INTRODUCTION TO TYPE THEORY

FORTE SHINKO

1. Introduction

In type theory, the basic object of study is a **type**. Here are some types:

- $\mathbb{N}$ (the natural numbers)
- $\mathbb{Z}$ (the integers)
- $\mathbb{S}^1$ (the circle)
- π_1 (the fundamental group)
- $=$ (the identity type)

Types can have **terms**. For example, 0 is a term of type $\mathbb{N}$, written as $0 : \mathbb{N}$. The expression $0 : \mathbb{N}$ is an example of a **type declaration**.

Another example of a type declaration is $\mathbb{Z} : \texttt{Type}$, saying that $\mathbb{Z}$ is a type. There is a "type of types" denoted `Type`, and it can cause inconsistencies if we're not careful. We can use **universes** to put this on solid formal ground, but we won't do so until needed.

Here are some interpretations of types:

- A type is a set, and a term is an element of this set.
- A type is a space, and a term is a point in this space.
- A type is a proposition, and a term is a proof of this proposition.

The basic statements in type theory are known as **judgments**. Our judgments will look like one of the following:

- $\Gamma \vdash A : \texttt{Type}$
- $\Gamma \vdash x : A$
- $\Gamma \vdash A \equiv B : \texttt{Type}$
- $\Gamma \vdash x \equiv y : A$

where Γ is a finite sequence of type declarations, known as a **context**. These intuitively mean that the assumptions in the left-hand context imply the right-hand side. The symbol $\equiv$ stands for **judgmental equality**. We'll need to differentiate it from the type $=$ which we'll see later (this type represents **propositional equality**).

Note that the context can be empty; for example, we'll have the judgment $\vdash 0 : \mathbb{N}$. We'll have certain **inference rules** saying how to derive judgments from other ones. A collection of such rules is known as a **type theory**.

There are some "obvious" judgments such as $x : A \vdash x : A$ which always hold. We won't list them all here; see the appendix for a full list.

2. Function types (aka λ-calculus)

Given two types A and B, there is a type $A \to B$. Formally, this rule is written as follows:

$$\frac{\vdash A : \texttt{Type} \qquad \vdash B : \texttt{Type}}{\vdash A \to B : \texttt{Type}}\ \lambda\text{-FORM}$$

This is the **formation rule** for function types.

Actually, the rule we wrote is a bit restrictive, since it only lets us form $A \to B$ when we have **no** assumptions. The formation rule in full generality is the following:

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma \vdash B : \texttt{Type}}{\Gamma \vdash A \to B : \texttt{Type}}\ \lambda\text{-FORM}$$

Date: August 31, 2019.

where Γ stands for any context.

So far, all we can do is form function types, but we don't even know how to make functions, that is, terms of $A \to B$. The **introduction rule** says that to specify a term of $A \to B$, it suffices to specify a term $b : B$ for every $x : A$. Formally, this is the following rule:

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : A \to B}\ \lambda\text{-INTRO}$$

Now we have terms of $A \to B$, but we have no rules that allow us to use them. Such a rule is called an **elimination rule**. For the function type, the elimination rule says that given a function $f : A \to B$ and a term $a : A$, we can form the term $f(a) : B$. Formally, this is the following rule:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B}\ \lambda\text{-ELIM}$$

The **computation** rule tells us what happens when the elimination rule is applied to a result of the introduction rule:

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.b)(a) \equiv b[a/x] : B}\ \lambda\text{-COMP}$$

where $b[a/x]$ is the term obtained from b by substituting a for every x.

In the case of function types, we also have what's known as a **uniqueness rule**:

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f \equiv \lambda x.f(x) : A \to B}\ \lambda\text{-UNIQ}$$

Actually, each of the rules above comes with a rule stating how it interacts with judgmental equality. For example, we have the following rule:

$$\frac{\Gamma \vdash A \equiv A' : \mathtt{Type} \qquad \Gamma \vdash B \equiv B' : \mathtt{Type}}{\Gamma \vdash A \to B \equiv A' \to B' : \mathtt{Type}}\ \lambda\text{-FORM-EQ}$$

Since these rules are fairly innocuous, we won't mention them anymore.

