Chapter 1

Type theory

Before we go on to discuss the specific aspects of Homotopy Type Theory we will have a look at the basic notions of Type Theory. Keep in mind that Type Theory is a foundational language, i.e. an alternative to Zermelo-Fraenkel set theory and at the same time it can be viewed as a programming language not to dissimilar to modern functional programming languages like Haskell. In particular the notion of a type is very different from the notion of a set in set theory and resembles more the types of strongly typed programming languages. When we write $3 : \mathbb{N}$ then this is a judgement in Type Theory while the similar looking statement $3 \in \mathbb{N}$ in set theory is a proposition. As a consequence we cannot talk about objects in isolation of their type. While this may seem to be a restriction at the first glance it is precisely this aspect which makes the homotopy interpretation possible.

We attempt here to give an informal presentation of Type Theory, sufficient for the purposes of this book, more formal accounts can be found in [**??**].

For the purpose of this book we consider two judgements:

$a : A$ a is an object of type A.

$a \equiv_A b$ a and b are definitionally equal objects of type A.

Judgements may depend on assumptions of the form $x : A$ where x is a variable and A is a type. As an example we may construct an object $m+n : \mathbb{N}$ under the assumptions that $m, n : \mathbb{N}$. Another example is that we assume that A is a type, $x, y : A$ and $p : x =_A y$ and from this we construct $p^{-1} : y =_A x$. Here we use types to represent propositions whose inhabitants are proofs. We may omit the names of proofs and instead say that from $x =_A y$ we can infer $y =_A x$ but since we are doing proof-relevant mathematics we will frequently refer back to proofs as objects, e.g. in this case we may want to establish that p^{-1} together with the proofs of transitivity and reflexivity behave like a group or more precisely a groupoid.

We emphasize the difference between equality type $a =_A b$ and the judgment $a \equiv_A b$. The statement $a =_A b$ requires a proof, i.e. the construction of an

inhabitant $p : a =_A b$. In contrast $a \equiv b$ can be decided mechanically just by expanding definitions. So if we define $c \coloneqq 1$ then $c \equiv 1$. Another source of definitional equality is the application of a function. If we define a function $f : \mathbb{N} \to \mathbb{N}$ as $f(n) \coloneqq n + n$ then $f(3) \equiv 3 + 3$ (this corresponds to β-equality in λ-calculus). As for $a : A$ the judgement $a \equiv b$ cannot be used within a proposition. In particular $a \equiv b \to b \equiv a$ is not a proposition (i.e. a type) in type theory. In contrast if $a =_A b \to b =_A a$ is a type and hence a proposition.

Given a type A, if from assuming $x : A$ we can deduce $B[x]$ is a type. (here we write $B[x]$ to make it clear that B may contain the variable x then we say $B[x]$ is a family of types in $\mathcal{U}$ indexed by $x : A$. An example is the type $Vec(A, n)$ of vectors (n-tuples) of elements of type A which is family indexed over $n : \mathbb{N}$. Another example is the type $x =_A x$ which is a family indexed over $x : A$.

We now introduce the basic type formers of Type Theory.

1.1 Universes

We introduce a sequence of symbols $\mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2, \ldots$ which stand for type theoretic universes, i.e. types whose elements are types. In a way it would be nice if we just had one universe $\mathcal{U}$ of all types including $\mathcal{U} : \mathcal{U}$. However, we can encode Russell's paradox in Type Theory and such a universe makes the type theory inconsistent (every type is inhbited). Instead we introduce a hierarchy of universes

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \ldots$$

We assume that the hierarchy of universes is cummulative, i.e. if $A : \mathcal{U}_i$ then $A : \mathcal{U}_{i+1}$. Hence the difference between $\mathcal{U}_0$ and $\mathcal{U}_1$ is that $\mathcal{U}_1$ contains the type $\mathcal{U}_0 : \mathcal{U}_1$ types build from $\mathcal{U}_0$. Similarily $\mathcal{U}_2$ contains all the types from $\mathcal{U}_1$ and additionally $\mathcal{U}_1$ and types build using $\mathcal{U}_1$.

We say that our type theory is predicative if types in a universe $\mathcal{U}_i$ are introduced only with reference to types in the same or lower universes. Most (all ?) constructions introduced in this book are predicative.

When we say A is a type we mean that it is an element of some universe $\mathcal{U}_i$. To avoid having to clutter up the text with indices we use the metavariable $\mathcal{U}$ to stand for any universe. That is A is a type is the same as saying $A : \mathcal{U}$.

