# Homotopy Type Theory Primer

Emily Pillmore

***Abstract*— This set of notes covers chapter 1 of *Homotopy Type Theory: Univalent Foundations of Mathematics*. We will cover in shorthand the notation, syntax, semantics, topics presented in the chapter, as well as show solutions for the exercises at the end. This will be handed out as reference material with the terminus of the chapter.**

## I. Function Types

Given types $A$ and $B$, we can construct the type $A \to B$ of **functions** with domain $A$ and codomain $B$. Function types are a primitive concept in type theory. Let $f : A \to B$ be a function and let $a : A$. We can **apply** the function $f$ to obtain an element in the codomain of $f$ of type $B$, denoted $f(a)$ or $fa$, called the **value** of $f$ at $a$. We may construct elements of type $A \to B$ in two equivalent ways: either by direct definition, or by $\lambda$**-abstraction**. Introducing a function by definition requires introducing the function by a name - say, $f$ - and defining $f : A \to B$ by giving an equation:

$$f(x) :\equiv \Phi \tag{1}$$

where $x$ is a variable, and $\Phi$ is an expression which may or may not use $x$. In order to be valid, one must check $\Phi : B$ assuming $x : A$. Now, we may compute $fa$ by replacing the variable $x$ in $\Phi$ with $a$. If we do do not wish to introduce a name for the function, we use $\lambda$-abstraction. Given an expression $\Phi$ of type $B$, which may or may not use $x : A$ as above, we write $\lambda(x : A).\Phi$ to indicate the function defined in (1). Thus, we have an equivalent definition:

$$\lambda(x : A).\Phi : A \to B \tag{2}$$

We generally omit the type of $x$ in a $\lambda$-abstraction and write $\lambda x.\Phi$, since typing $x : A$ is inferable from the judgment that the abstraction has type $A \to B$. The **scope** of the variable binding $\lambda x$ is the rest of the expression unless it is delimited with parentheses - *e.g.* $\lambda x.x+x$ should be parsed as $\lambda x.(x+x)$, and is semantically distinct from $(\lambda x.x)+x$. In fact, the latter example is not even well-typed! Sometimes, a "-" is used to denote a lambda abstraction. For example, $g(x, -) \cong \lambda y.g(x, y)$. Yet another useful notation is $x \mapsto \Phi$, which is equivalent to $\lambda x.\Phi$.

In order to apply an abstraction to a value, one uses a computation rule called $\beta$**-reduction**:

$$(\lambda x.\Phi)(a) \equiv \Phi' \tag{3}$$

where $\Phi'$ is the expression $\Phi$ in which all occurrences of $x$ have been replaced by $a$. Note that for any function $f : A \to B$, we can construct a lambda abstraction $\lambda x.fx$, which is "the function that applies $f$ to its argument". This abstraction is *definitionally* equal to $f$ by yet another computation rule called $\eta$**-conversion**:

$$f \equiv \lambda x.fx \tag{4}$$

This equality is called the **uniqueness principle for function types**; it shows that $f$ is uniquely determined by its values. As a consequence, we may view

$$fx :\equiv \Phi \tag{5}$$

as

$$f :\equiv \lambda x.\Phi. \tag{6}$$

We must be careful that we preserve the binding structures of expressions when doing calculations involving variables. *Binding structures* are the invisible link generated by binders such as $\lambda, \Pi$ and $\Sigma$ between the place where the variable is introduced, and where it is used. In order to avoid variable **capture** (or, breaking the bindings of other variables via blind substitution), we consider the notion of bound (or "dummy") variables. For example, for $f : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ and the definition $fx :\equiv \lambda y.x + y$, $y$ is a *bound* variable of the expression. Indeed, under the computation rule $\alpha$**-conversion**, we consider $\lambda y.x + y$ and $\lambda z.x + z$ to be judgmentally equal.

