CARLETON UNIVERSITY

SCHOOL OF
MATHEMATICS AND STATISTICS

HONOURS PROJECT

TITLE: Homotopy Type Theory

AUTHOR: Melanie Brown

SUPERVISOR: Dr Brett Stevens

DATE: 07 January 2020

Homotopy Type Theory

Melanie Brown — MATH 4905

1 Introduction

Homotopy type theory (HoTT) is an exciting new branch of mathematics. Using ideas from topology, category theory, and algebraic geometry, it can be used as an alternative to Zermelo-Fraenkel (ZF) set theory as a foundation for mathematics. The purpose of this paper is threefold: to raise awareness for this exciting field of study; to challenge the reader to work in an unfamiliar formal system; and finally, to showcase alternative viewpoints for well-studied areas of classical mathematics.

HoTT consists of a handful of additional axioms on top of a formal system known as **intensional Martin-Löf type theory** (IML). At the outset, this may seem like a very different system to the familiar universe of sets and elements of ZF theory. However, as the foundations are developed and we work our way up the ladder of abstraction, the reader will begin to notice the similarities and subtler differences between the two foundations.

The paper is structured as follows. We begin by discussing formal systems in general, how they function, and what it means to "do mathematics" inside one. Then, we formulate the structure of IML using the language of formal systems, and describe a more comfortable syntax moving forward. Next we discuss the history of HoTT and the areas in which it makes IML more expressive. Finally, we describe the familiar notions of sets, predicates, and algebraic structures, and discuss some of the differences between the sets of HoTT and their classical ZF-theoretic analogues.

2 Formal systems

A **formal system** consists of four pieces of information. Firstly, there are the **alphabet** and the **grammar**. The alphabet of a formal system is a finite collection of symbols, and the grammar is a collection of rules according to which symbols can be arranged. Combinations of symbols that obey the grammar are called **judgments** of the system. Additionally, any formal system has a collection of **axioms**, or **axiom schemata**, which are judgments or families of judgments that are asserted as "valid"; and, a collection of **inference rules**, such that any judgment inferred from the axioms is of equal "validity".

A formal system can be imagined as a game, say, chess. The alphabet corresponds to the board squares and the pieces on them, and the grammar is given by the structural rules: there cannot be two pieces on the same tile; there cannot be two squares of the same colour adjacent to each other; each player must have a single king; etc. The ways in which the pieces move correspond to the inference rules, and the only "axiom" is the initial board, shown here:

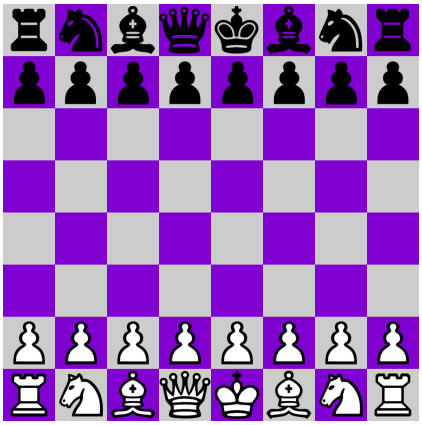

Figure 1: The axiom of the formal system of chess.

2.1 INFERENCE RULES

We write inference rules in the following way. For judgments $J_1, \ldots, J_k, J$, we say that the rule

$$\frac{J_1 \quad \cdots \quad J_k}{J} \quad \textsc{inference-ex}$$

has as **PREMISES** the judgments $J_1, \ldots, J_k$, as a **CONCLUSION** the judgment J, and is given the name INFERENCE-EX. Each inference rule has only one judgment as a conclusion, but may have multiple premises.

In a game, we cannot play the "next move" unless we have a position from which to begin. Just so, in a formal theory, we must have a **CONTEXT OF INFERENCE**, or a non-empty list of judgments, in order to apply the inference rules. Axioms A can be thought of as inference rules with an empty list of premises, so their inference rules would have the form

$$\frac{}{A} \quad \textsc{axiom-assert}.$$

Another inference rule with no premises that is present in all formal systems is

$$\frac{}{\text{ctx}} \quad \textsc{ctx-empty},$$

where for some (possibly empty) list of judgments $\Gamma = J_1, \ldots, J_k$, the judgment

$$\Gamma \ \text{ctx}$$

is the assertion that Γ is indeed a context of inference. Another inference rule, for a list of judgments Γ and judgment J, is available:

$$\frac{\Gamma \ \text{ctx} \quad J}{\Gamma, J \ \text{ctx}} \quad \textsc{ctx-extend},$$

which states that any context can be expanded to include additional judgments. One rule that is useful, but unnecessary, is the following:

$$\frac{J_1, \ldots, J_k \ \text{ctx}}{J_i} \quad \textsc{assumption-}i$$

for $1 \leq i \leq k$.

2.2 THEOREMS AND PROVABILITY

It is critical to remember that *judgments do not have truth values*: they represent combinations of symbols that are well-formed according to the grammar rules of the formal system. For a judgment J of the formal system, and a context Γ, the judgment

$$\Gamma \vdash J$$

means that Γ GIVES J; that is, that if we have all of the judgments in Γ, then we are free to use the additional judgment J as if it were an axiom. There is an inference rule that encapsulates this:

$$\frac{\Gamma \text{ ctx} \quad \Gamma \vdash J}{J} \quad \textsc{conclude-given}.$$

This concept can be illuminated further with the rule

$$\frac{J_1, \ldots, J_k \text{ ctx}}{J_1, \ldots, J_k \vdash J_i} \quad \textsc{given-}i$$

for each $1 \leq i \leq k$. This is very similar to ASSUMPTION-i; indeed, as alluded above, the latter is unnecessary because of the rule GIVEN-i, since by applying CONCLUDE-GIVEN we may conclude J_i.

The concept of *proof* in a formal system is located entirely within the domain of inference rules, when we ask a question like

Does the context Γ allow for the conclusion of the judgment J?

when we do not know that J is given by Γ. In this case, we say that Γ PROVES J. This is also a judgment.

example 2.2.1 Consider the following chessboard:

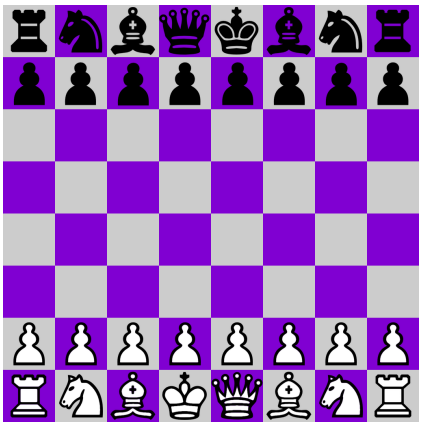

Figure 2: The white king has switched places with the white queen.

While this is a well-formed judgment according to the grammar, it is not a theorem: there is no way to reach this position from the initial board in Figure 1, since to swap the king and queen would require the move of a white pawn, which cannot move back to the home row, as well as taking more than one turn, forcing the player in black to move their pawns as well. Hence this is not a "theorem", or legal position, of chess.

2.3 DEFINITIONS

Often, mathematicians define new concepts in terms of old ones, perhaps for brevity or to highlight a particular perspective. We can do this in any formal system, by using a symbol to denote another, or another list of symbols (as we have done above with Γ). We will write

$$Y :\equiv X$$

to denote the judgment that the symbol Y is **DEFINED** to represent the same thing that is represented by the symbol X. That is, Y and X are semantically the same, and can be substituted for one another wherever they arise. This act of replacement forms an equivalence relation, which we call **SYNONYMY**. To say that the symbols X and Y are synonymous is a judgment, written $X \equiv Y$, and we write the inference rules to convey the equivalence in a context Γ:

$$\frac{\Gamma \vdash X}{\Gamma \vdash X \equiv X}\ \textsc{syn-refl} \qquad \frac{\Gamma \vdash X \quad \Gamma \vdash Y \quad \Gamma \vdash X \equiv Y}{\Gamma \vdash Y \equiv X}\ \textsc{syn-symm}$$

$$\frac{\Gamma \vdash X \quad \Gamma \vdash Y \quad \Gamma \vdash Z \quad \Gamma \vdash X \equiv Y \quad \Gamma \vdash Y \equiv Z}{\Gamma \vdash X \equiv Z}\ \textsc{syn-trans}.$$

Since definitions grant synonymy, we also have

$$\frac{Y :\equiv X}{Y \equiv X}\ \textsc{def-syn}.$$

3 INTENSIONAL TYPE THEORY

Intensional Martin-Löf type theory (IML) was developed during the 1970s and 1980s, with the most widely-accepted version published in 1982. This is the version that we will describe and use. In this formal system, we have new forms of judgments, the first being

$$X \text{ type}$$

for a symbol X in the alphabet, and the second form

$$\alpha : X$$

for symbols α, X. The first is referred to as the **TYPING** judgment: the symbol X is said to represent a **TYPE** that certain objects might have. The second is the **TERM-OF** judgment: it says that the symbol α represents a **TERM** having the type X. Of course, in order for the second judgment to make sense, we must be working in a context where X has already occurred in a judgment of the form "X type".

Synonymy in IML arises only between types or between terms of synonymous types. That is, within a context Γ, we have the inference rules

$$\frac{\Gamma \vdash X \text{ type} \quad \Gamma \vdash \alpha : X}{\Gamma \vdash \alpha \equiv \alpha}\ \textsc{syn-refl-terms},$$

$$\frac{\Gamma \vdash X \text{ type} \quad \Gamma \vdash Y \text{ type} \quad \Gamma \vdash X \equiv Y \quad \Gamma \vdash \alpha : X}{\Gamma \vdash \alpha : Y}\ \textsc{term-of-syn-types}.$$

We say, in English, the phrase "let X be a type" to refer to the judgment "X type" in the formal system, and the phrase "let α be a term of type X" to refer to the ordered pair of judgments "X type, $\alpha : X$", unless "X type" has already been declared; in this case we can refer simply to "$\alpha : X$".

3.1 Universes

The first type we will see is called the **base universe**, and is denoted $\mathbf{U}_0$. A **universe**, of which the base universe is the smallest, is a type, whose terms can also be considered types. There are two frameworks for this consideration: first, the *Russell-style* framework, which posits the following, for each $i = 0, 1, 2, \ldots$:

$$\frac{}{\mathbf{U}_i \ \text{type}}\ \textsc{univ-is-type}, \qquad \frac{}{\mathbf{U}_i : \mathbf{U}_{i+1}}\ \textsc{univ-contain},$$

$$\frac{\Gamma \ \text{ctx} \quad \Gamma \vdash X : \mathbf{U}_i}{\Gamma \vdash X \ \text{type}}\ \textsc{u-terms-are-types}, \qquad \frac{X : \mathbf{U}_i}{X : \mathbf{U}_{i+1}}\ \textsc{univ-subsume}.$$

The *Tarski-style* framework, however, in addition to UNIV-IS-TYPE, posits operators T_i and lift_{i+1} for each such i, that act on the terms of $\mathbf{U}_i$ in the following way:

$$\frac{}{u_i : \mathbf{U}_{i+1}}\ \textsc{canonical-term-}i, \qquad \frac{\Gamma \ \text{ctx} \quad \Gamma \vdash X : \mathbf{U}_i}{\Gamma \vdash T_i(X) \ \text{type}}\ \textsc{type-of-u-term},$$

$$\frac{}{T_{i+1}(u_i) \equiv \mathbf{U}_{i+1}}\ \textsc{type-of-canon}, \qquad \frac{\Gamma \ \text{ctx} \quad \Gamma \vdash X : \mathbf{U}_i}{\Gamma \vdash \mathsf{lift}_{i+1}(X) : \mathbf{U}_{i+1}}\ \textsc{lift-}i\textsc{-term}.$$

There are also some coherence rules, found in the last chapter of [2], that describe the way the operators T_i and lift_{i+1} interact with the other type formers we will see below. They are omitted here, save to mention that the coherences are as one would expect.

