A BRIEF INTRODUCTION TO TYPE THEORY AND THE UNIVALENCE AXIOM

JACKSON MACOR

Abstract. In this paper, we will introduce the basic concepts and notation of modern type theory in an informal manner. We will discuss functions, type formation, the nature of proof, and proceed to prove some basic results in sentential logic. These efforts will culminate in a proof of the axiom of choice in type theory. We will further introduce a few concepts in homotopy type theory, a modern invention which seeks to provide a foundation of mathematics without ZFC set theory. We will conclude with the univalence axiom, an indispensable tool in homotopy type theory, and use it to prove a stronger version of the axiom of choice.

Contents

1. Introduction

The notion of type theory begins with Bertrand Russell's efforts to resolve certain paradoxes in the set theory of his time, such as that which arises when one considers the set of all sets which do not contain themselves. For him, a type was the range of significance of a propositional function, that is, if $\phi(x)$ is a propositional function, the "place" from which x may be taken such that $\phi(x)$ is a sensible assertion is given the title "type". For instance, if $\phi(x)$ stands for "x is either true or false", then x should be a proposition. However, given his aim was to avoid paradoxes which arise when one considers objects such as the set of all sets which do not contain themselves, his investigations led him to formulate something much more akin to ordering in modern logic. Specifically, his logical types were the type of individuals, the type of first-order functions, the type of second-order functions, and so forth. In contrast, modern type theory is a completely distinct system which was developed much later in the 20th century.

The basic notion in type theory is that each object is assigned a type, and this type is something to which the object is explicitly linked. This is similar to the way in which mathematicians informally use the notation of first-order logic and set theory in stating things such as

$$(\forall x \in \mathbb{R})(x \leq 0 \vee x > 0).$$

This is in fact bad syntax in the notation of first-order logic, but in writing like this one implicitly assigns x to a type, that of the real numbers. To be completely correct, one ought to write

$$(\forall x)(x \in \mathbb{R} \to (x \leq 0 \vee x > 0))$$

which leaves one to wonder from where x is being drawn. As stated, in type theory, any object is assigned a type from which it cannot be separated. Thus, the type-theoretic statement of the above would be more akin to the first than the second.

There is of course much more to type theory than this mere notational convention. First of all, types are not sets (though a type may sometimes be a set), so types behave quite differently. As a general note, while a set is completely characterized by the elements which compose it, a type is not characterized by the objects which inhabit it. It exists independently from these objects. Moreover, while set theory is a two-layer system with the sets being supported by the system of first-order logic, type theory only deals with types. Working in type theory is thus substantially different from working in set theory.

We will now proceed to build our type-theoretic system. Section 2 will present the foundational tools of equality and functions along with the notation and rules governing their use. Section 3 will build upon these basics by introducing the formal procedure for type formation, allowing us to construct the dependent function type and the dependent pair type. With these notational and conceptual devices, we will present proofs of some basic results in sentential logic as well as the axiom of choice. Section 4 will expand upon our type-theoretic system by introducing the basics of homotopy type theory, including the univalence axiom. We will then use these new tools to prove a stronger version of the axiom of choice.

2. A Primer to Type Theory

The fundamental judgement in type theory is prescribing some object to a type. If an object a belongs to a type A, we write "$a : A$" and say "a is of type A", "a is an inhabitant of A", and the like. Types themselves inhabit types called *universes* which for bookkeeping purposes are arranged in a hierarchy. That is, we have a chain of types $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \ldots$. This chain is cumulative, in that if $A : \mathcal{U}_i$ for some i, then $A : \mathcal{U}_{i+1}$. This is important in formal type theory when, for instance, one constructs a new type out of given types, for one needs to keep track of which universe this new type inhabits with respect to the given types. However, since we are presenting type theory in a somewhat informal manner, and will not run into potential paradoxes, we will not maintain such rigorous bookkeeping. The notion of universes will be used, however, when describing type families.

One last matter must be noted before we start, that being the treatment of equality in type theory. In type theory, two objects can only be equal with respect to a certain type. So if we wish to state that a and b are equal inhabitants of A, we write $a = b : A$. Moreover, in keeping with the notion that type theory is a "single layer" system, the equality of two inhabitants of a given type is itself a type, so if a and b are any two inhabitants of a type A, we have a type $a =_A b$. When this type is inhabited, a and b are said to be *propositionally equal.* (Exactly what this means will be discussed later.) Another kind of equality is *judgmental equality* or *definitional equality*, in which case we would have instead written $a \equiv b : A$. In understanding the distinction, it is perhaps best to think of the prior as equality *a posteriori* and the latter as equality *a priori.* As another analogy, one may think of Frege's example of the morning star and the evening star. One does not know *a priori* that the two objects are the same, but through later investigation, one can determine *a posteriori* that they are both Venus. Returning to types, one may think of this analogy as applying to two types which are not judgmentally equal, but are determined to be propositionally equal (i.e. we construct an inhabitant of $A =_{\mathcal{U}} B)$. In general, judgmental equality is a stronger condition than propositional equality. That is, if we have $a \equiv b : A$, then $a =_A b$ is inhabited. We list the rules for judgmental equality for completeness.