Often we may find that we would like to be able to abstract upon the arity (the number of variables a function admits as inputs) of $f$ by having $f$ consume more than one input. The canonical way to do this in standard mathematics would be to give $f$ the type $f : A \times B \to C$, where $A \times B$ is the Cartesian product of $A$ and $B$, however, it is not always a given that such a product exists in the space where $f$ exists. Until the correct foundations are laid out, we will make use of another construction that avoids the use of product types called **currying**. The idea is that we consider a two-variable function to be an iterated function of type $f : A \to (B \to C)$. Thus, for $a : A$ and $b : B$, we may apply the function as so: $(fa)b$ or $f(a)(b)$ or $f(a, b)$, or even $fab$. Likewise, we may expand such a function $f$ equivalently in the following ways:

$$f(x, y) :\equiv \Phi \tag{7}$$

$$f :\equiv \lambda x.\lambda y.\Phi \tag{8}$$

$$f :\equiv x \mapsto y \mapsto \Phi. \tag{9}$$

$$f(-, -) :\equiv x \mapsto y \mapsto \Phi. \tag{10}$$

Currying a function of arity $> 2$ is a straightforward extension of what was just described.

## II. Universes

A **universe** is a type whose elements are types. To avoid the type-theoretic version of Russel's paradox ($\mathcal{U}_\infty : \mathcal{U}_\infty$), we introduce a hierarchy of universes

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \dots \tag{11}$$

where every universe $\mathcal{U}_i$ is an element of the subsequent universe $\mathcal{U}_{i+1}$. We assume our universes are **cumulative**, i.e. that all elements of the $i^{th}$ universe are also elements of the $(i+1)^{st}$ universe. When we say that $A$ is a type, we mean that it inhabits some universe $\mathcal{U}_i$. We can omit the index of $\mathcal{U}$ and just assume the indexing all works out in a consistent way for the time being, just saying $A : \mathcal{U}$. When some universe $\mathcal{U}$ is assumed, we may refer to the types belonging to it as **small types**.

To model a collection of types varying over a given type A, we may use functions $B : A \to \mathcal{U}$ whose codomain is a universe. These functions are called **families of types** (much like families of sets), sometimes called *dependent types*.

## III. Dependent function types ($\Pi$-types)

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

Given a type $A : \mathcal{U}$, and a family $B : A \to \mathcal{U}$, we can construct the type of dependent functions $\Pi_{(x:A)}B(x) : \mathcal{U}$. There are many alternative notations for this type:

$$\Pi_{(x:A)}B(x) \quad \prod_{(x:A)} B(x) \quad \Pi(x : A), B(x) \tag{12}$$

If $B : \mathcal{U}$ is a constant family (i.e. $\lambda(x : A).B) : A \to \mathcal{U}$), then the dependent product type is the ordinary function type:

$$\Pi_{(x:A)}B \equiv A \to B \tag{13}$$

All constructions of $\Pi$-types are generalizations of corresponding constructions on ordinary function types. We can introduce dependent functions by explicit definitions or by $\lambda$-abstraction: to define $f : \Pi_{(x:A)}B(x)$, we must have a name $f$, and an expression $\Phi : B(x)$ possibly involving the variable $x : A$ and we may write the following:

$$fx :\equiv \Phi \text{ for } x : A. \tag{14}$$

or, alternatively,

$$\lambda x.\Phi : \prod_{x:A} B(x). \tag{15}$$

As with standard function types, we can apply a dependent function type $f : \Pi_{(x:A)}B(x)$ to an argument $a : A$ to obtain an element $fa : B(a)$. The same computation rules for $\beta, \eta$, and $\alpha$ apply in a similar sense.

Another important class of dependent function types are functions which are **polymorphic** over a given universe. A polymorphic function is one which takes a type as one of its arguments and then acts on elements of that type (or of other types constructed from it). For example, the identity function, $id : \Pi_{(A:\mathcal{U})}A \to A$, which is defined as $id :\equiv \lambda(A : \mathcal{U}).\lambda(x : A).x$. Note that much like other binders, $\Pi$ automatically scopes over the rest of the expression unless delimited. Sometimes, we write some arguments to dependent functions as subscripts (e.g. $id_A(x) :\equiv x$). If the argument can be inferred from context, for instance, for some $a : A$, then we will just write $id(a)$, since it is unambiguous that we are working with $id_A$. Consider the following function:

$$swap : \prod_{(A:\mathcal{U})} \prod_{(B:\mathcal{U})} \prod_{(C:\mathcal{U})} (A \to B \to C) \to (B \to A \to C) \tag{16}$$

defined as

$$swap(A, B, C, g) :\equiv \lambda b.\lambda a.gab. \tag{17}$$

Note that just like we did for ordinary functions, we can use currying to define dependent functions with several arguments, however, in the dependent case, this is more complicated. The second codomain may dependent on the first one, and the codomain may depend upon both. For example, given an $A : \mathcal{U}$, and type families $B : A \to \mathcal{U}$ and $C : \Pi_{(x:A)}B(x) \to \mathcal{U}$, we can construct the type $\Pi_{(x:A)}\Pi_{(y:B(x))}C(x, y)$ of functions with two arguments.