> example 3.1.1: Key differences between frameworks
>
> We may have a canonical construction of some type X, that exists at each universe level. Let two such constructions $X_i : \mathbf{U}_i$ and $X_{i+1} : \mathbf{U}_{i+1}$ be given. In the Russell-style framework, where $\mathbf{U}_i : \mathbf{U}_{i+1}$, we have $X_i : \mathbf{U}_{i+1}$ through UNIV-SUBSUME, but then the types X_i and X_{i+1} are distinct, in that we can never obtain the judgment $X_i \equiv X_{i+1}$. In the Tarski-style framework, however, we can allow $\mathsf{lift}_{i+1}(X_i) \equiv X_{i+1}$, because the definition of X_i in $\mathbf{U}_i$ is "wrapped inside" the lift_{i+1} operator. The constructions of X_i and X_{i+1} are essentially the same, but on different levels; the wrapping lets us make precise that fact. See [1] for a deeper discussion.

The Russell-style framework's rule U-TERMS-ARE-TYPES allows us to write $X : \mathbf{U}_i$ to instantly infer that X is a well-formed type, while also specifying its place in the universe "hierarchy". We will work in this framework for the most part, for its simplicity, and because constructions in it can be translated without great effort into the Tarski-style framework. We will also frequently be working in a situation where the particular universe level does not matter, so in these cases we omit the subscript and simply write $X : \mathbf{U}$ to mean X type.

3.2 Making useful types

In order to actually make use of "types" in general, we must supply additional information. There are five collections of rules that we must specify in order to *use* a type:

1. **formation rules**, that specify what information is needed to make the type;
2. **construction rules**, that govern the creation of terms of the type;
3. **elimination rules**, that allow for the definition of *functions* using those terms;
4. **computation rules**, that demonstrate how eliminators act on constructors;
5. and **uniqueness principles**, that describe what terms of the type look like compared to the constituent information from the formation rules.

In many cases, the fifth set of rules is actually not required outright, but can be proven from the other four; hence the distinction between rules and principles. Here, a "function" refers to a λ-expression, as used in Church's λ-calculus, whose input has the type we are forming, and whose output is a sequence of symbols that refer to a term of another type. We write such expressions in the form

$$\lambda(\alpha : X).\ (\varphi(\alpha) : Y),$$

in a context where the judgment "Y type" has occurred.

We will see several examples of types formed from other types, as well as some concrete types, to ensure that we aren't doing work for no reason. Then, we'll see how we can use type formers to recreate classical structures in this new system.

For the moment, it will help to imagine types as classical sets, whose terms can be thought of as the elements of the set. But be careful: there is no membership predicate in type theory! Whereas in set theory, one may have elements without sets, it is impossible in type theory to have terms without types.

3.2.1 FUNCTION TYPES

Let Γ be a context that gives X type. Suppose that, with a term $x : X$, we can create a new type $Z(x)$. Then we are given the **(DEPENDENT) FUNCTION TYPE** (or **Π-TYPE**), written

$$\left(\prod_{(x:X)} Z(x)\right) : \mathbf{U}.$$

Formally, this is expressed by the rule

$$\frac{\Gamma \text{ ctx} \quad \Gamma \vdash X : \mathbf{U} \quad \Gamma, x : X \vdash Z(x) : \mathbf{U}}{\Gamma \vdash \prod_{(x:X)} Z(x) : \mathbf{U}}\ \Pi\text{-FORM}.$$

As described above, terms of this type are given by λ-expressions, like so:

$$\frac{\Gamma, x : X \vdash z : Z(x)}{\Gamma \vdash \left(\lambda(x : X).\ (z : Z(x))\right) : \prod_{(x:X)} Z(x)}\ \Pi\text{-CONSTR}.$$

The name "dependent" comes from the fact that the type $Z(x)$ depends on the term $x : X$ that was given. If in fact we have $\Gamma \vdash Z$ without requiring that assumption, we can form the type of **INDEPENDENT FUNCTIONS**, written

$$X \to Z :\equiv \prod_{(x:X)} Z.$$

The same inference rules apply, but the extra assumption $x : X$ next to the context Γ may be disregarded. For example, its formation rule:

$$\frac{\Gamma \text{ ctx} \quad \Gamma \vdash X : \mathbf{U} \quad \Gamma \vdash Z : \mathbf{U}}{\Gamma \vdash X \to Z : \mathbf{U}}\ \to\text{-FORM}.$$

The elimination for the dependent function type is given as follows:

$$\frac{\Gamma \vdash f : \prod_{(x:X)} Z(x) \quad \Gamma \vdash y : X}{\Gamma \vdash f(y) : Z(y)}\ \Pi\text{-ELIM},$$

where $Z(y)$ is the type given by replacing each instance of the symbol x, in both the expression f and the type $Z(x)$ as formed above, to the symbol y.

The computation rule for dependent functions is given by

$$\frac{\Gamma, x : X \vdash z : Z \quad \Gamma \vdash y : X}{\Gamma \vdash \left(\lambda(x : X).\ (z : Z(x))\right)(y) \equiv z[y/x] : Z(y)}\ \Pi\text{-COMP},$$

where $z[y/x]$ indicates the α-renaming of x to y in the sequence of symbols that give z in the assumptions. Once again, in the case where the type Z is independent of the term $x : X$, we may drop the assumption and we need perform no α-renaming in the elimination or computation rules.

The function type is the only case for which we will define a uniqueness rule.

$$\frac{\Gamma \vdash f : \prod_{(x:X)} Z(x)}{\Gamma \vdash f \equiv \big(\lambda(x : X).\ f(x)\big) : \prod_{(x:X)} Z(x)} \quad \Pi\text{-UNIQ}.$$

This rule states that a function is synonymous with the action of applying itself to an input.

Applying functions is the only way, save by definition, of concluding synonymy. We will see a weaker notion of "sameness", called **EQUALITY**, as another type former later on. The fact that equality is weaker than synonymy is what allows it to be more easily concluded: it is far less frequent that two things have the same definition than that they are equal!

The notion of **FUNCTION COMPOSITION** can be defined for dependent functions, but we will use this example to demonstrate independent functions. Let X, Y, Z be types, and suppose we have $f : X \to Y$ and $g : Y \to Z$. Then their composition

$$g \circ f :\equiv \big(\lambda(x : X).\ (g(f(x)))\big) : X \to Z.$$

This operation is synonymously associative: that is, supposing $h : Z \to W$, then

$$h \circ (g \circ f) \equiv (h \circ g) \circ f.$$

3.2.2 PAIR TYPES

Let Γ be a context giving the type X, and suppose that from some $x : X$ we can create the type $Y(x)$. Then Γ also gives the **(DEPENDENT) PAIR TYPE** (or **Σ-TYPE**), written

$$\big(\textstyle\sum_{(x:X)} Y(x)\big) : \mathbf{U}.$$

Formally, this is expressed by the rule

$$\frac{\Gamma \text{ ctx} \quad \Gamma \vdash X : \mathbf{U} \quad \Gamma, x : X \vdash Y(x) : \mathbf{U}}{\Gamma \vdash \sum_{(x:X)} Y(x) : \mathbf{U}} \quad \Sigma\text{-FORM}.$$

We want the terms of this type to be pairs of terms $(x : X, y : Y(x))$, where the type of the second term depends on the first term. Again, in the case where $Y(x)$ does not depend on the specific term of X, we have an **INDEPENDENT** pair type, also called the **PRODUCT TYPE**, that we denote by

$$X \times Y :\equiv \textstyle\sum_{(x:X)} Y.$$

The same tricks as above apply for creating independent pairs from the rules for dependent pairs: ignore the extra $x : X$ assumption immediately following the context Γ. For example, we can write

$$\frac{\Gamma \text{ ctx} \quad \Gamma \vdash X : \mathbf{U} \quad \Gamma \vdash Y : \mathbf{U}}{\Gamma \vdash X \times Y : \mathbf{U}} \quad \times\text{-FORM}.$$

The construction rule for pair types is given by

$$\frac{\Gamma, x : X \vdash Y(x) : \mathbf{U} \quad \Gamma \vdash a : X \quad \Gamma \vdash y : Y(a)}{\Gamma \vdash (a, y) : \sum_{(x:X)} Y(x)} \quad \Sigma\text{-INTRO}.$$

Again, the type $Y(a)$ is simply the α-renaming of x to a in the type $Y(x)$.

The elimination rule is often called the **INDUCTION PRINCIPLE** of the type. This is an allusion to the natural numbers case, whereby specifying a "basepoint" and "step", we obtain a way of describing functions out of any number. The analogy here is that, by specifying a first coordinate, and a second depending on that, we can describe how to obtain functions out of any pair:

$$\frac{\Gamma, z : \sum_{(x:X)} Y(x) \vdash C(z) : \mathbf{U} \quad \Gamma, x : X, y : Y(x) \vdash c : C((x,y)) \quad \Gamma \vdash p : \sum_{(x:X)} Y(x)}{\Gamma \vdash \mathsf{ind}_\Sigma(C, c[p/(x,y)], p) : C(p)} \ \Sigma\text{-ELIM}.$$

Inside the bracket in the conclusion, the term C refers to the **TYPE FAMILY** $C : \sum_{(x:X)} Y(x) \to \mathbf{U}$. A type family is a special function, whose output type is a universe; that is, its values are types. The type family C was specified in the first assumption of this rule, since we assume that for an unspecified $z : \sum_{(x:X)} Y(x)$ we could create some type $C(z)$; this is a special case of Π-FORM. Indeed, we can also consider $Y : X \to \mathbf{U}$ as a type family.

The computation rule for dependent pairs is given as follows:

$$\frac{\Gamma, z : \sum_{(x:X)} Y(x) \vdash C(z) : \mathbf{U} \quad \Gamma, x : X, y : Y(x) \vdash c : C((x,y)) \quad \Gamma \vdash a : X \quad \Gamma \vdash b : Y(a)}{\Gamma \vdash \mathsf{ind}_\Sigma(C, c[(a,b)/(x,y)], (a,b)) \equiv c[(a,b)/(x,y)] : C((a,b))} \ \Sigma\text{-COMP}.$$

As we noted above, computation rules specify how eliminators act on constructors, which in this case are the terms $a : X$ and $b : Y(a)$. We conclude here that eliminating the pair (a, b) through ind_Σ is synonymous with the term c where (x, y) has been α-renamed to (a, b).

We will no longer specify uniqueness principles, since the other type constructions we will see have provable, rather than given, such principles. Instead, let's study ind_Σ. We can consider this as a function! But what is its type? Let's suppose we are using $X : \mathbf{U}$, $Y : X \to \mathbf{U}$, and $\sum_{(x:X)} Y(x) : \mathbf{U}$, as above. Then

$$\mathsf{ind}_\Sigma : \prod_{(C:\sum_{(x:X)} Y(x) \to \mathbf{U})} \left(\prod_{(x:X)} \prod_{(y:Y(x))} C((x,y)) \right) \to \left(\prod_{(p:\sum_{(x:X)} Y(x))} C(p) \right).$$

Let's break this down.

