E/Ea Thompson

Homotopy Type Theory: A Construction of Mathematics

– In Pursuit of Abstract Nonsense –

Wednesday 30th August, 2023

Springer Nature

Preface

This text consists of a collection of Homotopy Type Theory notes taken over a summer reading group for HoTT.

Place(s),
month year

Firstname Surname
Firstname Surname

Contents

Part I

Foundations

Chapter 1

Type Theory

1.1 Type Theory versus Set Theory

Homotopy type theory, among other things, is a foundational language for mathematics. We compare and contrast with the set-theoretic Zermelo-Fraenkel construction. The set-theoretic foundation has two “layers": the deductive system of first-order logic, and, formulated inside this system, the axioms of a particular theory, such as ZFC. Thus, set theory is an interplay between sets (the objects of the second layer) and propositions (the objects of the first layer).

By constrast, type theory is its own deductive system: it need not be formulated inside any superstructure, such as first-order logic. Type theory has one basic notion: **types**. Propositions (i.e. statements which we can prove, disprove, assume, negate, and so on) are identified with particular types. The mathematical activity of proving atheorem is thus identified with a special case of the mathematical activity of constructing an object.

This leads to another difference. First, informally a deductive system is a collection of **rules** for deriving things called **judgments**. Thinking of a deductive system as a formal game, the judgments are “position" in the game which we read by following the game rules. We can also think of a deductive system as a kind of algebraic theory, in which case the judgments are the elements and the deductive rules are the operations. From a logical point of view, the judgements can be considered to be the external statements, living in the metatheory, as opposed to the internal statements of the theory itself.

For example, in first-order logic there is only one kind of judgment: that a given proposition has a proof. A rule of first-order logic is a rule of proof construction, which derives judgments from other judgments. The basic judgment of type theory, which in certain cases can be interpreted as “A has a proof", is written “$a : A$” and pronounced as “the term a has type A"m or in HoTT “a is a point of A". When A is a type representing a proposition, then a may be called a **witness** to the provability of A, or **evidence** of the truth of A. In this case the judgment $a : A$ is derivable in type theory for som a precisely when the analogous judgment “A has a proof" is derivable in first order logic.

Note, internally in type theory we cannot “prove" statements about judgments, nor can we “disprove" a judgment. Additionally, we cannot talk about an element “a" in isolation: every element by

its very nature is a term of some type, and that type is uniquely determined, generally speaking. That is, "$a : A$" is an **atomic statement** in type theory.

A last major difference between set and type theory is the treatment of equality. The classical notion of equality is a proposition (we can disprove or assume an equality). Since in type theory propositions are types, this means that equality is a type: for elements $a, b : A$, we have a type "$a =_A b$". When $a =_A b$ is inhabited (i.e. has terms), we say that a and b are **(propositionally) equal**.

However, we also need in type theory an equality judgment, existing at the same level as the judgment "$x : A$". This is called **judgmental equality** or **definitional equality**, and we write it as $a \equiv b : A$ or simply $a \equiv b : A$. This can be thought of as meaning "equal by definition". Whether or not two expressions are equal by definition is a matter of expanding out the definitions; in particular, it is algorithmically decidable (though the algorithm is meta-theoretic, not internal to the theory).

If we interpret a deductive system as an algebraic theory, judgmental equality is equality in that theory. A judgmental notion of equality allows us to control the other form of judgment, "$a : A$". Then we should have a rule saying that given the judgments $a : A$ and $A \equiv B$, we may derive the judgment $a : B$. For us type theory will be a deductive system based on two forms of judgment:

- $a : A$ - "a is an object of type A"
- $a \equiv b : A$ - "a and b are definitially equal objects of type A"

As we shall see later, for types A and B we use "$A \to B$" as notation for the type of functions from A to B. Then $f : A \to B$ can be interpreted as a typing judgment.

Judgments may depend on assumptions of the form $x : A$, where x is a variable and A is a type. For example, assuming A is a type, $x, y : A$, and $p : x =_A y$, we may construct an element $p^{-1} : y =_A x$. The collection of all such assumptions is called the **context**; from a topological point of view this is analogous to a parameter space. Technically the context must be an ordered list of assumptions since we can have later assumptions depend on previous ones.