## IV. Product Types

Given types $A, B : \mathcal{U}$, we introduce the type $A \times B : \mathcal{U}$ consisting of values $(a, b) : A \times B$ where $a : A$ and $b : B$, which we call their **cartesian product**. We introduce the nullary product called the **unit type $\mathbf{1}$** : $\mathcal{U}$. We intend that only element of $\mathbf{1}$ is some object $* : \mathbf{1}$. The way we construct pairs is obvious: given $a : A$, and $b : B$, we insert them into the tuple $(a, b) : A \times B$. Likewise, we can trivially construct $* : \mathbf{1}$ by simply summoning it from the void and using the symbol $*$. Now, how we *use* pairs is slightly less trivially, and we must introduce some elimination rules to do it. Suppose we could provide a function $g : A \to B \to C$ for a normal function $f : A \times B \to C$. For any such $g$, we can define the function $f$ by the following:

$$fab :\equiv g(a)(b) \tag{18}$$

Thus, we can identify the two via this rule. Rather than invoking this construction every time, we can define a **recursor** for product types that can be called once, universally, to express the rules we want:

$$rec_{A\times B} : \prod_{C:\mathcal{U}} (A \to B \to C) \to A \times B \to C \tag{19}$$

with the defining equation

$$rec_{A\times B}(C, g, (a, b)) :\equiv g(a)(b) \tag{20}$$

Then, we can define the canonical projections for product types thusly:

$$\pi_1 :\equiv rec_{A\times B}(A, \lambda a.\lambda b.a) \tag{21}$$

$$\pi_2 :\equiv rec_{A\times B}(A, \lambda a.\lambda b.b) \tag{22}$$

In order to define *dependent* functions over product types, we have to generalize the recursor. Given $C : A \times B \to \mathcal{U}$, we can define a function $f : \Pi_{(x:A\times B)} C(x)$ by providing a function

$$g : \prod_{x:A} \prod_{y:B} C((x, y)) \tag{23}$$

with the defining equation

$$f((x, y)) :\equiv g(x)(y). \tag{24}$$

Generally, the ability to define the 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, the ordered pairs. When we do as we did above and apply this principle once in the universal case, we call the resulting function **induction** for product types: given $A, B : \mathcal{U}$, we have

$$ind_{A\times B} : \prod_{C:A\times B\to\mathcal{U}} (\prod_{x:A} \prod_{y:B} C((x, y))) \to \prod_{x:A\times B} C(x) \tag{25}$$

with the defining equation

$$ind_{A\times B}(C, g, (a, b)) :\equiv g(a)(b). \tag{26}$$

Induction is also called the **(dependent) eliminator**, and recursion the **non-dependent eliminator**. Induction for the unit type turns out to be more useful than the recursor:

$$ind_{\mathbf{1}} : \prod_{C:\mathbf{1}\to\mathcal{U}} C(*) \to \prod_{x:\mathbf{1}} C(x) \tag{27}$$

with the defining equation

$$ind_{\mathbf{1}}(C, c, *) :\equiv c. \tag{28}$$

Induction enables us to prove the propositional uniqueness principle for **1**, which asserts that the only inhabitant is *.

## V. Dependent pair types ($\Sigma$-types)

It is often useful to generalize the product types from above 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 a **$\Sigma$-type**, because it corresponds to an indexed sum (in the sense of a coproduct) over a given type. Given a type $A : \mathcal{U}$, and a family $B : A \to \mathcal{U}$, the dependent pair type is written as $\Sigma_{(x:A)} B(x) : \mathcal{U}$. Alternative notations are:

$$\Sigma_{(x:A)} B(x) \tag{29}$$

$$\sum_{(x:A)} B(x) \tag{30}$$

$$\Sigma(x : A)B(x). \tag{31}$$

Similarly to the other binders, $\Sigma$ is automatically scoped over the rest of an expression unless delimited by parentheses.