- First, we specify a type family $C : \sum_{(x:X)} Y(x) \to \mathbf{U}$.
- Then, we have a two-argument dependent function, taking a term $x : X$ and, dependently, a term $y : Y(x)$, returning *dependently on both* a term $c : C((x, y))$. Recall that (x, y) specified in this way has the type $\sum_{(x:X)} Y(x)$, so it makes sense to apply C to it.
- Finally, the output type of ind_Σ: we have a function that takes any pair $p : \sum_{(x:X)} Y(x)$ and yields a term of $C(p)$.

The computation rule states that, given $(a, b) : \sum_{(x:X)} Y(x)$, the term given by $\mathsf{ind}_\Sigma(C, c, (a, b))$ is synonymous with the α-renaming of (x, y) in the definition of c to (a, b).

We can use this notation to capture the elimination and computation rules in a single function definition. Let's see this with $\mathsf{ind}_\times$. Suppose we have $X, Y : \mathbf{U}$, and $X \times Y : \mathbf{U}$. Then

$$\mathsf{ind}_\times : \prod_{(C:X\times Y\to \mathbf{U})} \left(\prod_{(x:X)} \prod_{(y:Y)} C((x,y)) \right) \to \left(\prod_{(p:X\times Y)} C(p) \right)$$
$$\mathsf{ind}_\times(C, f, (a, b)) :\equiv f(a, b).$$

Here, the term $f : \prod_{(x:X)} \prod_{(y:Y)} C((x, y))$ represents the two-argument function, so that the "dummy variables" $x : X$, $y : Y$ never have to be specified in the definition. Since $\mathsf{ind}_\times$ is *defined* as being the output of f applied to a and b, we *must* have the proper synonymy given by the computation rule.

For pair types, we have the **PROJECTION FUNCTIONS**

$$\mathsf{pr}_1 : \sum_{(x:X)} Y(x) \to X$$
$$\mathsf{pr}_2 : \prod_{(z:\sum_{(x:X)} Y(x))} Y(\mathsf{pr}_1(z)).$$

In the independent case, we simply write $\mathsf{pr}_1 : X \times Y \to X$ and $\mathsf{pr}_2 : X \times Y \to Y$.

3.2.3 CONCRETE TYPES

Our first concrete type, we want the **UNIT TYPE** $\mathbf{1} : \mathbf{U}_0$ to have a single term, canonically named $\star : \mathbf{1}$. We would like to be able to use it at any time. Thus its formation and construction rules are given by

$$\frac{\Gamma \ \mathrm{ctx}}{\Gamma \vdash \mathbf{1} : \mathbf{U}} \ \mathbf{1}\text{-FORM}, \qquad \frac{\Gamma \ \mathrm{ctx}}{\Gamma \vdash \star : \mathbf{1}} \ \mathbf{1}\text{-INTRO}.$$

We do not know *a priori* that $\star$ is the unique term of $\mathbf{1}$, but we can write our induction principle so that it behaves this way:

$$\begin{aligned} \mathrm{ind}_{\mathbf{1}} : \textstyle\prod_{(C:\mathbf{1}\to\mathbf{U})} C(\star) \to \prod_{(u:\mathbf{1})} C(u) \\ \mathrm{ind}_{\mathbf{1}}(C, c, \star) :\equiv c. \end{aligned}$$

Since the computation rule is the action of the eliminators on the constructors, the only thing to which we know how to apply $\mathrm{ind}_{\mathbf{1}}$ is $\star : \mathbf{1}$.

Another concrete type is the **EMPTY TYPE** $\mathbf{0} : \mathbf{U}$. It is special in that it has no construction rule, and therefore no computation rule. We have only

$$\frac{\Gamma \ \mathrm{ctx}}{\Gamma \vdash \mathbf{0} : \mathbf{U}} \ \mathbf{0}\text{-FORM}, \qquad \frac{\Gamma, x : \mathbf{0} \vdash C(x) : \mathbf{U} \quad \Gamma \vdash \, ! : \mathbf{0}}{\Gamma \vdash \mathrm{ind}_{\mathbf{0}}(C, !) : C(!)} \ \mathbf{0}\text{-ELIM}.$$

Since $\mathbf{0}$ has no constructors, if we find a term $! : \mathbf{0}$, we call it a **CONTRADICTION**.

Since type families from $\mathbf{0}$ cannot technically depend on the particular term of $\mathbf{0}$, since there are no constructors on which to act, the induction principle can instead be stated for a general type $C : \mathbf{U}$,

$$\mathrm{ind}_{\mathbf{0}}(C) : \mathbf{0} \to C,$$

with no definition. That is, if we manage to find a term of $\mathbf{0}$, we can create a term of any type. This captures the principle of *ex falso quodlibet*: from a contradiction, anything follows.

3.2.4 COPRODUCT TYPES

A type former may have more than one construction rule, at the cost of multiple computation rules. A simple example of this is the **COPRODUCT TYPE**, formed by the rule

$$\frac{\Gamma \ \mathrm{ctx} \quad \Gamma \vdash X : \mathbf{U} \quad \Gamma \vdash Y : \mathbf{U}}{\Gamma \vdash X + Y : \mathbf{U}} \ +\text{-FORM}.$$

Its constructors are the independent functions $\mathrm{in}\ell : X \to X + Y$ and $\mathrm{in}r : Y \to X + Y$. The induction principle is then defined by **PATTERN MATCHING** on each of the possible constructors:

$$\begin{aligned} \mathrm{ind}_{+} : \textstyle\prod_{(C:X+Y\to\mathbf{U})}(\prod_{(x:X)} C(x)) \to (\prod_{(y:Y)} C(y)) \to \prod_{(q:X+Y)} C(q) \\ \mathrm{ind}_{+}(C, c_\ell, c_r, \mathrm{in}\ell(x)) :\equiv c_\ell(x), \\ \mathrm{ind}_{+}(C, c_\ell, c_r, \mathrm{in}r(y)) :\equiv c_r(y). \end{aligned}$$

example 3.2.1: The type of *Booleans*, $\mathbf{2} :\equiv \mathbf{1} + \mathbf{1}$

We give the names $0_{\mathbf{2}} :\equiv \mathrm{in}\ell(\star)$ and $1_{\mathbf{2}} :\equiv \mathrm{in}r(\star)$ for clarity. By combining ind_{+} and $\mathrm{ind}_{\mathbf{1}}$, we obtain

$$\begin{aligned} \mathrm{ind}_{\mathbf{2}} : \textstyle\prod_{(C:\mathbf{2}\to\mathbf{U})} C(0_{\mathbf{2}}) \to C(1_{\mathbf{2}}) \to \prod_{(b:\mathbf{2})} C(b) \\ \mathrm{ind}_{\mathbf{2}}(C, c_0, c_1, 0_{\mathbf{2}}) :\equiv c_0, \\ \mathrm{ind}_{\mathbf{2}}(C, c_0, c_1, 1_{\mathbf{2}}) :\equiv c_1. \end{aligned}$$

This corresponds to the *if-then-else* construct in computer science, or rather, *then-else-if*.

3.2.5 NATURAL NUMBERS

Another type with multiple constructors is the type of **NATURAL NUMBERS**. In fact, this is a concrete type, and its constructors have types that are dissimilar in structure. The formation rule is

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathbf{N} : \mathbf{U}} \quad \mathbf{N}\text{-FORM}.$$

Its constructors are the terms $0 : \mathbf{N}$ and $\text{succ} : \mathbf{N} \to \mathbf{N}$. This is an example of an **INDUCTIVE TYPE**, one whose terms are generated by all possible combinations of its constructors. In this case, its well-formed terms are of the form $0 : \mathbf{N}$ or $\text{succ}(n) : \mathbf{N}$ for some $n : \mathbf{N}$. The induction principle now earns its namesake:

$$\begin{aligned} \text{ind}_{\mathbf{N}} : \textstyle\prod_{(C:\mathbf{N}\to\mathbf{U})} C(0) \to \left(\prod_{(n:\mathbf{N})} C(n) \to C(\text{succ}(n))\right) \to \prod_{(n:\mathbf{N})} C(n) \\ \text{ind}_{\mathbf{N}}(C, c_0, c_s, 0) :\equiv c_0, \\ \text{ind}_{\mathbf{N}}(C, c_0, c_s, \text{succ}(n)) :\equiv c_s(n, \text{ind}_{\mathbf{N}}(C, c_0, c_s, n)). \end{aligned}$$

Hence, functions out of the natural numbers are defined by primitive recursion, as is familiar in traditional mathematics.

3.2.6 EQUALITY TYPES

As previously foreshadowed, for each type X there is a type former $=_X : X \to X \to \mathbf{U}$ that represents equality between two terms of a type. That is, given a type X and two of its terms x, y, there is a type $x =_X y$, whose terms are proofs of the equality of x and y. When the type X is clear, we write $x = y$. The terms of this type are also called **PATHS** between x and y; here we begin to see the roots of homotopy theory. Let's consider its rules.

$$\frac{\Gamma \text{ ctx} \quad \Gamma \vdash X : \mathbf{U} \quad \Gamma \vdash x, y : X}{\Gamma \vdash x =_X y : \mathbf{U}} \quad =\text{-FORM}.$$

Nothing special here. It has only one constructor, named **REFLEXIVITY**:

$$\frac{\Gamma \vdash x : X}{\Gamma \vdash \text{refl}_x : x =_X x} \quad =\text{-CONSTR}.$$

Note that the construction rule also grants that synonymy entails equality. This is not to say the only terms of this type take the form of reflexivity; indeed, when considering $x = y$ where $x \not\equiv y$, we cannot simplify or inspect its terms. However, we still have the induction principle:

$$\begin{aligned} \text{ind}_{=} : \textstyle\prod_{(C:\prod_{(x,y:X)}(x=xy)\to\mathbf{U})} \left(\prod_{(x:X)} C(x, x, \text{refl}_x)\right) \to \prod_{(x,y:X)} \prod_{(p:x=_X y)} C(x, y, p) \\ \text{ind}_{=}(C, c, x, x, \text{refl}_x) :\equiv c(x). \end{aligned}$$

In this way, the type family $(x =) : X \to \mathbf{U}$ is inductively generated by paths of the form refl_x. If we imagine refl_x to be the "stay-still path", the type family is induced by moving a single endpoint of that path around the entire space.

Theorem 3.2.2: Path inversion

Let X be a type and $x, y : X$. Suppose $p : x = y$. Then there is a term $p^{-1} : y = x$, called the **INVERSE** of p.

Quick proof: According to the induction principle for equality types (also called **PATH INDUCTION**), it suffices to consider the case when $x \equiv y$ and $p \equiv \mathrm{refl}_x$ in order to generate the result for the entire type of paths $x = y$. But in that case, we can define $\mathrm{refl}_x^{-1} :\equiv \mathrm{refl}_x : x = x$. Hence through path induction, we obtain $p^{-1} : y = x$. ■

Detailed proof: Let $C : \prod_{(x,y:X)}(x = y) \to \mathbf{U}$ be defined by $C(x, y, p) :\equiv (y = x)$. Then we have a function

$$c :\equiv (\lambda(x : X).\ \mathrm{refl}_x) : \prod_{(x:X)} C(x, x, \mathrm{refl}_x).$$

By the induction principle for equality types, we have $\mathrm{ind}_=(C, c, x, y, p) : y = x$ for each $p : x = y$. Hence we can define

$$(-)^{-1} :\equiv \lambda(p : x = y).\ (\mathrm{ind}_=(C, c, x, y, p) : y = x),$$

so that $\mathrm{refl}_x^{-1} :\equiv c(x) \equiv \mathrm{refl}_x$. ■

Theorem 3.2.3: Path concatenation

Let X be a type and $x, y, z : X$. Suppose $p : x = y$ and $q : y = z$. Then there is a term $p \bullet q : x = z$, called the **CONCATENATION** of p and q.

