university of groningen / faculty of science and engineering / mathematics and applied mathematics

Type Theory and its Homotopical Interpretation

Bachelor's Project Mathematics

June 2025

Student: A.R. Matei

First supervisor: Prof. M. Seri

Second assessor: PhD. O. Lorscheid

Abstract

Martin Löf's Type Theory is a constructive formal system that can serve as the foundation of mathematics, offering an alternative to Zermelo–Fraenkel set theory. This thesis intends to be a tour of Martin Löf's Type Theory, aimed at people without a background in the topic. We present the core inference rules of the formal system, constructing a series of important types, and offering a first definition of the integers. Attempting to produce an alternative construction of the integers surfaces some missing pieces in our system, leading to the exploration of core ideas from Homotopy Type Theory and univalent mathematics. Finally, we use these tools to produce a second construction of the integers, and show that our definitions are homotopically equivalent.

1 Introduction

Martin Löf's Type Theory is a formal system built around collections called *types*. Said formal system can be used as the foundation of mathematics, offering an alternative to Zermelo–Fraenkel set theory. While one can think of *types* in a somewhat similar way to sets (as collections of various elements), the concepts do differ. For one, a value x in type theory always come bundled up with an associated type T, which we denote by $x : T$ (for example, $0 : \mathbb{N}$). Unlike set theory, where everything is conceptualised as a set, types and their elements live in somewhat separate worlds, although types themselves can be treated as values as part of so-called *universes*.

Set theory is usually formalized in the language of *first order logic*, while Martin Löf's Type Theory acts as its own stand-alone formal system. In particular, propositions are represented by a given kind of types, inhabited by the proofs of said propositions. Moreover, the fact proofs are represented by values means they can be passed around by functions like any other element. For example, we can represent proofs of the fact that $d|n$ (for $d, n : \mathbb{N}$) as a pair containing a natural $m : \mathbb{N}$ and a proof that $m \cdot d = n$.

The differences don't end here. Sets are characterised by nothing but the $\in$ relation. On the other hand, we introduce types by providing a series of ways to construct elements of said type, and a way to use up said elements (often referred to as the "induction principle" for the type).

The presence of equality types (i.e. the types of proofs that $a = b$ for $a, b : T$) is an important feature of Martin Löf's Type Theory. The natural question is how equality between types behaves. The *univalence principle* gives us an identification between two types being equal (in some sense), and them being equivalent. As an example, since we can construct a trivial equivalence between any two types containing precisely one element, the two types can be identified. This stands in stark contrast with set theory, where there exists a myriad of unique singleton sets (i.e. sets with a single element).

This thesis is intended to be a tour of Martin Löf's Type Theory, aimed at people without a background in the topic. We begin with section 2, where we introduce the formal system and its various inference rules, and define the so-called *dependent function types*, which are a more general kind of functions, where the codomain is allowed to vary based on the argument.

Then, section 3 introduces the natural numbers, together with their familiar induction principle, and operations like addition and multiplication. Section 4 then applies the same procedure we used to define the naturals, in order to define a myriad of other types, culminating in the definition of dependent pairs (a more general form of tuples, where the second element's domain is allowed to vary over the value of the first element), and finally, the integers.

Finally, we touch upon identity types (in some sense, the type of proofs that two elements are equal), and their homotopical interpretation. Indeed, it turns out one can think of identity types as paths on a space, carrying over many concepts from homotopy theory. We use this idea to provide a path-based definition of the integers, and construct a proof that our two definitions are homotopically equivalent. We will use the construction of the integers as our main through line into the theory, elucidating which kinds of constructions require the stronger machinery offered by the homotopical interpretation.

2 Martin Löf's Type Theory

This section attempts to outline the rules governing the formal system we'll be using throughout this thesis. Our system will be similar to the one described in [5, Appendix A.2]. We proceed by introducing a series of *inference rules* — distinct steps we are allowed to take while constructing a proof. When writing out such inference rules, the required hypotheses are written above the so-called *judgment line*, with the conclusion written underneath, and an optional rule name on the right:

$$\frac{\text{Hypothesis 1} \qquad \text{Hypothesis 2} \qquad \ldots}{\text{Conclusion}}\ \text{example-rule}$$

The individual hypotheses & conclusions take the form of various kinds of *judgments*. For example, a judgment might affirm that some T is a type, or that some variable a has a given type $a : T$. Such judgments do not live in a vacuum — they always exist within a *context*, which we often denote by Γ or Δ. A context is an ordered list of bindings

$$x_1 : A_1, x_2 : A_2, x_3 : A_3, \cdots, x_n : A_n.$$

The names x_i must be unique, and each type A_i is allowed to reference the variables defined before it. The context contains every variable we have access to throughout a proof, and gets tweaked by every inference rule we apply. Most of the time when doing mathematics, the context itself remains implicit, and although that's also the case throughout most of this thesis, we will explicitly carry around contexts when specifying the inference rules of the system.

Our theory has three kinds of judgments:

1. The judgment
$$\Gamma \vdash a : A$$
states that a has type A (in the context Γ). Intuitively, this corresponds to the $\in$ relation in set theory. Unlike said $\in$ relation, this judgment is not internal to the language of type theory. That is, although a statement like "If $2 \in P$ then $2 \in Q$" would make sense when talking about sets, one cannot state the direct analogue within our theory[1].

   Unlike set theory, where everything is a set, not everything is a type in our theory. For example, the natural numbers $\mathbb{N}$ form a type (which we will look at in section 3), yet the individual naturals (0, 1, and so on) are themselves not types.

   We thus need a judgment that affirms that some given A is a type. One solution would be to introduce some "type of all types" $\mathcal{U}$, with $\mathcal{U} : \mathcal{U}$. Doing so would introduce *Girad's paradox* (originally proven in [2]) — a type theoretic equivalent of *Russel's paradox*. Intuitively, this is reminiscent of there being no such thing as a "set of all sets" in ZFC.

   Instead, we postulate an infinite hierarchy of universes. That is, instead of a single "type of all types" $\mathcal{U}$, we introduce universes $\mathcal{U}_0, \mathcal{U}_1, \cdots$ with
$$\frac{}{\Gamma \vdash \mathcal{U}_l : \mathcal{U}_{l+1}}\ \mathcal{U}\text{-intro}$$

[1] Of course, one can express the same statement in Martin Löf's Type Theory, but in a different looking form (i.e. not in terms of the : judgment). For instance, the statement "If $2 \in P$ then $2 \in Q$" could get encoded as $P(2) \to Q(2)$, which we will see in action a bit further below.

This hierarchy is *cumulative*, in the sense that every type in $\mathcal{U}_l$ is also a type in the following universe, i.e. $\mathcal{U}_{l+1}$:

$$\frac{\Gamma \vdash A : \mathcal{U}_l}{\Gamma \vdash A : \mathcal{U}_{l+1}}\,\mathcal{U}\text{-cumul}$$

As an example, section 4.1 will introduce the type $\text{list}(T)$ of ordered lists with elements of type T. One would begin doing so via the following rule:

$$\frac{\Gamma \vdash T : \mathcal{U}_l}{\Gamma \vdash \text{list}(T) : \mathcal{U}_l}\,\text{list-form}$$

Intuitively, this rule can be read as saying that, given a type T that is well-formed in some universe $\mathcal{U}_l$, then the type $\text{list}(T)$ is also well-formed in the same universe. Note, on the other hand, that this doesn't offer us any way to conclude that "list" itself is a type — it is not, unless given an argument.

2. Next, the judgment

$$\Gamma \vdash a \doteq b : A$$

denotes $a : A$ and $b : A$ as being *judgmentally equal* (also known as *definitionally equal*) within the context Γ. Judgmental equality formalizes the idea of things being equal "by definition".

Definitional equality encodes the fundamental mechanics of the underlying theory. For example, let $\text{id}_{\mathbb{N}}$ be the identity function on the naturals. Intuitively, one might say that $\text{id}_{\mathbb{N}}(5)$ *evaluates* to 5. Within our theory, this would be stated as

$$\Gamma \vdash \text{id}_{\mathbb{N}}(5) \doteq 5 : \mathbb{N}.$$

Note that this judgment is, again, external to our theory. That is, it makes no sense for us to state "if $a \doteq b$ then ..." within the language of Martin Löf's Type Theory. For this reason, Martin Löf's Type Theory does not provide us with any tools for proving non-trivial definitional equalities. That is, something like $2 + 3 \doteq 3 + 2$ would hold by the definition of 2, 3, and $+$, yet one wouldn't be able to prove that $a + b \doteq b + a$ for all $a, b : \mathbb{N}$. In section 5, we will introduce a new form of equality (identity types), which is provable within the language itself.

We postulate that $\doteq$ is an equivalence relation:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \doteq a : A}\,\doteq\text{-refl} \qquad\qquad \frac{\Gamma \vdash a \doteq b : A}{\Gamma \vdash b \doteq a : A}\,\doteq\text{-sym}$$

$$\frac{\Gamma \vdash a \doteq b : A \qquad \Gamma \vdash b \doteq c : A}{\Gamma \vdash a \doteq c : A}\,\doteq\text{-trans}$$

Next, if A and B are judgmentally equal and $a : A$, then $a : B$:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash A \doteq B : \mathcal{U}_l}{\Gamma \vdash a : B}$$

We introduce an analogous rule for the $\doteq$ judgment:

$$\frac{\Gamma \vdash a \doteq b : A \qquad \Gamma \vdash A \doteq B : \mathcal{U}_l}{\Gamma \vdash a \doteq b : B}$$

3. The third (and final) kind of judgment we introduce states that some Γ is a valid context, and is written

$$\Gamma \ \mathbf{ctx}\,.$$

We only introduce two rules for constructing contexts.

- First, we have the empty context (denoted by $\cdot$):

$$\frac{}{\cdot \ \mathbf{ctx}} \text{ nil-ctx}$$

- Next, we can append things at the end of a context:

$$\frac{x_1 : A_1, \cdots, x_n : A_n \vdash A_{n+1} : \mathcal{U}_l}{x_1 : A_1, \cdots, x_{n+1} : A_{n+1} \ \mathbf{ctx}} \text{ cons-ctx}$$

Throughout this thesis, we will implicitly assert that every context used is well formed.

Having defined the underlying judgments, it is now time for us to go over the formal rules of the system. First, we postulate that contexts do indeed hold variables that can be referenced:

$$\frac{x_1 : A_1, ..., x_n : A_n \ \mathbf{ctx}}{x_1 : A_1, ..., x_n : A_n \vdash x_i : A_i} \text{ var}$$

Note that we will use Γ, Δ to denote the context obtained by concatenating the bindings in Γ and Δ. For example, our previous rule could be rewritten as:

$$\frac{\Gamma, x : A, \Delta \ \mathbf{ctx}}{\Gamma, x : A, \Delta \vdash x : A} \text{ var}$$

A question one might ask is what a "type" or "value" even is in the first place. Of course, an unsatisfactory definition for a type T is "a value such that $T : \mathcal{U}_l$", since this simply begs the question of what a value is. Our values are formal expressions that fall in one of a few categories, notably:

1. Variables bound in context, denoted by x, y, etc.
2. Constants referencing earlier derivations, examples of which include things like $\mathbb{N}$, id, etc.
3. Function introductions and applications, which we will introduce in section 2.2.

We will encounter the various kinds of values as we work our way to defining more constructs.

2.1 Weakening & Substitution

Next, we will introduce the so-called *weakening* and *substitution* rules, which allow us to expand/shrink the underlying context.

We postulate that we can introduce additional "unused" variables in the context while preserving our other conclusions[2]. For instance, if $b : B$ in some context Γ, Δ and A is a valid type in Γ, then $b : B$ still holds in a context with $x : A$ inserted[3] between Γ and Δ. We call this process *weakening* the context:

$$\frac{\Gamma \vdash A : \mathcal{U}_l \qquad \Gamma, \Delta \vdash b : B}{\Gamma, x : A, \Delta \vdash b : B}\ \text{:-wkg} \qquad \frac{\Gamma \vdash A : \mathcal{U}_l \qquad \Gamma, \Delta \vdash a \doteq b : B}{\Gamma, x : A, \Delta \vdash a \doteq b : B}\ \doteq\text{-wkg}$$

We will denote the term produced by replacing every occurrence of x in e with a by

$$e[a/x].$$

For instance, given that $b : B$ in a context containing $x : A$ (i.e. $\Gamma, x : A, \Delta$), and we have some $a : A$ which holds in the same context, except with $x : A$ removed (i.e. Γ, Δ), we can conclude that $b : B$ would hold in said smaller context as well, if we were to replace every occurrence of x in b and B with a. Moreover, we also have to apply the substitution to every binding in the context occurring after $x : A$ (i.e. to every binding in Δ), since bindings inside the context can depend on earlier bindings. Formally:

$$\frac{\Gamma, \Delta \vdash a : A \qquad \Gamma, x : A, \Delta \vdash b : B}{\Gamma, \Delta[a/x] \vdash b[a/x] : B[a/x]}\ \text{:-subst}$$

We introduce an analogous rule for the case when, instead of $b : B$, we have $b \doteq c : B$:

$$\frac{\Gamma, \Delta \vdash a : A \qquad \Gamma, x : A, \Delta \vdash b \doteq c : B}{\Gamma, \Delta[a/x] \vdash b[a/x] \doteq c[a/x] : B[a/x]}\ \doteq\text{-subst}$$

Finally, if $a \doteq c$, then replacing every occurrence of x in $b : B$ with a and c respectively should produce judgmentally equal values:

$$\frac{\Gamma, \Delta \vdash a \doteq c : A \qquad \Gamma, x : A, \Delta \vdash b : B}{\Gamma, \Delta[a/x] \vdash b[a/x] \doteq b[c/x] : B[a/x]}\ \doteq\text{-subst}_2$$

2.2 Π-types

Having introduced the basic structure of the theory, it is now time for us to construct some of its fundamental types. Defining types in Martin Löf's Type Theory requires a few kinds of rules:

1. *Formation rules* tell us when the type is well defined. For instance, we'll see in the first rule of (3) below that $\mathbb{N}$ is a well defined type in any context.

2. *Introduction rules* provide us with ways of creating concrete values of the types formed by the formation rules. For example, one of the introduction rules of $\mathbb{N}$ (namely the second rule of (3)) introduces the constant $0_{\mathbb{N}} : \mathbb{N}$ into existence.

[2] As noted by [5, A.2.2], the weakening and substitution rules we introduce are not technically needed. One could prove by induction on all the possible derivations that whenever these rules' hypotheses are derivable, then so is their conclusion, although this is outside our scope.

[3] As stated above, the names of the bindings contained in a context need to be distinct. Type theory papers often denote this requirement explicitly by $x \notin \Gamma, \Delta$ or $x \notin \mathrm{dom}(\Gamma, \Delta)$, although we will always implicitly assume this to be the case throughout our thesis.

3. *Elimination rules* provide us with ways of making use of the values of the types introduced by the introduction rules. For example, the elimination rule for functions (see rule (1)) allows us to evaluate said functions at a given argument.

4. *Computation rules* provide us with the ways the introduction and elimination rules interact via judgmental equalities. We'll see this in action towards the end of this section for the aforementioned idea of function evaluation.

Martin Löf's Type Theory is a dependent type theory. The word *dependent* alludes to the fact types can refer (i.e. "depend on") values. For example, one can construct a type $\mathrm{Vec}\,(n, T)$ of n-tuples with elements of type T (that is, what we would traditionally write out as T^n). The $n : \mathbb{N}$ in this expression is a concrete natural number (i.e. it is not a type), yet the type $\mathrm{Vec}\,(n, T)$ depends on it.

One important class of such dependent types are *dependent functions*, also known as Π-types, *dependent maps*, or *dependent products*. Unlike maps $A \to B$ (as traditionally defined in set theory), the codomain of a dependent function is not fixed, but can depend on the concrete value of the argument. For example, one can define a function which, given any natural $n : \mathbb{N}$, produces an element of type is-even$(2n)$ (the type of proofs that $2n$ is indeed even. See equation (5) for the proper definition of is-even).

To express this more formally, we need to introduce the idea of *type families*. Given some type $A : \mathcal{U}$ (we will worry about universe levels later), a type family P can be thought of as a map[4] $P : A \to \mathcal{U}$. That is, this map assigns every element in A to a type. Given such a setup, we can construct the type of dependent functions from $a : A$ to $P(a)$. We denote this by

$$\prod_{x:A} P(x).$$

The application of P at a given element $a : A$, i.e. $P(a)$, is referred to as the *fiber* of P at a. This naming will make more sense once we talk about the homotopical interpretation of the construction in section 5.

If the family $P(x)$ is constant for all $x : A$, then we recover the usual concept of functions $A \to B$. That is, given types $A, B : \mathcal{U}$, we have

$$A \to B \coloneqq \prod_{x:A} B.$$

A bit of care must be taken when encoding the idea as a formal inference rule. That is, given a type $A : \mathcal{U}_l$ living in the universe of level l and a type family $P : A \to \mathcal{U}_k$, we will introduce the resulting dependent function type into the universe of level $\max(l, k)$:

$$\prod_{a:A} P(a) : \mathcal{U}_{\max(l,k)}.$$

Moreover, we don't yet have access to the $\to$ notation for type families inside inference rules, since we are yet to formally introduce any kind of function types. We will thus move the argument $a : A$ of $P(a)$ into the context, and simply assert that $P : \mathcal{U}_k$. Formally:

$$\frac{\Gamma \vdash A : \mathcal{U}_l \qquad \Gamma, a : A \vdash P : \mathcal{U}_k}{\Gamma \vdash \prod_{a:A} P : \mathcal{U}_{\max(l,k)}}\ \Pi\text{-form}$$

[4]Of course, we haven't yet introduced any kind of maps formally. With non-dependent maps being a special case of dependent maps, one might worry about the circularity of our explanation. Worry not, for we shall introduce things properly once we get to the formal rules defining these constructions.

The slight of hand we performed by pulling the argument of P into the context might look a bit confusing, so we will break it down:

1. First, we require some type A living in context Γ.
2. Next, we require a type P living in context $\Gamma, a : A$. That is, P can not only reference the existing bindings A is able to reference, but can reference an additional binding $a : A$.
3. Then, we conclude that $\prod_{a:A} P$ is a well defined type in Γ.