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

$$(\sum_{(a:A)} B) \equiv (A \times B). \tag{32}$$

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

For example, the recursion principle says that to define a non-dependent function out of a $\Sigma$-type $f : (\Sigma_{(a:A)} B(x)) \to C$, we must provide a function $g : \Pi_{a:A} B(x) \to C$, and then we define $f$ via the defining equation:

$$f((a, b)) :\equiv g(a)(b). \tag{33}$$

Projections out of a $\Sigma$-type behave in a similar manner. To define $pr_1$, it is the usual construction, but in the case of the dependent parameter, a dependent function is needed:

$$\pi_1 : (\sum_{(a:A)} B(x)) \to A \tag{34}$$

$$\pi_2 : \prod_{p:\Sigma_{(a:A)} B(x)} B(\pi_1(p)). \tag{35}$$

Thus, we need the *induction* principle for $\Sigma$-types (the "dependent eliminator"). This says that to construct a function out of a $\Sigma$-type into a family $C : (\Sigma_{(a:A)} B(x) \to \mathcal{U}$, we need a function

$$g : \prod_{(a:A)} \prod_{(b:B(a))} C((a, b)). \tag{36}$$

We can then derive a function

$$f : \prod_{p:\sum_{(x:A)} B(x)} C(p) \tag{37}$$

along with a defining equation

$$f((a, b)) :\equiv g(a)(b). \tag{38}$$

Allowing for $C(p) :\equiv B(\pi_1(p))$, we can finally define the second dependent projection:

$$\pi_2 : \prod_{p:\Sigma_{(a:A)} B(a)} B(\pi_1(p)) \tag{39}$$

with the obvious equation $\pi_2((a, b)) :\equiv b$.

We can package the inductive principles up with its corresponding recursion principles by defining the recursor for $\Sigma$:

$$rec_{\Sigma_{(a:A)} B(a)} : \prod_{(C:\mathcal{U})} (\Pi_{(x:A)} B(x) \to C) \to (\Sigma_{(x:A)} B(x)) \to C \tag{40}$$

along with the corresponding induction operator:

$$ind_{\Sigma_{(a:A)} B(a)} : \prod_{(C:(\Sigma_{(x:A)} B(a))\to\mathcal{U})} (\Pi_{(a:A)} \Pi_{(b:B(a))} C((a, b)) \to \prod_{(p:\Sigma_{x:A)} B(x))} C(p) \tag{41}$$

with the defining equation

$$ind_{\Sigma_{(x:A)}B(x)}(C, g, (a, b)) :\equiv g(a)(b). \tag{42}$$

Dependent pair types are often used to defined types of mathematical structures, which commonly consist of several dependent pieces of data. For examples, suppose we wanted to define a **magma** to be a type $A$ together with a binary operation $m : A \to A \to A$. To be pedantic, the precise meaning of "together with" means that "a magma" is a *pair* $(A, m)$ consisting of a type $A : \mathcal{U}$, and an operation $m : A \to A \to A$. Since the type of the second component of $A \to A \to$ depends upon the first component, $A$, such pairs belong to a dependent pair type. Thus, the definition of a magma should be read as defining *the type of magmas* to be

$$Magma :\equiv \sum_{A:\mathcal{U}} (A \to A \to A).$$

Intuitively, we can build upon this structure and even define "pointed types", by affixing a basepoint $e : A$ to such a magma and define the type of pointed magmas:

$$PointedMagma :\equiv \sum_{A:\mathcal{U}} (A \to A \to A) \times A. \tag{43}$$

## VI. Coproduct Types

Given $A, B : \mathcal{U}$, we may introduce their **coproduct** type $A + B : \mathcal{U}$. This type corresponds to the *disjoint union* in set theory. We also introduce an incredibly important type the **empty type** $\mathbf{0} : \mathcal{U}$.

There are two ways to consturct elements of $A+B$, either as $inl(a) : A + B$, for $a : A$, or as $inr(b) : A + B$ for $b : B$. These names are shorthand for *left-injection* and *right injection*. There are no ways to construct the empty type, $\mathbf{0}$. To construct a non-dependent function $f : A + B \to C$, we need functions $g_0 : A \to C$ and $g_1 : B \to C$. Then $f$ is defined via the defining equations

$$f(inl(a)) :\equiv g_0(a) f(inr(b)) :\equiv g_1(b). \tag{44}$$

That is, the function $f$ is defined by **case analysis**. As before, we can derive a recursor:

$$rec_{A+B} : \prod_{(C:\mathcal{U})} (A \to C) \to (B \to C) \to A + B \to C \tag{45}$$

with the defining equations

$$rec_{A+B}(C, g_0, g_1, inl(a)) :\equiv g_0(a)$$

$$rec_{A+B}(C, g_0, g_1, inr(b)) :\equiv g_1(b).$$

We can *always* construct a function $f : \mathbf{0} \to C$ without having to give any defining equations, because there are no elements of $\mathbf{0}$ on which $f$ may be defined. Thus, the recursor for $\mathbf{0}$ is

$$rec_{\mathbf{0}} : \Pi_{(C:\mathcal{U})} \mathbf{0} \to C,$$

which constructs the canonical function from the empty type to any other type. Logically, it corresponds to the principle of *ex falso quodlibet*.