If the type A in an assumption $x : A$ represents a proposition, then the assumption is a type theoretic version of a hypothesis. Under this meaning of the word assumption, we can assume a propositional equality, by assuming a variable $p : x = y$, but we cannot assume a judgmental equality $x \equiv y : A$, since it is not a type that can have an element. However, if we have a type or an element which involves a variable $x : A$, then we can substitute any particular element $a : A$ for x to obtain a more specific type or element. By the same token we cannot **prove** a judgmental equality either, since it is not a type in which we can exhibit a witness. Nonetheless, we will sometimes state judgmental equalities as part of a theorem, for example "there exists $f : A \to B$ such that $f(x) \equiv y$".

1.2 Function Types

Given types A and B, we can construct the type $A \to B$ of **functions** with domain A and codomain B. Unlike in set theory, functions in type theory are a primitive concept. We explain the function type by describing what we can do with functions, how to construct them, and what equalities they induce.

Given a function $f : A \to B$ and an element of the domain $a : A$, we can **apply** the function to obtain an element of the codomain B, denoted $f(a) : B$, and called the **value** of f at a. But how can we construct elements of type $A \to B$? There are two equivalent ways: either by direct definition or by using λ-abstraction. Introducing a function by definition means that we introduce a function by giving it a name, let's say f, and saying we define $f : A \to B$ by giving an equation

$$f(x) :\equiv \Phi$$

where x is a variable and Φ is an expression which may use x. In order for this to be valid, we have to check that $\Phi : B$ assuming $x : A$.

We can compute $f(a)$ by replacing the variable x in Φ with a. For example, consider $f : \mathbb{N} \to \mathbb{N}$, which is defined by $f(x) :\equiv x + x$. Then $f(2)$ is judgmentally equal to $2 + 2$.

If we don't want to introduce a name for the function, we can use λ-**abstraction**. Given an expression Φ of type B which may use $x : A$, as above, we write $\lambda(x : A).\Phi$ to indicate the same function defined previously as f. Thus we have

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

For example we have the typing judgment

$$(\lambda(x : \mathbb{N}).x + x) : \mathbb{N} \to \mathbb{N}$$

As another example, for any types A and B and any $y : B$, we have a constant function $(\lambda(x : A).y) : A \to B$. The type of the variable x in the λ-abstraction is often ommitted, and infered from context. The scope of the variable binding "λx." is the entire rest of the expression, unless delimited with parentheses. Another equivalent notation is

$$(x \mapsto \Phi) : A \to B$$

We may also write something like $g(x, -)$ for $\lambda y.g(x, y)$. For λ-abstraction we have the following **computation rule**, which is definitional equality:

$$(\lambda x.\Phi)(a) \equiv \Phi'$$

where Φ' is the expression Φ in which all occurrences of x have been replaced by a. For any function $f : A \to B$ we can also construct a lambda abstraction function $\lambda x.f(x)$. Since this is by definition the function that applies f to its argument, we consider it to be definitionally equal to f:

$$f \equiv (\lambda x.f(x))$$

This equality is the **uniqueness principle for function types**. We can then read a definition of $f : A \to B$ by $f(x) :\equiv \Phi$ as $f :\equiv \lambda x.\Phi$.

As a matter of analogy, a λ-abstraction binds a dummy variable in exactly the same way that an integral does. Now, to define a function in several variables we could use the cartesian product, to be introduced later. A function with aprameters A and B and results in C would be given the type $f : A \times B \to C$. However, we have another choice that avoids product types which is called **currying**. Currying involves representing a function of two inputs $a : A$ and $b : B$ as a function which takes one input, $a : A$, and returns another function which then takes a second input $b : B$ and returns the result. That is, we consider two-variable functions to belong to an iterated function type, $f : A \to (B \to C)$.

We can define a named function $f : A \to (B \to C)$ by giving an equation

$$f(x)(y) :\equiv \Phi$$

where $\Phi : C$ assuming $x : A$ and $y : B$. Using λ-abstraction this corresponds to

$$f :\equiv \lambda x.\lambda y.\Phi$$

which may also be written as

$$f :\equiv x \mapsto (y \mapsto \Phi)$$

1.3 Universes and Families

So far we have been using the expression "A is a type" informally. We now make this precise with **universes**.