1.2 The dependent function types (Π-types)

Functions are a primitive concept in Type Theory, they are not reduced to relations as in set theory. The dependent function type is a generalisation of non-dependent function type $A \to B$ which allow the codomain to vary over the codomain over the domain. Thus using the proposition as types principle we can use function types to model not only implication but also universal quantification. Given a type A and a family $B[x]$ indexed over $x : A$ we can form the type $\Pi_{x:A} B[x]$ of dependent functions. Non dependent-functions arise

in the special case when B is a constant family (i.e. B does not dependent on x) in which case we write $A \to B$.

Given a dependent function $f : \Pi_{x:A}B[x]$ and $a : A$ we can apply f to a which we write as $f(a) : B[a]$. Such a function f may be introduced by a defining equation

$$f(x) \coloneqq b[x] \text{ for } x : A,$$

where $b[x]$ is a term for an object of type $B(x)$ for $x : A$ that may depend on the variable x ranging over objects of A. So if a is a term for an object of type A we may substitute a for x in the defining equation to give a definitional equality

$$f(a) \equiv b[a] : B[a]$$

and we have $f(a) : B[a]$. As usual we use the lambda abstraction notation $\lambda_{x:A}b[x]$ to name the function f so that if $a : A$ then

$$\lambda_{x:A}b[x](a) \equiv b[a] : B(a).$$

Assuming $a : A$ an example of a dependent function of type $\Pi_{n:\mathbb{N}}Vec(A, n)$ is the function which constructs an n-tuple of as. Another example of a dependent function is the proof of reflexivity which has the type $\Pi_{x:A}x = x$.

Dependent function types are used to represent:

- Conventional (non-dependent) functions as in $\mathbb{N} \to \mathbb{N}$,
- Implication as in $x =_A y \to y =_A x$,
- Universal quantification as in $\Pi_{n:\mathbb{N}}m + 0 =_{\mathbb{N}} m$.

1.3 The dependent pair types (Σ-types)

As before in the case of functions the dependent pair type is a generalisation of the ordinary cartesian product $A \times B$. As in the case of Π we assume that $B[x]$ is a family indexed by $x : A$ to form $\Sigma_{x:A}B[x]$. Elements of $\Sigma_{x:A}B[x]$ are pairs $(a, b) : \Sigma_{x:A}B[x]$ where $a : A$ and $b : B[a]$. If the family B does not depend on A we write $A \times B$.

If C is a family of types on $p : \Sigma_{x:A}B(x)$ and $c : \Pi(x : A)(y : B(x), C((x, y))$ then we may define $f : (\Pi z : \Sigma_{x:A}B(x))C(z)$ with defining equation

$$f((x, y)) \coloneqq c(x)(y) \text{ for } x : A, y : B(x).$$

As special cases we can derive the projections: to derive the 1st projection let $C_1(p) \coloneqq A$ be the constant family and $c_1 : \Pi(x : A)(y : B(x), A$ defined as $c_1(x)(y) \coloneqq x$ to derive $\pi_1 : \Sigma_{x:A}B(x)) \to A$ with the defining equation $\pi_1(x, y) \coloneqq x$. To derive the 2nd projection we use $C_2(p) \coloneqq B(\pi_1(p))$ and $c_2 : \Pi(x : A)(y : B(x)), B(\pi_1(x, y))$ note that the codomain $B(\pi_1(x, y))$ is definitionally equal to $B(x)$ and hence we can use $c_2(x)(y) \coloneqq y$ to construct $\pi_2 : \Pi(p : \Sigma_{x:A}B(x)), B(\pi_1 p)$ with the defining equation $\pi_2(x, y) \coloneqq y$.

As an example consider the type $\Sigma_{n:\mathbb{N}}Vec(A, n)$ of tuples of arbitrary length - this type is equivalent to the type of lists or finite sequences over A. Another

example is the type $\Sigma_{n:\mathbb{N}} n + n = n$ which expresses the (true) proposition that there exists a natural number which is equal to its doubling.