Quick proof: By path induction on p, we may assume $x \equiv y$ and $p \equiv \mathrm{refl}_x$. By induction on q, we may assume $x \equiv z$ and $q \equiv \mathrm{refl}_x$. Hence it suffices to set $\mathrm{refl}_x \bullet \mathrm{refl}_x :\equiv \mathrm{refl}_x : x = x$. ■

Detailed proof: Let $C : \prod_{(x,y:X)}(x = y) \to \mathbf{U}$ be defined by $C(x, y, p) :\equiv \prod_{(z:X)} \prod_{(q:y=z)}(x = z)$. Note that $C(x, x, \mathrm{refl}_x) \equiv \prod_{(z:X)} \prod_{(q:x=z)}(x = z)$. In order to apply the path induction principle, we need a function with the type $\prod_{(x:X)} C(x, x, \mathrm{refl}_x)$.

Let also $D : \prod_{(x,z:X)} \prod_{(q:x=z)} \mathbf{U}$ be defined by $D(x, z, q) :\equiv (x = z)$. Note that $D(x, x, \mathrm{refl}_x) \equiv (x = x)$. Then we have a function

$$d :\equiv (\lambda(x : X).\ \mathrm{refl}_x) : \prod_{(x:X)} D(x, x, \mathrm{refl}_x).$$

By the induction principle for equality types applied to D, we obtain a function

$$c :\equiv \mathrm{ind}_=(D, d) : \prod_{(x,z:X)} \prod_{(q:x=z)} D(x, z, q),$$

which is in fact $\prod_{(x,z:X)} \prod_{(q:x=z)}(x = z) \equiv \prod_{(x:X)} C(x, x, \mathrm{refl}_x)$. By the induction principle for equality types in the case of C, we now have a function of type

$$\mathrm{ind}_=(C, c) : \prod_{(x,y,z:X)}(y = z) \to (x = y) \to (x = z).$$

We set $(- \bullet -)$ to be this function, so that $\mathrm{refl}_x \bullet \mathrm{refl}_x :\equiv \mathrm{refl}_x$. ■

Inversion corresponds to symmetry of equality, and concatenation to its transitivity. Thus equality is an equivalence relation on every type X. Also, given $x, y : X$, and $p, q : x = y$, we have the type $p = q$, and for each $\alpha, \beta : p = q$, we have the type $\alpha = \beta$, and so on...

We can view terms of a type X as objects, and paths between them as morphisms, and paths between those paths as 2-morphisms, and so on. The structure generated by repeated applications of the equality type former corresponds to the category-theoretic notion of an ∞-GROUPOID, since each morphism is invertible (by 3.2.2). Every type is hence an ∞-groupoid. With the right definitions, it is actually a theorem that every topological space is a groupoid, and vice versa. In general, however, this statement is known as the *homotopy hypothesis*.

Theorem 3.2.4: Some important 2-paths

Let $X : \mathbf{U}$ with $x, y, z, w : X$. Suppose $p : x = y$, $q : y = z$, $r : z = w$. Then we can construct terms of the following types:

1. $p = \mathrm{refl}_x \bullet p$;
2. $p = p \bullet \mathrm{refl}_y$;
3. $p^{-1} \bullet p = \mathrm{refl}_y$;
4. $p \bullet p^{-1} = \mathrm{refl}_x$;
5. $(p^{-1})^{-1} = p$;
6. $p \bullet (q \bullet r) = (p \bullet q) \bullet r$.

One would expect that, just as is the case with synonymous symbols, equal terms should preserve their equality under a function. This is formalized by the following lemma.

Theorem 3.2.5: Path application

Let $X, Y : \mathbf{U}$ and $f : X \to Y$. There is a term

$$\mathrm{ap}_f : \prod_{(x,y:X)} (x = y) \to (f(x) = f(y)).$$

Proof: Let $x, y : X$ and suppose $p : x = y$. By path induction on p, we may assume $x \equiv y$ and $p \equiv \mathrm{refl}_x$. In this case, we also have $f(x) \equiv f(y)$ by α-renaming, and so we can define

$$\mathrm{ap}_f(\mathrm{refl}_x) :\equiv \mathrm{refl}_{f(x)} : f(x) = f(x).$$

■

Theorem 3.2.6: Transport

Let $X : \mathbf{U}$ and $Y : X \to \mathbf{U}$, and $x, y : X$. There is a term

$$\mathrm{transport}^Y : (x = y) \to (Y(x) \to Y(y)).$$

We write $p_* :\equiv \mathrm{transport}^Y(p)$ when Y is clear.

Proof: By path induction, we may assume $x \equiv y$ and $p \equiv \mathrm{refl}_x$. In this case, we have $Y(x) \equiv Y(y)$, so we can define

$$\mathrm{transport}^Y(\mathrm{refl}_x) :\equiv \mathrm{id}_{Y(x)}.$$

■

Theorem 3.2.7: Dependent path application

Let $X : \mathbf{U}$, $Y : X \to \mathbf{U}$, and $f : \prod_{(x:X)} Y(x)$. There is a term

$$\mathrm{apd}_f : \prod_{(x,y:X)} \prod_{(p:x=y)} (\mathrm{transport}^Y(p, f(x)) = f(y)).$$

We omit the specification of x and y since they are required for the definition of p.

Proof: It suffices by path induction to assume $x \equiv y$ and $p \equiv \mathrm{refl}_x$. In this case $Y(x) \equiv Y(y)$, so we can define

$$\mathrm{apd}_f(\mathrm{refl}_x) :\equiv \mathrm{refl}_{f(x)}.$$

■

Corollary 3.2.8: Dependent path application on path types

Let $X : \mathbf{U}$ and $a : X$. Then for any $x, y : X$ and $p : x = y$, we have

$$\begin{aligned}
\mathrm{transport}^{z \mapsto (a=z)}(p, q) &= (q \bullet p) : a = y && \text{for } q : a = x, \\
\mathrm{transport}^{z \mapsto (z=a)}(p, r) &= (p^{-1} \bullet r) : y = a && \text{for } r : x = a, \\
\mathrm{transport}^{z \mapsto (z=z)}(p, s) &= (p^{-1} \bullet s \bullet p) : y = y && \text{for } s : x = x.
\end{aligned}$$

This marks the end of our discussion of pure IML type theory. In the next section, we will discuss the history of type theory writ large, the additional axioms that homotopy type theory posits, and analyze some of the properties of path spaces of various types. These ideas are closely related, and we will see why shortly.

4 Homotopy Type Theory

The very first *theory of types* was given in Russell and Whitehead's *Principia Mathematica*, first published in 1910. Its goal was to minimize the number of axioms required to perform mathematics, and to solve the paradoxes that arose in the set theory of the time, especially the one of Russell's own namesake. That theory presented a grammar for formulae that prevented the sort of unrestricted application that made those paradoxes occur. However, it is rather infamous, in that the symbolic logic they developed required several hundred pages to prove the proposition "1 + 1 = 2", that is, $\mathrm{succ}(0) + \mathrm{succ}(0) = \mathrm{succ}(\mathrm{succ}(0))$.

Another type theory, of more computational value, is Church's *typed λ-calculus*: a variant of the untyped λ-calculus where functions are restricted to act on specific types, just like in *Principia*. Several other systems have been based on typed λ-calculus, such as System T, System F, and the rest of the "Lambda cube".

IML was designed on the principles of mathematical constructivism. It is *intensional*, in that synonymy and equality are distinct concepts. One limitation of IML as compared to HoTT is that, when forming a type in IML, we may only reference its terms, and not paths or higher paths between those terms. HoTT allows us to do this with *inductive types*; we will see that propositions and sets are examples of such types. This is where the name "*homotopy* type theory" comes from.

4.1 HOMOTOPIES OF FUNCTIONS

The classical notion of *homotopy* between two continuous functions $f, g : X \to Y$, where X and Y are topological spaces, is a continuous map $H : [0,1] \to X \to Y$ with

$$H(0,x) = f(x); \qquad H(1,x) = g(x).$$

If instead of f, g we consider paths $p, q : [0,1] \to X$ between x and y in a space X, we have the homotopy $H : [0,1] \times [0,1] \to X$ such that

$$\begin{aligned} H(0,t) &= p(t), & H(1,t) &= q(t), \\ H(s,0) &= x, & H(s,1) &= y, \end{aligned}$$

so that for each $s \in [0,1]$, the function $H(s) : [0,1] \to X$ is also a path between x and y. For two paths to share a homotopy is an equivalence relation, with symmetry as the inversion of the homotopy and transitivity as concatenation.

Let $X, Y : \mathbf{U}$ and $f, g : X \to Y$. A **HOMOTOPY** between f and g is a function that assigns to each $x : X$ a path between $f(x)$ and $g(x)$. That is,

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

Homotopies must be defined between independent functions, since the target terms $f(x)$ and $g(x)$ must have the same type in order to construct a path between them.

Note that, in this light, a homotopy is not quite the same as a path $f = g$; the only paths exist at the level of the functions' values. We will soon see how paths in the function type $X \to Y$ arise.

Lemma 4.1.1

Homotopy is an equivalence relation.

Proof: Let $X, Y : \mathbf{U}$ and $f, g, h : X \to Y$.

To show that $\sim$ is reflexive, consider that $f(x) \equiv f(x)$ for each $x : X$. Thus we have

$$\lambda(x : X).\ (\mathrm{refl}_{f(x)} : f(x) = f(x)) : f \sim f.$$

To show that $\sim$ is symmetric, consider a homotopy $H : f \sim g$. Define $J(x) :\equiv (H(x))^{-1}$ for each $x : X$. Thus $J : g \sim f$.

To show that $\sim$ is transitive, consider homotopies $J : f \sim g$ and $K : g \sim h$. Define $L(x) :\equiv J(x) \bullet K(x)$ for each $x : X$. Thus $L : f \sim h$. ■

Theorem 4.1.2: Naturality

Let $X, Y : \mathbf{U}$, $f, g : X \to Y$, and suppose $x, y : X$ with $p : x = y$. If $H : f \sim g$, then

$$H(x) \bullet \mathrm{ap}_g(p) = \mathrm{ap}_f(p) \bullet H(y).$$

Proof: By path induction, it suffices to assume $x \equiv y$ and $p \equiv \mathrm{refl}_x$. But in this case, we have

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

■

Corollary 4.1.3

If $f : X \to X$ with $H : f \sim \mathrm{id}_X$, then $H \circ f \sim \mathrm{ap}_f \circ H$. That is, $\prod_{(x:X)} H(f(x)) = \mathrm{ap}_f(H(x))$.