$\equiv$-Reflexivity: Given a type A and $a : A$, then $a \equiv a : A$.

$\equiv$-Symmetry: Given a type A and $a \equiv b : A$, then $b \equiv a : A$.

$\equiv$-Transitivity: Given a type A, if $a \equiv b : A$ and $b \equiv c : A$, then $a \equiv c : A$.

Equality of Types: Given types A and B and $a : A$, if $A \equiv B : \mathcal{U}$, then $a : B$.

Remark 2.1. Note that these rules do not gives us a general criterion for establishing judgmental type equality apart from $A \equiv A : \mathcal{U}$ by $\equiv$-Reflexivity. This marks one way in which types differ from sets, for while it is an assumption that two sets A and B are equal if and only if $(\forall x)(x \in A \leftrightarrow x \in B)$, the above rules only tell us that $a : A$ implies $a : B$ and $b : B$ implies $b : A$ is a necessary condition of two types A and B being judgmentally equal. This reinforces our earlier assertion that while sets are entirely characterized by their elements, types have content in their own right. We shall discuss equality in type theory in greater detail in section 4.

With this matter settled, we move on to the formation of the function type.

Given types A and B, we can form the function type $A \to B$, the inhabitants of which are functions with domain A and codomain B.

In type theory, functions are taken as primitive, as opposed to set theory in which they are defined to be particular elements of a Cartesian product. It is thus better not to think of a function in type theory as a set of ordered pairs, but as a rule, say f, applied to an inhabitant of A, say a, which when executed yields an inhabitant of B, denoted $f(a)$. As a result of these differing conceptions of functions, the notational mechanics of functions in type theory differ in some ways from those in set theory. Namely, type theory makes use of the λ-calculus of Church, which we now introduce.

Let us take $f : A \to B$ and define it by giving an equation

$$f(x) \equiv \phi : B$$

where x is an arbitrary inhabitant of A and ϕ is an expression possibly containing x which inhabits B for $x : A$, that is, for $x : A$, $\phi : B$. In order to make a function out of the expression ϕ, we use λ-abstraction, or λ-abstraction over x, and write

$$(\lambda(x:A).\phi):A\to B.$$

While the matter of differentiating between the mere expression ϕ and the function $(\lambda(x:A).\phi)$ is, for the mathematician, one of notation, it is an important bookkeeping tool in type theory. Specifically, in keeping with the rigor of type theory in assigning objects to types, we must keep a clear distinction between ϕ as an expression denoting some inhabitant of B and $\lambda x(x:A).\phi$ as a function which when applied to an inhabitant of A yields an inhabitant of B.

Example 2.2. Let A and B both be $\mathbb{R}$ and $f(x)\equiv x^2+2x+2$ (i.e. we take ϕ to be the expression x^2+2x+2). As it stands, x^2+2x+2 is not a function, but merely one unspecified inhabitant of $\mathbb{R}$. λ-abstraction over x allows this unspecified real number to range over all real numbers, so while x^2+2x+2 is not a function, $\lambda(x:\mathbb{R}).(x^2+2x+2)$ is a function from $\mathbb{R}$ to $\mathbb{R}$.

In the future, we shall write $\lambda x.\phi$ in place of $\lambda(x:A).\phi$. For instance, if we wish to apply the function $(\lambda(x:A).\phi):A\to B$ to $a:A$, we write $(\lambda x.\phi)(a):B$.

To better understand this notation, note for any function $f:A\to B$, we can construct another function $\lambda x.f(x):A\to B$ where $x:A$, which applies the expression $f(x)$, symbolically, to all inhabitants of A. Moreover, these functions are posited to be definitionally equal, that is,

$$f\equiv(\lambda x.f(x)):A\to B.$$

This is known as the uniqueness principle for function types, and is a general rule for the function type.

The notation of the λ-calculus may easily accommodate a notion of function composition. For instance, if $f:A\to B$ and $g:B\to C$, then for $x:A$, we have $f(x):B$, so we may write $g(f(x)):C$. We may further apply λ-abstraction to this expression to obtain a function which when applied to an inhabitant of A yields an inhabitant of C. Thus,

$$\lambda x.g(f(x)):A\to C$$

Moreover, this function is defined to be the composition of g with f, and we write

$$\lambda x.g(f(x))\equiv g\circ f:A\to C$$

The notation of the λ-calculus is also by no means limited to one variable. For instance, if $f:A\times B\to C$ (we describe the Cartesian product for types below), and $x:A$ and $y:B$, we write

$$\lambda x.\lambda y.f(x,y)\equiv f:A\times B\to C.$$

If we apply $\lambda x.\lambda y.f(x,y)$ to an inhabitant $a:A$, we obtain

$$(\lambda x.\lambda y.f(x,y))(a)\equiv\lambda y.f(a,y):B\to C.$$

We may of course apply $\lambda y.f(a,y)$ to an inhabitant $b:B$ and obtain

$$(\lambda y.f(a,y))(b) \equiv f(a,b) : C.$$

In this manner, $f : A \times B \to C$ is rather realized as $f : A \to (B \to C)$. Functions of several variables are often thought of in this way in type theory. As a result, one often sees $f(a)(b)$ as opposed to $f(a,b)$, though the choice of notation is rather fluid. However, if one sees $f(a,b)$, one should still think of it as a rule applied first to a and then after to b. This view of multivariable functions is called "currying" after the mathematician and logician Haskell Curry. With the basics of the λ-calculus outlined, we proceed to the formation of new types from given types.