Definition 1.1 A **universe** is a type whose elements are types. To avoid Russell's paradox we introduce a hierarchy of universes,

$$U_0 : U_1 : U_2 : \cdots$$

where every universe U_i is an element of the next universe U_{i+1}. Moreover, we assume that our universes are **cumulative**, so all elements of the ith universe are also elements of the $(i + 1)$th universe. This implies that elements no longer have unique types.

When we say A is a type, we mean it inhabits some universe U_i. We often write the universes just as U, with $U : U$ meaning $U_i : U_{i+1}$, creating a kind of **typical ambiguity**.

To model a collection of types varying over a given type A, we use function $B : A \to U$ whose codomain is a universe. These functionar are called **families of types** (or sometimes **dependent types**). An example of a type family is the family of finite sets $\mathrm{Fin} : \mathbb{N} \to U$.

A simpler example of a type family is the **constant** type family at a type $B : U$, which is of course the constant function $(\lambda(x : A).B) : A \to U$.

1.4 Dependent Function Types

In type theory we often use a more general version of function types, called $\prod$-**type** or **dependent function type**. The elements of a $\prod$-type are functions whose codomain type can vary depending on the element of the domain to which the function is applied, called **dependent functions**. The name "$\prod$-type" is used because this type can also be regarded as the cartesian product over a given type.

Definition 1.2 Given a type $A : U$, and a family $B : A \to U$, we may construct the type of dependent functions $\prod_{(x:A)} B(x) : U$.If B is a constant family, then the dependent product type is the ordinary function type:

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

To define $f : \prod_{(x:A)} B(x)$, where f is the name of a dependent function to be defined, we need an expression $\Phi : B(x)$ possibly involving the variable $x : A$, and we write

$$f(x) :\equiv \Phi, \quad \text{for } x : A$$

Alternatively, we can use λ-abstraction

$$\lambda x.\Phi : \prod_{x:A} B(x)$$

As with non-dependent functions, we can **apply** a dependent function $f : \prod_{(x:A)} B(x)$ to an argument $a : A$ to obtain an element $f(a) : B(a)$. The equalities are the same as the ordinary function type in the sense that for $a : A$, we have $f(a) \equiv \Phi'$ and $(\lambda x.\Phi)(a) \equiv \Phi'$, where Φ' is obtained by replacing all occurrences of x in Φ by a. Similarly we have the uniqueness principle $f \equiv (\lambda x.f(x))$.

An important class of dependent functions types are functions which are **polymorphic** over a given universe.

Definition 1.3 A function is polymorphic if it takes a type as one of its arguments, and then acts on elements of that type (or of other types constructed from it).

An example is the polymorphic identity function $\mathrm{Id} : \prod_{(A:U)} A \to A$, which we define by $\mathrm{Id} : \lambda(A : U).\lambda(x : A).x$.

We sometimes will write arguments of a dependent function type as subscripts, so $\mathrm{Id}(A, x) = \mathrm{Id}_A(x) :\equiv x$.

Example 1.1 A non-trivial example is the "swap" operation which switches the order of arguments of a curried two-argument function:

$$\mathrm{swap} : \prod_{(A:U)} \prod_{(B:U)} \prod_{(C:U)} (A \to B \to C) \to (B \to A \to C)$$

We define this by

$$\mathrm{swap}(A, B, C, g) :\equiv \lambda b.\lambda a.g(a)(b)$$

We can also write

$$\mathrm{swap}_{A,B,C}(g)(b, a) :\equiv g(a, b)$$

Note as with ordinary functions we use currying to define dependent functions of several arguments. However, the second domain in the dependent case may depend on the first one, and the codomain may depend on both.

1.5 Product Types

Given types $A, B : U$ we introduce the type $A \times B : U$, which we call their **cartesian product**. We also introduce a nullary product type, called the **unit type 1** : U. A term of $A \times B$ is a pair $(a, b) : A \times B$, where $a : A$ and $b : B$, and the only term of **1** is some particular object $\star : \mathbf{1}$. As with functions, ordered pairs are a primitive concept in Type theory.