4.2 TYPE EQUIVALENCE

Given types $X, Y : \mathbf{U}$ and a function $f : X \to Y$, we say that f is an **EQUIVALENCE** between X and Y if there is a function $g : Y \to X$ such that $g \circ f \sim \mathrm{id}_X$ and a function $h : Y \to X$ such that $f \circ h \sim \mathrm{id}_Y$. That is,

$$\mathrm{isEquiv}(f) :\equiv (\textstyle\sum_{(g:Y\to X)}(g \circ f \sim \mathrm{id}_X)) \times (\sum_{(h:Y\to X)}(f \circ h \sim \mathrm{id}_Y)).$$

A curious property of this type, since it is defined using the unique identity function, is that for any two terms $e_1, e_2 : \mathrm{isEquiv}(f)$, we always have $e_1 = e_2$. For this reason, we write the terms of the type

$$X \simeq Y :\equiv \textstyle\sum_{(f:X\to Y)} \mathrm{isEquiv}(f)$$

as simply the function, that is, $f : X \simeq Y$.

example 4.2.1 For any X, the function $\mathrm{id}_X : X \simeq X$, since we have a term

$$((\mathrm{id}_X, \lambda(x : X).\ \mathrm{refl}_x), (\mathrm{id}_X, \lambda(x : X).\ \mathrm{refl}_x)) : \mathrm{isEquiv}(\mathrm{id}_X).$$

We have a formulation that is simpler to use in practise, that uses only one "backwards" function, but is not guaranteed the same equality of terms as given by $\mathrm{isEquiv}(f)$. This is the type of **QUASI-INVERSES** of f, namely

$$\mathrm{qinv}(f) :\equiv \textstyle\sum_{(g:Y\to X)}(g \circ f \sim \mathrm{id}_X) \times (f \circ g \sim \mathrm{id}_Y).$$

Lemma 4.2.2

Let $X, Y : \mathbf{U}$. Then for any $f : X \to Y$, we have $\mathrm{isEquiv}(f) \to \mathrm{qinv}(f)$ and $\mathrm{qinv}(f) \to \mathrm{isEquiv}(f)$.

Proof: Let $f : X \to Y$ with $(g, \alpha, \beta) : \mathrm{qinv}(f)$. Then $((g, \alpha), (g, \beta)) : \mathrm{isEquiv}(f)$. Also, if $((g, \alpha), (h, \beta)) : \mathrm{isEquiv}(f)$, then define $\beta' : s \circ f \sim \mathrm{id}_X$ by

$$\beta'(x) :\equiv \beta(g(f(x)))^{-1} \bullet \mathrm{ap}_h(\alpha(f(x))) \bullet \beta(x);$$

then $(g, \alpha, \beta') : \mathrm{qinv}(f)$. ■

Since either can be obtained from the other, we can use quasi-inverses for their simplicity, but apply (4.2.2) when we need an equivalence with equal terms in the second coordinate. See (5.2.4) for a discussion on why this matters.

Theorem 4.2.3

Type equivalence is an equivalence relation.

Proof: We showed $\simeq$ is reflexive in (4.2.1).

To show $\simeq$ is symmetric, let $f : X \simeq Y$. We can take a quasi-inverse $f^{-1} : Y \to X$, and f^{-1} has a quasi-inverse f; apply (4.2.2) to show that $Y \simeq X$.

To show $\simeq$ is transitive, suppose $f : X \simeq Y$ and $g : Y \simeq Z$. We have $f^{-1} : Y \simeq X$ and $g^{-1} : Z \simeq Y$ as quasi-inverses, respectively. Observe that

$$(g^{-1} \circ f^{-1} \circ f \circ g) \sim g^{-1} \circ g \sim \mathrm{id}_Y,$$

$$(f^{-1} \circ g^{-1} \circ g \circ f) \sim f^{-1} \circ f \sim \mathrm{id}_X.$$

Thus $f^{-1} \circ g^{-1}$ is a quasi-inverse for $g \circ f : X \to Z$, and hence $g \circ f : X \simeq Z$ by (4.2.2). ■

Corollary 4.2.4

Let $f : X \simeq Y$ and $x, y : X$. Then $\mathrm{ap}_f : (x = y) \simeq (f(x) = f(y))$.

Beware: the quasi-inverse is not simply $\mathrm{ap}_{f^{-1}}$! In fact, if $(g, \alpha, \beta) : \mathrm{qinv}(f)$,

$$(\mathrm{ap}_f)^{-1} :\equiv \lambda(q : f(x) = f(y)).\ (\beta(x)^{-1} \bullet \mathrm{ap}_{f^{-1}}(q) \bullet \beta(y)) : (f(x) = f(y)) \simeq (x = y).$$

4.3 Function extensionality

Everything up to this point has used only the rules of IML. We are about to define our first supplementary axiom. Let $X, Y : \mathbf{U}$ and $f, g : X \to Y$. Consider $p : f = g$. We can use path induction on p and suppose that $f \equiv g$, $p \equiv \mathrm{refl}_f$ to define the function

$$\mathrm{happly} : (f = g) \to (f \sim g)$$
$$\mathrm{happly}(\mathrm{refl}_f) :\equiv \lambda(x : X).\ \mathrm{refl}_{f(x)}.$$

We really, really want $(f = g) \simeq (f \sim g)$, but unfortunately, there is no quasi-inverse to happly that can be defined using only the IML axioms. So, we define our own! Defining an axiom, in this case, is simply positing an atomic term of a type, namely

$$\mathrm{funext} : (f \sim g) \to (f = g), \tag{4.3.1}$$

which we define as the quasi-inverse to happly. This gives us the equivalence $\mathrm{happly} : (f = g) \simeq (f \sim g)$ through an application of (4.2.2).

Theorem 4.3.2

Let $X, Y : \mathbf{U}$, $f, g, h : X \to Y$, $x : X$, and $h : f = g$, $j : g = h$. Then:

1. $\mathrm{refl}_f = \mathrm{funext}(\lambda(x : X).\ \mathrm{refl}_{f(x)})$;
2. $h = \mathrm{funext}(\lambda(x : X).\ \mathrm{happly}(h, x))$;

3. $h^{-1} = \mathsf{funext}(\lambda(x : X).\ \mathsf{happly}(h, x)^{-1})$;
4. $h \bullet j = \mathsf{funext}(\lambda(x : X).\ \mathsf{happly}(h, x) \bullet \mathsf{happly}(j, x))$;
5. $\mathsf{happly}(\mathsf{funext}(h), x) = h(x)$.

Note that these are equalities, not synonymies.

In fact, the axiom of function extensionality is not strictly necessary in homotopy type theory; there is another axiom we will see from whence function extensionality follows, but the proof of entailment is outside the scope of this paper.

4.4 Uniqueness principles

We now have the tools to prove the uniqueness principles for most of the type formers we have seen. Recall that these principles are supposed to describe what terms of the type look like when compared to the constituent information from the formation rules; we demonstrate this using equivalences.

Theorem 4.4.1: Uniqueness principle for products

Let $X, Y : \mathbf{U}$ and $z, w : X \times Y$. Then

$$(z = w) \simeq (\mathsf{pr}_1(z) = \mathsf{pr}_1(w)) \times (\mathsf{pr}_2(z) = \mathsf{pr}_2(w)).$$

Proof: For $p : z = w$, the forward function is given by

$$f(p) :\equiv (\mathsf{ap}_{\mathsf{pr}_1}(p), \mathsf{ap}_{\mathsf{pr}_2}(p)),$$

and for $x : X$, $y : Y$, the backward function is defined using the induction principles for products and paths:

$$g(\mathsf{refl}_x, \mathsf{refl}_y) :\equiv \mathsf{refl}_{(x,y)}.$$

It suffices to show that they are quasi-inverses. But for $p_i : \mathsf{pr}_i(z) = \mathsf{pr}_i(w)$, $x : X$, and $y : Y$, we have

$$(g \circ f)(p_1, p_2) :\equiv (p_1, p_2),$$
$$(f \circ g)(\mathsf{refl}_x, \mathsf{refl}_y) :\equiv (\mathsf{refl}_x, \mathsf{refl}_y).$$

These outputs are synonymous with the inputs, and hence equal. Thus $g \circ f \sim \mathsf{id}_{z=w}$ and $f \circ g \sim \mathsf{id}_{(\mathsf{pr}_1(z)=\mathsf{pr}_1(w))\times(\mathsf{pr}_2(z)=\mathsf{pr}_2(w))}$, so by applying (4.2.2), we have an equivalence. ■

Theorem 4.4.2: Uniqueness principle for the unit type

For any $x : \mathbf{1}$, we have $x = \star$. Also, for any $x, y : \mathbf{1}$, we have $x = y \simeq \mathbf{1}$. Thus, in the unit type, every term is uniquely equal to $\star$.

Proof: Path induction on the type family $\lambda(x : \mathbf{1}).\ (x = \star)$. ■

Theorem 4.4.3: Uniqueness for Σ-types

Let $X : \mathbf{U}$ and $Y : X \to \mathbf{U}$, and $z, w : \sum_{(x:X)} Y(x)$. Then

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

Theorem 4.4.4: Uniqueness for coproducts

Let $X, Y : \mathbf{U}$ and $x, a : X, y, b : Y$. Then $\mathrm{in}\ell(x) = \mathrm{in}\ell(a) \simeq x = a$ and $\mathrm{in}r(y) = \mathrm{in}r(b) \simeq y = b$, and also $\mathrm{in}\ell(x) = \mathrm{in}r(y) \simeq \mathbf{0}$.

Theorem 4.4.5: Uniqueness of naturals

We have, for $m, n : \mathbf{N}$, the following:

- $0 = 0 \simeq \mathbf{1}$,
- $\mathrm{succ}(m) = 0 \simeq \mathbf{0}$ and $0 = \mathrm{succ}(n) \simeq \mathbf{0}$,
- $\mathrm{succ}(m) = \mathrm{succ}(n) \simeq m = n$.

4.5 UNIVALENCE

We have reached the second axiom that homotopy type theory posits on top of IML. This is the axiom of **UNIVALENCE**, that deals with applying path types to universes. We can already say that

$$\mathrm{eqtoequiv} : (X = Y) \to (X \simeq Y)$$

for any types $X, Y : \mathbf{U}$, by path induction and (4.2.1):

$$\mathrm{eqtoequiv}(\mathrm{refl}_X) :\equiv \mathrm{id}_X .$$

Wouldn't it be great if types that were equivalent were also equal? This is precisely what the univalence axiom states, defined as the quasi-inverse to the above:

$$\mathrm{ua} : (X \simeq Y) \to (X = Y). \tag{4.5.1}$$

It can be shown that our axiom of function extensionality follows from the univalence axiom.

Theorem 4.5.2

Let $X, Y, Z : \mathbf{U}$ and $f : X \simeq Y, g : Y \simeq Z$. Then

1. $\mathrm{refl}_X = \mathrm{ua}(\mathrm{id}_X)$;
2. $\mathrm{ua}(f)^{-1} = \mathrm{ua}(f^{-1})$;
3. $\mathrm{ua}(f) \bullet \mathrm{ua}(g) = \mathrm{ua}(g \circ f)$.

Theorem 4.5.3: Univalence and transport

Let $X : \mathbf{U}$, $Y : X \to \mathbf{U}$, $x, y : X$, and $p : x = y$. Then $\mathrm{ua}(p_*) =_{Y(x)=Y(y)} \mathrm{ap}_Y(p)$.

5 SETS AND PROPOSITIONS

So far, we have seen ways of making old types from new, and familiar constructions that can be formulated as types, like the natural numbers and Booleans. Now we will discuss the *propositions-as-types interpretation* of HoTT, and from this perspective develop a constructive theory of sets.