2.1. **Informal description.** In essence, our rules amount the following:

- For any types A and B, there is a type $A \to B$, called the **function type**. A term of $A \to B$ is called a **function**.
- Defining a function $f : A \to B$ is equivalent to defining a term $f(x) : B$ for every $x : A$.

2.2. **Interpretation.**

- For sets, this represents the set of functions from A to B.
- For spaces, this represents a space of functions from A to B (which is a bit vague).
- For propositions, this represents "A implies B", since having a proof of $A \to B$ is equivalent to having a proof of B for every proof of A.

2.3. **Example: the identity function.** To compare the informal approach with the formal one, we'll construct the identity function of a type A.

According to our informal description, we can define it as follows:

$$\mathrm{id}_A(a) :\equiv a$$

More formally, we would define $\mathrm{id}_A :\equiv \lambda x.x$, and the computation rule would show that $\mathrm{id}_A(a) \equiv a$.

Even more formally, we would introduce the following inference rule:

$$\frac{\Gamma \vdash A : \mathtt{Type}}{\Gamma \vdash \mathrm{id}_A \equiv \lambda x.x : A \to A}\ \text{id-DEF}$$

and we would derive

$$A : \mathtt{Type}, a : A \vdash \mathrm{id}_A(a) \equiv a$$

via **proof trees** as follows.

First we could derive $A : \mathtt{Type}, a : A \vdash (\lambda x.x)(a) \equiv a : A$ as shown below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\ \mathtt{ctx}}\ \textsc{ctx-emp}}{(A : \mathtt{Type})\ \mathtt{ctx}}\ \textsc{ctx-form}}{A : \mathtt{Type} \vdash A : \mathtt{Type}}\ \textsf{Vble}}{(A : \mathtt{Type}, a : A)\ \mathtt{ctx}}\ \textsc{ctx-ext}}{A : \mathtt{Type}, a : A \vdash A : \mathtt{Type}}\ \textsf{Vble}}{(A : \mathtt{Type}, a : A, x : A)\ \mathtt{ctx}}\ \textsc{ctx-ext}}{A : \mathtt{Type}, a : A, x : A \vdash x : A}\ \textsf{Vble} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\ \mathtt{ctx}}\ \textsc{ctx-emp}}{(A : \mathtt{Type})\ \mathtt{ctx}}\ \textsc{ctx-form}}{A : \mathtt{Type} \vdash A : \mathtt{Type}}\ \textsf{Vble}}{(A : \mathtt{Type}, a : A)\ \mathtt{ctx}}\ \textsc{ctx-ext}}{A : \mathtt{Type}, a : A \vdash a : A}\ \textsf{Vble}$$

$$\dfrac{\qquad\qquad}{A : \mathtt{Type}, a : A \vdash (\lambda x.x)(a) \equiv a : A}\ \lambda\textsc{-comp}$$

Then we could derive $A : \mathtt{Type}, a : A \vdash \mathrm{id}_A(a) \equiv (\lambda x.x)(a) : A$ as shown below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\ \mathtt{ctx}}\ \textsc{ctx-emp}}{(A : \mathtt{Type})\ \mathtt{ctx}}\ \textsc{ctx-form}}{A : \mathtt{Type} \vdash A : \mathtt{Type}}\ \textsf{Vble}}{(A : \mathtt{Type}, a : A)\ \mathtt{ctx}}\ \textsc{ctx-ext}}{A : \mathtt{Type}, a : A \vdash A : \mathtt{Type}}\ \textsf{Vble}}{A : \mathtt{Type}, a : A \vdash \mathrm{id}_A \equiv \lambda x.x : A \to A}\ \textsc{id-def} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\ \mathtt{ctx}}\ \textsc{ctx-emp}}{(A : \mathtt{Type})\ \mathtt{ctx}}\ \textsc{ctx-form}}{A : \mathtt{Type} \vdash A : \mathtt{Type}}\ \textsf{Vble}}{(A : \mathtt{Type}, a : A)\ \mathtt{ctx}}\ \textsc{ctx-ext}}{A : \mathtt{Type}, a : A \vdash a : A}\ \textsf{Vble}$$

$$\dfrac{\qquad\qquad}{A : \mathtt{Type}, a : A \vdash \mathrm{id}_A(a) \equiv (\lambda x.x)(a) : A}\ \lambda\textsc{-elim-eq}$$