Remark 1.1 We have a general pattern for introducing a new kind of type in Type theory. To specify a type, we specify:

(i) how to form new types of this kind, via **formation rules**. (Example: we can form the dependent function type $\prod_{(x:A)} B(x)$ when A is a type and $B(x)$ is a type for $x \in A$).

(ii) how to construct elements of that type. These are called the type's **constructors** or **introduction rules**. (Example: A function type has one constructor, λ-abstraction)

(iii) how to use elements of that type. These are called the type's **eliminators** or **elimination rules**. (Example: Function application for function types)

(iv) a **computation rule** (i.e. β-**reduction**), which expresses how an eliminator acts on a constructor. (Example: For functions, the computation rule states that $(\lambda x.\Phi)(a)$ is judmentally equal to the substitution of a for x in Φ.)

(v) an optional **uniqueness principle** (i.e. η-**expansion**), which expresses uniqueness of maps into or out of that type. For some types, the uniqueness principle characterizes maps into the type, by stating that every element of the type is uniquely determined by the results of applying eliminators to it, and can be reconstructed from these results by applying a constructor. (Example: For functions, the uniqueness principle says that any function f is judgmentally equal to the expanded function $\lambda x.f(x)$, and thus is uniquely determined by its values) For other types the uniqueness principle says that every map from that type is uniquely determined by some data.

When the uniqueness principle is not taken as a rule of judgmental equality, iti s often nevertheless provable as a propositional equality from the other rules for the type. In this case we call it a **propositional uniqueness principle**.

We will assert the uniqueness principle for products, “every element of $A \times B$ is a pair," as a propositional equality later. Now, how can we use pairs? Consider the definition of a non-dependent function $f : A \times B \to C$. Since we intend the only elements of $A \times B$ to be pairs, we expect to be able to define a function by prescribing the result when f is applied to a pair (a, b). We can prescribe these results by using currying, $g : A \to (B \to C)$. Thus we introduce the elimination rule for products, which says that for any such g, we can define a function $f : A \times B \to C$ by

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

Unlike in set theory, type theory assumes that a function on $A \times B$ is well-defined as soon as we specify its values on pairs, and from this we will be able to prove that every element of $A \times B$ is a pair. From a category-theoretic perspective we can say that we define the product $A \times B$ to be the left adjoint to the “exponential" $B \to C$.

As an example, we can derive the **projection** functions

$$\mathrm{pr}_1 : A \times B \to A$$
$$\mathrm{pr}_2 : A \times B \to B$$

with the defining equations

$$\mathrm{pr}_1((a, b)) :\equiv a$$
$$\mathrm{pr}_2((a, b)) :\equiv b$$

Rather than invoking the principle of function definition, every time we want to define a function, an alternative approach is to invoke it once in a universal case, and then apply the resulting function in all other cases. That is we may define a function of type

$$\mathrm{rec}_{A\times B} : \prod_{C:U} (A \to B \to C) \to A \times B \to C \tag{1.5.1}$$

with the defining equation

$$\mathrm{rec}_{A\times B}(C, g, (a, b)) :\equiv g(a)(b)$$

Then, we can define these functions by

$$\mathrm{pr}_1 :\equiv \mathrm{rec}_{A\times B}(A, \lambda a.\lambda b.a)$$
$$\mathrm{pr}_2 :\equiv \mathrm{rec}_{A\times B}(B, \lambda a.\lambda b.b)$$

We refer to the function $\mathrm{rec}_{A\times B}$ as the **recursor** for the product types. We also speak of the **recursion principle** for cartesian products, meaning the fact that we can define a function $f : A \times B \to C$ as above by its value on pairs.

We also have a recursor for the unit type

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

with the defining equation

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

To define dependent functions over the product type, we have to generalize the recursor. Given $C : A \times B \to U$, we may define a function $f : \prod_{(x:A\times B)} C(x)$ by providing a function $g : \prod_{(x:A)} \prod_{(y:B)} C((x, y))$ with defining equation

$$f((x, y)) :\equiv g(x)(y)$$

Then we can prove the propositional uniqueness principle. Specifically, we can construct a function

$$\mathrm{uniq}_{A\times B} : \prod_{x:A\times B} ((\mathrm{pr}_1(x), \mathrm{pr}_2(x)) =_{A\times B} x)$$

