we define our function g, i.e. the pair's first component, in such a way, that $\mathrm{pr}_1(g(x))$ is simply x. This way, $\alpha(g) \equiv \mathrm{id}_A$, so we can use $\mathrm{refl}_{\mathrm{id}_A}$ as the pair's second component. Our $g : A \to \sum_{(x:A)} P(x)$ must therefore be of form $\lambda(x : A).(x, ...)$, with the gap being of type $P(x)$. For this gap though, we can simply use our input function f, defining s by

$$s(f) :\equiv (\lambda(x : A).(x, f(x)), \mathrm{refl}_{\mathrm{id}_A}).$$

Now we observe that we can simplify $r(s(f))$ for any $f : \prod_{(x:A)} P(x)$ as follows:

$$\begin{aligned}
r(s(f)) &\equiv r\Big((\lambda x.(x, f(x)), \mathrm{refl}_{\mathrm{id}_A})\Big) \\
&\equiv \lambda x.\mathrm{happly}(\mathrm{refl}_{\mathrm{id}_A}, x)^P_*\Big(\mathrm{pr}_2((x, f(x)))\Big) \\
&\equiv \lambda x.(\mathrm{refl}_x)^P_*\Big(f(x)\Big) \\
&\equiv \lambda x.\mathrm{id}_{P(x)}\Big(f(x)\Big) \\
&\equiv \lambda x.f(x).
\end{aligned}$$

With this, theorem 1.7.1 gives us the equality between $r(s(f))$ and f, so that we now also have our desired homotopy, completing the proof. □

Theorem 4.4.5 *Weak function extensionality implies function extensionality.*

PROOF: We want to show function extensionality, i.e. inhabit

$$\prod_{(A:\mathcal{U})} \prod_{(P:A\to\mathcal{U})} \prod_{(f,g:\prod_{(x:A)} P(x))} \Big((f = g) \simeq \prod_{(x:A)} (f(x) = g(x))\Big).$$

Since we already defined a suitable function

$$\mathrm{happly}(f, g) : (f = g) \to \prod_{(x:A)} (f(x) = g(x))$$

in section 2.3, we only need to show

$$\prod_{(A:\mathcal{U})} \prod_{(P:A\to\mathcal{U})} \prod_{(f,g:\prod_{(x:A)} P(x))} \mathrm{isequiv}(\mathrm{happly}(f, g)).$$

Note that unlike before, we wrote happly as a function of its two defining functions, to highlight that we have different happly for different input functions. We can now rewrite $\mathrm{happly}(f, g)$, using a fiberwise transformation

$$\alpha :\equiv \lambda(g : \prod_{(x:A)} P(x)).\mathrm{happly}(f, g) : \prod_{(g:\prod_{(x:A)} P(x))} \Big((f = g) \to (f \sim g)\Big),$$

which allows us to apply lemma 4.3.9. This lemma's proposition of α being a fiberwise equivalence means, that we have $\mathrm{isequiv}(\mathrm{happly}(f, g))$ for all g, which, since our f was also arbitrary, is exactly what we need to prove the theorem. Therefore, by using lemma 4.3.9, we can prove our theorem by showing $\mathrm{isequiv}(\mathrm{total}(\alpha))$, i.e. we need to show that

$$\mathrm{total}(\alpha) : \Big(\sum_{g:\prod_{(x:A)} P(x)} (f = g)\Big) \to \Big(\sum_{g:\prod_{(x:A)} P(x)} (f \sim g)\Big)$$

forms an equivalence. By lemma 4.3.8, we know that the domain's type is contractible. If we can also show that the codomain is contractible, then we know that $\mathrm{total}(\alpha)$ is contractible by lemma 4.3.10. We can then use lemma 4.3.2 to obtain $\mathrm{isequiv}(\mathrm{total}(\alpha))$, proving our theorem. Lemma 4.3.1 tells us that the codomain is equivalent to the type

$$X :\equiv \prod_{(x:A)} \sum_{(u:P(x))} f(x) = u.$$

This equivalence provides us with the functions and the homotopy required to show that the codomain is a retract of X. By lemma 4.3.7 we therefore only need to show that X is contractible. By lemma 4.3.8, we know that $\sum_{(u:P(x))} f(x) = u$ is contractible, since $f(x) : P(x)$. We therefore know

$$\prod_{(x:A)} \mathrm{isContr}\Big(\sum_{(u:P(x))} f(x) = u\Big),$$

from which weak function extensionality gives us $\mathrm{isContr}(X)$. □

5 Conclusion

There are more applications of univalence that can be found in the HoTT-book [11], for example in the theory of higher inductive types (see chapter 6), in the theory of n-types (see chapter 7) and in the category theory within univalent type theory (see chapter 9). A milestone in the study of univalence is the cubical model of homotopy type theory, given in [3] and [5], where a constructive interpretation of univalence is given.

References

[1] S. Awodey, M. A. Warren. Homotopy theoretic models of identity types, Mathematical Proceedings of the Cambridge Philosophical Society, 2009, 146:45-55.

[2] E. Bishop: *Foundations of Constructive Analysis*, McGraw-Hill, 1967.

[3] C. Cohen, T. Coquand, S. Huber and A. Mörtberg: Cubical type theory: a constructive interpretation of the univalence axiom, 2016, to appear in the Proceedings of TYPES 2015.

[4] M. Hofmann, T. Streicher: The groupoid interpretation of type theory, G. Sambin, J. M. Smith (Eds.) *Twenty-five years of constructive type theory* (Venice, 1995), volume 36 of Oxford Logic Guides, Oxford University Press, 1998, 83-111.

[5] S. Huber: *Cubical Interpretations of Type Theory*, PhD Thesis, University of Gothenburg, 2016.

[6] C. Kapulkin, P. LeFanu Lumsdaine, V. Voevodsky: The simplicial model of univalent foundations, arXiv:1211.2851, 2012.

[7] P. Martin-Löf: An intuitionistic theory of types: predicative part, in H. E. Rose and J. C. Shepherdson (Eds.) *Logic Colloquium '73*, pp.73-118, North-Holland, 1975.

[8] P. Martin-Löf: *Intuitionistic type theory: Notes by Giovanni Sambin on a series of lectures given in Padua, June 1980*, Napoli: Bibliopolis, 1984.

[9] P. Martin-Löf: An intuitionistic theory of types, in G. Sambin, J. M. Smith (Eds.) *Twenty-five years of constructive type theory* (Venice, 1995), volume 36 of Oxford Logic Guides, Oxford University Press, 1998, 127-172.

[10] N. Rigas: The function type revisited, preprint, 2017.

[11] The Univalent Foundations Program: *Homotopy Type Theory: Univalent Foundations of Mathematics*, Institute for Advanced Study, Princeton, 2013.

[12] V. Voevodsky: A very short note on the homotopy λ-calculus, in http://www.math.ias.edu/∼vladimir/Site3/Univalent_Foundations_files /Hlambda_short_current.pdf, 2006.

Eigenständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfasst zu haben und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt zu haben.

München, den 29. Mai 2018

Armin Eghdami