5.1 PROPOSITIONS-AS-TYPES

A *proposition* is a statement about certain mathematical objects. For example, the statement "4 is a multiple of 2" is a proposition. A key fact about propositions is that they are subject to proof or disproof: in the case above, it suffices to demonstrate that 4 is the product of 2 and 2, since this is sufficient for it to be a multiple of 2. However, 1 cannot be written as a product of 2 and an integer, and so it is not a multiple of 2, which constitutes a disproof of the proposition "1 is a multiple of 2".

In type theory, the objects that are subject to proof are *types*. Their proofs are exactly the terms of those types, since we have seen cases where types may not have terms (namely, the empty type). However, classically, we do not much care for the precise proof of a proposition, so long as the statement holds; for example, there are hundreds of proofs of the Pythagorean theorem, but all are equally valid at demonstrating its truth.

We define the **TRUNCATION** of a type $X : \mathbf{U}$ as the type $\|X\| : \mathbf{U}$, along with the constructors

$$|\cdot| : X \to \|X\|, \qquad \text{witness} : \prod_{(x,y:\|X\|)}(x = y).$$

The type $\|X\|$ is truncated in that every term of X infers a term $|x| : \|X\|$, but we can treat any $|x|$ equally as any other term of $\|X\|$. Thus, the only information $\|X\|$ contains is whether or not the type X is inhabited. More generally, we say that a type $X : \mathbf{U}$ is a **PROPOSITION** if the type

$$\mathsf{isProp}(X) :\equiv \prod_{(x,y:X)}(x = y)$$

is inhabited. Indeed, if X is any type, then its truncation $\|X\|$ is a proposition by virtue of witness. The induction principle for truncation is given by

$$\mathsf{ind}_{\|\cdot\|} : \prod_{(C:\|X\|\to\sum_{(Y:\mathbf{U})}\mathsf{isProp}(Y))}\left(\prod_{(x:X)} C(|x|)\right) \to \prod_{(y:\|X\|)} C(y)$$
$$\mathsf{ind}_{\|\cdot\|}(C, f, |x|) :\equiv f(x).$$

Note the codomain of the type family: we cannot extract more information out of a proposition than inhabitedness, so the result type must also be a proposition. We write

$$\mathbf{Prop} :\equiv \sum_{(Y:\mathbf{U})} \mathsf{isProp}(Y).$$

We use truncation to hide the "extra information" in certain cases: for example, in the case of coproducts, we have additional information about which constructor was used to create the particular term of $P + Q$, unless we truncate to get rid of it. Also, in the dependent pair case, we have access to the first projection, and hence which particular x such that $Q(x)$, unless we truncate to obfuscate it. We can now compare the traditional logical notation with our type-theoretic notation. For types P, Q, we have the

following analogies for propositional logic:

Traditional	HoTT
$\neg P$	$P \to \mathbf{0}$
$P \wedge Q$	$P \times Q$
$P \vee Q$	$\|P + Q\|$
$P \Rightarrow Q$	$P \to Q$
$P \Leftrightarrow Q$	$P \simeq Q$
$\forall(x \in P) : Q(x)$	$\prod_{(x:P)} Q(x)$
$\exists(x \in P) : Q(x)$	$\left\|\sum_{(x:P)} Q(x)\right\|$

For familiar negations, such as disequality, we may write $x \neq y$ to mean that $\neg(x = y)$.

example 5.1.1: Not all types are propositions

Consider the natural numbers, **N**. The uniqueness principle states that $0 = 0$, but that for any $n : \mathbf{N}$, we have $0 \neq \mathrm{succ}(n)$. In particular, $0 \neq \mathrm{succ}(0)$, so **N** is not a proposition.

5.2 Sets

A **SET** in HoTT is a type that has propositional equality. That is, for $X : \mathbf{U}$ and $x, y : X$, we want

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

In this case, we treat all proofs of the equality of x and y as equal to one another, so that we may use them interchangeably.

Just like any type X can be made into a proposition through $\|X\|$, there is a way to extract only the terms and 1-paths of X, so that any information about higher paths is lost. We call this type former **SET-TRUNCATION** and write $\|X\|_0$; it looks quite similar to regular truncation, but with the constructors

$$|\cdot|_0 : X \to \|X\|_0, \qquad \mathrm{witness}_0 : \prod_{(x,y:\|X\|_0)} \prod_{(p,q:x=y)} (p = q).$$

Notice the similarity of the definitions of sets and propositions. It is common to call sets 0-**TYPES**, and propositions (-1)-**TYPES**. One would expect a 1-**TYPE** to have the property

$$\text{is-1-type}(X) :\equiv \prod_{(x,y:X)} \prod_{(p,q:x=y)} \prod_{(\alpha,\beta:p=q)} (\alpha = \beta);$$

a recursive definition, for $n : \mathbf{N}$, would be

$$\text{is-succ}(n)\text{-type}(X) :\equiv \prod_{(x,y:X)} \text{is-}n\text{-type}(x = y).$$

In general, n-types are also $\mathrm{succ}(n)$-types; we show the base case below.

Theorem 5.2.1: Propositions are sets

Let $X : \mathbf{U}$ with $f : \mathrm{isProp}(X)$. Then there is a term of $\mathrm{isSet}(X)$.

Proof: For any $x, y : X$ we have $f(x, y) : x = y$. Fix $x : X$ and define $g(y) :\equiv f(x, y)$. Then if $y, z : X$ with $p : y = z$, we have $\mathsf{apd}_g(p) : p_*(g(y)) = g(z)$, which because we are working in the path type gives $g(y) \bullet p = g(z)$, or $p = g(y)^{-1} \bullet g(z)$. Thus, for any $x, y : X$ and $p, q : x = y$, we have $p = g(x)^{-1} \bullet g(y) = q$. ■

Theorem 5.2.2: Propositionality is a proposition

Let $X : \mathbf{U}$. Then $\mathsf{isProp}(\mathsf{isProp}(X))$.

Proof: Suppose $f, g : \mathsf{isProp}(X)$. We want $f = g$, but by function extensionality, it suffices to show $\prod_{(x,y:X)} f(x, y) = g(x, y)$. But both sides of the equality are paths in X, which are equal since X is a set by (5.2.1) and one of $f, g : \mathsf{isProp}(X)$. ■

Because of (5.2.1) and (5.2.2), we can use the shorthand "let $X : \mathbf{Prop}$" to mean "let $X : \mathbf{U}$ such that $\mathsf{isProp}(X)$", since any two terms of the latter can be considered equal. Also, by a similar argument to (5.2.2), to be a set is a proposition, and so we can write "let $X : \mathbf{Set}$" to mean "let $X : \mathbf{U}$ with $\mathsf{isSet}(X)$".

Theorem 5.2.3

Universes are not sets; that is, their path types are not necessarily propositions.

Proof: It suffices to exhibit a type $X : \mathbf{U}$ such that $X = X$ is not a proposition. For this we use the type of Booleans, $\mathbf{2} : \mathbf{U}$. Note that there is a function

$$\begin{aligned} \mathsf{not} &: \mathbf{2} \to \mathbf{2} \\ \mathsf{not}(0_{\mathbf{2}}) &:\equiv 1_{\mathbf{2}} \\ \mathsf{not}(1_{\mathbf{2}}) &:\equiv 0_{\mathbf{2}}, \end{aligned}$$

which is an equivalence, since it is its own quasi-inverse, *i.e.* $\mathsf{not} \circ \mathsf{not} \sim \mathsf{id}_{\mathbf{2}}$. However, note that (by function extensionality) $\mathsf{not} \neq \mathsf{id}_{\mathbf{2}}$, so we have two unequal terms of $\mathbf{2} \simeq \mathbf{2}$. By univalence, each gives rise to a path in $\mathbf{2} = \mathbf{2}$, so we must have $\mathsf{ua}(\mathsf{not}) \neq \mathsf{ua}(\mathsf{id}_{\mathbf{2}})$. Hence $\mathbf{2} = \mathbf{2}$ is not a proposition. ■

Theorem 5.2.4

Let $X, Y : \mathbf{U}$ and suppose $f : X \to Y$ such that $\mathsf{qinv}(f)$ is inhabited. Then

$$\mathsf{qinv}(f) \simeq \prod_{(x:X)} (x = x).$$

Proof: By (4.2.2), we know that f is an equivalence, so we have $e : \mathsf{isEquiv}(f)$ and $(f, e) : X \simeq Y$. By univalence, we may assume that $f(e)$ is of the form $\mathsf{eqtoequiv}(p)$ for some $p : X = Y$. By path induction, it suffices to consider the case where $X \equiv Y$ and $p \equiv \mathsf{refl}_X$, in which case f is id_X. Thus it suffices to show $\mathsf{qinv}(\mathsf{id}_X) \simeq \prod_{(x:X)}(x = x)$. We have

$$\mathsf{qinv}(\mathsf{id}_X) \equiv \sum_{(g:X\to X)}(g \sim \mathsf{id}_X) \times (g \sim \mathsf{id}_X),$$

which by function extensionality is equivalent to

$$\sum_{(g:X\to X)}(g = \mathsf{id}_X) \times (g = \mathsf{id}_X).$$

It is not difficult, but here, to show that this is equivalent to

$$\sum_{(h:\sum_{(g:X\to X)}(g=\mathsf{id}_X))}(\mathsf{pr}_1(h) = \mathsf{id}_X).$$

But since id_X is unique, the type $\sum_{(g:X\to X)}(g = \mathsf{id}_X)$ is a proposition, with a canonical inhabitant $(\mathsf{id}_X, \mathsf{refl}_{\mathsf{id}_X})$. Thus the above type is equivalent to $\mathsf{id}_X = \mathsf{id}_X$, which by function extensionality is equivalent to $\prod_{(x:X)}(x = x)$ as required. ∎

The above theorem tells us that when X is not a set, for any $f : X \to Y$ we are not guaranteed that $\mathsf{qinv}(f)$ is a proposition. However, we can obtain a proposition by splitting the quasi-inverse into left- and right-inverses, like we did in the type $\mathsf{isEquiv}(f)$, because then the pre- and post-compositions with f must be homotopic to the identity functions on X and Y respectively, which are unique.

5.3 The axiom of choice

Perhaps the best example to explain the effects of truncation is the type-theoretic analogue of the **axiom of choice** (AC). In classical set theory, AC would be stated as follows:

if for all $x \in X$ there is some $y \in Y$ such that $R(x, y)$,
there is a function $f : X \to Y$ such that for all $x \in X$ we have $R(x, f(x))$.