We will see later that the identity type has a reflexivity element $\mathrm{refl}_x : x =_A x$ for any $x : A$. Given this, we can define

$$\mathrm{uniq}_{A\times B}((a, b)) :\equiv \mathrm{refl}_{(a,b)}$$

This construction works, because in the case that $x :\equiv (a, b)$, we can calculate

$$(\mathrm{pr}_1((a, b)), \mathrm{pr}_2((a, b))) \equiv (a, b)$$

using the defining equations for the projections. Therefore,

$$\mathrm{refl}_{(a,b)} : (\mathrm{pr}_1((a, b)), \mathrm{pr}_2((a, b))) = (a, b)$$

is well-typed, since both sides of the equality are judgmentally equal.

The ability to define dependent functions in this way means that to prove a property for all elements of a product, it is enough to prove it for its canonical elements. Applying this principle in the universal case, we call the resulting function **induction** for product types: given $A, B : U$ we have

$$\mathrm{ind}_{A\times B} : \prod_{C:A\times B\to U} \left(\prod_{(x:A)} \prod_{(y:B)} C((x, y)) \right) \to \prod_{x:A\times B} C(x)$$

with the defining equation

$$\mathrm{ind}_{A\times B}(C, g, (a, b)) :\equiv g(a)(b)$$

Similarly, we may speak of a dependent function defined on pairs being obtained from the **induction principle** of the cartesian product. Induction is also called the **dependent eliminator** since it describes how to use an element of the product type, and recursion is the **non-dependent eliminator**.

Induction for the unit type is

$$\mathrm{ind}_{\mathbf{1}} : \prod_{C:\mathbf{1}\to U} C(\star) \to \prod_{x:\mathbf{1}} C(x)$$

with the defining equation

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

This enables us to prove the propositional uniqueness principle for $\mathbf{1}$, which asserts that its only inhabitant is $\star$. That is, we can construct

$$\mathrm{uniq}_{\mathbf{1}} : \prod_{x:\mathbf{1}} x = \star$$

by using the defining equations

$$\mathrm{uniq}_{\mathbf{1}}(\star) :\equiv \mathrm{refl}_{\star}$$

or equivalently using induction

$$\mathrm{uniq}_{\mathbf{1}} :\equiv \mathrm{ind}_{\mathbf{1}}(\lambda x.x = \star, \mathrm{refl}_{\star})$$

1.6 Dependent Pair Types

We now look at generalizing the product type to allow the type of the second component of a pair to vary depending on the choice of the first component. This is called a **dependent pair type** or **$\sum$-type**, because it corresponds to an indexed sum (disjoint union or coproduct) in set thoery.

Given a type $A : U$ and a family $B : A \to U$, the dependent pair type is written as $\sum_{(x:A)} B(x) : U$. The way to construct elements of a dependent pair type is by pairing: we have $(a, b) : \sum_{(x:A)} B(x)$ given $a : A$ and $b : B(a)$. If B is constant, then the dependent pair type is the ordinary cartesian product type:

$$\left(\sum_{x:A} B\right) \equiv (A \times B)$$

All the constructions on $\sum$-types arise as generalizations of the ones for product types, with dependent functions often replacing non-dependent ones.

For instance, the recursion principle says that to define a non-dependent function out of a $\sum$-type $f : \left(\sum_{(x:A)} B(x)\right) \to C$, we provide a function $g : \prod_{(x:A)} B(x) \to C$, and then we can define f via the defining equation

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

For instance, we can derive the first projectino from a $\sum$-type:

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

by the defining equation $\mathrm{pr}_1((a, b)) :\equiv a$. However, since the type of the second component of the pair is $B(a)$, the second projection must be a dependent function, whose type involves the first projection function:

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

Thus we need the induction principle for $\sum$-types. This says that to construct a dependent function out of a $\sum$-type into a family $C : \left(\sum_{(x:A)} B(x)\right) \to U$, we need a function

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

We can then derive a function

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

with defining equation

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

Applying this with $C(p) :\equiv B(\mathrm{pr}_1(p))$, we can define $\mathrm{pr}_2 : \prod_{(p:\sum_{(x:A)} B(x))} B(\mathrm{pr}_1(p))$ with the equation $\mathrm{pr}_2((a,b)) :\equiv b$.