Every time we introduce a formation/introduction/elimination rule for a type, we will also introduce a local *congruence* rule that asserts that our new rule preserves definitional equalities. For example, for the rule above, we introduce the rule:

$$\frac{\Gamma \vdash A : \mathcal{U}_l \qquad \Gamma, a : A \vdash P, Q : \mathcal{U}_k \qquad \Gamma, a : A \vdash P \doteq Q : \mathcal{U}_k}{\Gamma \vdash \prod_{a:A} P \doteq \prod_{a:A} Q : \mathcal{U}_{\max(l,k)}}\ \Pi\text{-form-cong}$$

These rules are tedious to write down every time we introduce a new rule, and not particularly interesting, therefore we will omit explicitly writing them down, although know that they are indeed always there.

Next, we postulate an *introduction rule* that allows us to create values inhabiting the function types we've just defined. Notation-wise, we will use[5] $\lambda a.p$ to denote a function which, given the argument a, returns p (traditionally, one would write this as $a \mapsto p$). Continuing our previous example with $a : A$ and a family $P(a)$, we note that the p in $\lambda a.p$ is able to make references to a, and is required to have type $P(a)$. Formally,

$$\frac{\Gamma, a : A \vdash p : P}{\Gamma \vdash \lambda a.p : \prod_{a:A} P}\ \Pi\text{-intro}$$

Reading the rule in reverse shows that one can construct maps $\prod_{a:A} P$ by defining their return value once the argument has been moved into the context. This process is often referred to as λ-*abstraction*.

The elimination rule for dependent functions provides us with a way to evaluate said functions at any input. Given a function $f : \prod_{a:A} P(a)$ and some argument $x : A$, we will denote this function application by $f(a)$. Formally:

$$\frac{\Gamma \vdash f : \prod_{a:A} P \qquad \Gamma \vdash x : A}{\Gamma \vdash f(x) : P[x/a]}\ \Pi\text{-elim} \tag{1}$$

Intuitively, defining a function $\lambda a.p$ by λ-abstraction and then immediately applying it to some argument x should produce precisely the result obtained by substituting every occurrence of a in p with x. We encode this as a *computation rule*, expressing the result as a judgmental equality:

$$\frac{\Gamma, a : A \vdash p : P \qquad \Gamma \vdash x : A}{\Gamma \vdash (\lambda a.p)(x) \doteq p[x/a] : P[x/a]}\ \Pi\text{-comp}$$

The above rule is often referred to as β-*equivalence*, not to be confused with our final rule for Π-types — η-*equivalence*. In particular, η-equivalence acts as a sort of uniqueness

[5]This notation is used because at the core of our theory, we have a very special language called *the* λ-*calculus*.

rule for λ-abstraction. That is, any function $f : \prod_{a:A} P$ is judgmentally equal to a function defined via λ-abstraction (intuitively, any function can be thought of as being defined by λ-abstraction, since that is the only introduction rule we've postulated for Π-types). Seen another way, if β-equivalence allows us to collapse abstractions followed by applications into a judgmental equality, η-equivalence allows us to collapse applications followed by abstractions into a judgmental equality. Formally:

$$\frac{\Gamma \vdash f : \prod_{a:A} P}{\Gamma \vdash f \doteq \lambda x.f(x) : \prod_{a:A} P}\ \Pi\text{-uniq}$$

Non-dependent function correspond to functions in set theory. On the other hand, dependent functions correspond to generalized products (with the domain being the indexing set).

Moreover, non-dependent functions correspond to logical implications. That is, proving that some A implies B in type theory becomes a matter of constructing a map which, given some element of A, produces an element of B. Being able to evaluate a function $A \to B$ at an argument of type A in order to produce a value of type B then corresponds to *modus ponens*.

More interestingly, dependent functions can act as the type-theoretic correspondent of the universal quantifier $\forall$. That is, proving that $\forall x \in S.P(x)$ becomes a matter of constructing a map $\prod_{x:S} P(x)$.

2.3 The identity & function composition

We will put the rules introduced together in order to define a function

$$\mathrm{id} : \prod_{T:\mathcal{U}_l} T \to T.$$

The above type definition might seem a bit confusing. Indeed, we want our identity function to work for every type (within a certain universe, at least). To achieve this, we define a map which, given some type T, produces a new map $T \to T$ as a result. This process of defining "maps that produce maps" is our way of defining functions that take more than one argument. This process is called *currying*.

Although the proper notation for applying "id" would be, for example, $\mathrm{id}(\mathbb{N})(0_{\mathbb{N}})$, we allow shortening the above to $\mathrm{id}(\mathbb{N}, 0_{\mathbb{N}})$. In the case of type arguments, we will also often write them as subscripts, i.e. $\mathrm{id}_{\mathbb{N}}(0_{\mathbb{N}})$.

We start by constructing the desired function using the formal rules introduced above. We will do so by working backwards from the desired conclusion. That is, our goal is constructing the following:

$$\frac{\cdots}{\Gamma \vdash ? : \prod_{T:\mathcal{U}_l} T \to T}$$

Our only way of introducing new functions is by λ-abstraction. We thus apply the rule Π-intro twice (once for each argument):

$$\frac{\dfrac{\dfrac{\cdots}{\Gamma, T : \mathcal{U}_l, x : T \vdash ? : T}}{\Gamma, T : \mathcal{U}_l \vdash \lambda x.? : T \to T}\ \Pi\text{-intro}}{\Gamma \vdash \lambda T.\lambda x.? : \prod_{T:\mathcal{U}_l} T \to T}\ \Pi\text{-intro}$$

Note how applying the inference rules repeatedly yields a *derivation* (also known as a *proof tree*). We need to continue doing so until there's no unproven hypotheses left. Thankfully, we are one application of "var" away from this goal (since as stated earlier, we omit explicitly showing that each context is well constructed, as doing so is not a particularly interesting endeavour):

$$\dfrac{\dfrac{\dfrac{}{\Gamma, T : \mathcal{U}_l, x : T \vdash x : T}\;\text{var}}{\Gamma, T : \mathcal{U}_l \vdash \lambda x.x : T \to T}\;\Pi\text{-intro}}{\Gamma \vdash \lambda T.\lambda x.x : \prod_{T : \mathcal{U}_l} T \to T}\;\Pi\text{-intro}$$

Notation Remark. Constructing the formal derivation for every one of our definitions would get tedious very quickly, thus throughout this thesis, we will often define things using simpler syntax. That is, we would denote the definition above by

$$\begin{aligned} \mathrm{id} &: \prod_{T : \mathcal{U}_l} T \to T, \\ \mathrm{id}(T, x) &:= x. \end{aligned} \tag{2}$$

Formally, whenever we reference "id" at a latter point throughout the thesis, we are referencing its full derivation as part of the derivation we are currently constructing. Informally, we will simply reference "id" throughout our definitions as if it was a name in our context.

Next, we will define function composition (for non-dependent maps, that is). Given types A, B, C, we need to define a way to compose maps $A \to B$ and $B \to C$ into a map $A \to C$. Written as a type, we are defining an element

$$\mathrm{compose} : \prod_{A:\mathcal{U}_l} \prod_{B:\mathcal{U}_l} \prod_{C:\mathcal{U}_l} (A \to B) \to (B \to C) \to (A \to C).$$

Notation Remark. Writing out each type argument under a separate Π gets out of hand quickly, therefore we will allow defining more than one argument under a single Π-symbol (formally, this only acts as syntactic sugar over splitting the type into individual Π signs).

$$\mathrm{compose} : \prod_{A,\, B,\, C \,:\, \mathcal{U}_l} (A \to B) \to (B \to C) \to (A \to C).$$

The definition of this map looks like one would expect:

$$\mathrm{compose}(A, B, C, f, g) := \lambda x.g(f(x)).$$

Throughout this thesis, we will denote function composition with the usual $\circ$ notation, omitting explicitly specifying the types A, B, and C.

One would expect that $\mathrm{id}_Z(z) \doteq z$. Is this the case for our definition? It turns out this is indeed the case, although proving it using the formal inference rules is extremely tedious (do not worry about trying to parse this entire tree):

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\Gamma, Z : \mathcal{U}_l, z : Z, T : \mathcal{U}_l \vdash (\lambda x.x) : T \to T} \quad \dfrac{}{\Gamma, Z : \mathcal{U}_l, z : Z \vdash z : Z}\;\text{var}}{\Gamma, T : \mathcal{U}_l, z : Z \vdash (\lambda T.\lambda x.x)(Z) \doteq \lambda x.x : (Z \to Z)}\;\Pi\text{-comp} \quad \dfrac{}{\Gamma, Z : \mathcal{U}_l, z : Z \vdash z : Z}\;\text{var}}{\Gamma, T : \mathcal{U}_l, z : Z \vdash (\lambda T.\lambda x.x)(Z)(z) \doteq (\lambda x.x)(z) : Z}\;\Pi\text{-elim-cong} \quad \dfrac{\dfrac{}{\Gamma, Z : \mathcal{U}_l, z : Z, x : Z \vdash x : Z}\;\text{var} \quad \dfrac{}{\Gamma, Z : \mathcal{U}_l, z : Z \vdash z : Z}\;\text{var}}{\Gamma, T : \mathcal{U}_l, z : Z \vdash (\lambda x.x)(z) \doteq z : Z}\;\Pi\text{-comp}}{\Gamma, T : \mathcal{U}_l, z : Z \vdash (\lambda T.\lambda x.x)(Z)(z) \doteq z : Z}\;\doteq\text{-trans}$$

An advantage of defining functions using the shorthand syntax demonstrated in (2) is that one can read off the definitional equalities introduced by looking at the $\coloneqq$ symbols:

$$\mathrm{id}(Z, z) \coloneqq z \qquad \Longrightarrow \qquad \mathrm{id}(Z, z) \doteq z.$$

We will use this method to check whether the identity function is indeed the identity of the composition operator with some map $f : A \to B$:

$$\begin{aligned} \mathrm{id}_B \circ f &\doteq \lambda x.\, \mathrm{id}_B(f(x)) \\ &\doteq \lambda x. f(x) \\ &\doteq f. \end{aligned}$$

Note that we're implicitly using Π-intro-cong for the second equality, and η-equivalence for the third, together with $\doteq$-trans to tie everything together. The other direction follows similarly, except we also implicitly use Π-elim-cong for the second equality:

$$\begin{aligned} f \circ \mathrm{id}_A &\doteq \lambda x. f(\mathrm{id}_A(x)) \\ &\doteq \lambda x. f(x) \\ &\doteq f. \end{aligned}$$

Last but not least, one could use a similar chain of judgmental equalities to show the associativity of function composition:

$$\begin{aligned} (f \circ g) \circ h &\doteq (\lambda x. f(g(x))) \circ h \\ &\doteq \lambda y. (\lambda x. f(g(x)))(h(y)) \\ &\doteq \lambda y. f(g(h(y))) \\ &\doteq \lambda y. f((\lambda x. g(h(x)))(y)) \\ &\doteq f \circ (\lambda x. g(h(x))) \\ &\doteq f \circ (g \circ h). \end{aligned}$$

Function types are very important to our theory, hence we will see them used in various ways throughout this thesis. We are now ready to move on to defining more concrete types, like the natural numbers.

3 The naturals

Compared to dependent maps, the rules for the natural numbers are quite tame. First, we manifest the type of the naturals into existence via a formation rule. Then, we define all the ways one can construct a natural number. To do so, we first introduce a constant $0_{\mathbb{N}}$ (which, as the name suggests, represents the smallest natural number). Then, we introduce a map $\text{succ}_{\mathbb{N}}$ which, given any natural number, produces its successor. Note that said $\text{succ}_{\mathbb{N}}$ is synthetic in nature — there's no way to evaluate $\text{succ}_{\mathbb{N}}(0_{\mathbb{N}})$ any further. The important part is that it provides the necessary structure for the elimination rule (namely, the familiar notion of *natural induction*) to work its magic. Formally, the formation and introduction rules look as follows:

$$\frac{}{\vdash \mathbb{N} : \mathcal{U}_l}\ \mathbb{N}\text{-form} \qquad \frac{}{\vdash 0_{\mathbb{N}} : \mathbb{N}}\ 0_{\mathbb{N}}\text{-intro} \qquad \frac{}{\vdash \text{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}\ \text{succ}_{\mathbb{N}}\text{-intro} \tag{3}$$

With those two elements in place, we can repeatedly apply $\text{succ}_{\mathbb{N}}$ in order to define every natural. That is:

$$\begin{aligned} 1_{\mathbb{N}} &\coloneqq \text{succ}_{\mathbb{N}}(0_{\mathbb{N}}), \\ 2_{\mathbb{N}} &\coloneqq \text{succ}_{\mathbb{N}}(1_{\mathbb{N}}), \\ 3_{\mathbb{N}} &\coloneqq \text{succ}_{\mathbb{N}}(2_{\mathbb{N}}), \\ &\vdots \end{aligned}$$

As for the induction principle (the elimination principle for the naturals, or our way of "using up" the naturals we've constructed), we simply have the familiar notion of natural induction:

$$\frac{\Gamma \vdash P : \mathbb{N} \to \mathcal{U}_l \qquad \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \qquad \Gamma \vdash p_s : \prod_{n:\mathbb{N}} P(n) \to P(\text{succ}_{\mathbb{N}}(n))}{\Gamma \vdash \text{ind}_{\mathbb{N}}(P, p_0, p_s) : \prod_{n:\mathbb{N}} P(n)}\ \mathbb{N}\text{-ind}$$

Induction principles are a very fundamental concept, so let's take a second to walk through the definition above in more detail:

1. $\Gamma \vdash P : \mathbb{N} \to \mathcal{U}_l$ assigns a type to each natural, and represents the goal we are trying to prove. Under the classical idea of natural induction, it is the proposition we are trying to prove for every natural number. Since in type theory "proving" a proposition requires us to build out an element of its type, the type $P(n)$ describes what we are trying to construct in order for our proposition to hold at a certain n.

2. $\Gamma \vdash p_0 : P(0_{\mathbb{N}})$ is the base case. It proves that our goal holds at $0_{\mathbb{N}}$.

3. $\Gamma \vdash p_s : \prod_{n:\mathbb{N}} P(n) \to P(\text{succ}_{\mathbb{N}}(n))$ is the inductive step. Traditionally, we would write this out as $\forall n : \mathbb{N}, P(n) \to P(n+1)$. The Π-type encodes the $\forall$-quantifier, thus the type requires us to construct a map which, when given a natural $n : \mathbb{N}$ and an element of type $P(n)$, produces a new element of type $P(n+1)$.

4. Finally, $\Gamma \vdash \text{ind}_{\mathbb{N}}(p_0, p_s, :) \prod_{n:\mathbb{N}} P(n)$ is the conclusion of our induction. The type $\prod_{n:\mathbb{N}} P(n)$ is equivalent to what we would traditionally write as $\forall n : \mathbb{N}, P(n)$. That is, our reward for proving $P(0)$ and that $P(k) \to P(k+1)$ is concluding that P holds for every natural. We can evaluate an element of type $\prod_{n:\mathbb{N}} P(n)$ at any $n : \mathbb{N}$ to produce the proof that P holds at said n.

The expression on the left hand side, namely $\mathrm{ind}_{\mathbb{N}}(P, p_0, p_s)$, signals that we're evaluating the induction procedure with our two steps (the base case, and the inductive case) as arguments. Indeed, we could extract out P, p_0 and p_s as function arguments and introduce induction like this

$$\frac{}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}} : \prod_{P:\mathbb{N}\to\mathcal{U}_l} P(0_{\mathbb{N}}) \to (\prod_{n:\mathbb{N}} P(n) \to\ P(\mathrm{succ}_{\mathbb{N}}(n))) \to \prod_{n:\mathbb{N}} P(n)}\ \mathbb{N}\text{-ind},$$

although we avoided doing so for clarity's sake. Note that $\mathrm{ind}_{\mathbb{N}}$ acts like any other function in our theory (unlike function elimination, which required its own syntax), although it is synthetic in nature (just like $\mathrm{succ}_{\mathbb{N}}$). That is, induction alone cannot evaluate any further, although the *computation rules* for the naturals provide us with judgmental equalities governing the interaction between our constructors ($0_{\mathbb{N}}$ and $\mathrm{succ}_{\mathbb{N}}$) and induction principle.

Notation Remark. Throughout this thesis, we will often elide the first argument of $\mathrm{ind}_{\mathbb{N}}$ (i.e. P) when it can be inferred from context.

Finally, we need to provide the rules governing how our constructors interact with the induction principle:

$$\frac{\Gamma \vdash P : \mathbb{N} \to \mathcal{U}_l \qquad \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \qquad \Gamma \vdash p_s : \prod_{n:\mathbb{N}} P(n) \to\ P(\mathrm{succ}_{\mathbb{N}}(n))}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(P, p_0, p_s, 0) \doteq p_0 : P(0_{\mathbb{N}})}\ 0_{\mathbb{N}}\text{-ind}$$

$$\frac{\Gamma \vdash P : \mathbb{N} \to \mathcal{U}_l \qquad \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \qquad \Gamma \vdash p_s : \prod_{n:\mathbb{N}} P(n) \to\ P(\mathrm{succ}_{\mathbb{N}}(n))}{\Gamma, n : \mathbb{N} \vdash \mathrm{ind}_{\mathbb{N}}(P, p_0, p_s, \mathrm{succ}_{\mathbb{N}}(n)) \doteq p_s(P, \mathrm{ind}_{\mathbb{N}}(p_0, p_s, n)) : P(\mathrm{succ}_{\mathbb{N}}(n))}\ \mathrm{succ}_{\mathbb{N}}\text{-ind}$$

Intuitively, the rules above specify how we can evaluate $\mathrm{ind}_{\mathbb{N}}$, as long as we know which constructor was used when constructing the natural passed to $\mathrm{ind}_{\mathbb{N}}$ as an argument. In particular, when $\mathrm{ind}_{\mathbb{N}}$ is evaluated at $0_{\mathbb{N}}$, the returned value is p_0. Similarly, when $\mathrm{ind}_{\mathbb{N}}$ is evaluated at $\mathrm{succ}_{\mathbb{N}}(n)$, the returned value is $p_s(\mathrm{ind}_{\mathbb{N}}(P, p_0, p_s, n))$. One could repeatedly apply the rules above to observe the behaviour of $\mathrm{ind}_{\mathbb{N}}$ at any fixed natural.