3. Pure Type Theory

We will now form two new types from given types, namely the dependent function type and the dependent pair type. Both these types are reliant upon *families of types*, that is, some collection of types indexed by inhabitants of another type. For instance, if A is a type, we may have the family of types $B(x)$ where $x : A$. Instead of this notation, we write $B : A \to \mathcal{U}$, and the family of types is obtained by applying the function B to inhabitants of A. We next note that in type theory, new types are built out of given types by describing them in term of the following rules:

- A formation rule, which tells one how to form the new type.
- An introduction rule, which tells one how to construct inhabitants of that type.
- An elimination rule, which tells one how to use or apply inhabitants of that type.
- A computation rule, which expresses how an eliminator acts on a constructor.
- An optional uniqueness principle, which expresses uniqueness of maps into or out of that type.

We begin with the dependent function type, the inhabitants of which are functions for which the codomain is dependent upon the inhabitant of the domain to which it is applied.

Dependent Function Type

Π-Formation: Given the type A and the family of types $B : A \to \mathcal{U}$, we form the dependent function type $\prod\limits_{x:A} B(x)$.

Π-Introduction: Given the type A and the family of types $B : A \to \mathcal{U}$, if $x : A$ then $b : B(x)$, then $\lambda x.b : \prod\limits_{x:A} B(x)$.

Remark 3.1. Note, this also establishes the procedure for λ-abstraction over x.

Π-Elimination: Given $f : \prod\limits_{x:A} B(x)$ and $a : A$, then $f(a) : B(a)$.

Π-Computation: Given a type A and a family of types $B : A \to \mathcal{U}$, if $x : A$ then $b : B(x)$, and $a : A$, then $(\lambda x.b)(a) \equiv b[a/x] : B(a)$.

Remark 3.2. Note, $b[a/x]$ denotes the expression obtained by substituting $a : A$ for any possible occurrences of the variable x in the expression b.

Π-Uniqueness: Given $f : \displaystyle\prod_{x:A} B(x)$, $f \equiv \lambda x.f(x) : \displaystyle\prod_{x:A} B(x)$.

As one can see, the inhabitants of $\prod_{x:A} B(x)$ are functions from A to members of the family of types $B(x)$ where the precise codomain is dependent upon the inhabitant of A to which the function is applied. However, if B does not depend on x, then the codomain of each inhabitant of $\prod_{x:A} B(x)$ is simply B. Thus, we write

$$\prod_{x:A} B \equiv (A \to B) : \mathcal{U}.$$

Thus, in order to form the regular function type, $A \to B$, it is sufficient to take it as a special case of the dependent function type, which is why we did not describe the function type formally above. One can check that the uniqueness principle for functions types is the same as Π-Uniqueness where B does not depend on x.

Example 3.3. Let P denote the type of primes. We form the dependent function type $\prod_{p:P} \mathbb{Z}/(p)$. Now let $n : \mathbb{Z}$ and take the expression $(n \bmod p)$ for $p : P$. Note for any prime p, $(n \bmod p) : \mathbb{Z}/(p)$. Hence, $\lambda p.(n \bmod p) : \prod_{p:P} \mathbb{Z}/(p)$. Thus, the map $p \mapsto (n \bmod p)$ is a dependent function.

We now describe the dependent pair type, the inhabitants of which are ordered pairs in which the type of the second coordinate is dependent upon the first coordinate.

Dependent Pair Type

Σ-Formation: Given the type A and the family of types $B : A \to \mathcal{U}$, we form the dependent pair type $\displaystyle\sum_{x:A} B(x)$.

Σ-Introduction: Given the type A and the family of types $B : A \to \mathcal{U}$, if $a : A$ and $b : B(a)$, then $(a, b) : \displaystyle\sum_{x:A} B(x)$.

In order to set forth the rules for Σ-Elimination and Σ-Computation, we must introduce the induction operator for dependent pair types. The induction operator, denoted $\text{ind}_{\sum_{x:A} B(x)}$, is in fact precisely defined by the rules for Σ-Elimination and Σ-Computation. While the statements of these rules may appear mysterious at first, they ultimately express something quite simple, which we shall unpack below.

Σ-Elimination: Given a family of types $C : \Big(\displaystyle\sum_{x:A} B(x)\Big) \to \mathcal{U}$, $p : \displaystyle\sum_{x:A} B(x)$, and $g : C((x, y))$, then $\text{ind}_{\sum_{x:A} B(x)}(C, g, p) : C(p)$.

