Type Theory

Lectures by Paige Randall North
Notes by Jason Schuchardt

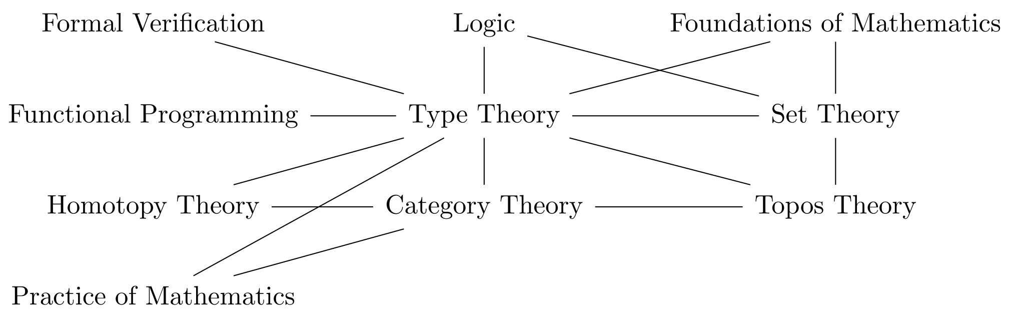
Ross Program, July 2019

Contents

1 Lecture 1

Type theory is a relatively new branch of math. It cannibalizes a lot of other areas of mathematics. In particular it has connections to logic, set theory, and functional programming. It is the basis of a lot of modern functional programming. It also takes a lot of inspiration from category theory, and its subfield, topos theory. It's also related to homotopy theory.

The map:



1.1 Basics

Basic objects are *types* and *terms*. Every term belongs to exactly one type. When the term t belongs to a type T, then we write $t : T$.

Example 1.1. In set theory, one writes $5 \in \mathbb{N}$. In type theory, one writes $5 : \mathbb{N}$.

In type theory, every term belongs to exactly one type. This is not the case in set theory.

Example 1.2. In set theory, one can think of 5 by itself. It is itself a set. One can then say that $5 \in \mathbb{N}$, and $5 \in \mathbb{R}$.

This is not the case in type theory. In type theory, one can only consider a term, like 5, as being part of a type, $\mathbb{N}$.

The terms $5 : \mathbb{N}$ and $5 : \mathbb{R}$ are completely different things. We might be able to compare them in a few classes, but they aren't immediately comparable. They are distinct.

History: The first person to come up with a sort of type theory was Bertrand Russell in 1902. He invented it to solve Russell's paradox (Is there a set that contains all the sets that don't contain themselves). However, it was very different from what we'll be talking about.

1.2 A first type theory: The Simply Typed Lambda Calculus

The simply typed lambda calculus (Church 1940) is the simplest type theory along the lines of what we're thinking about, and a model for computation (functional programming).

Definition 1.1. A simply typed lambda calculus with $\implies$ consists of the following:

- Atomic types $T_1, \ldots, T_n$.
- A type $S \implies T$ (S "implies" T) for every two types S and T.

- For each type T, we have variables
$$x_1^T, \dots, x_m^T : T,$$
and for any term $t : T$ and variable $x : S$, we have the term $\lambda x.t : S \implies T$.

  We say we are *abstracting x from t*. The term t might involve x (if it doesn't then the "function" we are defining is a constant function).

- For every term $f : S \implies T$, and every term $s : S$, we have a term $fs : T$. We call this term the *application of f to s*.

These are subject to the equations

- For every term $t : T$, variable $x : S$, term $s : S$,
$$(\lambda x.t)s = t[s/x],$$
where $t[s/x]$ is the term obtained from t by replacing every instance of x with s.

- For each $f : S \implies T$ and variable $x : S$ which doesn't occur in f (so we don't have variable naming conflicts basically). Then we have
$$\lambda x.(fx) = f : S \implies T$$

Another example of a type is $\mathbb{N}$. $\mathbb{N}$ is a type with terms $0 : \mathbb{N}$ and $Sn : \mathbb{N}$, where $n : \mathbb{N}$ is a term. 0 is a *closed term of* $\mathbb{N}$.

T_1 is an atomic type, but $T_1 \implies T_2$ is nonatomic. $(T_1 \implies T_2) \implies T_3$ is also nonatomic.

1.2.1 Things we can do!

Example 1.3. For every term $t : T$ not containing all variables of type S, there is a term of $S \implies T$. We need to supply a variable $x : S$ and our term $t : T$ to get such a term. For example $\lambda x_j^S.t : S \implies T$, where x_j^S doesn't occur in t.

Then
$$(\lambda x_j^S.t)s = t[s/x_j^S] = t.$$
This is a constant function that sends everything in S to our term t.

Example 1.4. For every type T, there is a term $T \implies T$. Let $x : T$ be a variable. We can build the identity function:
$$\lambda x.x : T \implies T.$$

The first equation tells us that
$$(\lambda x.x)t = x[t/x] = t,$$
which is why we can call it the identity function.

Question: Do we have a different identity function for each variable?

Answer: We can prove that they are the same. In type theory we distinguish between syntax and semantics. The formulas $\lambda x.x$ and $\lambda y.y$ are *different* formulas, in a sort of naive sense. However their semantics are somehow the same.

For the proof, we can use the function variable replacement rule (the second equation)

$$\lambda x.(fx) = f.$$

If x and y are variables, then let $f = \lambda y.y$.

$$\lambda y.y = f = \lambda x.(fx) = \lambda x.((\lambda y.y)x) = \lambda x.x.$$

This is a basic theory of functions. We have a type for functions, $S \implies T$, and an identity function, $\lambda x.x : T \implies T$.

Note! The only functions we have are ones for which we can write down an explicit formula. E.g. $\lambda x.x$, $\lambda x.t$.

This is *not* a theory of *mathematical* functions (in the sense of Set Theory), but *computational* functions!

What is a "mathematical" function. Officially, in set based math, a function $f : X \to Y$ is defined as a graph $G \subset X \times Y$. Intuitively, this set is something like a table, that records for each $x \in X$, the value $f(x) \in Y$. For example, for the function

$$\lambda x.x^2,$$

(the function $f(x) = x^2$) from $\mathbb{R}$ to $\mathbb{R}$. The set theoretic representation of this function will be as the set of pairs

$$\{(0,0), (1,1), (2,4), (\sqrt{2}, 2), \ldots\}$$

Even though the functions we consider in math usually have nice and finite formulas, a mathematical function doesn't have to have a formula, and might only be describable by such an infinite table.

Neither humans nor computers can work with infinite descriptions, so if we're interested in studying functions in a computational setting, we need a finite formula for every function.

In type theory, we have the interpretation

$$\text{Functions} \longleftrightarrow \text{Programs}.$$

Under this interpretation

$$\begin{aligned} \text{Types} &\longleftrightarrow \text{Specifications of programs} \\ \text{Terms} &\longleftrightarrow \text{Programs fulfilling the specification} \\ S \implies T &\longleftrightarrow \text{Programs which take input } s : S \text{ and return } t : T \end{aligned}$$

2 Lecture 2

2.1 Contexts

Contexts are important for math.

Example 2.1. For any natural number n, $2n$ is even.

The phrase "$2n$ is even" is the math part. The phrase "For any natural number n" is the context. It declares our variable basically.

In a type theoretic way, we might say: For $n : \mathbb{N}$, $2n$ is even.

Example 2.2. Another sentence might be "Suppose $f : \mathbb{R} \to \mathbb{R}$ is a continuous function such that $f(0) < 0$ and $f(1) > 0$, then there is some $c \in \mathbb{R}$, $0 < c < 1$ such that $f(c) = 0$." The context here declares the variable f.

Example 2.3. For $f : T$, there is some $\cdots$. We can encapsulate all of the hypotheses on f into the type T. We can have T be the type whose terms t are continuous functions $\mathbb{R} \to \mathbb{R}$ such that $t(0) < 0$ and $t(1) > 0$.

Example 2.4. For any variable $x : T$, there is a term $x : T$. In symbols, we might write:

$$x : T \vdash x : T,$$

where the left $x : T$ is the context, and $x : T$ is the product.

Example 2.5. We can also write something similar for function application. For any variable $f : S \implies T$, and any $x : S$, we get $fx : T$. In symbols, this is

$$f : S \implies T, x : S \vdash fx : T.$$

Example 2.6. In the natural numbers, we have the statement

$$\vdash 0 : \mathbb{N}.$$

This requires no context, since there are no hypotheses necessary.

2.2 Simply Typed Lambda Calculus Again

We'll redefine the STLC with some notation that became standard a bit over twenty years ago, that will help us fit it into a more modern framework.

There are three kinds of *judgments*:

1. (Is a type judgement) ____ TYPE (ex. $T_1 \implies T_2$ TYPE)
2. (Is of type judgement) ____ $\vdash$ ____ : ____ (ex. $f : S \implies T, x : S \vdash fx : T$)
3. (Equality judgement) ____ $\vdash$ ____ $=$ ____ : ____ (ex. $x : T \vdash (\lambda y.y)x = x : T$)

These are the things that we can say about our type theory. A *rule* consists of

$$\frac{\text{(some judgements)}}{\text{(a judgement)}}.$$

For example, we might have the rule

$$\frac{\vdash f : S \implies T, \vdash S : S}{\vdash f(S) : T}.$$

Definition 2.1. The *simply typed lambda calculus* is given by the following rules:

1. We have types:
$$\frac{}{\vdash T_1 \text{ TYPE}}, \dots, \frac{}{\vdash T_n \text{ TYPE}}$$
2. And we have terms, we think of as variables:
$$\frac{T \text{ TYPE}}{\vdash x_1^T : T}, \dots, \frac{T \text{ TYPE}}{\vdash x_m^T : T}$$
3. We have a secret judgement for contexts, satisfying the rules
$$\frac{}{\emptyset \text{ ctxt}}, \quad \frac{\Gamma \text{ ctxt}, \quad T \text{ TYPE}}{\Gamma, x_i^T : T \text{ ctxt}},$$
$$\frac{}{\Gamma, x : T, \Delta \vdash x : T}, \quad \frac{\Gamma \vdash t : T, \quad x : T \vdash s : S}{\Gamma \vdash s[t/x] : S}$$
4. We have rules for equality that are fairly obvious
$$\frac{t : T}{t = t : T}, \quad \frac{t = s : T}{s = t : T}, \quad \frac{t = s : T, \quad s = u : T}{t = u : T}$$
We also have an equality rule for substitution.
5. Rules for $\implies$-types
$$\frac{S \text{ TYPE}, \quad T \text{ TYPE}}{S \implies T \text{ TYPE}}, \quad \frac{S \text{ TYPE}, \quad T \text{ TYPE}, \quad x_i^S : S \vdash t : T}{\lambda x_i^S.t : S \implies T}$$
omitting the type judgements, we have
$$\frac{f : S \implies T, \quad s : S}{fs : T}, \quad \frac{x_i^S : S \vdash t : T, \quad s : S}{(\lambda x_i^S.t)s = t[s/x_i^S] : T}$$
$$\frac{f : S \implies T, \quad x_i^S : S}{\lambda x_i^S.(f x_i^S) = f : S \implies T}$$
when x_i^S doesn't appear in f.