While natural induction is often used to prove things about the naturals, it can do a lot more. In fact, since natural induction is our only elimination rule for the naturals, it turns out that any map defined on the naturals can be written in terms of induction. As an example, we are now going to define natural addition.

3.1 Natural addition

Our goal for this section is to define the function

$$\mathrm{add}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}.$$

We intend to perform induction on the second argument, therefore we will move the first argument into the context. Intuitively, we "fix" the first argument to some constant bound in our context, and attempt to define addition based on what value the second argument takes:

$$m : \mathbb{N} \vdash \mathrm{add}_{\mathbb{N}}(m) : \mathbb{N} \to \mathbb{N}.$$

We intend to apply the induction principle for $P(n) \coloneqq \mathbb{N}$. That is, the type we are trying to construct, namely $\mathbb{N}$, does not depend on the argument at all (since the addition of two naturals always results in another natural, no matter the input). Formally:

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) \doteq \mathbb{N} : \mathcal{U}_l}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(P) : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})}$$

The first argument of $\mathrm{ind}_{\mathbb{N}}$, which we will denote by $\text{add-0}_{\mathbb{N}}(m) : \mathbb{N}$, describes the 0-case, i.e. what happens when we add 0 to our fixed m. Similarly, the second argument of $\mathrm{ind}_{\mathbb{N}}$, which we will denote by $\text{add-succ}_{\mathbb{N}}(m) : \mathbb{N} \to \mathbb{N}$, describes the $n+1$ case, i.e. what happens when we add $n+1$ to our m. We define the two arguments as follows:

$$\begin{aligned} \Gamma, m : \mathbb{N} \vdash \quad & \text{add-0}_{\mathbb{N}}(m) && \coloneqq m && : \mathbb{N} \\ \Gamma, m : \mathbb{N} \vdash \quad & \text{add-succ}_{\mathbb{N}}(m) && \coloneqq \mathrm{succ}_{\mathbb{N}} && : \mathbb{N} \to \mathbb{N} \end{aligned}$$

We defined $\text{add-0}_{\mathbb{N}}(m)$, as m, because intuitively we have $m + 0 = m$ (we will soon show that this still holds for our definition). Similarly, we defined $\text{add-succ}_{\mathbb{N}}(m)$ as $\mathrm{succ}_{\mathbb{N}}$ because $m + (n + 1) = (m + n) + 1$. Note that, by the computation rules for induction, $\text{add-succ}_{\mathbb{N}}(m)$ is given the result of the n-case (i.e. $m + n$) as a parameter in order to compute $m + (n + 1)$, hence why we chose the definition above. We can now complete our induction in order to define the final addition function:

$$\frac{m : \mathbb{N} \vdash \text{add-0}_{\mathbb{N}}(m) \coloneqq m : \mathbb{N} \qquad m : \mathbb{N} \vdash \text{add-succ}_{\mathbb{N}}(m) \coloneqq \mathrm{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}{m : \mathbb{N} \vdash \mathrm{add}_{\mathbb{N}} \coloneqq \mathrm{ind}_{\mathbb{N}}\,(\text{add-0}_{\mathbb{N}}(m), \text{add-succ}_{\mathbb{N}}(m), :)\,\mathbb{N} \to \mathbb{N}}\,\mathrm{add}_{\mathbb{N}}$$

Notation Remark. Note that we're using the $\coloneqq$ notation inside judgments as syntactic sugar for writing out the expression the name is defined as.

When defining $\text{add-0}_{\mathbb{N}}(m)$ and $\text{add-succ}_{\mathbb{N}}(m)$, we claimed we did so in order to arrive at some properties, namely $m + 0 = 0$ and $m + (n + 1) = (m + n) + 1$ — do these actually hold for the function we've just defined? We can check by applying the computation rules for natural induction:

$$\begin{aligned} m : \mathbb{N} \vdash \mathrm{add}_{\mathbb{N}}(m, 0_{\mathbb{N}}) & \\ &\doteq \mathrm{ind}_{\mathbb{N}}(\text{add-0}_{\mathbb{N}}(m), \text{add-succ}_{\mathbb{N}}(m), 0_{\mathbb{N}}) \\ &\doteq \text{add-0}_{\mathbb{N}}(m) \\ &\doteq m. \\ m, n : \mathbb{N} \vdash \mathrm{add}_{\mathbb{N}}(m, \mathrm{succ}_{\mathbb{N}}(n)) & \\ &\doteq \mathrm{ind}_{\mathbb{N}}(\text{add-0}_{\mathbb{N}}(m), \text{add-succ}_{\mathbb{N}}(m), \mathrm{succ}_{\mathbb{N}}(n)) \\ &\doteq \text{add-succ}_{\mathbb{N}}(m, \mathrm{ind}_{\mathbb{N}}(\text{add-0}_{\mathbb{N}}(m), \text{add-succ}_{\mathbb{N}}(m), n)) \\ &\doteq \text{add-succ}_{\mathbb{N}}(m, \mathrm{add}_{\mathbb{N}}(m, n)) \\ &\doteq \mathrm{succ}_{\mathbb{N}}(\mathrm{add}_{\mathbb{N}}(m, n)). \end{aligned}$$

Notice that both equalities hold judgmentally. Since this is the case, can we, for example, also prove that $0 + n \doteq n$? The rules we've defined addition by are not enough to arrive at this conclusion. In fact, Martin Löf's Type Theory does not provide us with a way to prove such judgmental equalities! In section 5, we will define *identity types*, which allow us to prove such statements within the theory itself.

3.2 Natural multiplication and pattern matching

The process we've used to define natural addition was quite verbose and cumbersome. More succinctly, our definition would look something like this:

$$\begin{aligned} \mathrm{add}_{\mathbb{N}}(m, 0_{\mathbb{N}}) &\coloneqq m, \\ \mathrm{add}_{\mathbb{N}}(m, \mathrm{succ}_{\mathbb{N}}(n)) &\coloneqq \mathrm{succ}_{\mathbb{N}}(\mathrm{add}_{\mathbb{N}}(m, n)). \end{aligned}$$

Indeed, the above definition is enough to fully define the behaviour of our function. More generally, we can define any function $f : \prod_{n:\mathbb{N}} P(n)$ by specifying

$$\begin{aligned} f(0_{\mathbb{N}}) &\coloneqq p_0, \\ f(\mathrm{succ}_{\mathbb{N}}(n)) &\coloneqq p_s(n, f(n)). \end{aligned}$$

This is called *pattern matching* on n. In order to recover the definition by induction (i.e. $f \coloneqq \mathrm{ind}_{\mathbb{N}}(p_0, p_s)$), we need to replace every occurrence of $f(n)$ in $p_s(n, f(n))$ with a fresh variable $x : P(n)$. This way of defining functions is available in many programming languages (Haskell, Python, etc) and proof assistants (Lean, Agda, etc), and will make our job a lot easier. For instance, we can define the multiplication function on the naturals as follows:

$$\begin{aligned} \mathrm{mul}_{\mathbb{N}}(m, 0_{\mathbb{N}}) &\coloneqq 0_{\mathbb{N}} \\ \mathrm{mul}_{\mathbb{N}}(m, \mathrm{succ}_{\mathbb{N}}(n)) &\coloneqq \mathrm{add}_{\mathbb{N}}(m, \mathrm{mul}_{\mathbb{N}}(m, n)). \end{aligned}$$

Similarly to the way we defined $\circ$, this syntax for defining maps has the advantage of making the generated judgmental equalities trivial to read from the definition.

4 Inductive types

Our definition of the naturals is one example of the so-called *inductive types*. Many important types can be constructed this way, as we are going to see throughout this section. We will start by adding some additional data to the inductive constructor in order to arrive at the type of ordered lists $\mathrm{list}(T)$. We will then remove said constructor, leaving us with nothing but the base case, defining the unit type $\mathbf{1}$. We will then go further by removing the base case as well, defining the empty type $\mathbf{0}$. Finally, we will construct the type theoretic equivalents of disjoint unions and Cartesian products.

4.1 Ordered lists

Ordered list can be defined very similarly to the naturals. Unlike the naturals, ordered lists do not form a single type, but an entire type family. Indeed, given a type T of the contents of the list, we will define a new type $\mathrm{list}(T)$. We do so by introducing a constant empty list $[\ ]_T$, and a function $\mathrm{cons} : T \to \mathrm{list}(T) \to \mathrm{list}(T)$ which adds its first argument to the beginning of the list it's given as the second argument. Formally:

$$\frac{\Gamma \vdash T : \mathcal{U}_l}{\Gamma \vdash \mathrm{list}(T) : \mathcal{U}_l}\ \text{list-form} \qquad \frac{\Gamma \vdash T : \mathcal{U}_l}{\Gamma \vdash [\ ]_T : \mathrm{list}(T)}\ \text{nil-intro}$$

$$\frac{\Gamma \vdash T : \mathcal{U}_l}{\Gamma \vdash \mathrm{cons}_T : T \to \mathrm{list}(T) \to \mathrm{list}(T)}\ \text{cons-intro}$$

For example, we can construct the list $[1_{\mathbb{N}}, 2_{\mathbb{N}}, 3_{\mathbb{N}}]$ as follows:

$$[1_{\mathbb{N}}, 2_{\mathbb{N}}, 3_{\mathbb{N}}] \coloneqq \mathrm{cons}_{\mathbb{N}}(1_{\mathbb{N}}, \mathrm{cons}_{\mathbb{N}}(2_{\mathbb{N}}, \mathrm{cons}_{\mathbb{N}}(3_{\mathbb{N}}, [\ ]_{\mathbb{N}}))) : \mathrm{list}(\mathbb{N})$$

Notation Remark. Note that "list" is just a mere function $\mathrm{list} : \mathcal{U}_l \to \mathcal{U}_l$. Similarly, our notation for $[\ ]_T$ and cons_T is hiding away the fact that they are also just maps:

$$[\ ] : \prod_{T : \mathcal{U}_l} \mathrm{list}(T),$$
$$\mathrm{cons} : \prod_{T : \mathcal{U}_l} T \to \mathrm{list}(T) \to \mathrm{list}(T).$$

Recall how the second step in defining a type is describing a way to use values of said type, usually referred to as the *induction principle* for the type. Induction on the naturals is something most students will have been introduced to somewhat early in their mathematical education, and our definition is nothing but a more formal restatement of the same core idea. But what does it mean to apply induction on other types? Ordered lists have intentionally been chosen for demonstrating this, as they are, at their core, quite similar to the natural numbers. That is, they both consist of:

- Some "base value" (for the naturals, that is $0_{\mathbb{N}}$; for ordered lists, it is $[\ ]_T$).
- A way to construct "higher" values from existing ones ($\mathrm{succ}_{\mathbb{N}}$ in the case of the naturals, and cons_T in the case of ordered lists).

The induction principles of the two should therefore look somewhat similar. The difference between the constructions is the fact that $\text{cons}_T : T \to \text{list}(T) \to \text{list}(T)$ takes an additional argument (the one of type T representing the value that is to be added at the beginning of the list) compared to $\text{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}$. Indeed, all we have to do in order to adapt our prior idea of induction is to give the "inductive step" (the $P(n) \to P(n+1)$ case) access to the element being added to the list in addition to the induction hypothesis.

Putting it all together, in order to construct some element $P(l)$ for all lists $l : \text{list}(T)$, we need to construct it for the empty list (i.e. $P([\ ]_T)$, the base case) and a way to construct it for a list $\text{cons}_T(x, l)$ given $x : T$ and an element $P(l)$. Formally,

$$\frac{\Gamma \vdash P : \text{list}(T) \to \mathcal{U}_l \qquad \Gamma \vdash p_n : P([\ ]_T) \\ \Gamma \vdash p_c : \prod_{x:T} \prod_{l:\text{list}(T)} P(l) \to P(\text{cons}_T(x, l))}{\Gamma \vdash \text{ind}_{\text{list}(T)}(P, p_n, p_c) : \prod_{l:\text{list}(T)} P(l)}\ \text{list-ind}.$$

The computation rules work as expected, although we will omit the hypotheses for brevity's sake:

$$\frac{\cdots}{\Gamma \vdash \text{ind}_{\text{list}(T)}(P, p_n, p_c, [\ ]_T) \doteq p_n : P([\ ]_T)}$$

$$\frac{\cdots}{\Gamma, x : T, l : \text{list}(T) \vdash \text{ind}_{\text{list}(T)}(P, p_n, p_c, \text{cons}(x, l)) \doteq p_c(x, \text{ind}_{\text{list}(T)}(p_{\text{nil}}, p_c, l)) : P(\text{cons}(x, l))}$$

Do not worry if the above looks complicated — the computation rules become a lot clearer when used on functions defined by pattern matching. Pattern matching on ordered lists works similarly to pattern matching for $\mathbb{N}$. Indeed, we can define a function on all the ordered lists by specifying its behaviour at both constructors. As an example, we'll start off by constructing the $\text{concat}_T : \text{list}(T) \to \text{list}(T) \to \text{list}(T)$ function which concatenates (glues) together two lists. In a sense, this is the ordered-list version of the $\text{add}_{\mathbb{N}}$ function. Unlike our definition of $\text{add}_{\mathbb{N}}$, we will perform pattern matching on the first argument:

$$\begin{aligned} \text{concat}_T([\ ]_T, s) &\coloneqq s \\ \text{concat}_T(\text{cons}_T(h, l), s) &\coloneqq \text{cons}_T(h, \text{concat}_T(l, s)) \end{aligned}$$

For a slightly more involved example, consider the $\text{append}_T : T \to \text{list}(T) \to \text{list}(T)$ function which adds its first argument to the end of a list:

$$\begin{aligned} \text{append}_T(x, [\ ]_T) &\coloneqq \text{cons}_T(x, [\ ]_T) \\ \text{append}_T(x, \text{cons}_T(h, l)) &\coloneqq \text{cons}_T(h, \text{append}_T(x, l)) \end{aligned}$$

Let's try and digest what the definition is saying:

1. The first branch specifies the behaviour of appending an element $x : T$ to the end of the empty list $[\ ]_T$. The result of this computation is defined to be $\text{cons}_T(x, [\ ]_T)$, i.e. the single-element list produced by adding x at the start of the empty list.

2. The second branch (the inductive step) specifies what happens when appending an element $x : T$ at the end of the list $\text{cons}_T(h, l)$, a list produced by adding $h : T$ to the start of $l : \text{list}(T)$. We define the result of this branch as first appending x to the end of l (via $\text{append}_T(x, l)$) and then adding h back at the beginning of this newly created list. That is, we are essentially swapping the order of the "adding h at the start" and "adding x at the end" steps around.

4.2 The general procedure

Having constructed both $\mathbb{N}$ and list(T), we are now ready to tackle the more general procedure of defining inductive types. In particular, we can construct types *inductively* by providing the following kinds of rules:

1. *Formation rules* work as in described in section 2.2, declaring the type into existence, possibly depending on some parameters. For example, in the case of $\mathbb{N}$, the formation rule was the first rule of equation (3).

2. *Constructors* are functions that allow creating values of our type. For example, in the case of $\mathbb{N}$, we introduced the constructors $0_{\mathbb{N}}$ and $\text{succ}_{\mathbb{N}}$ in (3). These represent the *introduction rules* for the type.

3. The *induction principle* for a type defines the primary means of using up values of said type. That is, without this principle the type would seem "opaque" from the outside, i.e. there wouldn't be any way to distinguish between its various values. The induction principle represent the *elimination rule* for the type.

4. Finally, the *computation rules* explain what happens when we evaluate the induction principle at the various constructors.

Notation Remark. In order to simplify definitions, we can then *pattern match* on the declared type's constructors. This has the advantage of making definitions more readable, and more readily showing the judgmental equalities our definition produces as a consequence of the computation rules.

We will now construct a few more examples of such types.

4.3 The unit type

The *unit type* (also known as True or $\top$ in certain programming languages or proof assistants), denoted by $\mathbf{1}$, is an inductive type produced by a single constructor

$$\star : \mathbf{1}.$$

Intuitively, the unit type is what one would traditionally think of as a singleton set, i.e. a set with a single element. When it comes to logic, one can think of the unit type as corresponding to the "true" proposition (that is, a proposition that is always true), since it can always be constructed out of thin air.

So far, we've only defined induction on types that were "self-referential" in some way. Indeed, induction on both $\mathbb{N}$ and list(T) involves an inductive step that constructs our goal P at larger values from its values at smaller ones (for example, going from $P(n)$ to $P(n+1)$ in the case of the naturals). In the case of the unit type, there is no recursive constructor for us to do the above with, so can we even extend the notion of induction to it?

As a first step, we notice that $\star$ is quite similar to the other "base constructors" defined earlier (namely $0_{\mathbb{N}}$ and $[\ ]_T$). We can then define induction similarly to the way it was defined for $\mathbb{N}$ or list(T), but only taking in the arguments related to the base

constructors of said types. Indeed, given some type family $P : \mathbf{1} \to \mathcal{U}_l$, the induction principle looks as folows:

$$\mathrm{ind}_{\mathbf{1}} : \prod_{P:\mathbf{1}\to\mathcal{U}_l} P(\star) \to \prod_{u:\mathbf{1}} P(u). \tag{4}$$

The computation rule then states what happens when evaluating functions constructed by induction at $\star$. Unsurprisingly, the first (and only) argument is returned:

$$\mathrm{ind}_{\mathbf{1}}(p_\star, \star) \doteq p_\star.$$

Notation Remark. We treat both Π and $\to$ as having the same precedence, and being right-associative. That is, the above can be read as having the following parenthesis:

$$\mathrm{ind}_{\mathbf{1}} : \prod_{P:\mathbf{1}\to\mathcal{U}_l} \left(P(\star) \to \left(\prod_{u:\mathbf{1}} P(u) \right) \right).$$

4.4 The empty type

The empty type (also known as Void, False, or $\bot$ in certain programming languages or proof assistants), denoted by $\mathbf{0}$, is an inductive type with no constructors. Intuitively, the empty type is what one would traditionally think of as the empty set $\emptyset$. When it comes to logic, one can think of the empty type as corresponding to the "false" proposition (that is, a proposition that is always false), since there is no way for us to construct (i.e. "prove") it.