Σ-Computation: Given a family of types $C : \Big(\displaystyle\sum_{x:A} B(x)\Big) \to \mathcal{U}$, $g : C((x, y))$, $a : A$, and $b : B(a)$, then $\text{ind}_{\sum_{x:A} B(x)}(C, g, (a, b)) \equiv g(a, b) : C((a, b))$.

The inhabitants of $\sum_{x:A} B(x)$ are simply ordered pairs in which the first coordinate comes from A and the second from one of the members of the family of types

$B(x)$ where the precise member is determined by the first coordinate. Note, if B does not depend on x, then we simply have a type consisting of ordered pairs where the first coordinate comes from A and the second from B. Thus, we write

$$\sum_{x:A} B \equiv (A \times B) : \mathcal{U}$$

Thus, in order to form the Cartesian product type, $A \times B$, it is sufficient to take it as a special case of the dependent pair type, which is why we did not describe the Cartesian product type formally above.

An additional word concerning the induction operator for dependent pair types, Σ-Elimination, and Σ-Computation is warranted. In the statements of Σ-Elimination and Σ-Computation, C is a family of types indexed by inhabitants of $\sum_{x:A} B(x)$, g is a function mapping canonical inhabitants (ordered pairs) to the type in this family indexed by that inhabitant, and (a, b) is such an inhabitant. Σ-Elimination simply tells us $\mathrm{ind}_{\sum_{x:A} B(x)}$ is a map sending inhabitants of the type $\prod_{a:A} \prod_{b:B(a)} C((a, b))$ to inhabitants of the type $\prod_{p:\sum_{x:A} B(x)} C(p)$. That is, functions the domain of which is the canonical inhabitants of a dependent pair type (e.g. those of the form (a, b) where $a : A$ and $b : B(a)$), are mapped by Σ-Elimination via the induction operator to functions the domain of which is the arbitrary inhabitants of that dependent pair type (e.g. any $p : \sum_{x:A} B(x)$), such that the codomain of each function is preserved. Σ-Computation simply tells us, as the name suggests, how to compute $\mathrm{ind}_{\sum_{x:A} B(x)}(C, g, (a, b))$.

Remark 3.4. The canonical inhabitants of a type are defined in the introduction rules. One may think of canonical form as the elemental form of an inhabitant.

Having constructed the dependent pair type, we may proceed to define the projection functions for dependent pair types.

Definition 3.5 (Left Projection)**.** The left projection function, denoted pr_1, is defined by the following judgments:

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

$$\mathrm{pr}_1((a, b)) \equiv a.$$

Defining the other projection function is not so simple an affair, since the codomain of such a function will vary with the first coordinate of the argument. Specifically, it must be a dependent function the type of which involves the left projection function:

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

We now apply Σ-Elimination. Namely, to construct a dependent function from a dependent pair type to a family $C : (\sum_{x:A} B(x)) \to \mathcal{U}$, we need a function

$$g : \prod_{a:A} \prod_{b:B(a)} C((a, b)).$$

$\mathrm{ind}_{\sum_{x:A} B(x)}$ then gives us a function

$$f : \prod_{p:\sum_{x:A} B(x)} C(p)$$

defined by

$$f((a,b)) \equiv g(a)(b).$$

Letting $C(p) \equiv B(\mathrm{pr}_1(p))$ in the preceeding steps, we obtain the following definition:

Definition 3.6 (Right Projection)**.** The right projection function, denoted pr_2 is defined by the following judgments:

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

$$\mathrm{pr}_2((a,b)) \equiv b.$$

Remark 3.7. Note pr_2 is an output of the induction operator, which is why we had to go through the above construction. Furthermore, one can convince oneself that this is correct, since $B(\mathrm{pr}_1((a,b))) \equiv B(a)$ and indeed $b : B(a)$. In fact, pr_1 is also an output of the induction operator, though in that case, the type family in question is constant.

From the above, for $p : \sum_{x:A} B(x)$, one has the seemingly obvious equality

$$p \equiv (\mathrm{pr}_1(p), \mathrm{pr}_2(p)) : \sum_{x:A} B(x)$$

known as the uniqueness principle for dependent pair types. Informally, we know p must reduce to a canonical inhabitant of $\sum_{x:A} B(x)$, that is, $p \equiv (a,b)$ for some $a : A$ and $b : B(a)$, and clearly $p \equiv (a,b) \equiv (\mathrm{pr}_1((a,b)), \mathrm{pr}_2((a,b)))$, so by $\equiv$-Transitivity, $p \equiv (\mathrm{pr}_1(p), \mathrm{pr}_2(p))$. However, this notion of reduction to a canonical inhabitant is by no means acceptable in a formal proof, and in fact the formal proof is rather technical. So while we shall make use of this fact, we omit the proof here.[1] We note, however, that the above expressions are proved to be propositionally equal. That is,

$$p = (\mathrm{pr}_1(p), \mathrm{pr}_2(p)) : \sum_{x:A} B(x).$$

In fact, judgmental equality between these two expressions is not provable.