A whole bunch of turnstiles were omitted, which is because we want all the contexts here to implicitly be generic, they might all be some context Γ.

For example, we might change the function application rule to be

$$\frac{\Gamma \vdash f : S \implies T, \quad \Gamma \vdash s : S}{\Gamma \vdash fs : T}.$$

2.2.1 Free and bound variables

In computer science, there is a big distinction between free and bound variables.

Consider the statement

$$x : \mathbb{N} \vdash x^2 : \mathbb{N}.$$

Here x is *free*. We can then use our λ-abstraction rule, to form the function

$$\vdash \lambda x.x^2 : \mathbb{N} \implies \mathbb{N}.$$

In this sentence, x is *bound*. We say λ is a *binder* and that it *binds* x.

Another example of a binder, in $\mathbb{R}$ is

$$\int_0^1 x^2\, dx.$$

The integral sign, $\int_a^b dx$ binds the x in the formula x^2.

Recall the rule

$$\frac{\Gamma, x_1^S : S \vdash t : T}{\Gamma \vdash \lambda x_1^S.t : S \implies T}.$$

Example 2.7. We can derive a term of

$$S \implies (S \implies T) \implies T.$$

Intuitively, we can define $\epsilon(s)(f) : T$. This is the evaluation map, $\epsilon(s)(f) = f(s)$.

How do we get this? We apply our rules. We want to derive

$$s : S,\ f : S \implies T \vdash fs : T$$

so that we can apply λ-abstraction.

We have to start with our function application rule

$$\frac{s : S,\ f : S \implies T}{fs : T}.$$

We know

$$s : S,\ f : S \implies T \vdash s : S$$

and

$$s : S,\ f : S \implies T \vdash f : S \implies T,$$

so by our function application rule, we conclude

$$s : S,\ f : S \implies T \vdash fs : T.$$

Then by the lambda abstraction rule, we have

$$s : S \vdash \lambda f.fs : (S \implies T) \implies T,$$

and abstracting a second time, we have

$$\lambda s.\lambda f.fs : S \implies (S \implies T) \implies T,$$

as desired.

Question: Is the idea of a context, roughly the following?

In ordinary logic, we have formulas which have free variables in them. However, in type theory, all of our terms have types, so our free variables also need to have types. A context is a way of recording the free variables in your formulas and tracking their types.

Answer: Yes, that's one way of thinking about contexts.

Question: Are these rules reversible? Answer, no.

Question: Would we formulate ring theory in as judgements? Answer: No, instead the axioms of ring theory will be terms that live in certains types.

Question: What about associativity? Is that a judgment? Answer: No, rather $a + (b + c) = (a + b) + c$ is a type, and a proof of associativity is a term in this type.

3 Lecture 3

Correction:

Last time we wrote the rule

$$\frac{T \text{ TYPE}}{x_i^T : T},$$

but this should really be

$$\frac{T \text{ TYPE}}{x_i^T : T \vdash x_i^T : T},$$

3.1 And Types

Definition 3.1. The rules for $\wedge$-types are the following

1. $\wedge$-type formation:
$$\frac{S \text{ TYPE}, \quad T \text{ TYPE}}{S \wedge T \text{ TYPE}},$$
2. term construction
$$\frac{\Gamma \vdash s : S, \quad \Gamma \vdash t : T}{\Gamma \vdash (s,t) : S \wedge T},$$
3. and term deconstruction
$$\frac{\Gamma \vdash c : S \wedge T}{\Gamma \vdash \pi_1(c) : S, \quad \Gamma \vdash \pi_2(c) : T}$$
4. compatibility
$$\frac{\Gamma \vdash s : S, \quad \Gamma \vdash t : T}{\Gamma \vdash \pi_1(s,t) = s : S, \quad \Gamma \vdash \pi_2(s,t) = t : T}$$

Object		Programatic View		Logical View
Types	$\longleftrightarrow$	Program Spaces	$\longleftrightarrow$	Propositions
Terms	$\longleftrightarrow$	Programs	$\longleftrightarrow$	Proofs
Terms in context	$\longleftrightarrow$	Programs that take input	$\longleftrightarrow$	Proofs that take hypotheses

Table 1: Interpretations of type theory

Example 3.1. We get functions $\pi_1 : S \wedge T \implies S$ and $\pi_2 : S \wedge T \implies T$.

These are derived in the following way: We start with

$$c : S \wedge T \vdash c : S \wedge T,$$

by deconstruction this gives

$$c : S \wedge T \vdash \pi_1(c) : S,$$

and then by lambda abstraction, we get

$$\vdash \lambda c.\pi_1(c) : (S \wedge T) \implies S.$$

Whenever we get a function from

$$\frac{\Gamma, x : S \vdash y : T}{\Gamma \vdash \lambda x.y : S \implies T},$$

we call this *Currying*, and say we *curry* the x.

Example 3.2. We can derive a term of

$$(S \implies T) \wedge S \implies T.$$

We start with the following things we know:

$$c : (S \implies T) \wedge S \vdash \pi_1(c) : S \implies T,$$

and

$$c : (S \implies T) \wedge S \vdash \pi_2(c) : S.$$

Then function application gives

$$c : (S \implies T) \wedge S \vdash \pi_1(c)\ \pi_2(c) : T,$$

so finally, by lambda abstraction, we have

$$\vdash \lambda c.(\pi_1(c)\ \pi_2(c)) : (S \implies T) \wedge S \implies T$$

3.2 Philosophy

There is a logical interpretation of type theory.

For example, we might have the type 12 is composition, a proof might be $12 = 3 \cdot 4$. Another proposition/type would be T has a term, a proof might require hypotheses/context, and we could use the context to produce a term of T.

Example 3.3. $f : S \implies T$ is a function that turns a proof of S into a proof of T.

For example, if $S =$ "12 is composite.", and $T =$ "24 is composite.".

Example 3.4. A term of the type $c : S \wedge T$ represents a proof of S and T, since from c we can obtain proofs $\pi_1(c) : S$ and $\pi_2(c) : T$.

As a silly example, we could say $S =$ "12 is composite", and $T =$ "4 is composite".

In the logical interpretation, the term we just constructed of type $(S \implies T) \wedge S \implies T$, tells us that if we know $P \implies Q$, and P, then Q is true. This rule is called *modus ponens* in propositional logic.

This logic/program correspondence is called the Curry-Howard correspondence, and dates from 58-69. It is also called the Proofs-as-programs paradigm.

Usually in mathematics, proofs are often thought of as metamathematical. In type theory, proofs are actual mathematical objects. Thus we say type theory is *proof relevant.* It is often important that there is more than one proof.

Note that in type theory, proofs are always constructive. This is because we are always constructing a term in a type. This is constructive mathematics, and we don't use the law of excluded middle.

3.3 Disjunction: $\vee$-types

Definition 3.2. Rules for $\vee$-types.

1. type construction:
$$\frac{S \text{ TYPE}, \quad T \text{ TYPE}}{S \vee T \text{ TYPE}}$$
2. term construction:
$$\frac{\Gamma \vdash s : S}{\Gamma \vdash i_1(s) : S \vee T}$$
$$\frac{\Gamma \vdash t : T}{\Gamma \vdash i_2(t) : S \vee T}$$
3. term deconstruction:
$$\frac{\Gamma, x : S \vdash r : R, \quad \Gamma, y : T \vdash r' : R}{\Gamma, z : S \vee T \vdash j_{r,r'}(z) : R}$$
4. compatibility
$$\Gamma, x : S \vdash j_{r,r'}(i_1(x)) = r : R$$
$$\Gamma, y : S \vdash j_{r,r'}(i_2(y)) = r' : R$$

To construct a term of $S \vee T$, we can either supply an $s : S$, or a $t : T$. In the logical interpretation, we can say that to prove $S \vee T$, it suffices to prove S or T.

To construct a term of R from a term of $S \vee T$, we can supply functions $f : S \implies R$, and $g : T \implies R$. In the logical interpretation, this says that if we want to prove R using $S \vee T$, we can prove $S \implies R$ and $S \implies T$.

Example 3.5. From these rules, we get some functions:

1. $i_1 = \lambda s.i_1(s) : S \implies S \vee T$,
2. $i_2 = \lambda t.i_2(t) : T \implies S \vee T$,

Now we want to prove

$$R \wedge (S \vee T) \implies (R \wedge S) \vee (R \wedge T).$$

We begin with

$$\frac{x : R, y : S \vdash i_1(x, y) : (R \wedge S) \vee (R \wedge T), \quad x : R, z : T \vdash i_2(x, z) : (R \wedge S) \vee (R \wedge T)}{x : R, w : S \vee T \vdash j_{i_1,i_2}(x, w) : (R \wedge S) \vee (R \wedge T)}$$

Then we can conclude

$$v : R \wedge (S \vee T) \vdash j_{i_1,i_2}(\pi_1 v, \pi_2 v) : (R \wedge S) \vee (R \wedge T).$$

Finally, we lambda abstract to get

$$\vdash \lambda v.j_{i_1,i_2}(\pi_1 v, \pi_2 v) : R \wedge (S \vee T) \implies (R \wedge S) \vee (R \wedge T).$$

Question: In what sense is the string unambiguous? Can we work out things from the type? Answer: We're abbreviating slightly, but we can work out the types of the important things from the type of the whole expression.

Note from note taker, we're eliding a rather important piece from this proof, where we construct a function $i_r : S \implies R \wedge S$.