A more involved example is the type-theoretic axiom of choice. Assume there are types A, B and a family $R(x, y)$ indexed over $x : A$ and $y : B$. Then we can show that from assuming

$$p : \Pi(x : A)\Sigma(y : B), R(x, y)$$

we can define

$$\begin{array}{lcl} a(p) & : & \Sigma(f : A \to B)(\Pi x : A, R(x, (fx)) \\ a(p) & \coloneqq & (\lambda_{x:A}\pi_1(p(x)), \lambda_{x:A}\pi_2(p(x))) \end{array}$$

Dependent pair types are used to represent:

- Conventional (non-dependent) pairs as in $\mathbb{N} \times \mathbf{B}$,
- Conjunction as in $x =_A y \times y =_A z$,
- Existential quantification as in $\Sigma_{n:\mathbb{N}} n + n = n$.

1.4 Finite types

We use $\mathbf{0}$, $\mathbf{1}$ and $\mathbb{B}$ for the standard **empty type**, the standard **singleton type** and the standard **boolean type**, respectively. So $\mathbf{0}$ is not intended to have any elements, we have $\star : \mathbf{1}$ and $0_{\mathbf{B}}, 1_{\mathbf{B}} : \mathbf{B}$.

- If C is a family on $\mathbf{0}$ then we have $f : \Pi_{z:\mathbf{0}}(z)$ with no defining equation.
- If C is a family on $\mathbf{1}$ and $c : C(\star)$ then we have $f : \Pi_{z:\mathbf{1}}(z)$ with defining equation
$$f(\star) \coloneqq c.$$
- If C is a family on $\mathbf{B}$, $c_0 : C(0_{\mathbf{B}})$ and $c_1 : C(1_{\mathbf{B}})$ then we have $f : \Pi_{z:\mathbf{B}} C(z)$ with the defining equations
$$\begin{array}{lcl} f(0_{\mathbf{B}}) \coloneqq & c_0, & \text{and} \\ f(1_{\mathbf{B}}) \coloneqq & c_1. & \end{array}$$

If A and A' are types then $A + A'$ is their disjoint union. If $a : A$ then there is a copy $\mathsf{inleft}(a) : A + A'$ and if $a' : A'$ there is a copy $\mathsf{inright}(a') : A + A'$. If C is a family on $A + A'$, $c_{\mathsf{inleft}} : \Pi_{x:A} C(\mathsf{inleft}(x))$ and $c_{\mathsf{inright}} : \Pi_{x':A'} C(\mathsf{inright}(x'))$ then $f : \Pi_{z:A+A'} C(z)$ with defining equations

$$\left\{ \begin{array}{rcll} f(\mathsf{inleft}(x)) \coloneqq & c_{\mathsf{inleft}}(x) & \text{for } x : A & \text{and} \\ f(\mathsf{inright}(x')) \coloneqq & c_{\mathsf{inright}}(x') & \text{for } x' : A' & \end{array} \right.$$

We can define the disjoint union of two types $A, B : \mathcal{U}$ as

$$A + B \coloneqq \Sigma_{x:\mathbf{B}} F(x)$$

where $F : \mathbf{B} \to \mathcal{U}$ is defined as $F(0_{\mathbf{B}}) \coloneqq A$ and $F(1_{\mathbf{B}}) \coloneqq B$. The injections can be defined as:

$$\begin{array}{ll} \mathsf{inleft} : A \to A + B & \\ \mathsf{inleft}(a) \coloneqq (0_{\mathbf{B}}, a) & \mathsf{inright} : B \to A + B \\ \mathsf{inright}(b) \coloneqq (1_{\mathbf{B}}, b) & \end{array}$$

Todo: Derive eliminator.

Clearly, using $+$, 0 and 1 we can define all finite types.

Our technique to derive $+$ from Σ can also be used to derive $\times$ form Π namely given $A, B : \mathcal{U}$ we can define

$$A \times B \coloneqq \Pi_{x:\mathbf{B}} F(x)$$

where F is defined as for $+$ above. The projections can be derived by applications to $0_{\mathbf{B}}$ and $1_{\mathbf{B}}$. However, the general eliminator requires the principle of functional extensionality which we haven't introduced yet. Hence there are two ways to derive $A \times B$: either as a non-dependent Σ-type or as a special case of Π where the domain is boolean.

1.5 The identity types on a type

If $A : \mathcal{U}$ is a type and $a, a' : A$ then $a =_A a'$ is the identity type on A. In constructive intensional type theory objects in this type are intended to represent proofs of the proposition that a and a' are identical. So, in particular there is an object $\mathsf{refl}_a : a =_A a'$ whenever $a : A$. In Homotopy Type Theory, when A is understood as a space and a, a' are understood as points of the space the type $Id_A(a, a')$ is to be understood as the type of paths from the point a to the point a'.