Finally, we could put the two trees together to derive the desired judgment:

$$\dfrac{\begin{array}{c}\vdots\\ A : \mathtt{Type}, a : A \vdash \mathrm{id}_A(a) \equiv (\lambda x.x)(a) : A\end{array} \qquad \begin{array}{c}\vdots\\ A : \mathtt{Type}, a : A \vdash (\lambda x.x)(a) \equiv a : A\end{array}}{A : \mathtt{Type}, a : A \vdash \mathrm{id}_A(a) \equiv a}\ \equiv\textsc{-trans}$$

This is extremely tedious, so we will stick to the informal approach from now on.

Note that the fact that we can always construct the identity function corresponds to the fact that the proposition $A \to A$ always holds.

3. Dependent function types (aka Π-types)

There is a dependent version of the function type, where the target type depends on the input.

3.1. Informal description.

- Given a type A, and a type $B(x)$ for every $x : A$, there is a type $\prod_{x:A} B(x)$, called the **dependent function type** or **Π-type** (read "pi-type"). A term of $\prod_{x:A} B(x)$ is called a **dependent function**.
- Defining a dependent function $f : \prod_{x:A} B(x)$ is equivalent to defining a term $f(x) : B(x)$ for every $x : A$.

3.2. Interpretation.

- For sets, this is the product of a family $B(x)$ of sets indexed by $x : A$ (hence Π-type).
- For spaces, this is the space of sections of a fibration with base space A and fiber $B(x)$ over $x : A$.
- For propositions, this is a universal quantifier over $x : A$ (a term of $\prod_{x:A} B$ is a proof of $\forall x : A[B(x)]$).

3.3. Inference rules.

$$\dfrac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma, x : A \vdash B : \mathtt{Type}}{\prod_{x:A} B : \mathtt{Type}}\ \Pi\textsc{-form}$$

$$\dfrac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : \prod_{x:A} B}\ \Pi\textsc{-intro}$$

$$\dfrac{\Gamma \vdash f : \prod_{x:A} B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[a/x]}\ \Pi\textsc{-elim}$$

$$\dfrac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.b)(a) \equiv b[a/x] : B[a/x]}\ \Pi\textsc{-comp}$$

$$\frac{\Gamma \vdash f : \prod_{x:A} B}{\Gamma \vdash f \equiv \lambda x.f(x) : A \to B}\ \Pi\text{-UNIQ}$$

4. Product types

4.1. **Informal description.**

- Given types A and B, there is a type $A \times B$, called the **product type**.
- Given $a : A$ and $b : B$, there is a term $(a, b) : A \times B$.
- To define a (dependent) function f from $A \times B$, it suffices to define $f((a, b))$ for every $a : A$ and $b : B$.

This is like a Π-type over the "two-element type" (which we haven't defined yet).

4.2. **Interpretation.** The interpretation is as follows:

- For sets and spaces, this is the usual product.
- For propositions, this is $\wedge$. Given a proof of A and a proof of B, there is a proof of $A \times B$.

4.3. **Examples.** We can define the projection $\mathrm{pr}_1 : A \times B \to A$ via

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

and similarly for $\mathrm{pr}_2 : A \times B \to B$.

4.4. **Inference rules.**

$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma \vdash B : \mathtt{Type}}{\Gamma \vdash A \times B : \mathtt{Type}}\ \times\text{-FORM}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}\ \times\text{-INTRO}$$

$$\frac{\Gamma, p : A \times B \vdash C : \mathtt{Type} \qquad \Gamma, x : A, y : B \vdash c : C[(x, y)/p] \qquad \Gamma \vdash q : A \times B}{\Gamma \vdash \mathrm{ind}_{A\times B}(p.C, x.y.c, q) : C[q/p]}\ \times\text{-ELIM}$$

$$\frac{\Gamma, p : A \times B \vdash C : \mathtt{Type} \qquad \Gamma, x : A, y : B \vdash g : C[(x, y)/p] \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathrm{ind}_{A\times B}(p.C, x.y.g, (a, b)) \equiv g[a/x, b/y] : C[(a, b)/p]}\ \times\text{-COMP}$$

5. The unit type

This is the nullary product.