3.4 Bottom and Top

Definition 3.3. We have the rules for true or top: $\top$, and false or bottom: $\bot$

1. $$\frac{}{\top \text{ TYPE}}, \quad \frac{}{* : \top}$$
2. $$\frac{}{\bot \text{ TYPE}}, \quad \frac{\Gamma \vdash f : \bot, \quad S \text{ TYPE}}{\Gamma \vdash j_f : S}$$

Exercise. Prove $S \implies \top$, $\bot \implies S$.

Definition 3.4. Define $\neg S$ as $S \implies \bot$. We cannot construct a term of

$$S \vee \neg S$$

for every S.

This corresponds to unprovable statements in logic. This fact is related to Gödels incompleteness theorem.

A term of $S \vee \neg S$ is called the *law of excluded middle.*

We can't prove the law of excluded middle in type theory, and we can't prove its negation, so the law of excluded middle is *independent* of the theory.

We say a statment is *independent* of axioms/a proof system if when you add that statement, you cannot prove false, and when you add the negation of that statement, you still cannot prove false. The existence of independent statements means that the theory is incomplete.

We *model* theory in different categories. Since a model is more concrete, we often have statements that are true in the model that are not true in the theory.

When we construct models, we are looking for more complete systems.

We can construct a model in the category of sets. Types are sets, terms are elements, implies is the set of functions, $\wedge = \times$, $\vee = \sqcup$, $\top = \{*\}$, $\bot = \emptyset$. Then in **Set**,

$$S \vee \neg S = S \sqcup (S \to \emptyset).$$

Can we construct a term of this type? (I.e., an element of this set.) If $S = \emptyset$, then there is an element of $S \to \emptyset$, so $S \sqcup (S \to \emptyset)$ is nonempty, and if S is nonempty, then $S \sqcup (S \to \emptyset)$ is nonempty. Thus in either case, the law of excluded middle holds for **Set**.

The law of excluded middle fails for presheaves on the arrow category. I.e., the category

$$[2, \mathbf{Set}].$$

The objects are triples (A, B, f), with A, B, sets and $f : A \to B$ a function. The arrows in this category are pairs $(g, h) : (A, B, e) \to (C, D, f)$, of functions $g : A \to C$, $h : B \to D$ such that

$$\begin{tikzcd} A \arrow[r, "e"] \arrow[d, "g"] & B \arrow[d, "h"] \\ C \arrow[r, "f"] & D \end{tikzcd}$$

commutes.

Then $\bot = (\emptyset, \emptyset, \mathrm{id})$, $\top = (*, *, \mathrm{id})$. Then we can show that for some objects S in this category, $S \vee \neg S$ is 'empty' generally. Terms of an object S here are elements of $\hom(\top, S)$.

4 Lecture 4

4.1 Set interpretation of type theory

We were talking about the set interpreation of type theory last time. See a summary in Table 2.

Type	Set
Term	Element
Dependent term, $x : T \vdash s(x) : S$	Function $T \to S$
$S \implies T$	Set of functions, $\hom(S, T)$
$S \wedge T$	Cartesian product, $S \times T$
$S \vee T$	Disjoint union, $S \sqcup T$
$\bot$	$\emptyset$
$\top$	$\{*\}$
$\neg A := A \implies \bot$	$\hom(A, \emptyset)$
$A \vee \neg A$	$A \sqcup (A \to \emptyset) = \begin{cases} \{*\} & \text{if } A = \emptyset \\ A & \text{otherwise} \end{cases}$

Table 2: The correspondence between type theoretical notions and their interpretations in set theory.

4.2 Natural Numbers

What should they be?

In any type definition, we introduced terms. For example, we had

$$\frac{}{\vdash * : T} \qquad \frac{\vdash a : A}{\vdash i_1(a) : A \vee B} \qquad \frac{x : S \vdash t : T}{\vdash \lambda x.t : S \implies T}$$

These elements are called canonical terms. What are the canonical terms of $\mathbb{N}$?

Definition 4.1. The natural numbers type will be defined by the following rules:

1. The natural numbers is a type:

$$\frac{}{\mathbb{N} \text{ TYPE}}$$

2. Term construction

$$\frac{}{0 : \mathbb{N}} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash sn : \mathbb{N}}.$$

3. We also need a way to access the terms in the type, which we will do by specifying how we can build functions out of the type. This is like with the $\vee$ type, where we had the rule

$$\frac{\Gamma, a : A \vdash y : Z \qquad \Gamma, b : B \vdash z : Z}{\Gamma, c : A \vee B \vdash j_{y,z}(c) : Z}$$

For the natural numbers, we have the rule (which we might call induction)

$$\frac{T \text{ TYPE} \qquad \Gamma \vdash z : T \qquad \Gamma, t : T \vdash \sigma t : T}{\Gamma, n : \mathbb{N} \vdash j_{z,\sigma}(n) : T}$$

4. We need this to satisfy the following equality rules:

$$\Gamma \vdash j_{z,\sigma}(0) = z : T,$$

and

$$\Gamma, n : \mathbb{N} \vdash j_{z,\sigma}(sn) = \sigma(j_{z,\sigma}(n)) : T.$$

We say $\mathbb{N}$ is *inductively* or *recursively* defined. $\mathbb{N}$ is defined to be the type whose terms are $0 : \mathbb{N}$, $sn : \mathbb{N}$ for every n.

Contexts were also recursively defined, as a list, with the rules:

$$\frac{}{\emptyset \text{ ctxt}} \qquad \frac{\Gamma \text{ ctxt} \quad T \text{ TYPE}}{\Gamma, x : T}$$

The natural numbers form a sort of tree, as do contexts.

$$\begin{tikzcd}
0 \arrow[d, no head] & \emptyset \arrow[d, no head] & \Gamma \arrow[dl, no head] \\
s0 \arrow[d, no head] & x : \Gamma \arrow[d, no head] & \Delta \arrow[dl, no head] \\
s^2 0 \arrow[d, no head] & x : \Gamma, y : \Delta \arrow[d, no head] & E \arrow[dl, no head] \\
s^3 0 & x : \Gamma, y : \Delta, z : E &
\end{tikzcd}$$

On the left, we have the construction of $3 : \mathbb{N}$, and on the right, we have the construction of the context $x : \Gamma, y : \Delta, z : E$.

The types in the simply typed lambda calculs are also recursively defined. We started with $T_1, \ldots, T_n$. For example, the tree for $(T_1 \implies T_2) \implies (T_3 \implies T_4)$ is

$$\begin{tikzcd}
T_1 \arrow[d, no head] & T_2 \arrow[dl, no head] & & T_3 \arrow[d, no head] & T_4 \arrow[dl, no head] \\
T_1 \implies T_2 \arrow[drr, no head] & & & T_3 \implies T_4 \arrow[dl, no head] & \\
& & (T_1 \implies T_2) \implies (T_3 \implies T_4) & &
\end{tikzcd}$$

4.2.1 Examples of using the function construction rule for the natural numbers

We'll construct a function

$$f : T \implies T, n : \mathbb{N} \vdash f^n : T \implies T,$$

where $f^n = f \circ f \circ f \circ \cdots \circ f$ n times.

The plan is to define $c(f, n)$ by the rules $c(f, 0) = \mathrm{id}_T$ and $c(f, sn) = f \circ c(f, n)$.

The value at zero is

$$f : T \implies T \vdash \lambda x.x : T \implies T,$$

and the inductive step is

$$f : T \implies T, g : T \implies T \vdash f \circ g : T \implies T.$$

These yield

$$f : T \implies T, n : \mathbb{N} \vdash j_{\lambda x.x, f\circ -}(n) : T \implies T.$$

Then we know that $j(f, 0) = \lambda x.x$, and $j(f, sn) = f \circ j(f, n)$.

Question. It seems like we could do the composition in the other order, and its not clear that they are equal.

Answer: Yes. We need a stronger type theory to prove that. We can prove it for a specific number, e.g., we can prove $f \circ (f \circ f) = (f \circ f) \circ f$.

Example 4.1. Define the function $\lambda x.s^2x : \mathbb{N} \to \mathbb{N}$ using induction rather than lambda abstraction. (This is the function $x \mapsto x + 2$)

For 0, we have

$$\vdash s(s(0)),$$

and for the inductive step, we have

$$n : \mathbb{N} \vdash sn : \mathbb{N},$$

so applying the inductive function formation rule, we have

$$n : \mathbb{N} \vdash j_{s^2 0, s}(n) : \mathbb{N}.$$

Checking, we have $\vdash j(0) = s^2 0 : \mathbb{N}$, and $n : \mathbb{N} \vdash j(sn) = s(jn) : \mathbb{N}$.

Question: It seems like we could prove that this function is actually equal to $\lambda x.s^2x$ with some sort of induction. Would that be a function from $\mathbb{N}$ to some equality type?

Answer: Yes, but we don't have an equality type yet. Hopefully we'll talk about the dependently typed lambda calculus tomorrow, and then we can introduce that type.

Example 4.2. Want to define $n : \mathbb{N}, m : \mathbb{N} \vdash \text{add}(n,m) : \mathbb{N}$. For zero we have

$$n : \mathbb{N} \vdash n : \mathbb{N},$$

and for the inductive step, we have

$$n : \mathbb{N}, x : \mathbb{N} \vdash sx : \mathbb{N}.$$

This yields

$$n : \mathbb{N}, m : \mathbb{N} \vdash j_{n,\lambda x.sx}(m) : \mathbb{N}.$$

Checking this, we have $n : \mathbb{N} \vdash j_n(0) = n$, and $n : \mathbb{N}, m : \mathbb{N} \vdash j_n(sm) = sj_n(m) : \mathbb{N}$.

Notice the asymmetry here. We inducted on m, and we could have inducted on n. Metatheoretically, we can see that these two ways of defining addition are the same, but hopefully next time, we can prove that they are the same inside the type theory.

Example 4.3. Now we can define multiplication! $n : \mathbb{N}, m : \mathbb{N} \vdash \text{mult}(n,m) : \mathbb{N}$.

Once again, we start with 0:

$$n : \mathbb{N} \vdash 0 : \mathbb{N},$$

and the inductive step:

$$n : \mathbb{N}, x : \mathbb{N} \vdash \text{add}(x,n).$$

Then by induction, we get the function

$$n : \mathbb{N}, m : \mathbb{N} \vdash \text{mult}(n,m) : \mathbb{N},$$

and we know that $\text{mult}(n,0) = 0$, and $\text{mult}(n,sm) = \text{add}(\text{mult}(n,m),n)$.