We now give two rules for defining functions on paths, both of which are useful. Given the other rules of our type theory, when suitably understood, each rule can be derived from the other.

Martin-Löf Rule: Let $Path_A$ be the type $\Sigma_{x,x':A} Id_A(x, x')$, let C be a family on $Path_A$ and let $c : \Pi_{x:A} C(x, x, \mathsf{refl}_x)$. Then we have $f : \Pi_{u:Path_A} C(u)$ with defining equation

$$f(x, x, \mathsf{refl}_x) \coloneqq c(x) \text{ for } x : A.$$

Paulin-Mohring Rule: If $a : A$ let $Path_A(a)$ be the type $\Sigma_{y:A} Id_A(a, y)$, let C be a family on $Path_A(a)$ and let $c : C(a, \mathsf{refl}_a)$. Then we have $f : \Pi_{u:Path_A(a)} C(u)$ with defining equation

$$f(a, \mathsf{refl}_a) \coloneqq c.$$

We are going to use the Paulin-Mohring rule to show that $=$ is an equivalence relation. Assuming a type A we show that there is a function $p^{-1} : b = a$ given $p : a = b$ using the family $b = a$ and the defining equation

$$(\mathsf{refl}_a)^{-1} = \mathsf{refl}_a$$

Similarily we can show transitivity: assume there is $p : a = b$ and we want to derive $p \centerdot - : b = c \to a = b$ by

$$p \centerdot \mathsf{refl}_b \coloneqq p.$$

Since we already have the constructor refl we have shown that $=_A$ is an equivalence relation.

Todo: Derive transport and ap.

1.6 Inductive types

The paradigm example of an inductive type is the type $\mathbb{N}$ of unary natural numbers. These objects of type $\mathbb{N}$ are generated from the natural number 0 by repeatedly applying the successor operation. So we say that there are two **introduction rules** for $\mathbb{N}$.

$$0 : \mathbb{N}$$

and

$$\text{from } n : \mathbb{N} \text{ infer } \mathsf{succ}(n) : \mathbb{N}.$$

The inductive character of $\mathbb{N}$ is captured by the, so called, elimination and computation rules for $\mathbb{N}$. We prefer to call them the rules for defining a function on $\mathbb{N}$ by primitive recursion.

If C is a family on $\mathbb{N}$, $c_0 : C(0)$ and $c_{\mathsf{succ}}(x, y) : C(\mathsf{succ}(x))$ for $x : \mathbb{N}, y : C(x)$ then we may introduce a function $f : \Pi_{x:\mathbb{N}} C(x)$ by the following primitive recursion defining equations; one defining equation for each introduction rule.

$$\begin{cases} f(0) \coloneqq & c_0, \text{ and} \\ f(\mathsf{succ}(x)) \coloneqq & c_{\mathsf{succ}}(x, f(x)) \text{ for } x : \mathbb{N} \end{cases}$$

As an example we define the function $m + - : \mathbb{N} \to \mathbb{N}$ for any $m : \mathbb{N}$ using the constant family $\mathbb{N}$ and the defining equations:

$$\begin{aligned} m + 0 &\coloneqq m \\ m + \mathsf{succ}(n) &\coloneqq \mathsf{succ}(m + n) \end{aligned}$$

By abstracting m using the rules for Π-types we can derive the binary function $- + - : \mathbb{N} \to (\mathbb{N} \to nat)$.

We use the same principle to prove properties by induction. While it is trivial to see that $+$ is right neutral because $m + 0 \equiv m$ we need to prove that $0 + m =_{\mathbb{N}} m$ is inhabited. We do this by constructing a function $f : \Pi_{n:\mathbb{N}} 0 + n =_{\mathbb{N}}$

n. The family is $0 + n =_{\mathbb{N}} n$ over $n : \mathbb{N}$. In the case for 0 we have that $0 + 0 \equiv 0$ hence we can define

$$f(0) :\equiv \mathsf{refl}_0$$

For $\mathsf{succ}(n)$ we have that $0 + \mathsf{succ}(n) \equiv \mathsf{succ}(0 + n)$ hence we define

$$f(\mathsf{succ}(n)) :\equiv \mathsf{ap}_{\mathsf{succ}}(f(n)).$$

In this case we can show that the type is propositional, that is that all elements of the type are equal. In this situation we may omit any explicit reference to proof objects and the type-theoretic proof does not look different to a conventional one in predicate logic.