Having developed enough machinery to speak in type-theoretic terms, we move on to discuss the nature of proof and proposition in type theory. A key difference between type theory and the standard system of set theory and first-order logic is that type theory is not an analytic or semantic system. In set theory and first-order logic, a proposition is seen as simply being a representation of a state of affairs which is either true or false. A proposition in type theory on the other hand is not merely seen as true or false, but as the collection of all possible witnesses to its truth,

[1] See Corollary 2.7.3 in *Homotopy Type Theory* for the complete proof.

that is, inhabitants of that proposition. The motivation for this conception of the truth of a proposition lies in the origins of modern type theory in computer science. As a consequence of these origins, a proposition is conceived as a specification of a computation, and true propositions are those whose specification is satisfied by some terminating computation, that is, yield an inhabitant after being executed or applied. Thus, in the type-theoretic proofs we exhibit later, we shall proceed by constructing inhabitants of a certain type.

Example 3.8. To see how this idea plays out, let us consider the case where $B : A \to \mathcal{U}$ is a family of types for which $B(a)$ corresponds to the proposition "$B(a)$ holds for $a : A$". If $f : \prod_{x:A} B(x)$, then for each $a : A$, we can find an inhabitant $f(a) : B(a)$, from which we conclude that $B(x)$ holds for all $a : A$. Thus, the dependent function type gives us a way to represent universal quantification. Similarly, taking A as B as above, if $(a, b) : \sum_{x:A} B(x)$, then $b : B(a)$, so $B(x)$ is inhabited for some $a : A$. Thus, the dependent pair type similarly gives us a way to represent existential quantification in a propositional context. If B does not depend on x, $\prod_{x:A} B(x) \equiv A \to B$, so if $f : A \to B$, then for $a : A$, $f(a) : B$, from which we conclude B holds if A holds. Hence, the function type gives us a way to represent the sentential connective implication. Similarly, $\sum_{x:A} B(x) \equiv A \times B$, so if $(a, b) : A \times B$, then $a : A$ and $b : B$, from which we conclude both A and B hold. Hence, the Cartesian product type gives us a way to represent the sentential connective conjunction.

We include here a few type-theoretic proofs of basic results in sentential logic as further examples.

Proposition 3.9. *Let A be a type. Then $A \to A$ is inhabited.*

Proof. Take $x : A$. By Π-Introduction, $\lambda x.x : A \to A$. □

Taking A to be a propositional type, this simply tells us A implies A.

Proposition 3.10. *Take the type A and the type family $B(x) : A \to \mathcal{U}$. Then $\prod\limits_{x:A}(B(x) \to A)$ is inhabited.*

Proof. Take $x : A$ and $y : B(x)$. By λ-abstraction on y, $\lambda y.x : B(x) \to A$. By λ-abstraction on x, $\lambda x.\lambda y.x : \prod_{x:A}(B(x) \to A)$. □

Supposing B does not depend on x, $\prod_{x:A}(B(x) \to A) \equiv \prod_{x:A}(B \to A)$, and since $B \to A$ does not depend on x, we obtain $A \to (B \to A)$. Taking A and B to be propositions, showing the above type is inhabited establishes that $A \to (B \to A)$ is true, that is, A implies that B implies A.

Proposition 3.11. *Take the type A and type families $B : A \to \mathcal{U}$ and $C : (\sum_{x:A} B(x)) \to \mathcal{U}$. Then*

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

is inhabited.

Proof. Let $x : A$, $f : \prod_{x:A} B(x)$, and $g : \prod_{x:A} \Big(\prod_{y:B(x)} C(x,y) \Big)$.
By Π-Elimination:

$$f(x) : B(x) \text{ and } g(x) : \prod_{y:B(x)} C(x,y).$$

By Π-Elimination:

$$[g(x)](f(x)) : C(x, f(x))$$

By λ-abstraction on x:

$$\lambda x.[g(x)](f(x)) : \prod_{x:A} C(x, f(x)).$$

By λ-abstraction on f:

$$\lambda f.\lambda x.[g(x)](f(x)) : \prod_{f:\prod_{x:A} B(x)} \Big(\prod_{x:A} C(x, f(x)) \Big).$$

By λ-abstraction on g:

$$\lambda g.\lambda f.\lambda x.[g(x)](f(x)) : \bigg(\prod_{x:A} \Big(\prod_{y:B(x)} C(x,y) \Big) \bigg) \to \bigg(\prod_{f:\prod_{x:A} B(x)} \Big(\prod_{x:A} C(x, f(x)) \Big) \bigg).$$

□

If we suppose C does not depend on x or y, the above type simplifies to

$$(\textstyle\prod_{x:A}(B(x) \to C)) \to (\prod_{f:\prod_{x:A} B(x)}(A \to C)).$$

If we further suppose B does not depend on x, this type simplifies to

$$(A \to (B \to C)) \to ((A \to B) \to (A \to C)).$$

Taking A, B, and C to be propositions, showing the above type is inhabited establishes the truth of the second axiom of Hilbert's propositional calculus.