To cosntruct a dependent function $f : \Pi_{(x:A+B)} C(x)$ out of a coproduct, we assume as given the family $C : A + B \to \mathcal{U}$, and require

$$g_0 : \prod_{a:A} C(inl(a))$$

$$g_1 : \prod_{b:B} C(inr(b)).$$

This yields $f$ with the defining equations:

$$f(inl(a)) :\equiv g_0(a) \tag{46}$$

$$f(inr(b)) :\equiv g_1(b). \tag{47}$$

We package this scheme into the induction principle for coproducts:

$$ind_{A+B} : \prod_{C:A+B\to\mathcal{U}} (\Pi_{(a:A)} C(inl(a))) \to (\Pi_{(b:B)} C(inr(b))) \to (\Pi_{(x:A+B)} C(x).$$

As before, the recursor manifests when $C$ is a constant type family. The induction principle for the empty type may be defined thusly:

$$ind_{\mathbf{0}} : \prod_{C:\mathbf{0}\to\mathcal{U}} \prod_{z:\mathbf{0}} C(z) \tag{48}$$

which gives us a way to define a trivial dependent function out of the empty type.

## VII. Boolean Types

The type of booleans $\mathbf{2} : \mathcal{U}$ is intended to have exactly two elements, $0_{\mathbf{2}}, 1_{\mathbf{2}} : \mathbf{2}$. We can construct this type out of coproduct and unit types, via $\mathbf{1} + \mathbf{1}$, however, since it is used frequently, we will give the explicit rules here.

To derive a function $f : \mathbf{2} \to C$, we need $c_0, c_1 : C$ and add the defining equations

$$f(0_{\mathbf{2}}) :\equiv c_0$$

$$f(1_{\mathbf{2}}) :\equiv c_1.$$

The recursor corresponds to the if-then-else construct in functional programming:

$$rec_{\mathbf{2}} : \prod_{C:\mathcal{U}} C \to C \to \mathbf{2} \to C \tag{49}$$

with the defining equations

$$rec_{\mathbf{2}}(C, c_0, c_1, 0_{\mathbf{2}}) :\equiv c_0 \tag{50}$$

$$rec_{\mathbf{2}}(C, c_0, c_1, 1_{\mathbf{2}}) :\equiv c_1. \tag{51}$$

We package this up into the following induction principle:

$$ind_{\mathbf{2}} : \prod_{C:\mathbf{2}\to\mathcal{U}} C(0_{\mathbf{2}}) \to C(1_{\mathbf{2}}) \to \Pi_{(x:\mathbf{2})} C(x) \tag{52}$$

with the defining equations

$$inc_{\mathbf{2}}(C, c_0, c_1, 0_{\mathbf{2}}) :\equiv c_0 \tag{53}$$

$$ind_{\mathbf{2}}(C, c_0, c_1, 1_{\mathbf{2}}) :\equiv c_1. \tag{54}$$

We have remarked that $\Sigma$-types can be regarded as analogous to indexed disjoint unions, while coproducts are binary disjoint unions. It is natural to expect that a binary disjoint union $A + B$ could be constructed as an indexed one over $\mathbf{2}$. For this, we need a type family $P : \mathbf{2} \to \mathcal{U}$ such that $P(0_{\mathbf{2}} \equiv A$ and $P(1_{\mathbf{2}} \equiv B$. Indeed, we can obtain such a family precisely by the recursion principle for $\mathbf{2}$. Thus, we have defined

$$A + B :\equiv \sum_{x:\mathbf{2}} rec_{\mathbf{2}}(\mathcal{U}, A, B, x) \tag{55}$$

with

$$inl(a) :\equiv (0_{\mathbf{2}}, a) \tag{56}$$

$$inr(b) :\equiv (1_{\mathbf{2}}, b). \tag{57}$$

A similar construction exists for products.