4.3 The list type

Lists are defined by the rules:

1. $$\frac{T \text{ TYPE}}{\text{LIST}(T) \text{ TYPE}}$$

2. Canonical elements:

$$\frac{}{\text{nil} : \text{LIST}(T)}, \quad \frac{\Gamma \vdash \ell : \text{LIST}(T),\ t : T}{\Gamma \vdash \text{con}(\ell,t) : \text{LIST}(T)}$$

3. Induction:

$$\frac{\Gamma \vdash s : S, \quad \Gamma, x : \text{LIST}(T), y : T \vdash c(x,y) : S}{\Gamma, \ell : \text{LIST}(T) \vdash j_{s,c}(\ell) : S}$$

4. Where induction satisfies the coherence rules:

$$j_{s,c}(\text{nil}) = s, \text{ and } j_{s,c}(\text{con}(x,y)) = c(x,y)$$

5 Lecture 5

5.1 Dependent Type Theory

The syntax rules were passed out in class. They come from a standard reference.

What is dependent type theory?

For example, we can consider the type List(T). We can consider a pair of lists and send them to a list of pairs

$$([1,2,3],[3,2,1]) \mapsto [(1,3),(2,2),(3,1)]$$

We could then add them together and get the list $[3,4,3]$, and then add these together to get 10. However, we can't do this, because we can't talk about the length of a list in the type.

We want a type List(n,T) for each $n : \mathbb{N}$ and type T such that the "union over n" is List(T).

We talked about a type/proposition "12 is composite," but this is awkward. We're usually more interested in the predicate, " ____ is composite," or perhaps with a variable, "n is composite."

Types should also be dependent on hypotheses/variables.

Example 5.1. I.e., we want to be able to write something like this,

$$\frac{T \text{ TYPE}}{n : \mathbb{N} \vdash \text{List}(n,T) \text{ TYPE}},$$

where the type depends on the parameter n. Similarly, we might write our is composite predicate as

$$n : \mathbb{N} \vdash \text{isComp}(n) \text{ TYPE}$$

Note. A family of sets $T(x)$ indexed by Γ produces a natural function $\bigsqcup_{x\in\Gamma} T(x) \to \Gamma$, which takes an element of the disjoint union and returns the index of the set it came from. Conversely, given a function $\pi : T \to \Gamma$, we get a family $\pi^{-1}(x)$ over Γ.

This gives a 'bijection'

$$\{\text{families of sets indexed by } \Gamma\} \longleftrightarrow \{\text{functions with codomain } \Gamma\}$$

(ignoring size issues).

Example 5.2. For the list type, we have the following:

$$\text{for } n : \mathbb{N}, \text{ List(n,T)} \longleftrightarrow \bigcup_{n:\mathbb{N}} \text{List}(n,T) \to \mathbb{N}$$

Type	Programming	Logic	Set
$x : \Gamma \vdash T(x)$ TYPE	A program that takes input x "(Given a $n : \mathbb{N}$ print all factors of n)"	A predicate, "n is prime"	A family of sets $T(x)$ indexed by Γ.

5.2 Definition of Dependent Type Theory

Per Martin-Löf (70s-80s). There are four judgements:

1. Is a type in a context:
$$\Gamma \vdash T \text{ TYPE}$$
2. Type equality in a context:
$$\Gamma \vdash S = T$$
3. Is a term of a type in a context:
$$\Gamma \vdash t : T$$
4. Term equality in a context:
$$\Gamma \vdash s = t : T$$

In the context formation rules that we had before, we had

$$\frac{}{\emptyset \text{ ctxt}}, \quad \frac{\Gamma \text{ ctxt} \qquad T \text{ TYPE}}{\Gamma, x : T \text{ ctxt}}.$$

Now T might depend on Γ, so we should change this to:

$$\frac{}{\emptyset \text{ ctxt}}, \quad \frac{\Gamma \text{ ctxt} \qquad \Gamma \vdash T \text{ TYPE}}{\Gamma, x : T \text{ ctxt}}.$$

5.2.1 The types

$\perp$ type

$$\frac{}{\perp \text{ TYPE}} \quad \perp\text{-formation}$$

Before we had

$$\frac{T \text{ TYPE} \quad \Gamma \vdash f : \perp}{\Gamma \vdash}$$

Now we need to allow it to depend on the context

$$\frac{\Gamma, x : \perp \vdash C(x) \text{ TYPE} \quad \Gamma \vdash f : \perp}{\Gamma \vdash j_{c,f} : C(f)} \quad \perp\text{-elimination}$$

$\top$ type

1. $\top$-formation
$$\frac{}{\top \text{ TYPE}}$$
2. $\top$-introduction
$$\frac{}{* : \top}$$
3. $\top$-elimination
$$\frac{\Gamma, x : \top \vdash C(x) \text{ TYPE} \qquad \Gamma \vdash c : C(*)}{\Gamma, x : \top \vdash j_{C,c}(x) : C(x)}$$
4. $\top$-computation
$$\frac{\Gamma, x : \top \vdash C(x) \text{ TYPE} \qquad \Gamma \vdash c : C(*)}{\Gamma, x : \top \vdash j_{C,c}(*) = c : C(*)}$$

In general, we have these four sorts of rules, describing forming the type, introducing terms of the type, and elimination and computation rules describing how to break down the type.

The elimination rule says "If you have a predicate C on terms of a type, then if you want to prove C always holds, then it suffices to prove it just for the canonical terms of your type."

Question. We notice that the rules do not say that all elements of $\top$ are equal to the canonical element.

Answer: This is true, and we don't want this. However, we will soon be able to prove that they are equal in a different sense.

We want to prove that two terms are equal. In type theory, prove means construct a term of a type, so we should be constructing a term of an appropriate type. Let's define that type now.

5.2.2 The type Id

1. Id-formation
$$\frac{\Gamma \vdash T \text{ TYPE}}{\Gamma, s : T, t : T \vdash \mathrm{Id}_T(s,t) \text{ TYPE}}$$
2. Id-introduction
$$\frac{\Gamma \vdash T \text{ TYPE}}{\Gamma, t : T \vdash \mathrm{refl}_t : \mathrm{Id}_T(t,t)}$$
3. Id-elimination
$$\frac{\Gamma, s : T, t : T, p : \mathrm{Id}_T(s,t) \vdash C(s,t,p) \text{ TYPE} \qquad \Gamma, t : T \vdash c(t) : C(t,t,\mathrm{refl}_t)}{\Gamma, s : T, t : T, p : \mathrm{Id}_T(s,t) \vdash j(s,t,p) : C(s,t,p)}$$
4. Id-computation
$$\frac{\Gamma, s : T, t : T, p : \mathrm{Id}_T(s,t) \vdash C(s,t,p) \text{ TYPE} \qquad \Gamma, t : T \vdash c(t) : C(t,t,\mathrm{refl}_t)}{\Gamma, t : T \vdash j(t,t,\mathrm{refl}_t) = c(t) : C(t,t,\mathrm{refl}_t)}$$

5.2.3 Relations in sets

We are trying to emulate relations in sets. The idea is that "Equality is the smallest reflexive relation."

Given some function/relation $R \xrightarrow{i} X \times X$, we say that it is reflexive if there exists $r : X \to R$ such that $ir = \Delta$, i.e, if $ir(x) = \Delta(x) := (x, x)$.

For example, we have

$$\mathbb{R} \to [\leq] \hookrightarrow \mathbb{R} \times \mathbb{R},$$

where $[\leq] = \{(x, y) \in \mathbb{R} \times \mathbb{R} : x \leq y\}$. Then the left hand map is $r \mapsto (r, r)$.

The map i produces a family of sets $i^{-1}(x, y)$ indexed by $X \times X$. If i is an injection, then these preimages are either empty or contain exactly one element.

We can define equality to be the smallest reflexive relation. I.e., given a set X, we have the relation,

$$X \xrightarrow{\mathrm{id}_X} X \xrightarrow{\Delta} X \times X,$$

$$x \mapsto x \mapsto (x, x)$$

Then

$$\Delta^{-1}(x, y) = \begin{cases} \emptyset & \text{if } x \neq y \\ \{x\} & \text{if } x = y. \end{cases}$$

Given a reflexive relation, we get a function

$$\Delta^{-1}(x, y) \to i^{-1}(x, y).$$

In other words, given a reflexive relation, $i : R \to X \times X$, we get a map $f : X \to R$ such that $if = \Delta$. Indeed, this is obvious, f is the map r from the definition of reflexivity.

5.2.4 Relations in type theory

A relation on T is a type

$$x : T, y : T \vdash R(x, y) \text{ TYPE}.$$

"Says that s and t are related if there is a term in $R(s, t)$."

We will say that R is reflexive if

$$t : T \vdash \rho_t : R(t, t).$$

Now Id is the smallest reflexive relation, in the sense that

$$x : T, y : T, p : \mathrm{Id}_T(x, y) \vdash ? : R(x, y)$$

for any reflexive relation R.

We begin with saying $R(x, y)$ is a type,

$$x : T, y : T, p : \mathrm{Id}_T(x, y) \vdash R(x, y) \text{ TYPE}.$$

Then we know by reflexivity

$$x : T \vdash \rho_x : R(x, x).$$

Thus we can conclude from the Id elimination rule

$$x : T, y : T, p : \mathrm{Id}_T(x, y) \vdash j_\rho(x, y, p) : R(x, y),$$

as desired.

5.2.5 Axiom K or identity reflection rule

We don't use this anymore, but it was used from the 70s-80s up until about 2000, so it's important to talk about it.

The rule says

$$\frac{\Gamma \vdash T \text{ TYPE} \qquad \Gamma \vdash s : T \qquad \Gamma \vdash t : T \qquad \Gamma \vdash p : \mathrm{Id}_T(s, t)}{\Gamma \vdash s = t : T \qquad \Gamma \vdash p = \mathrm{refl}_s : \mathrm{Id}_T(s, t)}$$

This rule is very bad, and it prevents us from doing anything useful in type theory.

We have two different notions of equality in type theory, sitting at two different levels. We have the internal equality, Id, and we have the equality judgement which always implies the Id equality.

People realized that this wasn't useful. It collapses the identity type to reflect the equality judgement. However, if we allow the identity type to have richer structure, by taking away this rule, we can capture a much broader segment of mathematics more naturally.