5.1. **Informal description.**

- There is a type $\mathbf{1}$, called the **unit type**.
- There is a term $\star : \mathbf{1}$.
- To define a (dependent) function f from $\mathbf{1}$, it suffices to define $f(\star)$.

5.2. **Interpretation.** In line with the above interpretations of Π-types and product types, we have the following interpretations:

- For sets, $\mathbf{1}$ is a singleton.
- For spaces, $\mathbf{1}$ is the one-point space.
- For propositions, $\mathbf{1}$ is "true" (sometimes denoted $\top$).

5.3. **Inference rules.**

$$\frac{\Gamma\ \mathtt{ctx}}{\Gamma \vdash \mathbf{1} : \mathtt{Type}}\ \mathbf{1}\text{-FORM}$$

$$\frac{\Gamma\ \mathtt{ctx}}{\Gamma \vdash \star : \mathbf{1}}\ \mathbf{1}\text{-INTRO}$$

$$\frac{\Gamma, x : \mathbf{1} \vdash C : \mathtt{Type} \qquad \Gamma, y : \mathbf{1} \vdash c : C[y/x] \qquad \Gamma \vdash a : \mathbf{1}}{\Gamma \vdash \mathrm{ind}_{\mathbf{1}}(x.C, y.c, a) : C[a/x]}\ \mathbf{1}\text{-ELIM}$$

$$\frac{\Gamma, x : \mathbf{1} \vdash C : \mathtt{Type} \qquad \Gamma, y : \mathbf{1} \vdash c : C[y/x]}{\Gamma \vdash \mathrm{ind}_{\mathbf{1}}(x.C, y.c, \star) \equiv C[\star/y] : C[\star/x]}\ \mathbf{1}\text{-COMP}$$

6. Dependent pair types (aka Σ-types)

6.1. **Informal description.**

- Given a type A, and a type $B(x)$ for every $x : A$, there is a type $\sum_{x:A} B(x)$, called the **dependent pair type** or **Σ-type** (read "sigma-type").
- Given $a : A$ and $b : B(a)$, there is a term $(a, b) : \sum_{x:A} B(x)$.
- To define a (dependent) function f from $\sum_{x:A} B$, it suffices to define $f((a, b))$ for every $a : A$ and $b : B(a)$.

6.2. **Interpretation.**

- For sets, this is the disjoint union of a family $B(x)$ of sets indexed by $x : A$.
- For spaces, this is the total space of a fibration with base space A and fiber $B(x)$ over A.
- For propositions, this is an existential quantifier over $x : A$ (a term of $\sum_{x:A} B$ is a proof of $\exists x : A[B(x)]$).

6.3. **Inference rules.**

$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma, x : A \vdash B : \mathtt{Type}}{\Gamma \vdash \sum_{x:A} B : \mathtt{Type}}\ \Sigma\text{-FORM}$$

$$\frac{\Gamma, x : A \vdash B : \mathtt{Type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \sum_{x:A} B}\ \Sigma\text{-INTRO}$$

$$\frac{\Gamma, z : \sum_{x:A} B \vdash C : \mathtt{Type} \qquad \Gamma, x : A, y : B \vdash g : C[(x, y)/z] \qquad \Gamma \vdash p : \sum_{x:A} B}{\Gamma \vdash \mathrm{ind}_{\sum_{(x:A)} B}(z.C, x.y.g, p) : C[p/z]}\ \Sigma\text{-ELIM}$$

$$\frac{\Gamma, z : \sum_{x:A} B \vdash C : \mathtt{Type} \qquad \Gamma, x : A, y : B \vdash g : C[(x, y)/z] \qquad \Gamma \vdash a' : A \qquad \Gamma \vdash b' : B[a'/x]}{\Gamma \vdash \mathrm{ind}_{\sum_{(x:A)} B}(z.C, x.y.g, (a', b')) \equiv g[a', b'/x, y] : C[(a', b')/z]}\ \Sigma\text{-COMP}$$