Up to this point, it may seem as though the construction of this formal system is merely an exercise, and the method of proof presented above a curiosity. However, we now present a type-theoretic proof of the axiom of choice to show that this is not the case, and that type theory is in some ways stronger and more complete than set theory.

Theorem 3.12 (The Axiom of Choice, Version 1)**.** *Take a type A and type families $B : A \to \mathcal{U}$ and $C : (\sum_{x:A} B(x)) \to \mathcal{U}$. Then*

$$\bigg(\prod_{x:A} \Big(\sum_{b:B(x)} C(x,b) \Big) \bigg) \to \bigg(\sum_{g:\prod_{x:A} B(x)} \Big(\prod_{x:A} C(x, g(x)) \Big) \bigg)$$

is inhabited.

To see why this statement is the axiom of choice, one may think of A as an indexing set, $B(x)$ as a family of sets over A, and $C(x,b)$ as a relation where if $x : A$, there is at least one $b : B(x)$ such that $(x,b) : C$. In this case we may roughly translate the above statement as

$$(\forall x : A)(\exists b : B(x))(C(x,b)) \to (\exists g : \textstyle\prod_{x:A} B(x))(\forall x : A)(C(x,g(x))).$$

Thus, $g : \prod_{x:A} B(x)$ is simply a choice function. However, it should be remembered that this is a general statement about types.[2]

Example 3.13. Let $A \equiv \{2,3,5\}$, $B(2) \equiv \{7,9\}$, $B(3) \equiv \{11,13\}$, and $B(5) \equiv \{17,19,23\}$. Take $C \equiv \{(2,7),(2,1),(3,4),(3,13),(5,23),(5,9)\}$. In this case we may take the choice function defined by $g(2) \equiv 7$, $g(3) \equiv 13$, and $g(5) \equiv 23$. Note $g : \prod_{x:A} B(x)$ since $g(2) \equiv 7 : B(2)$, $g(3) \equiv 13 : B(3)$, and $g(5) \equiv 23 : B(5)$. Moreover, $(2,g(2)) \equiv (2,7) : C$, $(3,g(3)) \equiv (3,13) : C$, and $(5,g(5)) \equiv (5,23) : C$.

Proof. [3] Take $f : \displaystyle\prod_{x:A}\Big(\sum_{b:B(x)} C(x,b)\Big)$ and $x : A$.
By Π-Elimination:

$$f(x) : \sum_{b:B(x)} C(x,b).$$

By Left Projection:

$$\mathrm{pr}_1(f(x)) : B(x).$$

By Right Projection:

$$\mathrm{pr}_2(f(x)) : C(x, \mathrm{pr}_1(f(x))).$$

By Π-Introduction:

$$\lambda x.\mathrm{pr}_1(f(x)) : \prod_{x:A} B(x).$$

By Π-Computation:

$$(\lambda x.\mathrm{pr}_1(f(x)))(x) \equiv \mathrm{pr}_1(f(x)) : B(x).$$

By Substitution:

$$C(x, (\lambda x.\mathrm{pr}_1(f(x)))(x)) \equiv C(x, \mathrm{pr}_1(f(x))) : \mathcal{U}.$$

By Equality of Types:

$$\mathrm{pr}_2(f(x)) : C(x, (\lambda x.\mathrm{pr}_1(f(x)))(x)).$$

By λ-abstraction on x:

$$\lambda x.\mathrm{pr}_2(f(x)) : \prod_{x:A} C(x, (\lambda x.\mathrm{pr}_1(f(x)))(x)).$$

By Σ-Introduction:

[2] The situation concerning how to properly represent the axiom of choice in type theory is in fact more complicated than we are making it out to be. See section 3.8 in *Homotopy Type Theory* for a more complete discussion.

[3] This proof is based on one found on p. 50 of Sambin's *Intuitionistic Type Theory.*

$$(\lambda x.\mathrm{pr}_1(f(x)), \lambda x.\mathrm{pr}_2(f(x))) : \sum_{g:\prod_{x:A} B(x)} \Big(\prod_{x:A} C(x, g(x))\Big)$$

where $g \equiv \lambda x.\mathrm{pr}_1(f(x))$ on the right side.

By λ-abstraction on f:

$$\lambda f.(\lambda x.\mathrm{pr}_1(f(x)), \lambda x.\mathrm{pr}_2(f(x))) : \Big(\prod_{x:A}\Big(\sum_{b:B(x)} C(x,b)\Big)\Big) \to \Big(\sum_{g:\prod_{x:A} B(x)} \Big(\prod_{x:A} C(x, g(x))\Big)\Big).$$

□

As one would expect, this proof is not translatable into the notation of set theory and first-order logic, as it relies on the unique method of proof used in type theory. Whereas in set theory, we would have had to construct $g : \prod_{x:A} B(x)$ explicitly, it is sufficient in type theory to construct a map with domain $\prod_{x:A}(\sum_{b:B(x)} C(x,b))$ and codomain $\sum_{g:\prod_{x:A} B(x)}(\prod_{x:A} C(x, g(x)))$. The map $\lambda f.(\lambda x.\mathrm{pr}_1(f(x)), \lambda x.\mathrm{pr}_2(f(x)))$ is such a function, and the above proof simply constructs this function according to the rules we have listed above.