Without this rule, the type theory becomes naturally homotopical, whereas with this rule, we essentially have to build up homotopies from set theory and topological spaces. More on this in the future.

6 Lecture 6

6.1 Sum Types

If we have a dependent type,

$$x : S \vdash T(x) \text{ TYPE},$$

we want to take a union, which we write as a sum:

$$\sum_{x:S} T(x).$$

1. $\sum$-formation
$$\frac{\Gamma, x : S \vdash T(x) \text{ TYPE}}{\Gamma \vdash \sum_{x:S} T(x) \text{ TYPE}}$$

2. $\sum$-introduction
$$\frac{\Gamma \vdash s : S \qquad \Gamma \vdash t : T(s)}{\Gamma \vdash (s,t) : \sum_{x:S} T(x)}$$

3. $\sum$-elimination
$$\frac{\Gamma, z : \sum_{x:S} T(x) \vdash C(z) \qquad \Gamma, s : S, t : T(s) \vdash c(s,t) : C(s,t)}{\Gamma, z : \sum_{x:S} T(x) \vdash j_c(z) : C(z)}$$

4. $\sum$-computation
$$\frac{\Gamma, z : \sum_{x:S} T(x) \vdash C(z) \qquad \Gamma, s : S, t : T(s) \vdash c(s,t) : C(s,t)}{\Gamma, s : S, t : T(s) \vdash j_c(s,t) = c(s,t) : C(s,t)}$$

Example 6.1. There are two functions
$$z : \sum_{x:S} T(s) \vdash \pi_1(z) : S,$$
and
$$z : \sum_{x:S} T(s) \vdash \pi_2(z) : T(\pi_1(z)).$$

We need to use the elimination rule. We start with
$$z : \sum_{x:S} T(x) \vdash S \text{ TYPE},$$
and
$$s : S, t : T(s) \vdash s : S,$$
so by the elimination rule, we can produce
$$z : \sum_{x:S} T(x) \vdash \pi_1(z) : S.$$

For π_2, we have
$$z : \sum_{x:S} T(x) \vdash T(\pi_1 z) \text{ TYPE},$$
and
$$s : S, t : T(s) \vdash t : T(\pi_1(s,t)) = T(s).$$
Thus by the elimination rule, we have
$$z : \sum_{x:S} T(x) \vdash \pi_2(z) : T(\pi_1 z).$$

Types	Sets	Logic
$\sum_{x:S} T(x)$	$\coprod_{x\in S} T(x)$	To prove $\sum_{x:S} T(x)$, it suffices to prove $T(s)$ for a specific $s : S$. In other words, sum types are like existential quantifiers. "There exists $x : S$ such that $T(x)$ holds." In usual math, proving something exists, doesn't mean you can actually find it. For example, if we have a continuous function $f : \mathbb{R} \to \mathbb{R}$ such that $\lim_{x\to\infty} f(x) > 0$ and $\lim_{x\to-\infty} f(x) < 0$. Then $T(x)$ is "$f(x) = 0$," and $S = \mathbb{R}$. Then $\exists_{x:\mathbb{R}} T(x)$.

We automatically get $\wedge$-types. Consider two types $\vdash S$ TYPE and $\vdash T$ TYPE, then we can derive

$$x : S \vdash T \text{ TYPE},$$

so we can conclude from $\sum$-formation,

$$\vdash \sum_{x:S} T \text{ TYPE}.$$

Then

$$\frac{\vdash s : S \qquad \vdash t : T}{\vdash (s,t) : \sum_{x:S} T},$$

is our $\wedge$-introduction rule. For wedge-elimination, we already have

$$\frac{\vdash z : \sum_{x:S} T}{\vdash \pi_1 z : S} \qquad \frac{\vdash z : \sum_{x:S} T}{\vdash \pi_2(z) : T}.$$

Finally, $\wedge$-computation follows from the $\sum$-computation rule.

The $\sum$-type of a dependent type, $x : S \vdash T$, where T depends on S only *trivially* is an $\wedge$-type.

6.2 Fibers

In set theory, we had

$$\{\text{families indexed by } B\} \longleftrightarrow \{\text{functions with codomain } B\}.$$

In type theory, we want a correspondence

$$\{\text{types dependent on } B\} \longleftrightarrow \{\text{Functions with codomain } B,\ x : A \vdash a(x) : B\}.$$

We don't have nearly enough to prove this yet. Suppose we have a dependent type, $b : B \vdash E(b)$. Then we have

$$z : \sum_{b:B} E(b) \vdash \pi_1(z) : B,$$

which is the thing we want on the right hand side.

Can we go the other way? In set theory, what happens when we have a function with codomain B and want to produce a family indexed by B? We have

$$E \xrightarrow{f} B,$$

and we get

$$f^{-1}(b) = \{e \in E | fe = b\}.$$

We can't do this in type theory. In set theory, we have logic and the set theory as two different layers. The set theory is built on top of the logic. On the other hand, in type theory, everything is a type, and there is only one layer. Thus we need to reformulate this in type theory.

Define a set

$$[fe = b] = \begin{cases} \{e\} & \text{if } fe = b \\ \emptyset & \text{if } fe \neq b. \end{cases}$$

Then we could define $f^{-1}(b) = \bigsqcup_{e \in E} [fe = b]$. This contains an e if and only if $fe = b$.

Now this is something that we can emulate in type theory. In the type theory, if we have $e : E \vdash f(e) : B$, then we define

$$f^{-1}(b) := \sum_{e:E} \mathrm{Id}_B(fe, b).$$

The canonical terms in this type are (e, p), where p is a witness to the equality of $f(e)$ and b.

Now, we can write

$$b : B \vdash \sum_{e:E} \mathrm{Id}_B(fe, b).$$

This should be inverse to the previous process, but we don't yet have enough to prove that.

6.3 Unions

The $\sum$-type also generalizes the $\vee$-type that we had before. The $\sum$-types are unions of types $T(x)$ indexed by S. To get a twofold union, we need S to have two elements.

The type with two terms was defined on the first homework, $\mathbb{B}$, with the following rules.

1. $\mathbb{B}$-formation

$$\frac{}{\mathbb{B} \ \textsc{type}}$$

2. $\mathbb{B}$-introduction

$$\frac{}{0 : \mathbb{B} \qquad 1 : \mathbb{B}}$$

3. $\mathbb{B}$-elimination

$$\frac{\Gamma, b : \mathbb{B} \vdash C(b) \text{ TYPE} \qquad \Gamma \vdash c_0 : C(0) \qquad \Gamma \vdash c_1 : C(1)}{\Gamma, b : \mathbb{B} \vdash j_{c_0,c_1}(b) : C(b)}$$

4. $\mathbb{B}$-computation

$$\frac{\Gamma, b : \mathbb{B} \vdash C(b) \text{ TYPE} \qquad \Gamma \vdash c_0 : C(0) \qquad \Gamma \vdash c_1 : C(1)}{\Gamma \vdash j_{c_0,c_1}(0) = c_0 : C(0) \qquad \Gamma \vdash j_{c_0,c_1}(1) = c_1 : C(1)}$$

In dependent type theory, we assume that every type is a term of a universe, which is a type. Actually, there is a hierarchy of universes $U_1, U_2, \ldots$. With each universe a term of the next, and all the terms of U_i a term of U_{i+1}. This is the first time we've said that a term is of more than one type, but this is something that can be addressed in several ways.

We'll ignore this for now, and just talk about a universe U.

Now we have

$$b : \mathbb{B} \vdash U \text{ TYPE},$$

and

$$\vdash S : U \qquad \vdash T : U,$$

so by $\mathbb{B}$-elimination, we have

$$b : \mathbb{B} \vdash j_{S,T}(b) : U.$$

Then by $\sum$-formation, we have

$$\vdash \sum_{b:\mathbb{B}} j_{S,T}(b)$$

Then given $\vdash s : S$, we have

$$\vdash (0, s) : \sum_{b:\mathbb{B}} j_{S,T}(b).$$

Similarly, given $\vdash t : T$, we can form

$$\vdash (1, t) : \sum_{b:\mathbb{B}} j_{S,T}(b).$$

These give us our $\vee$-introduction rules.

7 Lecture 7

7.1 Product Types

Generalize the function type.

7.1.1 In Sets:

For a one element set $1 = \{*\}$, and a set X we have

$$\{\text{elements of } X\} \longleftrightarrow \{\text{functions } 1 = \{*\} \to X\} \longleftrightarrow X \longleftrightarrow \prod_{x \in \{*\}} X.$$

For a two element set, 2, and a set X, we have

$$\{\text{ordered pairs of } X\} \longleftrightarrow \{\text{functions } 2 \to X\} \longleftrightarrow X \times X \longleftrightarrow \prod_{x \in 2} X.$$

Then

$$\{\text{sequences in } X\} \longleftrightarrow \{\text{functions } \mathbb{N} \to X\} \longleftrightarrow X^{\mathbb{N}} \longleftrightarrow \prod_{x \in \mathbb{N}} X.$$

One way to think of functions is as tuples. For functions $A \to X$, we can think of these as tuples in X whose entries are labeled elements of A.

If we have an indexed family $E(b)$ over B, then we can form

$$\prod_{b \in B} E(b),$$

the set of generalized tuples x, where $x_b \in E(b)$.

This is in bijection with subsections of the map

$$\pi : \coprod_{b \in B} E(b) \to B.$$

A *subsection* of the map π is a map

$$s : B \to \coprod_{b \in B} E(b)$$

such that $\pi s = 1_B$. In other words, $s(b) \in \pi^{-1}(b)$ for all b.

This is the concept that we want to generalize in type theory. If E is trivially indexed by B, then

$$\prod_{b \in B} E \simeq \{*\}\text{subsections of } \coprod_{b \in B} E \simeq B \times E \to B \simeq \hom(B, E).$$

This is the sense in which we are generalizing functions.

7.2 Π-types

We start with the rules for Π-types.