7. Coproduct types

7.1. **Informal description.**

- Given types A and B, there is a type $A + B$, called the **coproduct type**.
- Given $a : A$, there is a term $\mathrm{inl}(a) : A + B$ (inl stands for "in left").
- Given $b : B$, there is a term $\mathrm{inr}(b) : A + B$.
- To define a (dependent) function f from $A + B$, it suffices to define $f(\mathrm{inl}(a))$ and $f(\mathrm{inl}(b))$ for every $a : A$ and $b : B$.

7.2. **Interpretation.**

- For sets and spaces, this is the usual disjoint union.
- For propositions, this is $\vee$. A proof of $A + B$ is either a proof of A or a proof of B.

7.3. **Inference rules.** Similar to those of Σ-types.

8. The empty type

This is the nullary coproduct.

8.1. **Informal description.**

- There is a type $\mathbf{0}$, called the **empty type**.
- For any type A, there is a function $f : \mathbf{0} \to A$.
- More generally, given a type $A(x)$ for every $x : \mathbf{0}$, there is a dependent function $f : \prod_{x:\mathbf{0}} A(x)$.

8.2. **Intepretation.** Specializing the previous interpretations of Σ-types and coproduct types gives the following interpretations of $\mathbf{0}$:

- For sets, $\mathbf{0}$ is the empty set.
- For spaces, $\mathbf{0}$ is the empty space.
- For propositions, $\mathbf{0}$ is "false" (sometimes denoted $\bot$). The fact that there is a function $\mathbf{0} \to A$ for any A means that given a proof of false, one can construct a proof of any proposition, ie. "false implies anything" (known as the *principle of explosion* or *ex falso quodlibet*).

8.3. **Example.** We can now define the negation $\neg A$ of a proposition A as follows:

$$\neg A :\equiv A \to \mathbf{0}$$

In other words, the negation of proposition holds if a proof of the proposition implies false.

8.4. **Inference rules.** Note that there is **no** introduction rule (and consequently no computation rule either).

$$\frac{\Gamma\ \mathtt{ctx}}{\Gamma \vdash \mathbf{0} : \mathtt{Type}}\ \mathbf{0}\text{-FORM}$$

$$\frac{\Gamma, x : \mathbf{0} \vdash C : \mathtt{Type} \qquad \Gamma \vdash a : \mathbf{0}}{\Gamma \vdash \mathrm{ind}_{\mathbf{0}}(x.C, a) : C[a/x]}\ \mathbf{0}\text{-ELIM}$$

9. The Boolean type

This is the "two-element type" mentioned above.

9.1. **Informal description.**

- There is a type $\mathbf{2}$ (sometimes also denoted $\mathtt{Bool}$), called the **Boolean type** or the **type of booleans**. A term of $\mathbf{2}$ is called a **boolean**.
- There are terms $0, 1 : \mathbf{2}$.
- To define a (dependent) function f from $\mathbf{2}$, it suffices to define $f(0)$ and $f(1)$.

Note that $\mathbf{1} + \mathbf{1}$ is also a perfectly good candidate for a Boolean type. This raises the question: how is $\mathbf{2}$ related to $\mathbf{1} + \mathbf{1}$? Obviously they're not judgmentally equal, but perhaps they're "isomorphic" in some way. Unfortunately, our type theory is not strong enough to prove this yet, but we'll see later that we can use univalence to show that these types are propositionally equal.

10. The natural numbers

10.1. **Informal description.**

- There is a type $\mathbb{N}$, called **the natural numbers**. A term of $\mathbb{N}$ is called a **natural number**.
- There is a term $0 : \mathbb{N}$ and a function $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$.
- To define a function f from $\mathbb{N}$, it suffices to define $f(0)$, and to define $f(\mathsf{succ}(n))$ in terms of $f(n)$.
- More precisely, given $f(0) : A(0)$ and $g : \prod_{n:\mathbb{N}} A(n) \to A(\mathsf{succ}(n))$, there is a dependent function $f : \prod_{n:\mathbb{N}} A(n)$ satisfying $f(\mathsf{succ}(n)) \equiv g(n)(f(n))$.