We have two different interpretations of this statement in type theory: the first, for types $X, Y : \mathbf{U}$ and a relation $R : X \to Y \to \mathbf{U}$, states that

$$\mathsf{ac}_\infty : \left(\prod_{(x:X)} \sum_{(y:Y)} R(x, y)\right) \to \left(\sum_{(f:X\to Y)} \prod_{(x:X)} R(x, f(x))\right),$$

which is actually provable, since we can define for $g : \prod_{(x:X)} \sum_{(y:Y)} R(x, y)$ the term

$$\mathsf{ac}_\infty(g) :\equiv (\lambda(x : X).\ \mathsf{pr}_1(g(x)), \lambda(x : X).\ \mathsf{pr}_2(g(x))) : \sum_{(f:X\to Y)} \prod_{(x:X)} R(x, \mathsf{pr}_1(g(x))).$$

Since we can prove this, it is not an axiom. What is usually meant by "axiom of choice" is the *choice of a specific element from an otherwise inscrutable set*, whereas in the above we have all of the information about the terms $y : Y$ for which $R(x, y)$. To capture this effectively, we need truncation to hide the constructive information. In this case, we let $X : \mathbf{Set}$ and $A : X \to \mathbf{Set}$, and $P : \prod_{(x:X)} A(x) \to \mathbf{Prop}$. Then

$$\mathsf{AC} :\equiv \left(\prod_{(x:X)} \left\|\sum_{(a:A(x))} P(x, a)\right\|\right) \to \left\|\sum_{(g:\prod_{(x:X)} A(x))} \prod_{(x:X)} P(x, g(x))\right\|.$$

It can also be stated, for $X : \mathbf{Set}$ and $Y : X \to \mathbf{Set}$, as "the cartesian product of nonempty sets is nonempty":

$$\mathrm{AC} :\equiv \left(\prod_{(x:X)} \|Y(x)\|\right) \to \left\|\prod_{(x:X)} Y(x)\right\|;$$

In fact, AC is an equivalence itself, due to the fact that both sides are propositions and that the RHS always entails the LHS. To see that both formulations of AC are equivalent, set

$$Y(x) :\equiv \sum_{(a:A(x))} P(x,a),$$

and apply the following lemma.

Lemma 5.3.1: Inward universal property of pair type families

Let $X : \mathbf{U}$, $A : X \to \mathbf{U}$, and $P : \prod_{(x:X)} A(x) \to \mathbf{U}$. Then

$$\left(\prod_{(x:X)} \sum_{(a:A(x))} P(x,a)\right) \simeq \left(\sum_{(g:\prod_{(x:X)} A(x))} \prod_{(x:X)} P(x,g(x))\right).$$

Proof: The forward function is given by sending $f : \prod_{(x:X)} \sum_{(a:A(x))} P(x,a)$ to

$$(\mathrm{pr}_1 \circ f, \mathrm{pr}_2 \circ f) : \sum_{(g:\prod_{(x:X)} A(x))} \prod_{(x:X)} P(x,g(x)).$$

We want to show that $\lambda((g,h)).\ \left(\lambda(x:X).\ (g(x),h(x))\right)$ is a quasi-inverse to the above. The round-trip composite acting on $f : \prod_{(x:X)} \sum_{(a:A(x))} P(x,a)$ yields the function

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

which for each $x : X$ gives

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

by uniqueness in dependent pair types. Function extensionality then grants

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

Now, given $(g,h) : \sum_{(g:\prod_{(x:X)} A(x))} \prod_{(x:X)} P(x,g(x))$, the round-trip composite in the other direction yields $(\lambda(x:X).\ g(x), \lambda(x:X).\ h(x))$, which is synonymous to (g,h) by uniqueness in function types. ■

The simpler statement for independent pairs, for $X : \mathbf{U}$ and $Y, Z : X \to \mathbf{U}$, is precisely

$$\left(\prod_{(x:X)} Y(x) \times Z(x)\right) \simeq \left(\prod_{(x:X)} Y(x)\right) \times \left(\prod_{(x:X)} Z(x)\right).$$

example 5.3.2: Outward universal property for pair types

Also called the **CURRY-HOWARD EQUIVALENCE**, for types $X, Y, Z : \mathbf{U}$ it is given by

$$(X \times Y) \to Z \ \simeq\ X \to (Y \to Z),$$

whereas for $X, Y : \mathbf{U}$ and $Z : X \times Y \to \mathbf{U}$, we have

$$\prod_{(w:X\times Y)} Z(w) \ \simeq\ \prod_{(x:X)} \prod_{(y:Y)} Z((x,y)).$$

Note that the reverse function is exactly the induction principle for products. This is also known to category theorists as (a special case of) the *tensor-hom adjunction*.

The axiom of choice (for sets) implies the **LAW OF EXCLUDED MIDDLE** (for propositions):

$$\mathrm{LEM} :\equiv \prod_{(P:\mathbf{Prop})} P + \neg P.$$

Thus, assuming AC for the universe $\mathbf{U}_i$ has the consequence that the type

$$\left(\sum_{(X:\mathbf{U}_i)} \mathrm{isProp}(X)\right) \simeq \mathbf{2};$$

this is due to the fact that $P \simeq \mathbf{1}$ if P is an inhabited proposition, and $P \simeq \mathbf{0}$ otherwise, and so $P + \neg P$ is always uniquely inhabited. The equivalence sends all propositions P for which $\mathrm{LEM}(P)$ is an $\mathrm{in}\ell$ to $1_{\mathbf{2}}$, and those that are $\mathrm{in}r$'s to $0_{\mathbf{2}}$. Also, if a proposition is *not uninhabited*, then it must be the case that it is inhabited, so LEM is equivalent to the **LAW OF DOUBLE NEGATION**.

Recall that a proposition P is **DECIDABLE** if we have $P + \neg P$. Assuming the axiom of choice, then, grants that **all propositions are decidable** at the particular universe level. This is a very strong assumption, but not entirely unexpected, given the hesitation towards its acceptance even in classical mathematics.

5.4 PROPOSITIONAL RESIZING AND SUBSETS

Lemma 5.4.1: Propositional subsumption

There is a map

$$\left(\sum_{(X:\mathbf{U}_i)} \mathrm{isProp}(X)\right) \to \left(\sum_{(X:\mathbf{U}_{i+1})} \mathrm{isProp}(X)\right).$$

Proof (Russell): Immediate from UNIV-SUBSUME. ■

Proof (Tarski): Let $X : \mathbf{U}_i$ with $\mathrm{isProp}(X)$. Then $\mathrm{lift}_{i+1}(X) : \mathbf{U}_{i+1}$. Also, for any $x, y : T_i(X)$, we have $x =_X y : \mathbf{U}_i$, and so $\mathrm{lift}_{i+1}(x =_X y) : \mathbf{U}_{i+1}$. But $\mathrm{witness}(x, y) : T_i(x =_X y)$ for every $x, y : X$. Thus we must have a term with type $T_{i+1}(\mathrm{lift}_{i+1}(x =_X y))$ induced by $\mathrm{witness}(x, y)$. ■

Another axiom that we may choose to accept is the axiom of **PROPOSITIONAL RESIZING**: that there is a quasi-inverse to the map defined in the above lemma. Upon accepting this axiom, propositions at every universe level are equivalent to some proposition in the base universe $\mathbf{U}_0$. We can then define

$$\mathbf{\Omega} :\equiv \sum_{(P:\mathbf{U}_0)} \mathrm{isProp}(P),$$

and for any $X : \mathbf{U}_i$ we can define maps $X \to \mathbf{\Omega}$ by composing any map $X \to \sum_{(P:\mathbf{U}_i)} \mathrm{isProp}(P)$ with the maps given by propositional resizing.

Indeed, when $X : \mathbf{Set}$, we can define the **POWERSET** $\mathbf{P}(X) :\equiv X \to \mathbf{\Omega}$. Terms $\gamma : \mathbf{P}(X)$ are called **SUBSETS** of the set X, and there is a **MEMBERSHIP PREDICATE**, defined as the adjoint to evaluation:

$$\begin{aligned} \in &: X \to \mathbf{P}(X) \to \mathbf{\Omega} \\ x \in \gamma &:\equiv \gamma(x). \end{aligned}$$

Since γ is a proposition family, we write $x \in \gamma$ when $\gamma(x) \simeq \mathbf{1}$ and $x \notin \gamma$ when $\gamma(x) \simeq \mathbf{0}$. We say that γ is a **DECIDABLE SUBSET** if $(x \in \gamma) + (x \notin \gamma)$.

We also have the familiar subset operations. Given $X : \mathbf{Set}$ and $\gamma, \varphi : \mathbf{P}(X)$, we use the traditional logical notation to define

$$\begin{aligned} \cup, \cap, \setminus &: \mathbf{P}(X) \to \mathbf{P}(X) \to \mathbf{P}(X) \\ \gamma \cup \varphi &:\equiv \lambda(x : X).\, \gamma(x) \vee \varphi(x), \\ \gamma \cap \varphi &:\equiv \lambda(x : X).\, \gamma(x) \wedge \varphi(x), \\ \gamma \setminus \varphi &:\equiv \lambda(x : X).\, \gamma(x) \wedge \neg\varphi(x). \end{aligned}$$

The **COMPLEMENT** of a subset is given by

$$\begin{aligned} -^{\complement} &: \mathbf{P}(X) \to \mathbf{P}(X) \\ \gamma^{\complement} &:\equiv \lambda(x : X).\, \neg\gamma(x). \end{aligned}$$

We also have the **INCLUSION** operator,

$$\begin{aligned} \subseteq &: \mathbf{P}(X) \to \mathbf{P}(X) \to \Omega \\ \gamma \subseteq \varphi &:\equiv \textstyle\prod_{(x:X)} (\gamma(x) \to \varphi(x)). \end{aligned}$$

Each $\gamma(x) \to \varphi(x)$ is a proposition: if a map exists, it is unique, since its codomain is a proposition. Thus the whole Π-type is a proposition. We could have used the traditional notation, in which case

$$\gamma \subseteq \varphi :\equiv \forall(x : X).\, (x \in \gamma) \Rightarrow (x \in \varphi).$$

5.5 Contrast with ZF sets

Although there is a notion of membership in HoTT, it is not the same as the one present in ZF. In HoTT, membership occurs within a specified type, and terms are members only of subsets of that type, since the notion of a term not belonging to a type is nonsensical. However in ZF, there is a global membership predicate, and sets can be constructed without first defining a set in which they are contained.

ZF set theory also posits the **AXIOM OF EXTENSIONALITY**:

$$\forall \gamma, \varphi :\ (\gamma \subseteq \varphi \wedge \varphi \subseteq \gamma) \Rightarrow (\gamma = \varphi).$$

But its analogue in HoTT is actually provable: two subsets are equal if each contains the other.

Theorem 5.5.1: Set extensionality

Let $X : \mathbf{Set}$ and $\gamma, \varphi : \mathbf{P}(X)$. Then

$$(\gamma \subseteq \varphi \wedge \varphi \subseteq \gamma) \Rightarrow (\gamma = \varphi).$$

Proof: We want to show

$$\textstyle\prod_{(x:X)} (\gamma(x) \to \varphi(x)) \times (\varphi(x) \to \gamma(x)) \to (\gamma(x) = \varphi(x)).$$

Fix $x : X$. The types $\gamma(x), \varphi(x)$ are propositions, so the two functions in the pair must be each other's quasi-inverse. *(continued)*

Proof (continued): Hence the domain becomes

$$\gamma(x) \Leftrightarrow \varphi(x) \;\equiv\; \gamma(x) \simeq \varphi(x),$$

but now the desired function

$$(\gamma(x) \simeq \varphi(x)) \to (\gamma(x) = \varphi(x))$$

is simply the univalence axiom. ■

In fact, (5.5.1) is an equivalence, with the reverse function given by eqtoequiv and the universal property.

5.6 ALGEBRAIC STRUCTURES

Traditionally, the description of an algebraic structure takes place over a set, so that there are no higher coherence paths to keep track of; we will stick with this characterization. We will present the notion of a semigroup and a monoid in the language of HoTT.

We can also create a structure on a general type through set-truncation in order to use subsets. These are useful because being a member of a subset is a proposition, which grants the propositionality of many other properties.