1. Π-formation $$\frac{\Gamma \vdash B : U \qquad \Gamma, x : B \vdash E(x) : U}{\Gamma \vdash \prod_{x:B} E(x) : U}$$

2. Π-introduction
$$\frac{\Gamma, x : B \vdash e(x) : E(x)}{\Gamma \vdash \lambda x.e(x) : \prod_{x:B} E(x)}$$

3. Π-elimination
$$\frac{\Gamma \vdash f : \prod_{x:B} E(x) \qquad \Gamma \vdash b : B}{\Gamma \vdash fb : E(b)}$$

4. Π-computation
$$\frac{\Gamma, x : B \vdash e(x) : E(x) \qquad \Gamma \vdash b : B}{\Gamma \vdash (\lambda x.e(x))b = e(b) : E(b)}$$

5. Π-uniqueness
$$\frac{\Gamma \vdash f : \prod_{x:B} E(x)}{\Gamma \vdash \lambda x.fx = f : \prod_{x:B} E(x)}$$

7.2.1 ∧-types

We can form ∧-types out of Π-types in the exact same way as we form ∨-types from Σ-types.

We have
$$\vdash S_0 : U \qquad \vdash S_1 : U,$$
so
$$b : \mathbb{B} \vdash S(b) : U.$$
Then
$$\vdash \prod_{b:\mathbb{B}} S(b) : U.$$

Then if $\vdash s_0 : S_0$, $\vdash s_1 : S_1$, $b : \mathbb{B} \vdash s(b) : S(b)$, and we have
$$\vdash \lambda b.s(b) : \prod_{b:\mathbb{B}} S(b).$$

Lastly, we get the π_1, π_2 maps as
$$\frac{\vdash p : \prod_{b:\mathbb{B}} S(b)}{\vdash p0 : S(0) \qquad \vdash p1 : S(1)}$$

We'll denote $S_0 \wedge S_1$ by $S_0 \times S_1$ from now on, as we're a bit more interested in the set interpretation than the logic interpretation.

7.2.2 $\implies$-types

Starting with types $\vdash S : U$ and $\vdash T : U$, we have $x : S \vdash T : U$, and we can form

$$\vdash \prod_{x:S} T : U.$$

Then we have

$$\frac{x : S \vdash f(x) : T}{\vdash \lambda x.f(x) : \prod_{x:S} T}.$$

From now on, we'll write this as $\to$, since, once again, the set interepretation is somehow closer to what we're doing now.

7.2.3 Logic interpretation

To prove $\prod_{x:S} T(x)$, we have to prove $T(x)$ for all $x : S$. Looks like $\forall_{x:S} T(x)$. "For all $x \in S$, $T(x)$ holds."

As with the Σ-type/$\exists$ correspondence, this product $\forall$ type is somehow stronger than the logic $\forall$.

Life hack. Read Π as $\forall$, and Σ as $\exists$. This will make formulas in type theory make much more sense to you.

7.3 Returning to Id-types.

They form an equivalence relation. A relation on T is a dependent type $s : T, t : T \vdash R(s,t) : U$. We can think of a relation as a function $R : T \times T \to U$.

In type theory, and functional programming, it is often better to instead think about $R : T \to (T \to U)$. These two types are equivalent in a way that we can't really talk about yet.

We can define the type of relations on T.

$$\mathrm{Rel}(T) := T \to T \to U.$$

Note that we are not writing parentheses. Functions types are assumed to associate to the left. Then

$$T : U \vdash \mathrm{Rel}(T) = T \to T \to U : U.$$

Given $a : T$ and $b : T$, we have

$$\mathrm{Id}_T(a,b) : U.$$

This gives us a dependent type

$$x : T, y : T \vdash \mathrm{Id}_T(x,y) : U.$$

Lambda abstracting gives us

$$x : T \vdash \lambda y.\mathrm{Id}_T(x,y) : \prod_{y:T} U,$$

and again, we have

$$\vdash \lambda x.\lambda y.\mathrm{Id}_T(x,y) : \prod_{x:T}\prod_{y:T} U.$$

As mentioned before, we'll write this last type as $T \to T \to U = \mathrm{Rel}(T)$. We'll call this term $\mathrm{Id}_T := \lambda x.\lambda y.\mathrm{Id}_T(x,y)$.

How do we show that something is reflexive? We want to define a type isRefl so that the type being inhabited corresponds to a proof of reflexivity:

$$T : U, R : \mathrm{Rel}(T) \vdash \mathrm{isRefl}(R) : U$$

$$s,t : T \vdash R(s,t) : U$$

$$t : T \vdash \mathrm{refl}_t : R(t,t) \vdash \lambda t.\mathrm{refl}_t : \prod_{t:T} \mathrm{Rel}(t,t).$$

This last type should be our predicate isRefl:

$$\mathrm{isRefl}(R) := \prod_{t:T} R(t,t).$$

Then to prove that Id is reflexive, we have

$$T : U, t : T \vdash \lambda t.\mathrm{refl}_t : \prod_{t:T} \mathrm{Id}(t,t),$$

and this type is precisely isRefl(Id).

Now for symmetry, once again, we want to define a predicate type isSym(R). Translating from logic, it should be

$$\mathrm{isSym}(R) := \prod_{s,t:T} R(s,t) \to R(t,s).$$

Now let's prove that Id_T is symmetric. we start with $s,t : T, p : \mathrm{Id}_T(s,t) \vdash \mathrm{Id}_T(t,s) : U$. Now $t : T \vdash \mathrm{refl}_t : \mathrm{Id}_T(t,t)$, so by the elimination rule for the identity type,

$$s,t : T, p : \mathrm{Id}_T(s,t) \vdash j_{\mathrm{refl}_t}(s,t,p) : \mathrm{Id}_T(t,s) : U.$$

Then by lambda abstracting, we get

$$\vdash \lambda s,t,p.j_{\mathrm{refl}_t}(s,t,p) : \prod_{s,t:T}\prod_{p:\mathrm{Id}_T(s,t)} \mathrm{Id}_T(t,s) = \mathrm{isSym}(R).$$

Finally, we just need to prove transitivity. This time the type should be

$$\prod_{r,s,t:T} R(r,s) \to R(s,t) \to R(r,t).$$

Now we prove that the identity type is transitive. We start with $r,s,t : T$, $p : \mathrm{Id}_T(r,s)$, $q : \mathrm{Id}_T(s,t)$, and we want to get $\mathrm{Id}_T(r,t)$.

It suffices to prove

$$t : T, r, s : T, p : \mathrm{Id}_T(r,s) \vdash ? : \mathrm{Id}_T(s,t) \to \mathrm{Id}_T(r,t) : U.$$

$$t : T \vdash \lambda x.x : \mathrm{Id}_T(r,t) \to \mathrm{Id}_T(r,t),$$

so if we have

$$t : T, r, s : T, p : \mathrm{Id}_T(r,s) \vdash j_{\lambda x.x}(t,r,s,p) : \mathrm{Id}_T(s,t) \to \mathrm{Id}_T(r,t)$$

then we can lambda abstract, getting

$$\lambda r, s, t, p. j_{\lambda x.x}(t,r,s,p) : \prod_{r,s,t} \mathrm{Id}_T(r,s) \to \mathrm{Id}_T(s,t) \to \mathrm{Id}_T(r,t) = \mathrm{isTrans}(\mathrm{Id}_T).$$

Lastly, we can define an equivalence relation in the type theory, we can define

$$T : U, R : \mathrm{Rel}(T) \vdash \mathrm{isEquivRel}(R) := \mathrm{isRefl}(R) \times \mathrm{isSym}(R) \times \mathrm{isTrans}(R) : U.$$

Then we have

$$T : U \vdash ((r,s),t) : \mathrm{isEquivRel}(\mathrm{Id}_T).$$

8 Lecture 8

8.1 Identity terms in Σ-types

Last time, we constructed

$$f : \prod_{s,s' : \sum_{x:B} E(x)} \left[\mathrm{Id}(s,s') \to \sum_{p : \mathrm{Id}(\pi_1 s, \pi_1 s')} \mathrm{Id}(p_* \pi_2 s, \pi_2 s') \right].$$

By Σ-elimination, we can assume that $s = (b,e), s' = (b',e')$ are canonical. By Id-elimination, it suffices to define the function on canonical terms of the identity type, $\mathrm{refl}_{(e,b)}$. Then constructing this function is equivalent to producing

$$\prod_{b:B, e:E(b)} \sum_{p:\mathrm{Id}(b,b)} \mathrm{Id}(p_* e, e).$$

Undoing the Π, we want to produce

$$b : B, e : E(b) \vdash \sum_{p:\mathrm{Id}(b,b)} \mathrm{Id}(p_* e, e).$$

I.e., we want to produce a pair (p,q), with $p : \mathrm{Id}(b,b)$, $q : \mathrm{Id}(p_* e, e)$. We can take $p = \mathrm{refl}_b$, then $p_* e = e$ by definition, so we can produce $q = \mathrm{refl}_e : \mathrm{Id}(e,e) = \mathrm{Id}(p_* e, e)$. Thus $(\mathrm{refl}_b, \mathrm{refl}_e)$ is in the sum type.

Assuming terms are canonical is often called induction, since in the natural numbers, defining a function out of $\mathbb{N}$ by specifying where the canonical terms $0 : \mathbb{N}$, $sn : \mathbb{N}$ go is induction.

Now we want to construct maps in the other direction:

$$g : \prod_{s,s':\sum_{x:B} E(x)} \left[\left(\sum_{p:\mathrm{Id}(\pi_1 s, \pi_1 s')} \mathrm{Id}(p_* \pi_2 s, \pi_2 s') \right) \to \mathrm{Id}(s, s') \right]$$

Want

$$s, s' : \sum_{x:B} E(x), t : \sum_{p:\mathrm{Id}(\pi_1 s, \pi_1 s')} \mathrm{Id}(p_* \pi_2 s, \pi_2 s') \vdash ? : \mathrm{Id}(s, s').$$

Induct on t, so assume we have (p, q), with $p : \mathrm{Id}(\pi_1 s, \pi_1 s')$, $q : \mathrm{Id}(p_* \pi_2 s, \pi_2 s')$. It suffices to construct

$$s, s' : \sum_{x:B} E(x), p : \mathrm{Id}(\pi_1 s, \pi_1 s'), q : \mathrm{Id}(p_* \pi_2 s, \pi_2 s') \vdash ? : \mathrm{Id}(s, s')$$

Then inducting on s, s', it suffices to prove

$$b, b' : B, e : E(b), e' : E(b'), p : \mathrm{Id}(b, b'), q : \mathrm{Id}(p_* e, e') \vdash ? : \mathrm{Id}((b, e), (b', e')).$$

Can't induct on q, since $p_* e$ is not free essentially. However, we can induct on p, so it suffices to prove

$$b : B, e, e' : E(b), q : \mathrm{Id}(e, e') \vdash ? : \mathrm{Id}((b, e), (b, e')).$$

Now we may induct on q. It now suffices to construct

$$b : B, e : E(b) \vdash ? : \mathrm{Id}((b, e), (b, e)).$$

However, we can now construct this term. It is $\mathrm{refl}_{(b,e)}$.