The unit type we've defined previously can be thought of as what remains when taking away the $\mathrm{succ}_{\mathbb{N}}$ constructor from $\mathbb{N}$, or the cons_T constructor from $\mathrm{list}(T)$. In a sense, the empty type goes even further by taking away the remaining constructor as well, leaving us with no way to construct the type.

The induction principle, given some type family $P : \mathbf{0} \to \mathcal{U}_l$, might seem a bit tricky to define. Normally, we would take one argument (case) for each constructor of the type at hand. As there are no constructors, we will simply take no arguments, and conclude that any proposition defined on $\mathbf{0}$ holds right out of the gate. That is:

$$\mathrm{ind}_{\mathbf{0}} : \prod_{P:\mathbf{0}\to\mathcal{U}_l} \prod_{e:\mathbf{0}} P(e).$$

Notice that there's no computation rules! Indeed, we normally write out one computation denoting what happens when evaluating functions constructed by induction at each constructor of the underlying type, but as there's no constructors, neither are there any such rules for us to define.

When $P(e)$ does not actually depend on $e : \mathbf{0}$, the induction principle produces the special case

$$\text{ex-falso} : \mathbf{0} \to P.$$

Intuitively, this matches the fact that a false statement implies any other statement in classical logic. We can use this to encode negations, since a proposition is false precisely when it implies "false":

$$\neg P :\equiv P \to \mathbf{0}.$$

While this technique of proving that a statement is false might seem familar, it is important for it not to be conflated with *proof by contradiction*. When proving a statement P by contradiction, we assume $\neg P$ is true and attempt to arrive at a falsehood. That is, we attempt to prove $\neg P \to \mathbf{F}$, which we can write down using our newly defined empty type as $\neg P \to \mathbf{0}$, i.e. $\neg\neg P$. In classical logic, P and $\neg\neg P$ are equivalent propositions. One direction of the correspondence is easy to construct:

$$\begin{aligned}&\text{double-neg-intro} : P \to \neg\neg P\\ &\text{double-neg-intro}(p) := \lambda f.f(p)\end{aligned}$$

Let's digest this definition a bit further. The type $\neg\neg P$ is defined as $(P \to \mathbf{0}) \to \mathbf{0}$. We thus have to construct a value of type $(P \to \mathbf{0}) \to \mathbf{0}$ given $p : P$. We do so by introducing a function going from $f : P \to \mathbf{0}$ to some value of type $\mathbf{0}$. Since we have no way to construct said value of type $\mathbf{0}$ out of thin air, our only choice is to evaluate the function $f : P \to \mathbf{0}$ at $p : P$ in order to produce the resulting $f(p) : \mathbf{0}$.

Proving by contradiction requires us to be able to reverse this map. That is, it requires us to have a map $\neg\neg P \to P$. This is called *double negation elimination*, and is not something we can perform without knowing more about the underlying type. Note that double negation elimination is equivalent to the law of the excluded middle, which is also something we don't generally have access to (it is something we could assume, but it would not come packaged with any computation rules).

4.5 Coproducts

The coproduct of two types A and B, denoted by $A + B$, is the equivalent to what we would traditionally think of as the disjoint union of two sets. That is, an element of the coproduct must be either an element of the left hand side, or an element of the right hand side. Formally, we have two constructors

$$\begin{aligned}\text{left} &: A \to A + B,\\ \text{right} &: B \to A + B.\end{aligned}$$

Coproducts are a bit different from the inductive types we've defined earlier. They are neither self-referential, nor have a constant "base case". Recall that induction principles define the primary means of consumption for values of a given type. Intuitively, in order to use up a value of type $A + B$, we need to be able to handle values of type A (we will turn this into the first argument given to the induction principle) and a way to handle values of type B (we will turn this into the second argument to said induction procedure).

Formally, given a type family $P : A + B \to \mathcal{U}_l$, the induction principle requires us to construct P for both the left and the right hand sides in order to construct P over any element of the type. That is,

$$\text{ind}_{A+B} : \prod_{P:A+B\to\mathcal{U}_l} \left(\prod_{a:A} P(\text{left}(a))\right) \to \left(\prod_{b:B} P(\text{right}(b))\right) \to \prod_{c:A+B} P(c).$$

The computation rules specify what happens when evaluating functions constructed by induction at either constructor:

$$\Gamma, a : A \vdash \mathrm{ind}_{A+B}(p_l, p_r, \mathrm{left}(a)) \doteq p_l(a),$$
$$\Gamma, b : B \vdash \mathrm{ind}_{A+B}(p_l, p_r, \mathrm{right}(b)) \doteq p_r(b).$$

Intuitively, the coproduct of two types corresponds to the $\vee$ operator in logic. That is, we can normally prove $A \vee B$ by either proving A or proving B. Similarly, we can construct an element of type $A + B$ by constructing an element of type A and applying left, or constructing an element of type B and applying right.

Unlike classical logic, we can then easily tell which constructor was used by pattern matching. For example, the law of the excluded middle states that $P \vee \neg P$ holds for all propositions P. Just because we affirm that either P or $\neg P$ must hold for some complicated proposition does not immediately give us either a proof of P, nor a proof of $\neg P$. On the other hand, we *must* first construct a value of type A or B in order to construct a value of type $A + B$. Instead, coproducts correspond to the $\vee$ operator in intuitionistic logic. In order to translate the proper law of the excluded middle into our theory, we need the concept of *propositions*, which we will introduce in section 5.3.

As an example for how we can pattern match on coproducts, consider the swap function $\mathrm{swap}_{A+B} : A + B \to B + A$

$$\begin{aligned}\mathrm{swap}_{A+B}(\mathrm{left}\, a) &:\equiv \mathrm{right}\, a\\ \mathrm{swap}_{A+B}(\mathrm{right}\, b) &:\equiv \mathrm{left}\, b.\end{aligned}$$

4.6 The booleans

An immediate application of the types defined above is defining the type of booleans. A boolean can either be the constant "false" or the constant "true". That is, we define the two constructors

$$\mathrm{false} : \mathbf{2}, \qquad\qquad \mathrm{true} : \mathbf{2}\,.$$

In a sense, booleans are reminiscent of the base constructors we defined for things like the naturals or the unit type. The difference is the fact we have two such constant constructors instead of one. Similarly to how we tackled the unit type, the induction principle will take one argument for each such base constructor.

Given a type family $P : \mathbf{2} \to \mathcal{U}_l$, the induction principle requires us to construct both $P(\mathrm{false})$ and $P(\mathrm{true})$ in order to construct a map defined on all booleans. Formally:

$$\mathrm{ind}_{\mathbf{2}}(P) : \prod_{P:\mathbf{2}\to\mathcal{U}_l} P(\mathrm{false}) \to P(\mathrm{true}) \to \prod_{b:\mathbf{2}} P(b).$$

The computation rules work as expected:

$$\begin{aligned}\mathrm{ind}_{\mathbf{2}}(p_f, p_t, \mathrm{false}) &\doteq p_f,\\ \mathrm{ind}_{\mathbf{2}}(p_f, p_t, \mathrm{true}) &\doteq p_t.\end{aligned}$$

Those familiar with programming might find the above familiar. Indeed, the induction principle for booleans is essentially an "if, then, else" expression, with the arguments in reverse order. The last argument is the condition; the function returns the second argument when the condition is true, and the first one otherwise.

As an example, we can define the negation function $\mathrm{neg} : \mathbf{2} \to \mathbf{2}$ by pattern matching as follows

$$\begin{aligned} \mathrm{neg}(\mathrm{false}) &\coloneqq \mathrm{false}, \\ \mathrm{neg}(\mathrm{true}) &\coloneqq \mathrm{true}\,. \end{aligned}$$

Alternatively, we could define the booleans without introducing an additional inductive type by combining our earlier constructions. We will denote this alternative construction by $\mathbf{2}'$:

$$\mathbf{2}' \coloneqq \mathbf{1} + \mathbf{1} \qquad \mathrm{false}' \coloneqq \mathrm{left}(\star) : \mathbf{2}' \qquad \mathrm{true}' \coloneqq \mathrm{right}(\star) : \mathbf{2}'\,.$$

We can then construct an induction principle similar to $\mathrm{ind}_{\mathbf{2}}$ by combining the induction principles for unit types and coproducts

$$\mathrm{ind}_{\mathbf{2}'}(P, p_f, p_t, b) \coloneqq \mathrm{ind}_{\mathbf{1}+\mathbf{1}}(P, \mathrm{ind}_{\mathbf{1}}(p_f), \mathrm{ind}_{\mathbf{1}}(p_t), b).$$

The above might look a bit intimidating. We could alternatively define induction by pattern matching

$$\begin{aligned} \mathrm{ind}_{\mathbf{2}'}(P, p_f, p_t, \mathrm{false}') &\coloneqq p_f, \\ \mathrm{ind}_{\mathbf{2}'}(P, p_f, p_t, \mathrm{true}') &\coloneqq p_t. \end{aligned}$$

While simple in nature, this detour demonstrates how we can combine our existing types into new ones. This is a very powerful idea that is used all throughout mathematics and programming. In section 5.1, we will show that our two constructions of the booleans are indeed equivalent.

4.7 Dependent pairs

Having just defined coproducts, the natural follow up would be trying to define products. Indeed, we could easily define $A \times B$ with a single constructor $A \to B \to A \times B$. It turns out we can define a more general family of types by making the first arrow dependent. That is, for a type A and a family $P : A \to \mathcal{U}_l$, we will define the type of dependent pairs $\sum_{a:A} P(a)$ using the constructor

$$\mathrm{pair}_{A,P} : \prod_{a:A} \left(P(a) \to \sum_{b:A} P(b) \right).$$

Let's step back for a second and walk through everything in more detail. The type $\sum_{a:A} P(a)$ is written the way it is in order to clearly show that our second element of the pair (the one of type $P(a)$) can depend on the first (the one of type A, bound to the name a). The constructor "pair" takes two different arguments: one of type A, bound here to the name a, and another of type $P(a)$. Writing both function as Π-types might make the type a bit more clear:

$$\mathrm{pair}_{A,P} : \prod_{a:A} \prod_{p:P(a)} \sum_{b:A} P(b).$$

The concept of dependent pairs is extremely important, and appears all throughout mathematics. For instance, consider the type is-even(n) defined for $n : \mathbb{N}$:

$$\text{is-even}(n) :\equiv \sum_{h:\mathbb{N}} h + h = n. \tag{5}$$

Intuitively, the type is-even(n) correponds to proofs that n is an even natural number. We can prove that a number is even by finding another natural h (its "half") such that $h + h = n$. In classical logic, we would write the above as

$$\exists h : \mathbb{N}, h + h = n.$$

Indeed, Σ-types correspond, in a sense, to the $\exists$-qualifier in classical logic. Again, this correspondence is not perfect — being able to prove that $\exists n : \mathbb{N}, P(n)$ in classical logic does not mean we have a way to compute such a natural. On the other hand, we can always extract the individual elements of some dependent pair. As a consequence, constructing an element of type $\sum_{a:A} B(a)$ is a much stronger requirement than proving $\exists a : A, B(a)$. Instead, dependent pairs correspond to the $\exists$-qualifier in intuitionistic logic. Similarly to the earlier discussion regarding coproducts, we will come back to this topic later when introducing the concept of *propositions*.

Having returned from our little detour, we now need to define the induction principle for dependent pairs. Given some type family $P(s)$ defined for $s : \sum_{a:A} B(a)$, the induction principle only requires us to construct P for the one and only constructor in order to construct it for any element of our type. Formally,

$$\text{ind}_\Sigma(P) : \left(\prod_{a:A} \prod_{b:B(a)} P(\text{pair}(a, b)) \right) \to \prod_{s:\sum_{a:A} B(a)} P(s).$$

The computation rule shows that what we are essentially doing is distributing the two arguments of the pair constructor to the first argument given to the induction principle, a concept often referred to as *uncurrying*:

$$\text{ind}_\Sigma(p, \text{pair}(a, b)) \doteq p(a, b).$$

Of course, when the type of the second element in the pair does not depend on the value of the first, we recover our initial product type:

$$A \times B :\equiv \sum_{a:A} B,$$
$$\text{pair} : A \to B \to A \times B.$$

Notably, non-dependent products correspond to the $\wedge$-operator in classical logic. Similarly to how one needs to prove both A and B in order to prove $A \wedge B$, one needs to construct elements of both A and B in order to apply "pair" and construct an element of type $A \times B$. In particular, we also recall that $A \times B \to A$ and $A \times B \to B$ in classical logic. Can we construct those same functions for our type?

Moreover, non-dependent products also correspond to the usual concept of the cartesian product in set theory. For a cartesian product $A \times B$ of two sets, we have the projections $\text{pr}_l : A \times B \to A$ and $\text{pr}_r : A \times B \to B$. Can we define those same projections for our products?

Indeed, the above can be defined by pattern matching. For the non-dependent case, we have:

$$\begin{aligned} &\mathrm{pr}_l : A \times B \to A, && \mathrm{pr}_r : A \times B \to B, \\ &\mathrm{pr}_l(\mathrm{pair}(a,b)) \coloneqq a. && \mathrm{pr}_r(\mathrm{pair}(a,b)) \coloneqq b. \end{aligned}$$

The general dependent case is a bit trickier. The first projection works the same way as in the non-dependent case

$$\begin{aligned} &\mathrm{pr}_l : \sum_{a:A} B(a) \to A, \\ &\mathrm{pr}_l(\mathrm{pair}(a,b)) \coloneqq a. \end{aligned}$$

The second projection is where the trouble occurs. That is, we want to project $\sum_{a:A} B(a)$ onto $B(a)$, but the latter is not a valid term, as a is no longer bound to anything. In order to get an element of type A to evaluate B at, we will make use of the first projection we've just defined

$$\begin{aligned} &\mathrm{pr}_r : \prod_{s:\sum_{a:A} B(a)} B(\mathrm{pr}_l(s)), \\ &\mathrm{pr}_r(\mathrm{pair}(a,b)) \coloneqq b. \end{aligned}$$

4.8 The integers as an inductive type

The natural follow-up to defining the naturals is, of course, defining the integers $\mathbb{Z}$. There's many ways to approach this, and we will try to present a few as we build out the theory along this thesis. The most common way of approaching the situation is to define the integers as quotients of natural differences, i.e. as pairs $a - b$ for $a, b \in \mathbb{N}$ under some equivalence relation. Unfortunately, Martin Löf's Type Theory does not offer us a way to construct quotients. Indeed, we require so-called "higher inductive types" for that, something we will touch upon later down the line. For now, we will go down the "boring" path of defining the integers as two separate copies of the naturals glued together

$$\mathbb{Z} \coloneqq \mathbb{N} + \mathbb{N}.$$

The first copy of the naturals represents the negative integers, starting at -1 and going down towards $-\infty$. The second copy represents the non-negative integers, starting at 0 at going up towards ∞:

$$\begin{aligned} &\vdots \\ -3_{\mathbb{Z}} &\coloneqq \mathrm{left}(2_{\mathbb{N}}) \\ -2_{\mathbb{Z}} &\coloneqq \mathrm{left}(1_{\mathbb{N}}) \\ -1_{\mathbb{Z}} &\coloneqq \mathrm{left}(0_{\mathbb{N}}) \\ 0_{\mathbb{Z}} &\coloneqq \mathrm{right}(0_{\mathbb{N}}) \\ 1_{\mathbb{Z}} &\coloneqq \mathrm{right}(1_{\mathbb{N}}) \\ 2_{\mathbb{Z}} &\coloneqq \mathrm{right}(2_{\mathbb{N}}) \\ 3_{\mathbb{Z}} &\coloneqq \mathrm{right}(3_{\mathbb{N}}) \\ &\vdots \end{aligned}$$

Intuitively, we've defined $\text{left}(n)$ to represent $-(n+1)$ and $\text{right}(n)$ to represent $+n$. As we've simply combined existing types, we don't need to introduce further rules into the theory (like an induction principle) in order to make use of our definition of the integers. Let's define integer addition $\text{add}_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ by pattern matching:

$$\begin{aligned}
\text{add}_{\mathbb{Z}}(\text{right}(x), \text{right}(y)) &:= \text{add}_{\mathbb{N}}(x, y), \\
\text{add}_{\mathbb{Z}}(\text{left}(x), \text{left}(y)) &:= \text{left}(\text{add}_{\mathbb{N}}(x, y) + 1), \\
\text{add}_{\mathbb{Z}}(\text{left}(x), \text{right}(y)) &:= ?, \\
\text{add}_{\mathbb{Z}}(\text{right}(x), \text{left}(y)) &:= ?.
\end{aligned}$$

The mixed sign cases are indeed a bit difficult to define right out of the gate. To do so, we'll introduce a helper function $\text{sub}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N} \to \mathbb{Z}$ for subtracting natural numbers:

$$\begin{aligned}
\text{sub}_{\mathbb{N}}(0_{\mathbb{N}}, 0_{\mathbb{N}}) &:= \text{right}(0_{\mathbb{N}}), \\
\text{sub}_{\mathbb{N}}(a+1, b+1) &:= \text{sub}_{\mathbb{N}}(a, b), \\
\text{sub}_{\mathbb{N}}(0_{\mathbb{N}}, b+1) &:= \text{left}(b), \\
\text{sub}_{\mathbb{N}}(a+1, 0_{\mathbb{N}}) &:= \text{right}(a+1).
\end{aligned} \tag{6}$$

We can now return to our definition of addition:

$$\begin{aligned}
\text{add}_{\mathbb{Z}}(\text{right}(x), \text{right}(y)) &:= \text{right}(\text{add}_{\mathbb{N}}(x, y)), \\
\text{add}_{\mathbb{Z}}(\text{left}(x), \text{left}(y)) &:= \text{left}(\text{add}_{\mathbb{N}}(x, y) + 1), \\
\text{add}_{\mathbb{Z}}(\text{left}(x), \text{right}(y)) &:= \text{sub}_{\mathbb{N}}(y, x+1), \\
\text{add}_{\mathbb{Z}}(\text{right}(x), \text{left}(y)) &:= \text{sub}_{\mathbb{N}}(x, y+1).
\end{aligned}$$

Pattern matching on our glued together copies of the naturals might feel a bit strange. Let's take a second to unpack what's happening here:

1. The first branch handles the addition of two non-negative integers. The underlying logic can be off-loaded to natural addition.