4. Homotopy Type Theory

From the above, it is evident that type theory is a system of some power. However, one way in which the system we have described so far is lacking is its treatment of equality. We have the rules of judgmental equality listed above, but these have limited usefulness. For instance, as noted in remark 2.1, it does not provide us with a way to show two types are equal. Furthermore, if one wishes to show two objects, whether they be types or inhabitants, are equal and it is not deducible from the rules for judgmental equality that they are so, one must construct an inhabitant of their identity type in the manner of the above proofs. For instance, to show that $A, B : \mathcal{U}$ are propositionally equal, one must construct an inhabitant of $A =_{\mathcal{U}} B$. As seen in the above proofs, such constructions can often be tedious and unenlightening. The only generality we have is that if $a \equiv b : A$, then $a =_A b$ is inhabited. This inhabitant is called the reflexivity function.

$$\mathrm{refl} : \prod_{a:A}(a =_A a)$$

refl is thus a map from inhabitants of A to inhabitants of the types of the form $a =_A a$.

This is where homotopy type theory becomes useful. To very briefly sketch how it contributes to type theory, under a homotopic interpretation, types are regarded as spaces and inhabitants of identity types as continuous paths between points in that space. Built upon this intuition, we can establish a general criterion for propositional equality called the *univalence axiom*. But first, we must lay forth a few definitions.

Definition 4.1. Let $f, g : \prod_{x:A} B(x)$ be two sections of a type family $B : A \to \mathcal{U}$. A homotopy from f to g is a dependent function of the type

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

This definition of homotopy is in fact consistent with our normal notion for homotopy as a continuous deformation of paths. For saying $\prod_{x:A}(f(x) = g(x))$ is inhabited tells us there is a continuous map from $x : A$ to paths between $f(x)$ and $g(x)$, which is the same as giving us a continuous deformation of f into g.

With a suitable notion of homotopy, we may define a quasi-inverse of a function.

Definition 4.2. For a function $f : A \to B$, a quasi-inverse of f is a triple (g, α, β) consisting of a function $g : B \to A$ and homotopies $\alpha : f \circ g \sim \mathrm{id}_B$ and $\beta : g \circ f \sim \mathrm{id}_A$. The type of quasi-inverses of f is denoted $\mathrm{qinv}(f)$.

In many ways, quasi-inverses replace the concept of literal inverses in set theory. We prefer this since the notions of isomorphism and bijection are reserved for types which behave like sets. One significant role played by quasi-inverses is in establishing equivalences between types. In the same way one would use the existence of an inverse to establish that the cardinalities of two sets are equal or to show two algebraic objects are isomorphic, showing that a quasi-inverse exists for a map between two types is sufficient to show that they are equivalent. However, unlike standard set-based topology, in which demonstrating the existence of a quasi-inverse directly establishes homotopy equivalence between two spaces, homotopy type theory makes use of an alternate criterion for equivalence, which in practice is technically advantageous.[4]

Definition 4.3. Given types A and B, the type of equivalences between A and B, denoted $\mathrm{isequiv}(f)$, is defined by

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

An inhabitant of this type is a quadruple of the form (g, α, h, β) such that $\alpha : f \circ g \sim \mathrm{id}_B$ and $\beta : h \circ f \sim \mathrm{id}_A$.

Definition 4.4 (Equivalence)**.** Given types A and B, an equivalence from A to B is defined to be a function $f : A \to B$ along with an inhabitant of $\mathrm{isequiv}(f)$. The type of equivalences is denoted $(A \simeq B)$, and we write

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

Proposition 4.5. *Given types A and B, for each $f : A \to B$, there is a function $\mathrm{qinv}(f) \to \mathrm{isequiv}(f)$.*

Proof. It is evident from the definitions that $(g, \alpha, \beta) \mapsto (g, \alpha, g, \beta)$ is such a function. □

Hence, to show two types A and B are equivalent, it is sufficient to find a map $f : A \to B$ for which there exists a quasi-inverse, for in such a case, one shows $\mathrm{qinv}(f)$ is inhabited, from which the above map tells you $\mathrm{isequiv}(f)$ is inhabited, from which one knows by Σ-Introduction that $\sum_{f:A\to B} \mathrm{isequiv}(f)$, hence $(A \simeq B)$, is inhabited, from which one concludes that A and B are equivalent.

[4] See sections 2.4, 4.1, and remark 2.10.4 of *Homotopy Type Theory* for more details.

So far, the above content seems to merely be another exercise in formalism. However, the notion of equivalence in homotopy type theory is turned into something which provides genuine type-theoretic content by the univalence axiom, proposed by Vladimir Voevodsky, which we now define.