Question: If we were to write out the logic using the rules we've gotten, the flow of logic would go from bottom to top? Yes. But day to day, this is how we write proofs in type theory. But we could write out the term. If we wrote out the resulting term, we would get something like

$$\lambda s, s'.\lambda t.j_{j_{r_{b,e}}}(s, s', t)$$

To show that we have a quasiequivalence term in

$$\prod_{s,s':\sum_{x:B} E(x)} (g_{s,s'} \circ f_{s,s'} \sim 1)$$

we expand out the type, to get that we need to construct a term of type

$$\prod_{s,s':\sum_{x:B} E(x)} \prod_{u:\mathrm{Id}(s,s')} \mathrm{Id}(g(f(u)), u).$$

Induct on u. We need

$$\prod_{s:\sum_{x:B} E(x)} \mathrm{Id}(g(f(\mathrm{refl}_s)), \mathrm{refl}_s).$$

Recall that $f(\mathrm{refl}_s) = (\mathrm{refl}_{\pi_1 s}, \mathrm{refl}_{\pi_2 s})$, and $g(\mathrm{refl}_b, \mathrm{refl}_e) = \mathrm{refl}_s$. Thus $g(f(\mathrm{refl}_s)) = \mathrm{refl}_s$. Then

$$\lambda s.\mathrm{refl}_{\mathrm{refl}_s} : \prod_{s:\sum_{x:B} E(x)} \mathrm{Id}(\mathrm{refl}_s, \mathrm{refl}_s).$$

Now we want to go the other way, and construct

$$\prod_{t,t':\Sigma} (f_{t,t'} \circ g_{t,t'} \sim 1),$$

where Σ is the sum type in the domain of g. Unfold the homotopy ($\sim$) type

$$\prod_{t,t':\Sigma} \prod_{t:\Sigma} \mathrm{Id}(f(g(t)), t)$$

Induct on s, s', t to get that we need

$$\prod_{b,b':B} \prod_{e:E(b)} \prod_{e':E(b')} \prod_{p:\mathrm{Id}(b,b')} \prod_{q:\mathrm{Id}(p_* e, e')} \mathrm{Id}(f(g(p,q)), (p,q)).$$

Induct on p, so we now need

$$\prod_{b:B} \prod_{e,e':E(b)} \prod_{q:\mathrm{Id}(e,e')} \mathrm{Id}(f(g(\mathrm{refl}_b, q)), (\mathrm{refl}_b, q)).$$

Inducting on q, we now want

$$\prod_{b:B} \prod_{e:E(b)} \mathrm{Id}(f(g(\mathrm{refl}_b, \mathrm{refl}_e)), (\mathrm{refl}_b, \mathrm{refl}_e)).$$

As before $g(\mathrm{refl}_b, \mathrm{refl}_e) = \mathrm{refl}_{(b,e)}$, and $f(\mathrm{refl}_{(b,e)}) = (\mathrm{refl}_b, \mathrm{refl}_e)$. Thus this last type contains the term

$$\mathrm{refl}_{(\mathrm{refl}_b, \mathrm{refl}_e)}$$

We have now constructed a term in

$$\prod_{s,s':\sum_{x:B} E(x)} \mathrm{Id}(s, s') \simeq \prod_{p:\mathrm{Id}(\pi_1 s, \pi_1 s')} \mathrm{Id}(p_* \pi_2 s, \pi_2 s').$$

8.2 Identity terms in Π-types

Given $f, g : \prod_{x:B} E(x) \vdash \mathrm{Id}(f,g)$. We have both $\mathrm{Id}(f,g) : \mathcal{U}$ and $f \sim g : \mathcal{U}$. We'd like to say that these are equivalent types.

However, we can't prove this. There are models of type theory for which this is false. So we can't prove this. Thus we postulate this, since it's reasonable to assume. We assume that we have a term

$$\mathrm{FunExt} : \mathrm{Id}(f,g) \simeq (f \sim g).$$

This is an axiom.

9 Lecture 9

9.1 Identity types

Last time we talked about identity terms in Σ-types, and we gave an explicit description. Can we do this for other types? Unfortunately, in general this isn't possible for the other types. There are models of type theory, in which the things we'd want to be the same aren't the same. So we postulate this for our type theory, since we don't care about those models.

9.1.1 Π types

$$\mathrm{Id}_{\prod_{x:B} E(x)}(f, f') \simeq ?$$

We already talked about homotopy,

$$f \sim f' := \prod_{x:B} \mathrm{Id}(fx, f'x).$$

We have

$$\mathrm{idToHo} : \prod_{f, f' : \prod_{x:B} E(x)} \mathrm{Id}(f, f') \to (f \sim f').$$

How does this work? By induction, we need to produce

$$\prod_{f : \prod_{x:B} E(x)} (f \sim f) = \prod_{f} \prod_{x:B} \mathrm{Id}(fx, fx),$$

and we already have a term in this last type,

$$\lambda f, x.\mathrm{refl}_{fx}.$$

We would hope that there is a function in the other direction. I.e., we want a term of type

$$\mathrm{isEquiv}(\mathrm{idToHo}).$$

This doesn't exist in general, so we postulate the existence of such a term,

$$\mathrm{FunExt}_{f,f'} : \mathrm{isEquiv}(\mathrm{idToHo}_{f,f'}).$$

This is called function extensionality.

If you are interested in bad models, which don't satisfy this, you can check out: von Glehn. Dialectica Models of TT.

9.1.2 Id types

We could postulate the following:

$$\mathrm{UIP} : \mathrm{Id}_{\mathrm{Id}(s,t)}(p, q) \simeq \top$$

UIP stands for uniqueness of identity proofs. However, we definitely don't assume this one. It's not equivalent to assuming axiom K that we talked about before, but it's morally the same, so we don't want to assume this either.

9.1.3 U types

$$\mathrm{Id}_U(S,T) \simeq ?$$

Should ? be $S \simeq T$?

Just like with Π-types, we can produce a map

$$\mathrm{idToEquiv} : \prod_{S,T:U} \mathrm{Id}_U(S,t) \to (S \simeq T).$$

We can postulate

$$\mathrm{UA} : \mathrm{isEquiv}(\mathrm{idToEquiv}),$$

and this axiom is called the univalence axiom.

Univalence implies function extensionality. However, univalence and uniqueness of identity proofs are incompatible. If you assume both, you can prove false.

9.2 Programs

Mathematicians and theoretical CS people are interested in the following two programs.

9.2.1 Univalent foundations

1. It's a new foundation of mathematics.
2. It can be formalized/computer-checked.
3. Dependent type theory with Σ, Π, Id, $\top$, $\bot$, $\mathbb{B}$, $\mathbb{N}$, UA, propositional truncation, and propositional resizing. (These last two we haven't seen, and we won't talk about the last, since it's not clear that it's consistent).
4. Invented by Vladimir Voevodsky. He won a Fields medal for work in motivic homotopy theory. His proofs were found to have a lot of holes, though he managed to fix the ones found in his Fields medal work.
5. UniMath (GitHub)

9.2.2 Homotopy Type Theory

1. Basically the same, but not purporting to be a new foundation for mathematics.
2. DTT with Σ, Π, Id, certain (higher) inductive types, and univalence.
3. Emphasizes the connections between the type theory and classical homotopy theory.

9.3 h-levels

h stands for homotopy. Consider a type T. It has the terms $r, s, t : T$. Think of T as a space, with r, s, t points in the space. It has the paths $p, q : \mathrm{Id}(s, t)$. I.e., terms of the identity type represent paths between the points in the space. It also has paths $\alpha, \beta : \mathrm{Id}(p, q)$. We can think of these as deformations of paths in the space. I.e., these are homotopies of paths.

Write $\mathrm{Id}(s, t)$ as $s = t$, and $s = t$ as $s \equiv t$ now.

Stratify types into their homotopy levels.

Level 0: Types $\simeq \top$.

Level 1: Types $\simeq \bot$ or $\top$.

Level 2: Types that look like sets. (Types with terms)

Level 3: Types that look like graphs. (groupoids) (Types with terms and paths, but no paths between paths)

Level 4: 2-groupoids.

Now let's be rigorous.

9.3.1 h-level 0 (contractible)

Some people say that this is level -2 to make the numbering work out with the way we number groupoids.

$$T : U \vdash \mathrm{isContr}(T) : U,$$

where

$$\mathrm{isContr}(T) := \sum_{t:T} \prod_{s:T} t = s.$$

Proposition 9.1. $\top$ *is contractible.*

Proof. Take $* : \top$, then we want $? : \prod_{x:\top} * = x$. By induction, we can assume $x \equiv *$ by the elimination rule for $\top$. Then we have $\mathrm{refl}_* : * = *$. This gives us $\jmath_{\mathrm{refl}_*} : \prod_{x:\top} * = x$, as desired. □

Thus up to homotopy, $\top$ is a one point space.

Proposition 9.2. *We have a map* $\mathrm{isContr}(S) \to (S \simeq \top)$ *(actually this is an equivalence).*

Proof. Consider $(t, p) : \mathrm{isContr}(S)$, where $t : S$, and $p : \prod_{x:S} t = x$. There is a function $g : S \to \top$, given by $\lambda s.*$. There is a function $f : \top \to S$ by $* \mapsto t$.

We have $g \circ f \sim \mathrm{id}_\top := \prod_{x:\top} gfx = x$, which by induction, we can prove by proving on the canonical term, $*$. On the canonical term, we have $gf* \equiv gt \equiv *$, and so we have $\mathrm{refl}_* : gf* = *$.

Now we need to prove $f \circ g \sim \mathrm{id}_S := \prod_{x:S} fgx = x$. However, $gx \equiv *$, and $f* \equiv t$, so $fgx \equiv t$. Then p is already of this type. I.e., by definition, we have $p : \prod_{x:S} fgx = x$.