2. The second branch handles the addition of $-(x+1)$ and $-(y+1)$. Intuitively, we want the result to be $-x-y-2$, which we can write as $-((x+y+1)+1)$ to fit our constructors.

3. The remaining two branches are mirrors of the same underlying case — adding a positive and a negative integer together. Intuitively, computing $x + (-(y+1))$ is the same as computing $x - (y+1)$, which we do via our previously defined $\text{sub}_{\mathbb{N}}$.

The definitions above pattern match on both arguments. This works by essentially applying induction for each of the arguments provided. It might be worth taking a second to think about how one would go about rewriting such definitions to use the underlying induction principles.

While the definition above works perfectly well for practical purposes (indeed, it is how integers are defined in Lean's mathlib), it can feel a bit clunky to work with. We have to manually handle the negative and positive cases each time, all while remembering to constantly insert calls to $\text{succ}_{\mathbb{N}}$ in the right spots. On the other hand, this definition has the advantage of simplicity and not requiring any more advanced theory. Indeed, the quotient based definition we will arrive at later in this thesis comes with its own set of practical annoyances (in particular, having to show that paths between elements of the type are preserved by every function we define out of the type).

As stated earlier, we don't technically *need* an induction principle for the integers, but constructing one might still be useful. If the induction principle for the naturals requires us to be able to climb up the natural ladder (i.e. going from $P(n)$ to $P(n+1)$), it turns out we can prove statements for all the integers if we are able to climb both ways (i.e. show that $P(n) \leftrightarrow P(n+1)$). We start by defining what $n+1$ means for the integers, i.e. $\mathrm{succ}_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z}$:

$$\begin{aligned}
\mathrm{succ}_{\mathbb{Z}}(\mathrm{right}(a)) &:\equiv \mathrm{right}(a+1),\\
\mathrm{succ}_{\mathbb{Z}}(\mathrm{left}(0_{\mathbb{N}})) &:\equiv \mathrm{right}(0_{\mathbb{N}}),\\
\mathrm{succ}_{\mathbb{Z}}(\mathrm{left}(a+1)) &:\equiv \mathrm{left}(a).
\end{aligned}$$

We then define the type of bidirectional maps:

$$A \leftrightarrow B :\equiv (B \to A) \times (A \to B).$$

Given some type $P : \mathbb{Z} \to \mathcal{U}_l$, our induction principle will have the type

$$\mathrm{ind}_{\mathbb{Z}}(P) : P(0_{\mathbb{Z}}) \to \left(\prod_{x:\mathbb{Z}} P(x) \leftrightarrow P(\mathrm{succ}_{\mathbb{Z}}(x))\right) \to \prod_{x:\mathbb{Z}} P(x).$$

That is, constructing the type at 0 (the base case) and providing a way to go both up and down the ladder should allow us to construct P for any integer. We can now prove the induction principle by pattern matching

$$\begin{aligned}
&\mathrm{ind}_{\mathbb{Z}}(p_z, p_i, \mathrm{right}(0_{\mathbb{N}})) &&:\equiv p_z,\\
&\mathrm{ind}_{\mathbb{Z}}(p_z, p_i, \mathrm{right}(\mathrm{succ}_{\mathbb{N}}(a))) &&:\equiv \mathrm{pr}_r(p_i(\mathrm{right}(a)))(\mathrm{ind}_{\mathbb{Z}}(p_z, p_i, \mathrm{right}(a))),\\
&\mathrm{ind}_{\mathbb{Z}}(p_z, p_i, \mathrm{left}(0_{\mathbb{N}})) &&:\equiv \mathrm{pr}_l(p_i(\mathrm{left}(0_{\mathbb{N}})))(p_z),\\
&\mathrm{ind}_{\mathbb{Z}}(p_z, p_i, \mathrm{left}(\mathrm{succ}_{\mathbb{N}}(a))) &&:\equiv \mathrm{pr}_l(p_i(\mathrm{left}(\mathrm{succ}_{\mathbb{N}}(a))))(\mathrm{ind}_{\mathbb{Z}}(p_z, p_i, \mathrm{left}(a))).
\end{aligned}$$

The above can look a bit intimidating. Let's break it down:

1. The first branch is the simplest, as it simply returns the base case.

2. Every time we need to go from $P(k)$ to $P(k+1)$ throughout our definition, we will do so by $\mathrm{pr}_r(p_i(k))$. Indeed, $p_i(k) : P(k) \leftrightarrow P(\mathrm{succ}_{\mathbb{Z}}(k))$, thus taking the right projection allows us to take the direction we're interested in

   $$\mathrm{pr}_r(p_i(k)) : P(k) \to P(\mathrm{succ}_{\mathbb{Z}}(k)).$$

   We then evaluate this at $\mathrm{right}(a)$, since by definition $\mathrm{succ}_{\mathbb{Z}}(\mathrm{right}(a)) \doteq \mathrm{right}(a+1)$. This yields

   $$\mathrm{pr}_r(p_i(\mathrm{right}(a))) : P(\mathrm{right}(a)) \to P(\mathrm{right}(a+1)).$$

   Constructing $P(\mathrm{right}(a))$ can be done inductively as $\mathrm{ind}_{\mathbb{Z}}(p_z, p_i, \mathrm{right}(a))$. The details are obfuscated a bit since we're pattern matching, but we're essentially performing natural induction starting at 0 and climbing up to the value we're interested in.

3. In the previous branch, we used $\mathrm{pr}_r(p_i(k)) : P(k) \to P(\mathrm{succ}_{\mathbb{Z}}(k))$ to access the forwards direction of the bidirectional map. We can go the other way around by taking the left projection

$$\mathrm{pr}_l(p_i(k)) : P(\mathrm{succ}_{\mathbb{Z}}(k)) \to P(k).$$

The two remaining branches are essentially performing natural induction along the other direction. For one, notice the judgmental equalities induced by our definition of $\mathrm{succ}_{\mathbb{Z}}$:

$$\begin{aligned} \mathrm{succ}_{\mathbb{Z}}(\mathrm{left}(0_{\mathbb{N}})) &\doteq \mathrm{right}(0_{\mathbb{N}}), \\ \mathrm{succ}_{\mathbb{Z}}(\mathrm{left}(a+1)) &\doteq \mathrm{left}(a). \end{aligned}$$

The remaining branches are thus applying the aforementioned first projection to $\mathrm{left}(0_{\mathbb{N}})$ and $\mathrm{left}(\mathrm{succ}_{\mathbb{N}}(a))$ respectively:

$$\begin{aligned} \mathrm{pr}_l(p_i(\mathrm{left}(0_{\mathbb{N}}))) &: P(\mathrm{right}(0_{\mathbb{N}})) \to P(\mathrm{left}(0_{\mathbb{N}})), \\ \mathrm{pr}_l(p_i(\mathrm{left}(\mathrm{succ}_{\mathbb{N}}(a)))) &: P(\mathrm{left}(a)) \to P(\mathrm{left}(\mathrm{succ}_{\mathbb{N}}(a))). \end{aligned}$$

Constructing the arguments required for evaluating the above works similarly to the first two branches. That is, the third branch constructs $P(\mathrm{right}(0_{\mathbb{N}}))$ as p_z (the base case), and the fourth branch constructs $P(\mathrm{left}(a))$ by induction as $\mathrm{ind}_{\mathbb{Z}}(p_z, p_i, \mathrm{left}(a))$.

Although the explanation might've been a bit verbose for the sake of explicitness, the important highlight is that we've constructed our own induction principle without introducing new rules into the type theory! (Similarly to the second definition of the booleans in section 4.6).

5 Identity types

When talking about the naturals, one issue we stumbled upon is not being able to prove that $0 + n \doteq n$ holds judgmentally. Indeed, Martin Löf's Type Theory does not provide us with the tools to prove such judgmental equalities. Instead, this section will introduce the type $a =_A b$ which represents proofs that $a, b : A$ are, in a sense, equal. Together with the associated induction principle, this will let us prove equalities in the language of type theory itself.

Given some type A and $a, b : A$, we introduce the *identity type* $a =_A b$ generated by the constructor refl_a:

$$\text{refl}_a : \prod_{a:A} a =_A a.$$

Identity types are the heart and soul of *Homotopy Type Theory*. So far, we've been making analogies to set theory and logic. Under the homotopical interpretation of type theory, types can be thought of as (abstract) spaces, and values can be thought of as points in those spaces. Type families can be thought of as fibrations, dependent functions can be thought of as sections, and dependent sums can be thought of as total spaces of said fibrations. Identity types then act as *paths* along those spaces, hence them also being commonly referred to as *path types*.

Given a fixed starting point $a : A$ and $P(x, i)$ defined for all $x : A$ and $i : a =_A x$, the induction principle states that it suffices to prove P holds for the refl case in order for it to hold for all paths starting at said point. This principle is called *path induction*, and looks as follows

$$\text{ind}_=(a, P) : P(a, \text{refl}_a) \to \prod_{x:A} \prod_{i:a=_A x} P(x, i). \tag{7}$$

At face value, this looks somewhat similar to the induction principle for the unit type, as defined in (4) — both types seem to be generated by a single base constructor, after all. The induction principle for identity types is a bit more subtle though! Note that, while the starting point is fixed, the endpoint *must* vary freely in order for us to be able to apply the induction principle. For example, assume we're defining a loop as a point together with a path from said point to itself. We cannot then apply the induction principle to attempt and prove that all loops can be identified with each other, since the endpoint is by the definition of a loop constrained to staying constant (indeed, otherwise we would be able to prove that a torus is contractible). Requiring an endpoint that is free to wiggle around the space prevents this issue.

One important thing to note is that since identity types are just that — types — one can construct the type of paths between paths on a given space, e.g. $p =_{a=_A b} q$. Under the homotopy theory analogy, the type of paths corresponds to the *path space* of the underlying type. Paths between paths can then be thought of as homotopies between said paths. With this in mind, the limitations of induction outlined above make sense — they parallel the way in which loops on a space can't generally be smoothly deformed into eachother (think of a loop around a torus not being contractible into a single point).

Since we've defined identity types as being generated by refl alone, it might seem unclear how we could ever end up with non-trivial paths. We will later introduce the concept of *higher inductive types*. These constructions are a more general kind of inductive types — instead of constructors only defining ways to construct values of the type

being defined, constructors for higher inductive types are allowed to also introduce paths between the various elements of the type, paths between paths on the type, and so on and so forth all the way up the metaphorical tower.

The computation rule for path-induction works as expected

$$\mathrm{ind}_{=}(P, p, a, \mathrm{refl}_a) \doteq p : P(a, \mathrm{refl}_a).$$

Paths assemble into a structure known as a *groupoid*. In fact, they form an infinite tower of such structures, known as an ∞-*groupoid*. We won't go into details regarding what that means, although we will define the basic operations of path inversion and concatenation (the base groupoid operations). Those operations can be thought of as analogous to symmetry and transitivity of equality in logic.

Lemma 1. Given a path $a =_A b$, we can construct a new path $b =_A a$. Formally, there is a map:

$$\mathrm{inv}_{=} : \prod_{A:\mathcal{U}_l} \prod_{a,b:A} a =_A b \to b =_A a.$$

We denote the inverse of a path p by p^{-1}.

Proof. We proceed by pattern matching on the given path,

$$\mathrm{inv}_{=}(A, a, a, \mathrm{refl}_a) :\equiv \mathrm{refl}_a\,.$$

□

Lemma 2. Given paths $a =_A b$ and $b =_A c$, we can construct a new path $a =_A c$. Formally, there is a map:

$$\mathrm{trans}_{=} : \prod_{A:\mathcal{U}_l} \prod_{a,b,c:A} a =_A b \to b =_A c \to a =_A c.$$

We denote the concatenation of p and q by $p \centerdot q$.

Proof. We proceed by pattern matching on the given paths,

$$\mathrm{trans}_{=}(A, a, a, a, \mathrm{refl}_a, \mathrm{refl}_a) :\equiv \mathrm{refl}_a\,.$$

□

Moreover, the groupoid operations must satisfy certain laws. We will not prove that every law holds for $=$ here, although we will prove the ones we're going to use later down the line.

Lemma 3. Given a path $p : a = b$, we have $p \centerdot p^{-1} = \mathrm{refl}_a$.

Proof. We proceed by pattern matching on p. It suffices to show that $\mathrm{refl}_a \centerdot \mathrm{refl}_a = \mathrm{refl}_a$, which holds judgmentally (by the definition of $\centerdot$). □

Lemma 4. Given a path $p : a = b$, we have $(p^{-1})^{-1} = p$.

Proof. We proceed by pattern matching on p. Indeed, this holds judgmentally

$$\left(\mathrm{refl}_a^{-1}\right)^{-1} \doteq \mathrm{refl}_a^{-1} \doteq \mathrm{refl}_a\,.$$

□

Functions in Homotopy Type Theory act as functors on the underlying groupoids of the types involved. Indeed, consider a function $f : A \to B$. This function not only maps the values in A to values in B, but also maps paths in A to paths in B and so on and so forth. Indeed, we can define this action by pattern matching

$$\begin{aligned}&\mathrm{ap} : (A \to B) \to \prod_{a,b:A} \prod_{p:a=_A b} f(a) =_B f(b),\\ &\mathrm{ap}(f, a, a, \mathrm{refl}_a) \coloneqq \mathrm{refl}_{f(a)}\,.\end{aligned}$$

Before moving on, we will prove a lemma regarding the way "ap" interacts with function composition.

Lemma 5. Applying "id" to a path preserves said path. Moreover, "ap" distributes over function composition. Formally, there are maps

$$\begin{aligned}\mathrm{ap}_\circ &: \prod_{\substack{f:A\to B\\ g:B\to C}} \prod_{\substack{a,b:A\\ p:a=b}} \mathrm{ap}(g, \mathrm{ap}(f, p)) = \mathrm{ap}(g \circ f, p).\\ \mathrm{ap}_{\mathrm{id}} &: \prod_{\substack{a,b:A\\ p:a=b}} \mathrm{ap}(\mathrm{id}_A, p) = p.\end{aligned}$$

Proof. Both maps can be constructed by path-induction. Written as pattern matching:

$$\begin{aligned}\mathrm{ap}_\circ(f, g, a, a, \mathrm{refl}_a) &\coloneqq \mathrm{refl}_{\mathrm{refl}_{g(f(a))}},\\ \mathrm{ap}_{\mathrm{id}}(a, a, \mathrm{refl}_a) &\coloneqq \mathrm{refl}_{\mathrm{refl}_a}\,.\end{aligned}$$

Indeed, we are allowed to use $\mathrm{refl}_{\mathrm{refl}_{\dots}}$ in both cases because the equalities hold judgmentally:

$$\begin{aligned}\mathrm{ap}(g, \mathrm{ap}(f, \mathrm{refl}_a)) \doteq \mathrm{ap}(g, \mathrm{refl}_{f(a)}) &\doteq \mathrm{refl}_{g(f(a))},\\ \mathrm{ap}(\mathrm{id}_A, \mathrm{refl}_a) &\doteq \mathrm{refl}_a\,.\end{aligned}$$

□

The natural follow up to the above is trying to define a simmilar operation for dependent functions. Unfortunately, we cannot do that directly — given a dependent function $f : \prod_{a:A} B(a)$, the endpoints $f(a), f(x)$ might have different types, disallowing us from connecting them via a path (we cannot directly connect points in different spaces). To get around this limitation, we must introduce the *transport* operation. This operation corresponds to $=$-elimination in logic (the idea that, given $P(a)$ and $a = b$, then $P(b)$).

Given a family of types $P : A \to \mathcal{U}_l$, we have

$$\begin{aligned}&\mathrm{tr}_P : \prod_{a,x:A} a = x \to P(a) \to P(x),\\ &\mathrm{tr}_P(a, a, \mathrm{refl}_a, p_a) \coloneqq p_a.\end{aligned}$$

Notation Remark. Note that we will omit explicit path endpoints for operations like tr when they can be inferred from context. We will do the same when it comes to the underlying type for paths (i.e. we might write $a = x$ instead of $a =_A x$). Modern proof assistants offer syntax for declaring dependent functions as implicit, which will make their arguments get automatically inferred during usage.

One thing worth noting is that transporting along a constant type family does not affect the input.

Lemma 6. Given types A, B with points $x, y : A$ and $b : B$ together with a path $p : x =_A y$, we have $\mathrm{tr}_B(p, b) = b$. Formally, we have a mapping

$$\text{tr-const}_{A,B} : \prod_{x,y:A} \prod_{p:x=y} \prod_{b:B} \mathrm{tr}_B(p, b) = b.$$

Proof. We construct this mapping by pattern matching. Note that we are able to use refl_b to construct $\mathrm{tr}_B(\mathrm{refl}_x, b) = b$ since, by the definition of transport, this equality holds judgmentally, i.e. $\mathrm{tr}_B(\mathrm{refl}_x, b) \doteq b$:

$$\text{tr-const}_{A,B}(\mathrm{refl}_x, b) :\equiv \mathrm{refl}_b\,.$$

□

Another interesting case is that of transporting a path along another path:

Lemma 7. Given types A and B together with points $x, y : A$, maps $f, g : A \to B$, and paths $p : x = y$ and $q : f(x) = g(x)$, we have

$$\mathrm{tr}_B(p, q) = \mathrm{ap}(f, p)^{-1} \circ q \circ \mathrm{ap}(g, p).$$

Proof. We proceed by pattern matching on p. Indeed, it remains for us to construct a path

$$\mathrm{tr}_B(\mathrm{refl}_x, q) = \mathrm{ap}(f, \mathrm{refl}_x)^{-1} \circ q \circ \mathrm{ap}(g, \mathrm{refl}_x),$$

but the above is definitionally equal to

$$q = \mathrm{refl}_{f(x)}^{-1} \circ q \circ \mathrm{refl}_{g(x)},$$

which is moreover judgmentally equal to

$$q = q,$$

thus our proof can be finalized by refl_q. □

With the transport operation under our toolbelt, we can now define the *dependent action of functions on paths*:

$$\begin{aligned}&\mathrm{apd} : \prod_{f:\prod_{a:A} B(a)} \prod_{a,b:A} \prod_{p:a=b} \mathrm{tr}_B(p, f(a)) = f(b),\\ &\mathrm{apd}(f, a, a, \mathrm{refl}_a) :\equiv \mathrm{refl}_{f(a)}\,.\end{aligned}$$

5.1 Homotopies and equivalences

An interesting question one can ponder is what it means for two functions to be equal. The traditional approach to this question is identifying two functions precisely when they produce the same outputs given the same input. It turns out that under our previously introduced homotopical analogy, this concept of "functions providing the same outputs given the same input" corresponds to the idea of homotopies between functions. Indeed, given two functions $f, g : \prod_{a:A} B(a)$, let

$$f \sim g \coloneqq \prod_{a:A} f(a) = g(a).$$

Traditionally, one thinks of a homotopy between $f, g : A \to B$ as a continuous deformation between the maps, i.e. a map $A \times [0,1] \to B$ that agrees with the original functions on the boundaries of the interval. Indeed, by fixing some constant $t \in [0,1]$, one gets back a specific instance of a map $A \to B$. If instead of fixing t one chooses to fix $a \in A$, then what is left is a map $[0,1] \to B$, which by definition is just a path in B. Indeed, homotopies can be thought of as (continuous) pointwise paths between two maps, which is where our analogy comes from.