Axiom 4.6 (Univalence)**.** *For any $A, B : \mathcal{U}$, the canonical map*

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

is an equivalence. Hence, we have

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

What this axiom does is give us a general tool for showing two types are propositionally equal which, as noted above, was lacking in pure type theory. Specifically, by the above note, to show two types are propositionally equal, it suffices to find a function between those types for which there exists a quasi-inverse. To show that this tool affords substantial logical content to type theory, we conclude with a stronger version of the axiom of choice.

Theorem 4.7 (Axiom of Choice, Version 2)**.** *The function defined in Theorem 3.12 has a quasi-inverse. Hence, we have*

$$\Big(\prod_{x:A}\Big(\sum_{b:B(x)} C(x,b)\Big)\Big) = \Big(\sum_{g:\prod_{x:A} B(x)} \Big(\prod_{x:A} C(x, g(x))\Big)\Big) : \mathcal{U}.$$

Proof. [5] Take the map

$$(j,k) \mapsto \lambda x.(j(x), k(x)).$$

We show this is a quasi-inverse of $\lambda f.(\lambda x.\mathrm{pr}_1(f(x)), \lambda x.\mathrm{pr}_2(f(x)))$. First, note by function composition that

$$\lambda f.(\lambda x.\mathrm{pr}_1(f(x)), \lambda x.\mathrm{pr}_2(f(x))) \equiv \lambda f.(\mathrm{pr}_1 \circ f, \mathrm{pr}_2 \circ f).$$

Hence, the map defined in Theorem 3.12 is simply $f \mapsto (\mathrm{pr}_1 \circ f, \mathrm{pr}_2 \circ f)$. Let $f : \prod_{x:A}(\sum_{b:B(x)} C(x,b))$. f is thus mapped to $(\mathrm{pr}_1 \circ f, \mathrm{pr}_2 \circ f)$. By $(j,k) \mapsto \lambda x.(j(x), k(x))$, this is mapped to $\lambda x.((\mathrm{pr}_1(f(x)), \mathrm{pr}_2(f(x)))$. By the uniqueness principle for Σ-types, for $x : A$,

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

So by Π-Uniqueness

$$\lambda x.(\mathrm{pr}_1(f(x)), \mathrm{pr}_2(f(x))) = \lambda x.f(x) = f.$$

[5] This proof is based on that of Theorem 2.15.7 in *Homotopy Type Theory*.

Now let $(j, k) : \sum_{g:\prod_{x:A} B(x)}(\prod_{x:A} C(x, g(x)))$. This is mapped to $\lambda x.(j(x), k(x))$, which is then mapped to $(\mathrm{pr}_1 \circ (\lambda x.(j(x), k(x))), \mathrm{pr}_2 \circ (\lambda x.(j(x), k(x))))$ by the map defined in Theorem 3.12. Note for any $x : A$,

$$(\mathrm{pr}_1(j(x), k(x)), \mathrm{pr}_2(j(x), k(x))) = (j(x), k(x)).$$

So by function composition and Π-Uniqueness,

$$(\mathrm{pr}_1 \circ (\lambda x.(j(x), k(x))), \mathrm{pr}_2 \circ (\lambda x.(j(x), k(x)))) = (\lambda x.j(x), \lambda x.k(x)) = (j, k).$$

Hence, taking α and β to both be the propositional identity function, joining them to the map $(j, k) \mapsto \lambda x.(j(x), k(x))$ gives us a triple which is indeed a quasi-inverse of the map defined in Theorem 3.12. □

Thus, with the introduction of the univalence axiom and the other machinery of homotopy type theory, we are able to show that the antecedent and the consequent of the statement of the axiom of choice are in fact logically identical.

It is to be expected that in such a short piece we have only scratched the surface of type theory. This is even more so in the case of homotopy type theory, which is the subject of more interest to the modern mathematician. However, the above can hopefully serve as a primer for those who may wish to explore homotopy type theory in greater detail.

Acknowledgments. I would like to thank my mentors, Sean Howe and Abigail Ward, for introducing me to the topic of this paper, their help in working through difficulties, and their patience in making sense of a somewhat unorthodox subject. I would also like to thank Professor Peter May for organizing and directing the REU, along with Professor Michael Shulman for ensuring the accuracy of this paper.

References

[1] Gabbay, Dov M.; Kanamori, Akihiro; Woods, John. Ed. *Handbook of the History of Logic*, Vol. 6 *Sets and Extensions in the Twentieth Century.* Elsevier: Amsterdam, 2012.

[2] Russell, Bertrand and Whitehead, Alfred North. *Principia Mathematica*, Vol. I. Cambridge University Press: New York, 1925.

[3] Sambin, Giovanni. *Intuitionistic Type Theory.* Bibliopolis: Naples, 1984.

[4] *Homotopy Type Theory: Univalent Foundations of Mathematics* The Univalent Foundations Program, 2013. Available http://homotopytypetheory.org/book/.