## VIII. Natural Numbers

The concrete type $\mathbb{N} : \mathcal{U}$ denotes the type of natural numbers - the most basic types of numbers. The elements of $\mathbb{N}$ are constructed using $0 : \mathbb{N}$ and the successor operation $succ : \mathbb{N} \to \mathbb{N}$. When denoting natural numbers, we adopt the usual decimal notation $1 :\equiv succ(0), 2 : succ(1), \ldots$

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

$$f(0) :\equiv c_0, \tag{58}$$

$$f(succ(n)) :\equiv c_s(n, f(n)). \tag{59}$$

We say that $f$ is defined by **primitive recursion**. Indeed, primitive recursion principles can be package into a recursor:

$$rec_{\mathbb{N}} : \prod_{C:\mathcal{U}} C \to (\mathbb{N} \to C \to C) \to \mathbb{N} \to C \tag{60}$$

with the defining equations

$$rec_{\mathbb{N}}(C, c_0, c_s, 0) :\equiv c_0 \tag{61}$$

$$rec_{\mathbb{N}}(C, c_0, c_s, succ(n)) :\equiv c_s(n, rec_{\mathbb{N}}(C, c_0, c_s, n)) \tag{62}$$

Of course, all functions definable using only primitive recursion will be *computable*, though presence of higher function types means we can define more than the usual primitive recursive functions. We can now generalize primitive recursion to dependent functions to obtain an induction principle. Assume a given family $C : \mathbb{N} \to \mathcal{U}$, an element $c_0 : C(0)$, and a function $c_s : \Pi_{n:\mathbb{N}} C(n) \to C(succ(n))$; then we can construct $f : \Pi_{n:\mathbb{N}} C(n)$ with the defining equations:

$$f(0) :\equiv c_0 \tag{63}$$

$$f(succ(n)) :\equiv c_s(n, f(n)). \tag{64}$$

We can also package this into a single function:

$$ind_{\mathbb{N}} : \prod_{C:\mathbb{N}\to\mathcal{U}} C(0) \to (\Pi_{n:\mathbb{N}} C(n) \to C(succ(n))) \to \Pi_{n:\mathbb{N}} C(n) \tag{65}$$

with the defining equations

$$ind_{\mathbb{N}}(C, c_0, cs, 0) :\equiv c_0 \tag{66}$$

$$ind_{\mathbb{N}}(C, c_0, cs, succ(n)) :\equiv c_s(n, ind_{\mathbb{N}}(C, c_0, c_s, n)). \tag{67}$$

## IX. Identity Types

The *proposition* that two elements of the same type $a, b : A$ are equal must correspond to some *type*. Since this proposition depends on what $a$ and $b$ are, these **equality types** or **identity types** must be type families dependent on two copies of A.

We may write the family as $Id_A : A \to A \to \mathcal{U}$ (not to be mistaken for the identity *function* $id_A$), so that $Id_A(a, b)$is the type representing the proposition of equality between $a$ and $b$. It is standard to use the equality symbol for this; thus $a =_A b$ will also be a notation used fro the type $Id_A(a, b)$ corresponding to the proposition that $a$ equals $b$. For clarity, we may also write $a =_A b$ to specify the type $A$. If we have an inhabitant of $a =_A b$, then we may say that $a$ and $b$ are equal, or sometimes propositionally equal if we want to emphasize that this is different from the judgmental equality $a \equiv b$. Just as in the propositions-as-types versions of "or" and "there exists" we can include more information than just the fact that the proposition is true, nothing prevents the type $a = b$ from also including more information. Indeed, this is the cornerstone of the homotopical interpretation where we regard witnesses of $a = b$ as *paths* or *equivalences* between $a$ and $b$ in the space $A$. Just as there can be more than one path between two points of a space, there can be more than one witness that two objects are equal.