This relation can be proven to respect all the usual groupoid laws. We will not do so here, as we aren't planning to make use of said laws. The interested reader should check [3, Chapter 9]. One can define a map $f = g \to f \sim g$ by path-induction. Unfortunately, we don't have a way to define the reverse direction, i.e. $f \sim g \to f = g$. This can be introduced into the system as an axiom, referred to as *functional extensionality*, which turns out to be a downstream consequence of the more important *univalence principle*. For the purpose of this thesis, we will leave things as-is and not worry about the paths between functions, but those interested should feel free to consult [3, Chapter 13].

A map $f : A \to B$ is traditionally said to be a homotopy equivalence if there's an inverse $g : B \to A$ such that $f \circ g \sim \mathrm{id}_A$ and $g \circ f \sim \mathrm{id}_B$. We will refer to such maps as *bi-invertible maps*[6]. On the other hand, one can define equivalences slightly differently, by allowing the left and right inverses to differ. That is, we say a map $f : A \to B$ is an equivalence when there exists:

1. A left-inverse $l : B \to A$ such that $l \circ f \sim \mathrm{id}_A$. This is also called a *retraction* of f.
2. A right-inverse $r : B \to A$ such that $f \circ r \sim \mathrm{id}_B$. This is also called a *section* of f.

Formally, we define a type $\text{is-equiv}(f)$ that witnesses the fact f is an equivalence by

$$\text{is-equiv}(f) \coloneqq \left(\sum_{l:B\to A} l \circ f \sim \mathrm{id}_A \right) \times \left(\sum_{r:B\to A} f \circ r \sim \mathrm{id}_B \right).$$

We use $A \simeq B$ to denote the type of equivalences $A \to B$. Formally,

$$A \simeq B \coloneqq \sum_{f:A\to B} \text{is-equiv}(f).$$

[6]The naming around this can be a bit confusing. Indeed, [5] uses the name "bi-invertible map" to refer to a specific definition of equivalences (namely, the one we are about to provide), yet [3] uses the name like we just did.

Given a proof that f is a bi-invertible map, one can construct a proof that f is an equivalence, and vice versa. The reason we defined equivalences the way we did is a bit technical for now, but the idea is that we want is-equiv(f) to be a *proposition*. That is, we want proofs that is-equiv(f) to be unique (up to identification). We will touch upon propositions for a bit in section 5.3.

As a first example, we will show that $\mathrm{neg} : \mathbf{2} \to \mathbf{2}$ is an equivalence, with itself as both inverses. To do so, we must construct a homotopy $h : \mathrm{neg} \circ \mathrm{neg} \sim \mathrm{id}_{\mathbf{2}}$. Recall that homotopies are just pointwise-paths, thus the above can be defined by pattern matching:

$$\begin{aligned} h(\mathrm{false}) &:= \mathrm{refl}_{\mathrm{false}}, \\ h(\mathrm{true}) &:= \mathrm{refl}_{\mathrm{true}}\,. \end{aligned}$$

While the above might look a bit anti-climatic, we are able to use refl on both branches because the equalities hold judgmentally, i.e.

$$\begin{aligned} (\mathrm{neg} \circ \mathrm{neg})(\mathrm{false}) &\doteq \mathrm{neg}(\mathrm{neg}(\mathrm{false})) \\ &\doteq \mathrm{neg}(\mathrm{true}) \\ &\doteq \mathrm{false} \\ &\doteq \mathrm{id}_{\mathbf{2}}(\mathrm{false}), \\ (\mathrm{neg} \circ \mathrm{neg})(\mathrm{true}) &\doteq \mathrm{neg}(\mathrm{neg}(\mathrm{true})) \\ &\doteq \mathrm{neg}(\mathrm{false}) \\ &\doteq \mathrm{true} \\ &\doteq \mathrm{id}_{\mathbf{2}}(\mathrm{true}). \end{aligned}$$

Since neg is both its own left and its own right inverse, the homotopy defined above is enough to conclude neg is an equivalence.

For a slightly more interesting example, we will show that $\mathbf{2}$ and $\mathbf{2}'$ are equivalent, i.e. that $\mathbf{2} \simeq \mathbf{2}'$. We start by defining a map $\mathcal{C}_{\mathbf{2}}^{\mathbf{2}'} : \mathbf{2} \to \mathbf{2}'$ by pattern matching:

$$\begin{aligned} \mathcal{C}_{\mathbf{2}}^{\mathbf{2}'}(\mathrm{false}) &:= \mathrm{false}', \\ \mathcal{C}_{\mathbf{2}}^{\mathbf{2}'}(\mathrm{true}) &:= \mathrm{true}'\,. \end{aligned}$$

Our goal is then to construct an element of is-equiv($\mathcal{C}_{\mathbf{2}}^{\mathbf{2}'}$), showing that the function we've just defined is indeed an equivalence. To do so, we define a map $\mathcal{C}_{\mathbf{2}'}^{\mathbf{2}} : \mathbf{2}' \to \mathbf{2}$, which acts as both a left and a right inverse for $\mathcal{C}_{\mathbf{2}}^{\mathbf{2}'}$:

$$\begin{aligned} \mathcal{C}_{\mathbf{2}'}^{\mathbf{2}}(\mathrm{false}') &:= \mathrm{false}, \\ \mathcal{C}_{\mathbf{2}'}^{\mathbf{2}}(\mathrm{true}') &:= \mathrm{true}\,. \end{aligned}$$

It now remains to construct homotopies

$$\begin{aligned} h_{\mathbf{2}} &: \mathcal{C}_{\mathbf{2}'}^{\mathbf{2}} \circ \mathcal{C}_{\mathbf{2}}^{\mathbf{2}'} \sim \mathrm{id}_{\mathbf{2}}, \\ h_{\mathbf{2}'} &: \mathcal{C}_{\mathbf{2}}^{\mathbf{2}'} \circ \mathcal{C}_{\mathbf{2}'}^{\mathbf{2}} \sim \mathrm{id}_{\mathbf{2}'}\,. \end{aligned}$$

Defining said homotopies is a bit of a routine operation, as all the equalities hold judgmentally

$$\begin{aligned} h_{\mathbf{2}}(\mathrm{false}) &:= \mathrm{refl}_{\mathrm{false}}, \\ h_{\mathbf{2}}(\mathrm{true}) &:= \mathrm{refl}_{\mathrm{true}}, \\ h_{\mathbf{2}'}(\mathrm{false}') &:= \mathrm{refl}_{\mathrm{false}'}, \\ h_{\mathbf{2}'}(\mathrm{true}') &:= \mathrm{refl}_{\mathrm{true}'}\,. \end{aligned}$$

The examples above are not particularly interesting to write out. Defining equivalences becomes a bit more involved when non-trivial paths enter the mix, as we will see in theorem 1. For now, we will end things off by noting the existence of the trivial equivalence induced by the identity function.

Lemma 8. Given a type A, the identity function induces an equivalence id-equiv : $A \simeq A$.

Proof. We will prove that id_A is an equivalence (with itself as its own inverse). Note that $\mathrm{id}_A \circ \mathrm{id}_A \doteq \mathrm{id}_A$ holds judgmentally, thus $\mathrm{id}_A \circ \mathrm{id}_A \sim \mathrm{id}_A$ finalizing our proof. □

5.2 The integers as a higher inductive type

In section 4.8, we defined the integers by glueing together two copies of the naturals. We will now attempt to define the integers as pairs of naturals subject to an equivalence relation. In set theoretic terms, we can describe any integer as a difference of two naturals, i.e. $a - b$ for $a, b \in \mathbb{N}$. Since every integer can be described as an infinite number of such differences, we define an equivalence relation

$$(a - b \sim c - d) \coloneqq (a + d = b + c)$$

We can then define our final set by $\mathbb{Z} \coloneqq (\mathbb{N} \times \mathbb{N})/\sim$. Translating this definition to Martin Löf's Type Theory is not possible out of the box, as said type theory does not offer us a way to take the quotients of types. On the other hand, set quotients are well defined (see [3, Chapter 18]) in Homotopy Type Theory, although they require a fair bit of additional machinery to set up. In this section, we will attempt to reconstruct the integers in a way that is similar in spirit to the quotient-based definition. By the end of this section, we will have constructed an equivalence between the new, and old definitions of $\mathbb{Z}$.

In order to prevent confusion, we will refer to the redefined type of integers as $\mathbb{Z}_{\mathrm{hit}}$ (here "hit" stands for *higher inductive type*). We start off by introducing a single constructor taking in a pair of naturals

$$\mathrm{int} : \mathbb{N} \to \mathbb{N} \to \mathbb{Z}_{\mathrm{hit}}$$

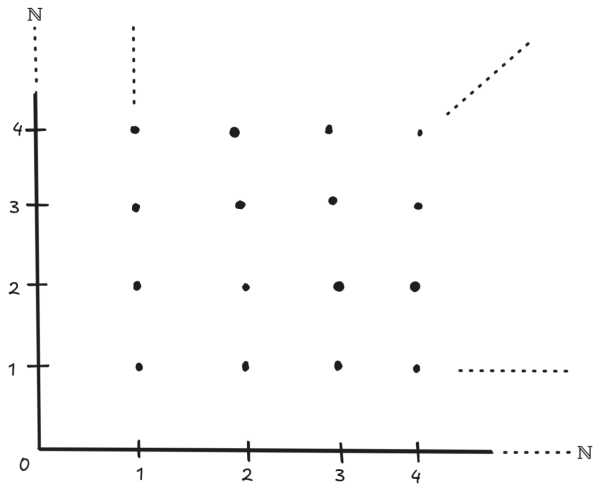


Figure 1: The type generated by int — essentially just $\mathbb{N} \times \mathbb{N}$.

Of course, this definition is incomplete. The elements $\mathrm{int}(0,0)$ and $\mathrm{int}(1,1)$ should be equivalent: intuitively, they represent the differences $0-0=1-1$. To achieve this, we will construct $\mathbb{Z}_{\mathrm{hit}}$ as a *higher inductive type*. Higher inductive types work similarly to inductive types. The main difference is that, in addition to providing constructors for their values, such types can also introduce additional non-trivial path constructors between said values, or between paths on said values, and so on up the infinite tower. We will thus introduce an additional constructor

$$\mathrm{step} : \prod_{a,b:\mathbb{N}} \mathrm{int}(a,b) = \mathrm{int}(a+1,b+1).$$

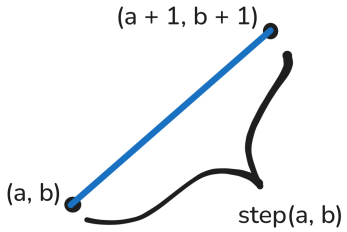


Figure 2: The step constructor connects equivalent pairs together.

It might seem like the path constructor we've just introduced is "too weak" — does it even capture the equivalence relation we started with? The answer is a resounding *yes*! We can concatenate invocations of "step" together to prove that pairs further away from eachother are equivalent.

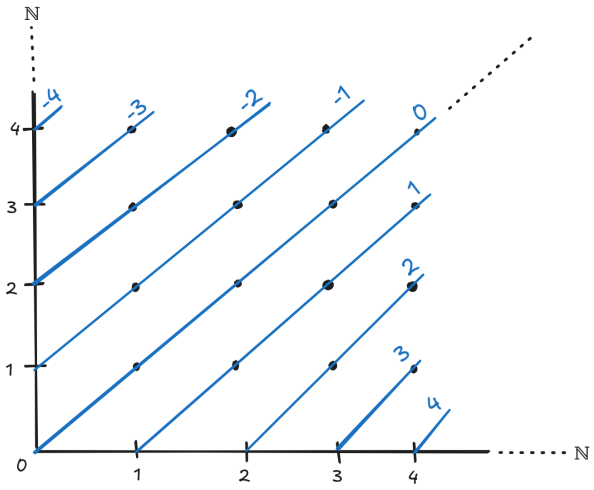


Figure 3: The type generated by int and step — each diagonal line represents one integer.

Intuitively, the paths we've added to our type constrain functions consuming values of said type to not being able to distinguish between different values on each "diagonal". This will become more apparent in the induction principle for $\mathbb{Z}_{\mathrm{hit}}$. That is, given some type family $P : \mathbb{Z}_{\mathrm{hit}} \to \mathcal{U}_l$, what does it take to define a map $\prod_{z:\mathbb{Z}_{\mathrm{hit}}} P(z)$? Such a function needs to both:

1. ...be defined on the "int" constructor. We achieve this by taking an argument of type $p : \prod_{a,b:\mathbb{N}} P(\mathrm{int}(a,b))$. This is reminiscent of induction for pair types.

2. ...preserve the paths generated by "step". That is, for all $a, b : \mathbb{N}$, we need to have some path between $p(a, b)$ and $p(a + 1, b + 1)$. Since the two could have different types (P varies based on the input), we need to make use of the transport operation to formulate the path we are looking for, i.e.

$$\mathrm{tr}_P(\mathrm{step}(a, b), p(a, b)) = p(a + 1, b + 1).$$

Putting it all together, the induction principle for our type looks as follows

$$\mathrm{ind}_{\mathbb{Z}_{\mathrm{hit}}}(P) : \prod_{p:\prod_{a,b:\mathbb{N}} P(\mathrm{int}(a,b))} \left(\prod_{a,b:\mathbb{N}} \mathrm{tr}_P(\mathrm{step}(a, b), p(a, b)) = p(a + 1, b + 1) \right) \to \prod_{z:\mathbb{Z}_{\mathrm{hit}}} P(z).$$

This might look a bit intimidating. In practice, we can define such functions by pattern matching instead, as we will see once we introduce the computation rules into the mix.

For now, let's see what happens in the non-dependent case, i.e. when $P(z) \coloneqq A$ is a constant family. We no longer need to make use of the transport operation (as seen by tr-const, transporting along a constant family yields a point that can be identified with the starting position).

$$\mathrm{rec}_{\mathbb{Z}_{\mathrm{hit}}} : \prod_{p:\mathbb{N}\to\mathbb{N}\to A} \left(\prod_{a,b:\mathbb{N}} p(a, b) = p(a + 1, b + 1) \right) \to (\mathbb{Z}_{\mathrm{hit}} \to A).$$

Notation Remark. Here "rec" stands for the *recursion principle* (also known as *non-dependent elimination*) of the type.

Since $\mathbb{Z}_{\mathrm{hit}}$ is a higher inductive type, the computation rules will not only involve evaluating the resulting function at points inside $\mathbb{Z}_{\mathrm{hit}}$, but also involve the (dependent) action of said function on the paths generated by "step". The computation rule for "int" works as expected

$$\mathrm{ind}_{\mathbb{Z}_{\mathrm{hit}}}(p,\ p_{\mathrm{step}},\ \mathrm{int}(a, b)) \doteq p(a, b).$$

On the other hand, the computation rule for "step" provides us with an identification (not a judgmental equality!) of the action $\mathrm{ind}_{\mathbb{Z}_{\mathrm{hit}}}$ has on step:

$$\mathrm{apd}(\mathrm{ind}_{\mathbb{Z}_{\mathrm{hit}}}(p, p_{\mathrm{step}}),\ \mathrm{step}(a, b)) \doteq p_{\mathrm{step}}(a, b).$$

Having constructed $\mathbb{Z}_{\mathrm{hit}}$ together with its induction principle, we are now ready to attempt to define an equivalence between the two definitions of the integers.

Theorem 1. There exists an equivalence $\mathbb{Z} \simeq \mathbb{Z}_{\mathrm{hit}}$.

Proof. We will denote this equivalence by $\mathcal{C}_{\mathbb{Z}}^{\mathbb{Z}_{\mathrm{hit}}} : \mathbb{Z} \to \mathbb{Z}_{\mathrm{hit}}$. Moreover, we will denote the (both left, and right) inverse by $\mathcal{C}_{\mathbb{Z}_{\mathrm{hit}}}^{\mathbb{Z}} : \mathbb{Z}_{\mathrm{hit}} \to \mathbb{Z}$.

The $\mathbb{Z} \to \mathbb{Z}_{\mathrm{hit}}$ direction is not complicated. In particular, we have

$$\begin{aligned} \mathcal{C}_{\mathbb{Z}}^{\mathbb{Z}_{\mathrm{hit}}}(\mathrm{right}(n)) &\coloneqq \mathrm{int}(n, 0), \\ \mathcal{C}_{\mathbb{Z}}^{\mathbb{Z}_{\mathrm{hit}}}(\mathrm{left}(n)) &\coloneqq \mathrm{int}(0, n + 1). \end{aligned}$$

The $\mathbb{Z}_{\text{hit}} \to \mathbb{Z}$ direction is a bit trickier, since we need to keep track of the paths in $\mathbb{Z}_{\text{hit}}$. On values, the definition follows from our prior definition of $\text{sub}_{\mathbb{N}}$ in equation (6):

$$\mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(\text{int}(a,b)) \coloneqq \text{sub}_{\mathbb{N}}(a,b).$$

Since we are in the non-dependent case, it remains for us to provide paths

$$\text{apd}(\mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}, \text{step}(a,b)) : \mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(a,b) = \mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(a+1,b+1).$$

Notation Remark. Note that we're making a slight abuse of notation here. Of course, apd is a function we've already defined in section 5. We're using this notation to mask the fact we're introducing an implicit p_{step} used for induction when defining $\mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}$, similarly to how pattern matching on constructors for standard inductive types works. Similarly to the aforementioned pattern matching for inductive types, we use this notation because it makes the computation rules for our function easily readable directly from the definition.