5.6.1 SEMIGROUPS

A **SEMIGROUP STRUCTURE** on a type $X : \mathbf{Set}$ is a subset $M : \mathbf{P}(X)$ and an operation $\diamond : X \to X \to X$, such that

$$\mathrm{closed}_M(\diamond) :\equiv \textstyle\prod_{(x,y:X)} M(x) \times M(y) \to M(x \diamond y);$$

or in traditional logical notation,

$$\forall(x, y \in M).\ x \diamond y \in M;$$

and such that

$$\mathrm{assoc}_M(\diamond) :\equiv \forall(x, y, z \in M).\ x \diamond (y \diamond z) =_X (x \diamond y) \diamond z.$$

Lemma 5.6.1

Closedness and associativity are propositions, since M is a family of propositions and paths in sets are propositions.

Since these properties are propositions, we may omit the naming of a specific term so long as we have given a proof of its existence, or assumed it. For this reason, we can say "let $(M, \diamond)$ be a semigroup on X" to mean

"let $X : \mathbf{Set}$ and $M : \mathbf{P}(X)$ and $\diamond : X \to X \to X$ with $\mathrm{closed}_M(\diamond)$ and $\mathrm{assoc}_M(\diamond)$",

or write $\mathrm{Semigroup}_X(M, \diamond)$ to denote the conjunction of the witnesses for the required properties.

Theorem 5.6.2: Subsemigroup test

Let $(M, \diamond)$ be a semigroup on $X : \mathbf{Set}$. If there is a subset $L : \mathbf{P}(X)$ with $L \subseteq M$ such that $\mathrm{closed}_L(\diamond)$, then $(L, \diamond)$ is a semigroup on X.

Proof: Recall $\text{closed}_L(\diamond) :\equiv \forall(x, y \in L).\ x \diamond y \in L$. But this suffices for $\text{assoc}_L(\diamond)$, since $L \subseteq M$ permits us to use the path given by $\text{assoc}_M(\diamond)$. ■

5.6.2 MONOIDS

A **MONOID STRUCTURE** is a semigroup structure with a neutral term for the binary operation. That is, any semigroup $(M, \diamond)$ on $X : \mathbf{Set}$ can be turned into a monoid by supposing some $e : X$ with $e \in M$ and

$$\text{neutral}_\diamond(e) :\equiv \forall(x : X).\ (e \diamond x =_X x) \wedge (x \diamond e =_X x).$$

Since paths in sets are propositions, clearly $\text{neutral}_\diamond(e)$ is a proposition. Monoid structures on sets X are hence written $\text{Monoid}_X(M, \diamond, e)$.

Lemma 5.6.3

Monoid units are unique.

Proof: Let $X : \mathbf{Set}$ permit a semigroup structure $(M, \diamond)$ and suppose $e_1, e_2 : X$ with $e_1, e_2 \in M$ and $\text{neutral}_\diamond(e_1)$, $\text{neutral}_\diamond(e_2)$. Then the first projection of $\text{neutral}_\diamond(e_1, e_2)$ is of type $e_1 \diamond e_2 = e_2$, and the second projection of $\text{neutral}_\diamond(e_2, e_1)$ has type $e_1 \diamond e_2 = e_1$. Concatenating the inverse of the latter with the former gives a term of $e_1 = e_2$. ■

Theorem 5.6.4: Submonoid test

Let $(M, \diamond, e)$ be a monoid structure on $X : \mathbf{Set}$. If there is a subset $L \subseteq M$ with $\text{closed}_L(\diamond)$ and $e \in L$, then $(L, \diamond, e)$ is a monoid structure on X.

5.6.3 ORDERS AND LATTICES

Let $X : \mathbf{Set}$. A **PARTIAL ORDER** on a subset $M : \mathbf{P}(X)$ is a binary proposition family $\leq_M : X \to X \to \mathbf{Prop}$ that is reflexive, antisymmetric, and transitive:

$$\forall(x, y, z \in M).\ (x \leq_M x) \wedge (x \leq_M y \wedge y \leq_M x \Rightarrow x =_X y) \wedge (x \leq_M y \wedge y \leq_M z \Rightarrow x \leq_M z).$$

Note that every member of M generates two subsets of X through partial application of the partial order; that is, for $x : X$ with $x \in M$,

$$\text{lteq}_M(x) :\equiv (- \leq_M x) : \mathbf{P}(X), \qquad \text{gteq}_M(x) :\equiv (x \leq_M -) : \mathbf{P}(X).$$

Note that $\text{lteq}_M(x), \text{gteq}_M(x) \subseteq M$. Those well-versed in lattice theory would recognize these as the **PRINCIPAL IDEALS** and, dually, the **PRINCIPAL FILTERS** of the partial order.

Given a partial order structure $(M, \leq_M)$ on a set X and a subset $L \subseteq M$, a **LOWER BOUND** of L is a term $\ell \in M$ satisfying

$$\ell\text{bd}_L(\ell) :\equiv \forall(x \in L).\ (\ell \leq_M x),$$

and an **UPPER BOUND** for L is a term $u \in M$ satisfying

$$\text{ubd}_L(u) :\equiv \forall(x \in L).\ (x \leq_M u).$$

Indeed, $\ell\mathrm{bd}_L, \mathrm{ubd}_L : \mathbf{P}(X)$ and $\ell\mathrm{bd}_L, \mathrm{ubd}_L \subseteq M$. The **INFIMUM** (**SUPREMUM**) of L is the upper bound of all lower bounds (lower bound of all upper bounds). That is,

$$\inf_L(\ell) :\equiv \ell\mathrm{bd}_L(\ell) \wedge \mathrm{ubd}_{\ell\mathrm{bd}_L}(\ell),$$
$$\sup_L(u) :\equiv \mathrm{ubd}_L(u) \wedge \ell\mathrm{bd}_{\mathrm{ubd}_L}(u).$$

Lemma 5.6.5

Infima and suprema are unique.

Proof: Let X : **Set** admit a partial order structure $(M, \leq_M)$, and let $L \subseteq M$. Suppose $j, k \in \inf_L$. Note that $\mathrm{ubd}_{\ell\mathrm{bd}_L}(j,k) : (k \leq_M j)$ and $\mathrm{ubd}_{\ell\mathrm{bd}_L}(k,j) : (j \leq_M k)$. Applying antisymmetry yields the result. The proof is similar for suprema. ■

5.7 SPHERES AND HOMOTOPY GROUPS

Of course, one cannot mention homotopy theory without discussing spheres. Indeed, the spheres are generated iteratively by the **SUSPENSION** type former Sus, where given X : **U** the type $\mathrm{Sus}(X)$ has constructors

$$N, S : \mathrm{Sus}(X), \qquad \mathrm{merid} : X \to (N = S).$$

The suspension takes terms of X and turns them into paths between two additional points, imagined as the north and south poles of the new type. The spheres are indexed by **N** and are denoted

$$\begin{aligned} \mathbf{S}^0 &:\equiv \mathrm{Sus}(\mathbf{0}) \quad \simeq \mathbf{2}, \\ \mathbf{S}^{\mathrm{succ}(n)} &:\equiv \mathrm{Sus}(\mathbf{S}^n). \end{aligned}$$

We have $N : \mathbf{S}^1$ and $\mathrm{merid}(0_\mathbf{2}) \bullet \mathrm{merid}(1_\mathbf{2})^{-1} : N = N$. For simplicity, we give them the names

$$\mathrm{base} : \mathbf{S}^1, \qquad \mathrm{loop} : \mathrm{base} = \mathrm{base}.$$

Since $0_\mathbf{2} \neq 1_\mathbf{2}$, we have $\mathrm{merid}(0_\mathbf{2}) \neq \mathrm{merid}(1_\mathbf{2})$, and so $\mathrm{refl}_{\mathrm{base}} \neq \mathrm{loop}$. Thus $\mathbf{S}^1$ is a 1-type. The induction principle for suspensions takes the following form:

$$\mathrm{ind}_{\mathrm{Sus}(X)} : \prod_{(C:\mathrm{Sus}(X)\to\mathbf{U})} \prod_{(n:C(N))} \prod_{(s:C(S))} (\prod_{(x:X)} \mathrm{merid}(x)_*(n) = s) \to \prod_{(x:\mathrm{Sus}(X))} C(x)$$
$$\begin{aligned} \mathrm{ind}_{\mathrm{Sus}(X)}(C, n, s, m, N) &:\equiv n, \\ \mathrm{ind}_{\mathrm{Sus}(X)}(C, n, s, m, S) &:\equiv s, \end{aligned}$$

and for each $x : X$ we have $\mathrm{apd}_{\mathrm{ind}_{\mathrm{Sus}(X)}(C,n,s,m)}(\mathrm{merid}(x)) = m(x)$.

To define homotopy groups, we must consider an additional type former Ω, which given X : **U** and a "basepoint" term $x : X$ is the type

$$\Omega(X, x) :\equiv (x = x, \mathrm{refl}_x),$$

called the **LOOP SPACE AT** x, where the new "basepoint" term is refl_x. The **ITERATED LOOP SPACE** is also indexed by **N** and is defined as

$$\begin{aligned} \Omega^0(X, x) &:\equiv (X, x) \\ \Omega^{\mathrm{succ}(n)}(X, x) &:\equiv \Omega^n(\Omega(X, x)). \end{aligned}$$

The **HOMOTOPY GROUPS** of a type $X : \mathbf{U}$ with basepoint $x : X$ are then the types

$$\pi_n(X, x) :\equiv \|\Omega^n(X, x)\|_0 ,$$

whose group structure is inherited from path inversion and concatenation. It is not too convoluted to prove that $\pi_1(\mathbf{S}^1)$ is equivalent to the group of integers; indeed, the key observation is that $\mathsf{refl}_{\mathsf{base}}$ maps to 0 and loop maps to 1 under the equivalence, whereas inversion becomes negation and concatenation becomes addition.

6 Conclusion

We have seen what it means for a framework to be a formal system, how homotopy type theory differs from set-based descriptions of mathematics, and the formulation of several familiar concepts inside HoTT. Perhaps the key observation is that, by treating general formulae as functions with specific domains, we can rigidify the world of classical mathematics into a constructive, proof-relevant framework.

There are other areas of set theory that can be synthesized in HoTT; perhaps the simplest notion is that of **CARDINAL NUMBERS**, where the type $\mathbf{Card} :\equiv \|\mathbf{Set}\|_0$, the set-truncation of the type of sets. In this case we have cardinal addition induced by the coproduct, cardinal multiplication induced by the pair, and cardinal exponentiation induced by the function type former. There is also a type for the real numbers, but there are multiple ways to construct it and their equivalence follows only from AC.

The interpretations such as propositions-as-types, and programs-as-proofs, make HoTT as a deductive system more suitable than ZF for computer-assisted theorem proving. Indeed, much of the theory has already been formalized in the Coq programming language, and is freely available to download and tinker with. There are extensions of HoTT, as well; a notable one is called cubical type theory, in which univalence is not an axiom, but a theorem! One drawback is that the IML equality types are not synonymous with the notion of paths in the models of cubical type theory, but they are provably equivalent.

References

[1] Zhaohui Luo. Notes on universes in type theory. 2012.

[2] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis; `https://archive-pml.github.io/martin-lof/pdfs/Bibliopolis-Book-retypeset-1984.pdf`, 1984.

[3] nLab. Homotopy type theory, Univalence axiom, Homotopy hypothesis, Homotopy n-type, Predicative mathematics, Cubical type theory, Computational trinitarianism, accessed 13/11/2019.

[4] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[5] Wikipedia. Formal system, Formal grammar, Rule of inference, Intuitionistic type theory, Principia Mathematica, Typed lambda calculus, Lambda cube, accessed 13/11/2019.