This completes the proof. □

HW: Consider a dependent type, $b : B \vdash E(b) : U$ such that $b : B \vdash c(b) : \mathrm{isContr}(E(b))$, then

$$\sum_{b:B} E(b) \simeq B.$$

This generalizes a problem on HW3, where we prove

$$\sum_{b:B} \top \simeq B.$$

Proposition 9.3. *Given $f : A \to B$,*

$$\mathrm{isEquiv}(f) \simeq \prod_{b:B} \mathrm{isContr}\left(\sum_{a:A} fa = b\right)$$

I.e., f being an equivalence is equivalent to all the fibers of f being contractible.

Proposition 9.4. *For any type T, and $t : T$, the type*

$$\sum_{u:T} t = u$$

is contractible.

Note that this is an English translation of the type theory statement:

$$T : U, t : T \vdash ? : \mathrm{isContr}\left(\sum_{u:T} t = u\right)$$

Topologically (roughly), this is the statement that the universal cover of a path connected space is simply connected. Note that we have a point in this space for every path out of t.

Proof. Center of construction, (t, refl_t).

$$\prod_{s:\sum_{u:T} t=u} ((t, \mathrm{refl}_t) = s) \simeq \sum_{p:t=\pi_1 s} (p_* \mathrm{refl}_t = \pi_2 s)$$

HW: prove that we have $\mathrm{refl}_{\mathrm{refl}_t} : \mathrm{refl}_{t,*}\mathrm{refl}_t = \mathrm{refl}_t$ by showing $\mathrm{refl}_{t,*}\mathrm{refl}_t \equiv \mathrm{refl}_t$. □

10 Lecture 10

Last time we talked about contractibility (level 0), and we defined

$$\mathrm{isContr}(T) := \sum_{t:T} \prod_{s:T} s = t.$$

10.1 Propositions (h level 1)

This time we will talk about propositions.

Definition 10.1.
$$T : U \vdash \mathrm{isProp}(T) : U,$$
where
$$\mathrm{isProp}(T) := \prod_{s,t:T} \mathrm{isContr}(s = t).$$

This roughly means that T is a proposition if and only if $T \simeq \bot$ or $T \simeq \top$. (This isn't strictly true. We would need law of excluded middle to prove it.)

Proposition 10.1. *If $T \simeq \bot$, then T is a proposition. If $t : T$ and T is a proposition, then $T \simeq \top$.*

Proof. We begin by showing $\bot$ is a proposition.
$$\mathrm{isProp}(\bot) = \prod_{x,y:\bot} \mathrm{isContr}(x = y).$$
Then by $\bot$-elimination, there is always a dependent function out of bottom into any type. Thus $\mathrm{isProp}(\bot)$ is true.

Suppose we have $t : T, p : \mathrm{isProp}(T)$. We want to construct a pair of a term $t : T$, and an element of $\prod_{s:T} s = t$. We already have a $t : T$, and we have
$$p : \mathrm{isProp}(T) := \prod_{s,t:T} \mathrm{isContr}(s = t).$$
Then
$$pt : \prod_{s:T} \mathrm{isContr}(s = t).$$
Then
$$\lambda s.\pi_1(pts) : \prod_{s:T} s = t$$
is our desired term. □

Why do we consider propositions?

Consider a dependent type, $b : B \vdash E(b)$. For example, $n : \mathbb{N} \vdash \mathrm{isEven}(n) : U$. Then we might want to define the 'subtype' of B consisting of all $b : B$ such that $E(b)$ is inhabited. In set theory, we can just form
$$\{b \in B \mid E(b)\}.$$
In type theory on the other hand, the closest thing we can produce is the Σ-type:
$$\sum_{b:B} E(b).$$

Then we might define the type of even natural numbers to be

$$E := \sum_{n:\mathbb{N}} \mathrm{isEven}(n).$$

Now we might run into a problem if our predicate is very complicated, for example, we might have many proofs of $E(b)$, then

$$\sum_{b:B} E(b)$$

might be very complicated, and not behave like a subtype.

However if each of $E(b)$ is a proposition, then it will behave like a subtype should behave. For example,

$$n : \mathbb{N} \vdash \mathrm{isEven}(n) := \sum_{m:\mathbb{N}} 2m = n.$$

is a proposition.

Proposition 10.2. *Consider $b : B \vdash E(b)$, together with*

$$b : B \vdash p(b) : \mathrm{isProp}(E(b)).$$

If we have

$$s, t : \sum_{b:B} E(b)$$

such that $\pi_1 s = \pi_1 t$, then $s = t$.

Proof. Consider $q : \pi_1 s = \pi_1 t$. We know

$$s = t \simeq \sum_{a:\pi_1 s = \pi_1 t} a_* \pi_2 s = \pi_2 t$$

We already have $q : \pi_1 s = \pi_1 t$. We now just need to find something of type $q_* \pi_2 s = \pi_2 t$. This is an identity type between terms of $E(\pi_1 t)$. However, we know that $E(\pi_1 t)$ is a proposition, so $q_* \pi_2 s = \pi_2 t$ is contractible. It is therefore inhabited by some term c. Then (q, c) is a term in our sum type. $\square$

Side note, once again, recall that all English statements are just translations of type theoretical statements. The proposition above is the following statement

$$B : U, E : B \to U, p : \prod_{b:B} \mathrm{isProp}(E(b)), s, t : \sum_{b:B} E(b), q : \pi_1 s = \pi_1 t \vdash ? : s = t : U.$$

Proposition 10.3. isEquiv(f)*,* isContr(T)*,* isProp(T) *are always propositions.* isQEquiv *is not a proposition generally. (This is why we say that equivalences are better behaved than quasiequivalences.)*

Example 10.1.

$$\sum_{f:A\to B} \mathrm{isEquiv}(f) \quad \text{“}\subset\text{”} \quad A \to B$$

10.2 Sets (h level 2)

Definition 10.2.

$$T : U \vdash \mathrm{isSet}(T) : U,$$

where

$$\mathrm{isSet}(T) := \prod_{s,t:T} \mathrm{isProp}(s = t)$$

Picture: We think of T as a space, with terms r, s, t being points of the space. Then we might have $p, q : s \to t$ paths. The space of paths is either empty or contractible. So if p and q are both paths, then we have a term of the identity type $p = q$.

Example 10.2 (Groups)**.**

$$\mathbb{G}\mathrm{roup} := \sum_{G:U} \sum_{p:\mathrm{isSet}(G)} \sum_{m:G\times G\to G} \sum_{e:G} \sum_{i:G\to G}$$

$$\left(\prod_{a,b,c:G} m(a, m(b, c)) = m(m(a, b), c) \right)$$

$$\times \left(\prod_{g:G} m(g, e) = g \times m(e, g) = g \right)$$

$$\times \left(\prod_{g:G} m(g, ig) = e \times m(ig, g) = e \right)$$

Now we need G to be a set. All identity types are propositions. This tells us that the equalities we impose are forming a nice subtype.

If G were not a set, then we might need equalities between our equalities and equalities between all the equality equalities and so on. We'd need infinitely many equalities, which would be bad.

Proposition 10.4. *$\bot$, $\top$, $\mathbb{B}$, $\mathbb{N}$ are sets.*

10.3 h-levels

Definition 10.3. $T : U, n : \mathbb{N} \vdash \mathrm{isnType}(n, T) : U$, where

$$\mathrm{isnType}(n, T) := \begin{cases} \mathrm{isnType}(0, T) := \mathrm{isContr}(T) \\ \mathrm{isnType}(sn, T) := \prod_{s,t:T} \mathrm{isnType}(n, s = t) \end{cases}$$

10.3.1 Topology

$\pi_0(X)$ is the set of path-connected components. If $X = \mathbb{T}^2 \sqcup \mathbb{T}^2$, then $\pi_0(\mathbb{T}^2 \sqcup \mathbb{T}^2) = 2$, and $\pi_1(\mathbb{T}^2) = \mathbb{Z} \times \mathbb{Z}$. We also have $\pi_2(X)$, $\pi_3(X)$, and so on.

Being contractible is $\pi_n(X) = *$. Being proposition is $\pi_0(X) \in \{0, 1\}$ and $\pi_n(X) = *$ for $n > 0$. Being a set means π_0 can be anything, and $\pi_n(X) = *$ for $n > 0$. Being a 3-type means $\pi_0(X), \pi_1(X)$ can be anything $\pi_n(X) = *$ for $n > 1$.

This gives us a stratification of types and a way to measure their complexity.

Proposition 10.5. *If T is an n-type, then T is an $n+1$-type.*

Proof. Induct on n. $n = 0$. Suppose T is a 0-type, so we have

$$c : \text{isContr}(T) := \sum_{t:T} \prod_{s:T} (t = s).$$

We need to show

$$\text{isProp}(T) := \prod_{s,s':T} \text{isContr}(s = s').$$

Fix s, s'. Since we have $c = (t, p)$, we get $p(s) : t = s$ and $p(s') : t = s'$. Then we have $p(s)^{-1} \cdot p(s') : s = s'$. Now we have

$$\prod_{s,s':T} \prod_{q:s=s'} p(s)^{-1} \cdot p(s') = q.$$

Then we induct on q. Thus we just need to show

$$\prod_{s:T} p(s)^{-1} \cdot p(s) = \text{refl}_s.$$

Let's quickly prove $p^{-1} \cdot p = \text{refl}_s$ for all paths p. Induct on p, then we want to show

$$\text{refl}_s^{-1} \cdot \text{refl}_s = \text{refl}_s,$$

but this is true by definition of inverses and path composition for reflexivities.

So we have a term in

$$\prod_{s,s':T} \text{isContr}(s = s') =: \text{isProp}(T).$$

Now the inductive step. Want to show that

$$\text{isnType}(n, T) \to \text{isnType}(sn, T).$$

By definition,

$$\text{isnType}(sn, T) := \prod_{x,y:T} \text{isnType}(n, x = y).$$

Then by the induction, we have $\ell_{n,x=y} : \text{isnType}(n, x = y) \to \text{isnType}(sn, x = y)$. Then composing this with $p : \text{isnType}(n, T)$, we obtain a term of $\text{isnType}(sn, T)$. □

Corollary 10.1. $\top$ *is a proposition, set.* $\bot$ *is a set.*

Proposition 10.6. $\mathbb{B}$ *is a set.*

Proposition 10.7. *If we have* $\mathbb{B}$ *and the univalence axiom, then* U *is not a set. Indeed, it is not an* n*-type for any* n*.*