In this case, notice that the equality we are looking for holds judgmentally

$$\begin{aligned}\mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(a+1,b+1) &\doteq \text{sub}_{\mathbb{N}}(a+1,b+1)\\ &\doteq \text{sub}_{\mathbb{N}}(a,b)\\ &\doteq \mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(a,b).\end{aligned}$$

We can thus complete our definition by specifying

$$\text{apd}(\mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}, \text{step}(a,b)) \coloneqq \text{refl}\,. \tag{8}$$

Of course, defining the two directions of the map is not enough to prove that $\mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}}$ is an equivalence. We still have to provide the two homotopies, i.e.

$$\begin{aligned}h_{\mathbb{Z}} &: \mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}} \circ \mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}} \sim \text{id}_{\mathbb{Z}},\\ h_{\mathbb{Z}_{\text{hit}}} &: \mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}} \circ \mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}} \sim \text{id}_{\mathbb{Z}_{\text{hit}}}\,.\end{aligned}$$

We'll start with $h_{\mathbb{Z}}$, as it doesn't require us to track any paths. In particular, we can simply fill in the required definition by pattern matching. Recall that in order to construct the required homotopy $h_{\mathbb{Z}}$, we need to construct pointwise paths between $\mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}} \circ \mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}}$ and $\text{id}_{\mathbb{Z}}$. We will proceed by pattern matching:

$$\begin{aligned}h_{\mathbb{Z}}(\text{left}(n)) &\coloneqq \text{refl},\\ h_{\mathbb{Z}}(\text{right}(n)) &\coloneqq \text{refl}\,.\end{aligned}$$

Although a bit anti-climatic, we can use refl in both branches, as the required equalities hold judgmentally:

$$\begin{aligned}\left(\mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}} \circ \mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}}\right)(\text{left}(n)) &\doteq \mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(\mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}}(\text{left}(n)))\\ &\doteq \mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(\text{int}(0,n+1))\\ &\doteq \text{sub}_{\mathbb{N}}(0,n+1)\\ &\doteq \text{left}(n)\\ &\doteq \text{id}_{\mathbb{Z}}(\text{left}(n)),\\ \left(\mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}} \circ \mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}}\right)(\text{right}(n)) &\doteq \mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(\mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}}(\text{right}(n)))\\ &\doteq \mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(\text{int}(n,0))\\ &\doteq \text{sub}_{\mathbb{N}}(n,0)\\ &\doteq \text{right}(n)\\ &\doteq \text{id}_{\mathbb{Z}}(\text{right}(n)).\end{aligned} \tag{9}$$

It remains for us to construct the other homotopy, namely $h_{\mathbb{Z}_{\text{hit}}}$. Similarly to the other direction, we proceed by pattern matching on the "int" constructor:

$$h_{\mathbb{Z}_{\text{hit}}}(\text{int}(a, b)) \coloneqq ?.$$

The required equality does not hold judgmentally in the general $\text{int}(a, b)$ case, thus we have to do a bit more manual work by pattern matching on a and b:

$$\begin{aligned}
h_{\mathbb{Z}_{\text{hit}}}(\text{int}(0_{\mathbb{N}}, 0_{\mathbb{N}})) &\coloneqq \text{refl}, \\
h_{\mathbb{Z}_{\text{hit}}}(\text{int}(a+1, 0_{\mathbb{N}})) &\coloneqq \text{refl}, \\
h_{\mathbb{Z}_{\text{hit}}}(\text{int}(0_{\mathbb{N}}, a+1)) &\coloneqq \text{refl}, \\
h_{\mathbb{Z}_{\text{hit}}}(\text{int}(a+1, b+1)) &\coloneqq ?.
\end{aligned}$$

One can, with a bit of pondering, convince themselves that the first three branches hold judgmentally by unwrapping the definition of $\text{sub}_{\mathbb{N}}$, as we did in equation (9). As for the last branch, that is where our path constructor "step" comes into play. Indeed, being able to identify $\text{int}(a, b)$ with $\text{int}(a+1, b+1)$ is precisely why we introduced this constructor in the first place. In particular, we will construct this path by concatenating three smaller paths

1. First, we go from $\mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}}(\mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(\text{int}(a+1, b+1)))$ to $\mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}}(\mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(\text{int}(a, b)))$ using refl, since the equality holds judgmentally

$$\begin{aligned}
\mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}}(\mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(\text{int}(a+1, b+1))) &\doteq \mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}}(\text{sub}_{\mathbb{N}}(a+1, b+1)) \\
&\doteq \mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}}(\text{sub}_{\mathbb{N}}(a, b)) \\
&\doteq \mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}}(\mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(\text{int}(a, b))).
\end{aligned}$$

2. Then, we use the induction hypothesis $h_{\mathbb{Z}_{\text{hit}}}(\text{int}(a, b))$ to arrive at $\text{int}(a, b)$.

3. Finally, we go from $\text{int}(a, b)$ to $\text{int}(a+1, b+1)$ by using $\text{step}(a, b)$.

Figure 4: Commutative diagram depicting the way we constructed the path

$$\begin{array}{ccc}
\mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}}(\mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(\text{int}(a+1, b+1))) & \overset{\text{refl}}{=\!=} & \mathcal{C}^{\mathbb{Z}_{\text{hit}}}_{\mathbb{Z}}(\mathcal{C}^{\mathbb{Z}}_{\mathbb{Z}_{\text{hit}}}(\text{int}(a, b))) \\
{\scriptstyle h_{\mathbb{Z}_{\text{hit}}}(\text{int}(a,b)) \,\blacksquare\, \text{step}(a,b)} \Big\| & & \Big\| {\scriptstyle h_{\mathbb{Z}_{\text{hit}}}(\text{int}(a,b))} \\
\text{int}(a+1, b+1) & \underset{\text{step}(a,b)}{=\!=\!=} & \text{int}(a, b)
\end{array}$$

Adding this to our definition yields the following map

$$\begin{aligned}
h_{\mathbb{Z}_{\text{hit}}}(\text{int}(0_{\mathbb{N}}, 0_{\mathbb{N}})) &\coloneqq \text{refl}, \\
h_{\mathbb{Z}_{\text{hit}}}(\text{int}(a+1, 0_{\mathbb{N}})) &\coloneqq \text{refl}, \\
h_{\mathbb{Z}_{\text{hit}}}(\text{int}(0_{\mathbb{N}}, a+1)) &\coloneqq \text{refl}, \\
h_{\mathbb{Z}_{\text{hit}}}(\text{int}(a+1, b+1)) &\coloneqq h_{\mathbb{Z}_{\text{hit}}}(\text{int}(a, b)) \,\blacksquare\, \text{step}(a, b).
\end{aligned}$$

Although it might seem like we are very close to the finish line, what remains is constructing

$$\mathrm{apd}(h_{\mathbb{Z}_{\mathrm{hit}}}, \mathrm{step}(a,b)) : \mathrm{tr}(\mathrm{step}(a,b), h_{\mathbb{Z}_{\mathrm{hit}}}(\mathrm{int}(a,b))) = h_{\mathbb{Z}_{\mathrm{hit}}}(\mathrm{int}(a+1,b+1)).$$

We recall from lemma 7 that

$$\begin{aligned}\mathrm{tr}(\mathrm{step}(a,b), h_{\mathbb{Z}_{\mathrm{hit}}}(\mathrm{int}(a,b))) = \mathrm{ap}(\mathcal{C}_{\mathbb{Z}}^{\mathbb{Z}_{\mathrm{hit}}} \circ \mathcal{C}_{\mathbb{Z}_{\mathrm{hit}}}^{\mathbb{Z}}, \mathrm{step}(a,b))^{-1} \\ \blacksquare\ h_{\mathbb{Z}_{\mathrm{hit}}}(\mathrm{int}(a,b)) \\ \blacksquare\ \mathrm{ap}(\mathrm{id}_{\mathbb{Z}_{\mathrm{hit}}}, \mathrm{step}(a,b)).\end{aligned} \tag{10}$$

Moreover, we recall from lemma 5 that $\mathrm{ap}(\mathrm{id}_{\mathbb{Z}_{\mathrm{hit}}}, \mathrm{step}(a,b)) = \mathrm{step}(a,b)$ and

$$\mathrm{ap}(\mathcal{C}_{\mathbb{Z}}^{\mathbb{Z}_{\mathrm{hit}}} \circ \mathcal{C}_{\mathbb{Z}_{\mathrm{hit}}}^{\mathbb{Z}}, \mathrm{step}(a,b)) = \mathrm{ap}(\mathcal{C}_{\mathbb{Z}}^{\mathbb{Z}_{\mathrm{hit}}}, \mathrm{ap}(\mathcal{C}_{\mathbb{Z}_{\mathrm{hit}}}^{\mathbb{Z}}, \mathrm{step}(a,b))).$$

Next, we notice by (8) that $\mathrm{ap}(\mathcal{C}_{\mathbb{Z}_{\mathrm{hit}}}^{\mathbb{Z}}, \mathrm{step}(a,b)) = \mathrm{refl}$, thus

$$\begin{aligned}\mathrm{ap}(\mathcal{C}_{\mathbb{Z}}^{\mathbb{Z}_{\mathrm{hit}}} \circ \mathcal{C}_{\mathbb{Z}_{\mathrm{hit}}}^{\mathbb{Z}}, \mathrm{step}(a,b)) &= \mathrm{ap}(\mathcal{C}_{\mathbb{Z}}^{\mathbb{Z}_{\mathrm{hit}}}, \mathrm{ap}(\mathcal{C}_{\mathbb{Z}_{\mathrm{hit}}}^{\mathbb{Z}}, \mathrm{step}(a,b))) \\ &= \mathrm{ap}(\mathcal{C}_{\mathbb{Z}}^{\mathbb{Z}_{\mathrm{hit}}}, \mathrm{refl}) \\ &= \mathrm{refl}\,.\end{aligned}$$

Substituting this into (10) yields

$$\begin{aligned}\mathrm{tr}(\mathrm{step}(a,b), h_{\mathbb{Z}_{\mathrm{hit}}}(\mathrm{int}(a,b))) &= \mathrm{refl}^{-1} \blacksquare h_{\mathbb{Z}_{\mathrm{hit}}}(\mathrm{int}(a,b)) \blacksquare \mathrm{step}(a,b) \\ &= h_{\mathbb{Z}_{\mathrm{hit}}}(\mathrm{int}(a,b)) \blacksquare \mathrm{step}(a,b) \\ &= h_{\mathbb{Z}_{\mathrm{hit}}}(\mathrm{int}(a+1,b+1)),\end{aligned}$$

where the last equality holds judgmentally. This finalizes our construction!

□

5.3 Contractible types, propositions, and sets

A type A is said to be *contractible* if we can construct an element of type

$$\text{is-contr}_A \coloneqq \sum_{a:A} \prod_{b:A} a = b.$$

Traditionally, a space A is said to be contractible if id_A is homotopically equivalent to const_a (the constant map) for some point $a : A$. The two definitions are definitionally equal, i.e.

$$\sum_{a:A} \mathrm{const}_a \sim \mathrm{id}_A \doteq \sum_{a:A} \prod_{b:A} \mathrm{const}_a(b) = \mathrm{id}_A(b) \doteq \sum_{a:A} \prod_{b:A} a = b \doteq \text{is-contr}_A\,.$$

The first element of the pair (the point $a : A$) is called the *center of contraction*, with the second element being the *contraction* of A. Intuitively, contractible types have a single distinguishable value.

We'll show that contractible types have contractible path types, and are only equivalent to other contractible types.

Lemma 9. Path spaces of contractible types are themselves contractible. Formally, we can construct a map

$$\text{contr-path-space}_A : \text{is-contr}_A \to \prod_{x,y:A} \text{is-contr}_{x=y} .$$

Proof. Let $(a, c) : \text{is-contr}_A$. We'll define the two components of $\text{is-contr}_{x=y}$ separately:

1. The center of contraction needs to have type $x = y$. We can construct this by $c(x)^{-1} \cdot c(y)$, as can be shown in the following diagram

$$x \overset{c(x)^{-1}}{=\!=} a \overset{c(y)}{=\!=} y .$$

2. The contraction on $x = y$ requires us to show that, for a path $p : x = y$, we have $c(x)^{-1} \cdot c(y) = p$. We proceed by path induction on p. Indeed, when p is refl_x, we are left to prove $\text{refl}_x = c(x)^{-1} \cdot c(x)$, which follows by the groupoid laws.

□

Lemma 10. Contractible types are only equivalent to other contractible types. Formally, there is a map

$$\text{contr-equiv}_{A,B} : \prod_{f:A\to B} \text{is-equiv}(f) \to \text{is-contr}_A \to \text{is-contr}_B .$$

Proof. Let $f : A \to B$ be an equivalence, and $(a, c) : \text{is-contr}_A$. We have to construct an element of type is-contr_B. For the center of contraction, we can take $f(a)$. It remains to construct paths $f(a) = b$ for all $b : B$.

Let g be the right-inverse of f. That is, we have a homotopy $h : f \circ g \sim \text{id}_B$. Since we can identify a with $g(b)$ via $c(g(b))$, and this path can be lifted into B via $\text{ap}(f)$, we can use this together with the aforementioned homotopy to complete our path:

$$f(a) \overset{\text{ap}(f,c(g(b)))}{=\!=} f(g(b)) \overset{h(b)}{=\!=} b .$$

□

A simple example of a contractible type is the unit type $\mathbf{1}$ we've defined in section 4.3. To construct a proof that $\text{is-contr}_{\mathbf{1}}$, we can pick $\star$ as the center of contraction. It then remains to show that for any $v : \star$, we have some path $\star = v$. Of course, we can proceed by induction on v. This requires us to construct $\star = \star$, which we can do by reflexivity, i.e.

$$(\star, \text{ind}_{\mathbf{1}}(\text{refl}_\star)) : \text{is-contr}_\star .$$

We've seen that contractible types have contractible path types, but is the converse also the case? A simple counterexample arises in the $\mathbf{0}$ type. Indeed, we can prove by induction, without providing any additional arguments, that given any two points $a, b : \mathbf{0}$, we have $\text{is-contr}_{a=b}$ (of course, one cannot actually provide any points $a, b : \mathbf{0}$, thus any statement about them will hold), yet $\mathbf{0}$ itself is not contractible, as there's no point to take as the center of contraction.

We call such types *propositions*, and denote the type of such proofs by is-prop:

$$\text{is-prop}_A : \prod_{a,b:A} \text{is-contr}_{a=b}\,.$$

The name comes from the idea that propositions in Homotopy Type Theory act similarly to propositions in classical logic. In a sense, propositions have at most 1 distinguishable value. If such a value is given, we can conclude the type is also contractible. The converse turns out to also be true.

Lemma 11. A type A is a proposition precisely when there is a map $A \to \text{is-contr}_A$.

Proof. We'll prove the statement in both directions:

$\to$ If A is a proposition, then we can construct the map as follows. We are given some $a : A$, and must show that is-contr$_A$. Let a be the center of contraction. We are left to construct paths $\prod_{b:A} a = b$, but recall that is-contr$_{a=b}$, hence we can simply take the center of contraction:

$$P : \text{is-prop}_A \to A \to \text{is-contr}_A,$$
$$P(p, a) :\equiv (a, \lambda b.\, \text{pr}_l(p(a, b))).$$

$\leftarrow$ We are given $p : A \to \text{is-contr}_A$ and $a, b : A$, and have to construct a proof that is-contr$_{a=b}$. By $p(a)$ we have a proof that A is contractible. We've already seen a bit earlier that contractible types have contractible path types, thus the rest follows immediately:

$$P : (A \to \text{is-contr}_A) \to \text{is-prop}_A,$$
$$P(p, a, b) :\equiv \text{contr-path-space}_A(p(a), a, b).$$

□

As a corollary of the above, all contractible types are also propositions, since given $p : \text{is-contr}_A$, we can construct the map $\lambda x.p : A \to \text{is-contr}_A$, proving that is-prop$_A$ as well.

As hinted at throughout section 4.5, propositions in Homotopy Type Theory correspond to the classical logic idea of propositions, since we cannot distinguish between their various values. One can define a so-called *propositional truncation operator* in order to turn arbitrary types into propositions. The interested reader is recommended to check out [3, Chapter 14].

An important instance of propositions is "is-equiv", although we will not prove that here (doing so would not be particularly complicated, had we taken the time to characterise paths on Σ-types properly).

Before moving on, it is important to note that, similarly to contractible types, propositions are only equivalent to other propositions.

Lemma 12. There is a map

$$\text{prop-equiv}_{A,B} : \prod_{f:A\to B} \text{is-equiv}(f) \to \text{is-prop}_A \to \text{is-prop}_B\,.$$

Proof. We are given an equivalence $f : A \to B$ together with a proof that is-prop$_A$. Let $b : B$. By lemma 11, it suffices for us to construct an element is-contr$_B$ to conclude our proof. Moreover, by the same lemma, we must have a map $p_a : A \to$ is-contr$_A$. Let $l : B \to A$ be the left-inverse of f. By $p_a(l(b))$ we have en element of type is-contr$_A$, but by lemma 10 this (together with f being an equivalence) induces an element is-contr$_B$, finalizing our proof. □

We can keep adding more levels to our metaphorical tower. In particular, most of the types we've defined earlier (the naturals, integers, etc) have propositions as their path types. We call a type for which the types of paths are always propositions a *set*, and denote proofs of said fact by is-set$_A$. This is in reference to sets (in set theory) having element equality be a proposition. Formally,

$$\text{is-set}_A :\equiv \prod_{a,b:A} \text{is-prop}_{a=b}\,.$$

One thing to note is that all propositions are sets (and as a consequence, all contractible types are sets as well). In particular, both $\mathbf{1}$ and $\mathbf{0}$ are sets. In fact, most of the types we've introduced in section 4 are examples of sets (when given sets as type arguments, that is). We'll set out to prove that $\mathbb{Z}$ is a set, which will require us to do so for both $\mathbb{N}$ and $A + B$ (given that A and B are themselves sets).