10.2. **Examples.**

- We define $\mathsf{double} : \mathbb{N} \to \mathbb{N}$ as follows:

$$\mathsf{double}(0) :\equiv 0$$
$$\mathsf{double}(\mathsf{succ}(n)) :\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{double}(n)))$$

  More precisely, we define $\mathsf{double}(0) :\equiv 0$ and $g(n)(m) :\equiv \mathsf{succ}(\mathsf{succ}(m))$, and this determines a function $\mathsf{double} : \mathbb{N} \to \mathbb{N}$ such that $\mathsf{double}(\mathsf{succ}(n)) \equiv g(n)(\mathsf{double}(n)) \equiv \mathsf{succ}(\mathsf{succ}(\mathsf{double}(n)))$. We will take the first approach from now on.
- We define $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ as follows:

$$0 + m :\equiv m$$
$$\mathsf{succ}(n) + m :\equiv \mathsf{succ}(n + m)$$

  This is really shorthand for the following:

$$0+ :\equiv \mathrm{id}_{\mathbb{N}}$$
$$\mathsf{succ}(n)+ :\equiv \mathsf{succ} \circ (n+)$$

11. Identity types

11.1. **Informal description.**

- Given a type A and term $a, b : A$, there is a type $a = b$, called the **identity type**. A term of $a = b$ is called a **path** (from a to b). If $a = b$ is inhabited, we will often say that a and b are **propositionally equal**.
- Given $a : A$, there is a term $\mathrm{refl}_a : a = a$.
- To define a function $f : \prod_{x,y:A}((x = y) \to B)$, it suffices to define $f(x, x, \mathrm{refl}_x) : B$ for every $x : A$. This is known as **path induction**. More generally, path induction says that to define a dependent function $f : \prod_{x,y:A} \prod_{p:x=y} B(p)$, it suffices to define $f(x, x, \mathrm{refl}_x) : B(\mathrm{refl}_x)$ for every $x : A$.
- There is also another induction principle, known as **based path induction**. It says that given $a : A$, to define a function $f : \prod_{x:A}((a = x) \to B)$, it suffices to define $f(a, \mathrm{refl}_a) : B$. There is also an analogous dependent version.

In fact, path induction and based path induction are equivalent, so we only need to include one of them in our type theory.

11.2. **Interpretation.**

- For spaces, $a = b$ is the space of paths from a to b.
- For propositions, $a = b$ is the proposition "a is equal to b".

11.3. **Examples.** We can now state propositions such as the following:

Proposition 1. *Let A and B be types, let $f : A \to B$ and let $x, y : A$. If $x = y$, then $f(x) = f(y)$.*

What does it mean to state this proposition? Recall that a proposition is just a type, and in this case, it is the following type:

$$\prod_{A,B:\mathsf{Type}} \prod_{f:A\to B} \prod_{x,y:A} ((x = y) \to (f(x) = f(y)))$$

The "proof" of the proposition will be the explicit construction of a term of this type.

Proof. By path induction, it suffices to assume $x = x$. But we always have $f(x) = f(x)$, so we are done. □

We can use this to prove the following fact about $\mathbb{N}$.

Proposition 2. *For every $n : \mathbb{N}$, we have $n + 0 = n$.*

Proof. We proceed by induction on n.

For $n \equiv 0$, we have $0 + 0 \equiv 0 = 0$.

Now suppose that $n + 0 = n$. Then $\mathsf{succ}(n + 0) = \mathsf{succ}(n)$ by the above proposition, and thus $\mathsf{succ}(n) + 0 \equiv \mathsf{succ}(n + 0) \equiv \mathsf{succ}(n)$. □

Note that we only need the implication $(n + 0 = n) \to (\mathsf{succ}(n + 0) = \mathsf{succ}(n))$, but to get this, we need to prove the more general proposition $(x = y) \to (\mathsf{succ}(x) = \mathsf{succ}(y))$, since we need to apply path induction. This is a common theme when working with the identity type.