The formation rule says that given a type $A : \mathcal{U}$ and two elements $a, b : A$, we can form the type $(a =_A b) : \mathcal{U}$ in the same universe. The way to construct $a =_A b$ is to know that $a$ and $b$ are the same. Thus, the introduction rules is a dependent function

$$refl : \prod_{a:A} (a =_A a) \tag{68}$$

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

In particular, this means that if $a$ and $b$ are judgmentally equal, then we also have an element $refl_a : a =_A b$. 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 for the identity types is one of the most subtle parts of type theory, and crucial to the homotopic interpretation. We may begin by considering an important consequence of it, that "equals may be substituted for equals", as expressed by the following:

**Indiscernability of identicals:** For every family

$$C : A \to \mathcal{U}, \tag{69}$$

there is a function

$$f : \prod_{x,y:A} \prod_{p:x=_A y} C(x) \to C(y) \tag{70}$$

such that

$$f(x, x, refl_x) :\equiv id_{C(x)}. \tag{71}$$

This says that for every family of types $C$ that respects equality, then applying $C$ to equal elements of $A$ results in a function between the resulting types. The displayed equality states that the function associated to reflexivity is the identity function. This indiscernability can be regarded as the recursion principle for identity types. In order to define an induction principle for identity types, we must not only consider maps out of $x =_A y$, but also families over it. Put differently, we consider not only allowing equals to be substituted for equals, but also taking into account the evidence $p$ for equality.

## X. Path Induction

The induction principle for identity types is called **path induction**, in view of the homotopical interpretation of the type theory. It can be seen as stating that the family of identity types freely generated by the elements of the form $refl_x : x = x$.

**Path induction:** Given a family

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

and a function

$$c : \prod_{x:A} C(x, x, refl_x)$$

there is a function

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

such that

$$f(x, x, refl_x) :\equiv c(x).$$

The general, inductive form of the rule allows $C$ to depend on the witness $p : x = y$ to the identity between $x$ and $y$. In the premise, we not only replace $x, y$ by $x, x$, but also simultaneously replace $p$ by $refl_x$: to prove a property for all elements $x, y$ and paths $p : x = y$ between them, and the path is $refl_x : x = x$. If we were viewing types just as sets, it would be unclear what this buys us, but since there may be many identifications $p : x = y$ between $x$ and $y$, it makes sense to keep track of them in considering families over the type $x =_A y$.

If we package path induction into a single function, it takes the form

$$ind_{=_A} : \prod_{C:\prod_{(x,y:A)}(x=_A y)\to\mathcal{U}} (\Pi_{x:A} C(x, x, refl_x) \to \prod_{x,y:A} \prod_{p:x=_A y} C(x, y, p) \tag{72}$$

with the equality

$$ind_{=_A}(C, c, x, x, refl_x) :\equiv c(x). \tag{73}$$

The function $ind_{=_A}$ is traditionally called $J$. Given a proof $p : a = b$, path induction requires us to replace both $a$ and $b$ with the same unknown element $x$. Thus, in order to define an element of a family $C$ for all equal elements of $A$, it suffices to define it on the diagonal. In some proofs, however, it is simpler to make use of an equation $p : a = b$ by replacing all occurrences of $b$ with $a$ (or vice versa), because it is sometimes easier to to do the remainder of a proof for one of those specific elements, rather than some unknown $x$. This motivates a second induction principle, which says that the family of types $a =_A x$ is generated by the element $refl_a : a = a$. This second principle is equivalent to the previous.

**Based path induction:** Fix an element $a : A$, and suppose we have a family

$$C : \prod_{x:A} (a =_A x) \to \mathcal{U} \tag{74}$$

and an element

$$c : C(a, refl_a). \tag{75}$$

Then, we obtain a function

$$f : \prod_{x:A} \prod_{p:a=x} C(x, p) \tag{76}$$

such that

$$f(a, refl_a) :\equiv c. \tag{77}$$

This principle says that to define an element of this family for all $x$ and $p$, it suffices to consider just the case where $x$ is $a$ and $p$ is $refl_a : a = a$. Thus, packaged as a function, based path induction becomes:

$$ind'_{=_A} : \prod_{a:A} \prod_{C:\prod_{(x:A)}(a=_A x)\to\mathcal{U}} C(a, refl_a) \to \prod_{a:A} \prod_{p:a=_A x} C(x, p) \tag{78}$$

with the equality

$$ind'_{=_A}(a, C, c, a, refl_a) :\equiv c. \tag{79}$$

## XI. DISEQUALITY

**Disequality** is the negation of equality:

$$(x \neq_A y) :\equiv \neg(x =_A y). \tag{80}$$

If $x \neq y$, we say that $x$ and $y$ are **unequal** or **not equal**. Just like negation, disequality plays a less important role here than it does in classical logic. Due to the absence of LEM, we cannot, for instance, prove that two elements are equal by proving the negation of their unequality; this would be an application of the classical law of double-negation.