Definition 1.4 The recursor for $\sum$ is

$$\mathrm{rec}_{\sum_{(x:A)} B(x)} : \prod_{(C:U)} \left(\prod_{(x:A)} B(x) \to C \right) \to \left(\sum_{(x:A)} B(x) \right) \to C$$

with defining equation

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

and the corresponding induction operation

$$\mathrm{ind}_{\sum_{(x:A)} B(x)} : \prod_{(C:(\sum_{(x:A)} B(x)) \to U)} \left(\prod_{(a:A)} \prod_{(b:B(a))} C((a,b)) \right) \to \prod_{(p:\sum_{(x:A)} B(x))} C(p)$$

with the defining equation

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

1.7 Coproduct Types

Given $A, B : U$, we introduce the **coproduct type** $A + B : U$.

1.8 The type of booleans

1.9 The natural numbers

The rules we have introduced so far do not allow us to construct any infinite types. The simplest infinite type is the type $\mathbb{N} : U$ of natural numbers. The elements of $\mathbb{N}$ are constructed using $0 : \mathbb{N}$ and the successor operation $\mathrm{succ} : \mathbb{N} \to \mathbb{N}$. When denoting natural numbers we write $1 :\equiv \mathrm{succ}(0), 2 :\equiv \mathrm{succ}(1), 3 :\equiv \mathrm{succ}(2)$,...

The essential property of the naturals is that we can define functions by recursion and perform proofs by induction. To construct a non-dependent function $f : \mathbb{N} \to C$ out of the natural numbers by recursion, it is enough to provide a starting point $c_0 : C$ and a "next step" function $c_s : \mathbb{N} \to C \to C$. This gives rise to f with the defining equations

$$\begin{aligned} f(0) &:\equiv c_0, \\ f(\mathrm{succ}(n)) &:\equiv c_s(n, f(n)) \end{aligned}$$

We say that f is defined by **primitive recursion**.

We can define multi-variable functions by primitive recursion as well, by currying and allowing C to be a function type. As in the previous cases we can package the principle of primitive recursion into a recursor:

$$\mathrm{rec}_{\mathbb{N}} : \prod_{(C:U)} C \to (\mathbb{N} \to C \to C) \to \mathbb{N} \to C$$

with defining equations

$$\begin{aligned} \mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, 0) &:\equiv c_0 \\ \mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, \mathrm{succ}(n)) &:\equiv c_s(n, \mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, n)) \end{aligned}$$

Note all functions definable only using the primitive recursion principle will be **computable**.

1.10 Pattern Matching and Recursion

1.11 Propositions as Types

1.12 Identity Types

By the propositions-as-types conception, the proposition that two elements of the same type $a, b : A$ are equal must correspond to some type. These **equality types** or **identity types** must be type families dependent on two copies of A.

We may write the family as $\mathrm{Id}_A : A \to A \to U$, so that $\mathrm{Id}_A(a, b)$ is the type representing the proposition of equality between a and b, written previously as "$a =_A b$" to specify the type A. If we have an term of $a =_A b$, we say that it is evidence of the equality of a and b, or that a and b are **propositionally equal**. The type $a =_A b$ can contain more information than just equality. In the homotopical interpretation, terms of $a =_A b$ are paths or equivalences between a and b in the space A. In this way we can interpret $a =_A b$ as the type of identifications of a and b.

The formation rule for the identity type says that given a type $A : U$, and two elements $a, b : A$, we can form the type $(a =_A b) : U$ in the same universe. The basic way to construct an element of $a =_A b$ is to know that a and b are the same. Thus, the introduction rule is a dependent function

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

called **reflexivity**, which says that every element of A is equal to itself (in a specified way). We regard refl_a as being the constant path at the point a.

If a and b are judgmentally equal, $a \equiv b$, then we also have an element $\mathrm{refl}_a : a =_A b$. This is well-typed because $a \equiv b$ means that also the type $a =_A b$ is judgmentally equal to $a =_A a$, which is the type of refl_a.

The induction principle (i.e. elimination rule) for the identity types is one of the subtlest parts of type theory, and crucial to homotopy interpretation. We begin by consiering an important consequence of it.

Theorem 1.1 (Indiscernability of Identicals) *For every family $C : A \to U$, there is a function*

$$f : \prod_{(x,y:A)} \prod_{(p:x=_A y)} C(x) \to C(y)$$

such that

$$f(x, x, \mathit{refl}_x) :\equiv id_{C(x)}$$

This says that every family of types C respects equality, in the sense C to equal elements of A also results in a function between the resulting types. The displayed equality states that the function associated to reflexivity is the identity function. This can be regarded as a recursion principle for the identity type. Just as $\mathrm{rec}_{\mathbb{N}}$ gives a specified map $\mathbb{N} \to C$ for any other type C of a certain sort, indiscernability of identicals gives a specified map from $x =_A y$ to certain other reflexive, binary relations on A, namely those of the form $C(x) \to C(y)$ for some unary predicate $C(x)$.

1.12.1 Path Induction

The induction principle for the identity type is called **path induction**. It can be seen as stating that the family of identity types is freely generated by the elements of the form $\mathrm{refl}_x : x = x$.

Definition 1.5 (Path Induction) Given a family $C : \prod_{x,y:A}(x =_A y) \to U$ and a function $c : \prod_{x:A} C(x, x, \mathrm{refl}_x)$, there is a function

$$f : \prod_{(x,y:A)} \prod_{(p:x=_A y)} C(x, y, p)$$

such that $f(x, x, \mathrm{refl}_x) :\equiv c(x)$.

Path induction allows us to define specified functions which exhibit appropriate computational behaviour. Consider the case when C does not depend on p, so $: A \to A \to U$, which can be regarded

as a predicate depending on two elements of A. We are interesting in when $C(x, y)$ holds for some $x, y : A$. The hypotheses of path induction say that we know $C(x, x)$ holds for all $x : A$, so C is a reflexive relation. The conclusion then tells us that $C(x, y)$ holds whenever $x = y$.

Packaging up path induction into a single function gives

$$\mathrm{ind}_{=_A} : \prod_{(C:\prod_{(x,y:A)}(x=_A y)\to U)} \left(\prod_{(x:A)} C(x, x, \mathrm{refl}_x) \right) \to \prod_{(x,y:A)} \prod_{(p:x=_A y)} C(x, y, p)$$

with the equality

$$\mathrm{ind}_{=_A}(C, c, x, x, \mathrm{refl}_x) :\equiv c(x)$$

Theorem 1.2 (Based Path Induction) *Fix an element $a : A$, and suppose we are given a family $C : \prod_{x:A}(a =_A x) \to U$, and an element $c : C(a, \mathit{refl}_a)$. Then we obtain a function*

$$f : \prod_{(x:A)} \prod_{(p:a=x)} C(x, p)$$

such that $f(a, \mathit{refl}_a) :\equiv c$.

Packaged as a function, based path induction becomes:

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

with the equality

$$\mathrm{ind}'_{=_A}(a, C, c, a, \mathrm{refl}_a) :\equiv c$$

This is equivalent to path induction.

Remark 1.2 Note it is not the identity type that is inductively defined, but the identity family. In particular, path induction says that the family of types $(x =_A y)$, as x, y vary over all elements of A, is inductively defined by the elements of the form refl_x. This means that to give an element of any other family $C(x, y, p)$ dependent on a generic element (x, y, p) of the identity type, it suffices to consider the cases of the form (x, x, refl_x). This says that triples (x, y, p) of endpoints of the path p, is inductively generated by the constant loops at each point x.

Chapter 2

Homotopy Type Theory

2.1 Types as Higher Groupoind

Lemma 2.1 *For every type A and every $x, y : A$, there is a function*

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

denoted $p \mapsto p^{-1}$, such that $\mathit{refl}_x^{-1} \equiv \mathit{refl}_x$ for each $x : A$. We call p^{-1} the inverse of p.

Chapter 3

Sets and Logic

Chapter 4

Equivalences

Chapter 5

Induction

Chapter 6

Higher Inductive Types

Chapter 7

Homotopy n-Types

Part II

Mathematics

Chapter 8

Homotopy Theory

Chapter 9

Category Theory

Chapter 10

Set Theory

Chapter 11

Real Numbers

Part III

COQ Proof Assistant

Chapter 12

Coding in Coq

Appendix A

Formal Type Theory

All's well that ends well