Lemma 13. When A and B are sets, so is $A + B$. Formally, there is a map

$$\text{coprod-set}_{A,B} : \text{is-set}_A \to \text{is-set}_B \to \text{is-set}_{A+B}\,.$$

Proof. Before constructing the map, we will show that given points $a : A$ and $b : B$, the existence of a path $\text{left}(a) = \text{right}(b)$ is a contradiction. That is, we will define a map

$$\text{coprod-disj}_{A,B} : \prod_{a:A}\prod_{b:B} \text{left}(a) = \text{right}(b) \to \mathbf{0}.$$

To construct such a map, we define a type family $P(t)$ defined for all $t : A + B$:

$$\begin{aligned} P(\text{left}(a)) &:\equiv \mathbf{1}, \\ P(\text{right}(b)) &:\equiv \mathbf{0}. \end{aligned}$$

The important thing to note here is that we can trivially construct an element $P(\text{left}(a))$ by $\star$, yet having such a value and a path $\text{left}(a) = \text{right}(b)$ would allow us to construct $\mathbf{0}$ by transport, which is exactly what we're looking for:

$$\text{coprod-disj}_{A,B}(a, b, p) :\equiv \text{tr}_P(p, \star) : \mathbf{0}.$$

We can now construct coprod-set by pattern matching:

$$\begin{aligned} &\text{coprod-set}(p_A, p_B, \text{left}(a), \text{left}(b)) &&:\equiv \text{ap}(\text{left}, p_A(a, b)), \\ &\text{coprod-set}(p_A, p_B, \text{right}(a), \text{right}(b)) &&:\equiv \text{ap}(\text{right}, p_B(a, b)), \\ &\text{coprod-set}(p_A, p_B, \text{left}(a), \text{right}(b)) &&:\equiv \lambda q.\, \text{ex-falso}(\text{coprod-disj}_{A,B}(a, b, q)), \\ &\text{coprod-set}(p_A, p_B, \text{right}(a), \text{left}(b)) &&:\equiv \lambda q.\, \text{ex-falso}(\text{coprod-disj}_{A,B}(b, a, q^{-1})). \end{aligned}$$

Let's break the above apart:

1. Every branch is given proofs $p_A : \text{is-set}_A$ and $p_B : \text{is-set}_B$ together with two points in $A + B$. In the case of the first branch, both points are constructed using "left". We can thus simply apply p_A at a and b to produce a proof $\text{is-prop}_{a=b}$, which can be lifted into a proof $\text{is-prop}_{\text{left}(a)=\text{left}(b)}$ by using "ap". The second branch works similarly.

2. On the third branch, we're given points constructed using different constructors (left and right respectively). We have to produce an element of type $\text{is-prop}_{\text{left}(a)=\text{right}(b)}$, but by definition, that is in fact just

$$\text{is-prop}_{\text{left}(a)=\text{right}(b)} \doteq \prod_{q,r:\text{left}(a)=\text{right}(b)} \text{is-contr}_{q=r},$$

thus we construct a mapping from the first path given $(q : \text{left}(a) = \text{right}(b))$, and use the $\text{coprod-disj}_{A,B}$ we've defined earlier to yield a contradiction, which can then be exploited using ex-falso to produce the desired result. The fourth branch works similarly.

□

Lemma 14. $\mathbb{N}$ is a set.

With a bit of work, we could approach this by following the same procedure we applied for coproducts. Nevertheless, we will try to be a bit more systematic. In particular, we will introduce the so-called *observational equality* on the naturals — a type $\text{Eq}_{\mathbb{N}}(a, b)$ defined for $a, b : \mathbb{N}$ that describes the intuitive idea of natural equality, defined by looking at the individual values:

$$\begin{aligned}
\text{Eq}_{\mathbb{N}}(0, 0) &:= \mathbf{1}, \\
\text{Eq}_{\mathbb{N}}(a + 1, b + 1) &:= \text{Eq}_{\mathbb{N}}(a, b), \\
\text{Eq}_{\mathbb{N}}(0, b + 1) &:= \mathbf{0}, \\
\text{Eq}_{\mathbb{N}}(a + 1, 0) &:= \mathbf{0}.
\end{aligned}$$

Does this notion of equality agree with the existence of paths between the naturals? It turns out that indeed, this is the case! The above relation fully characterizes paths on $\mathbb{N}$, and its type is equivalent to that of paths on the naturals.

A significant chunk of this subsection will focus on constructing the aforementioned equivalence. Since introducing easier criterions for constructing equivalences is outside the scope of this thesis[7], we will construct the proof from first principles. Said proof will not make use of any foreign concepts, although it can be easy to get lost in the details. The first time reader is recommended to skip over this proof.

Lemma 15. There exists an equivalence

$$\text{observe}_{\mathbb{N}} : \prod_{a,b:\mathbb{N}} (a = b) \simeq \text{Eq}_{\mathbb{N}}(a, b).$$

Proof. We start by constructing the direction $\text{observe}^{\to}_{\mathbb{N}} : a = b \to \text{Eq}_{\mathbb{N}}(a, b)$ by natural induction on the first argument, and path induction on the rest. Written using pattern matching,

$$\begin{aligned}
\text{observe}^{\to}_{\mathbb{N}}(0, 0, \text{refl}_0) &:= \star, \\
\text{observe}^{\to}_{\mathbb{N}}(a + 1, a + 1, \text{refl}_{a+1}) &:= \text{observe}^{\to}_{\mathbb{N}}(a, a, \text{refl}_a).
\end{aligned}$$

[7] For a different approach, see [3, Theorem 11.3.1]

We will now construct the other direction, namely $\mathrm{observe}^{\leftarrow}_{\mathbb{N}} : \mathrm{Eq}_{\mathbb{N}}(a,b) \to a = b$ by pattern matching yet again:

$$\begin{aligned}
&\mathrm{observe}^{\leftarrow}_{\mathbb{N}}(0,0,p) &&:\equiv \mathrm{refl}_0,\\
&\mathrm{observe}^{\leftarrow}_{\mathbb{N}}(a+1,b+1,p) &&:\equiv \mathrm{ap}(\mathrm{succ}_{\mathbb{N}}, \mathrm{observe}^{\leftarrow}_{\mathbb{N}}(a,b,p)),\\
&\mathrm{observe}^{\leftarrow}_{\mathbb{N}}(0,b+1,p) &&:\equiv \text{ex-falso}(p),\\
&\mathrm{observe}^{\leftarrow}_{\mathbb{N}}(a+1,0,p) &&:\equiv \text{ex-falso}(p).
\end{aligned}$$

Let's break this down:

1. The first branch constructs the result by path reflexivity.
2. The second branch is given $p : \mathrm{Eq}_{\mathbb{N}}(a+1,b+1)$, but by definition we have
$$\mathrm{Eq}_{\mathbb{N}}(a+1,b+1) \doteq \mathrm{Eq}_{\mathbb{N}}(a,b),$$
thus we can simply lift $\mathrm{observe}^{\leftarrow}_{\mathbb{N}}(a,b,p) : a = b$ into $a+1 = b+1$ by applying $\mathrm{succ}_{\mathbb{N}}$ to both sides using "ap".
3. The last two branches make use of the fact that $p : \mathbf{0}$ to produce the required results out of "thin air" (of course, this branch can never be reached in practice).

We now claim this map is an equivalence. In particular, we will show (for all $a, b : \mathbb{N}$) that

$$\text{is-equiv}(\mathrm{observe}^{\rightarrow}_{\mathbb{N}}(a,b)).$$

We already have $\mathrm{observe}^{\leftarrow}_{\mathbb{N}}(a,b)$ as both a left and a right inverse, so it remains to construct homotopies

$$\begin{aligned}
h_{=}(a,b) &: \mathrm{observe}^{\leftarrow}_{\mathbb{N}}(a,b) \circ \mathrm{observe}^{\rightarrow}_{\mathbb{N}}(a,b) \sim \mathrm{id}_{a=b},\\
h_{\mathrm{Eq}_{\mathbb{N}}}(a,b) &: \mathrm{observe}^{\rightarrow}_{\mathbb{N}}(a,b) \circ \mathrm{observe}^{\leftarrow}_{\mathbb{N}}(a,b) \sim \mathrm{id}_{\mathrm{Eq}_{\mathbb{N}}(a,b)}.
\end{aligned}$$

We will start with $h_{=}$, which we define by pattern matching:

$$\begin{aligned}
&h_{=}(0,0,\mathrm{refl}_0) &&:\equiv \mathrm{refl}_{\mathrm{refl}_0},\\
&h_{=}(a+1,a+1,\mathrm{refl}_{a+1}) &&:\equiv ?.
\end{aligned}$$

The second branch is a bit tricky to fill in. We need to construct a path

$$\mathrm{observe}^{\leftarrow}_{\mathbb{N}}(a+1,a+1,\mathrm{observe}^{\rightarrow}_{\mathbb{N}}(a+1,a+1,\mathrm{refl}_{a+1})) = \mathrm{refl}_{a+1}.$$

Unfolding the underlying functions yields the following (judgmentally equal) type

$$\mathrm{ap}(\mathrm{succ}_{\mathbb{N}}, \mathrm{observe}^{\leftarrow}_{\mathbb{N}}(a,a,\mathrm{observe}^{\rightarrow}_{\mathbb{N}}(a,a,\mathrm{refl}_a))) = \mathrm{refl}_{a+1}.$$

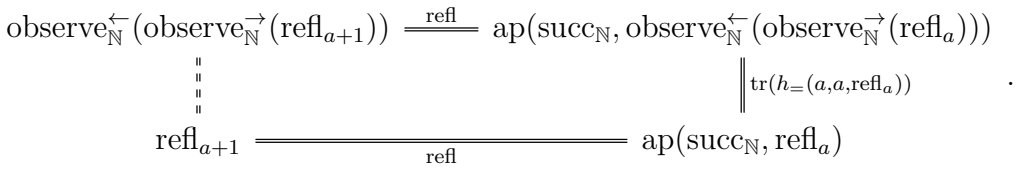
We can transport along

$$h_{=}(a,a,\mathrm{refl}_a)^{-1} : \mathrm{refl}_a = \mathrm{observe}^{\leftarrow}_{\mathbb{N}}(a,a,\mathrm{observe}^{\rightarrow}_{\mathbb{N}}(a,a,\mathrm{refl}_a)),$$

which means we only have to produce an element of type

$$\mathrm{ap}(\mathrm{succ}_{\mathbb{N}}, \mathrm{refl}_a) = \mathrm{refl}_{a+1},$$

which holds judgmentally. The path we've just constructed can be illustrated using the following diagram



Putting the above together completes the homotopy:

$$\begin{aligned} h_{=}(0,0,\mathrm{refl}_0) &:\equiv \mathrm{refl}_{\mathrm{refl}_0}, \\ h_{=}(a+1,a+1,\mathrm{refl}_{a+1}) &:\equiv \mathrm{tr}(h_{=}(a,a,\mathrm{refl}_a), \mathrm{refl}_{\mathrm{refl}_{a+1}}). \end{aligned}$$

Next, we construct $h_{\mathrm{Eq}_{\mathbb{N}}}$ by pattern matching on the two naturals:

$$\begin{aligned} h_{\mathrm{Eq}_{\mathbb{N}}}(0,0,p) &:\equiv \mathrm{refl}_{\star}, \\ h_{\mathrm{Eq}_{\mathbb{N}}}(a+1,b+1,p) &:\equiv ?, \\ h_{\mathrm{Eq}_{\mathbb{N}}}(0,b+1,p) &:\equiv \text{ex-falso}(p), \\ h_{\mathrm{Eq}_{\mathbb{N}}}(a+1,0,p) &:\equiv \text{ex-falso}(p). \end{aligned}$$

Again, the second branch is the tricky bit. We need to construct a path

$$\mathrm{observe}^{\rightarrow}_{\mathbb{N}}(a+1,b+1,\mathrm{observe}^{\leftarrow}_{\mathbb{N}}(a+1,b+1,p)) = p.$$

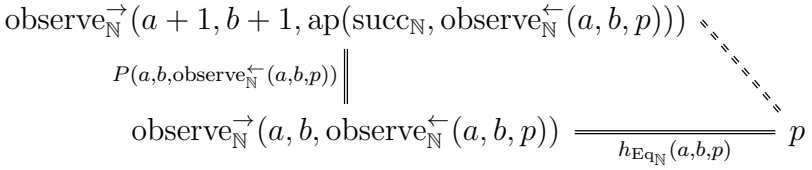
Unfolding the underlying functions yields the following (judgmentally equal) type

$$\mathrm{observe}^{\rightarrow}_{\mathbb{N}}(a+1,b+1,\mathrm{ap}(\mathrm{succ}_{\mathbb{N}},\mathrm{observe}^{\leftarrow}_{\mathbb{N}}(a,b,p))) = p.$$

Notice that we can collapse the first two function applications. Indeed, we have

$$\begin{aligned} &P : \prod_{a,b:\mathbb{N}} \prod_{p:a=b} \mathrm{observe}^{\rightarrow}_{\mathbb{N}}(a+1,b+1,\mathrm{ap}(\mathrm{succ}_{\mathbb{N}},p)) = \mathrm{observe}^{\rightarrow}_{\mathbb{N}}(a,b,p), \\ &P(a,a,\mathrm{refl}_a) :\equiv \mathrm{refl}_{\mathrm{observe}^{\rightarrow}_{\mathbb{N}}(a,a,\mathrm{refl}_a)}. \end{aligned}$$

Together with $h_{\mathrm{Eq}_{\mathbb{N}}}(a,b,p)$, this is enough to complete the path, as can be seen in the following diagram



Substituting this into our original definition finalizes the homotopy

$$\begin{aligned} h_{\mathrm{Eq}_{\mathbb{N}}}(0,0,\star) &:\equiv \mathrm{refl}_{\star}, \\ h_{\mathrm{Eq}_{\mathbb{N}}}(a+1,b+1,p) &:\equiv P(a,b,\mathrm{observe}^{\leftarrow}_{\mathbb{N}}(a,b,p)) \centerdot h_{\mathrm{Eq}_{\mathbb{N}}}(a,b,p), \\ h_{\mathrm{Eq}_{\mathbb{N}}}(0,b+1,p) &:\equiv \text{ex-falso}(p), \\ h_{\mathrm{Eq}_{\mathbb{N}}}(a+1,0,p) &:\equiv \text{ex-falso}(p). \end{aligned}$$

We can now conclude that $a = b$ and $\mathrm{Eq}_{\mathbb{N}}(a,b)$ are indeed equivalent types! □

We now have all the pieces required to conclude that $\mathbb{N}$ is a set. Together with lemma 13, this is enough to conclude that $\mathbb{Z}$ is a set as well.

Proof of lemma 14. Recall from the definition of is-set$_{\mathbb{N}}$ that all we have to do is construct elements is-prop$_{a=b}$ for all $a, b : \mathbb{N}$. By lemma 15, $a = b$ is equivalent to $\mathrm{Eq}_{\mathbb{N}}(a, b)$. Since the latter is always a proposition (this can be proven explicitly by pattern matching, referencing the fact both $\mathbf{0}$ and $\mathbf{1}$ are propositions), it follows by lemma 12 that is-prop$_{a=b}$ as well, finalizing our proof. □

As a last result regarding sets, it is indeed the case that sets (similarly to propositions and contractible types) can only be equivalent to other sets

Lemma 16. There is a map

$$\text{set-equiv}_{A,B} : \prod_{f:A\to B} \text{is-equiv}(f) \to \text{is-set}_A \to \text{is-set}_B\,.$$

To prove this lemma, we will make use of a very useful observation regarding equivalences of propositions. Notably, since propositions have no interesting higher path structure, bidirectional maps between propositions can in fact be turned into equivalences of propositions.

Lemma 17. Given propositions A and B and a bidirectional map $(f, g) : A \leftrightarrow B$, then is-equiv$(f)$ (and by symmetry, is-equiv(g)).

Proof. We will only prove that is-equiv(f) (since the other direction follows by symmetry). First, let g be both the left and right inverse for f. In order to finalize the proof, we must construct homotopies

$$h_A : g \circ f \sim \mathrm{id}_A,$$
$$h_B : f \circ g \sim \mathrm{id}_B\,.$$

Given $a : A$, by $p :$ is-prop$_A$, we have a path $p(g(f(a)), a) : g(f(a)) = a$. This is enough to construct the homotopy h_A (with h_B following by symmetry). □

Proof of lemma 16. Let $f : A \to B$ be an equivalence, and $p :$ is-set$_A$. By the definition of is-set$_B$, we have to construct proofs prop$_{x=y}$ for all $x, y : B$. Let g be the right inverse of f. We will construct a bidirectional map

$$(l, r) : (g(x) = g(y)) \leftrightarrow (x = y).$$

For the $(x = y) \to (g(x) = g(y))$ direction, we can simply take $l :\equiv \mathrm{ap}(g)$. The remaining direction is a tad trickier. In particular, given a path $q : g(x) = g(y)$, we can apply f to get an element $\mathrm{ap}(f, q) : f(g(x)) = f(g(y))$. Since g is the right-inverse for f, we have a homotopy $h : f \circ g \sim \mathrm{id}_B$. We can concatenate paths induced by this homotopy to both ends of $\mathrm{ap}(f, q)$ to get it to have the correct type:

$$r(q) :\equiv h(x)^{-1} \centerdot \mathrm{ap}(f, q) \centerdot h(y).$$

This can more clearly be seen in the following diagram:

$$x \overset{h(x)^{-1}}{=\!=} f(g(x)) \overset{\mathrm{ap}(f,q)}{=\!=} f(g(y)) \overset{h(y)}{=\!=} y\ .$$

By lemma 17, the bidirectional map we've just defined induces an equivalence between $g(x) = g(y)$ and $x = y$. Moreover, by $p(g(x), g(y))$ and lemma 12, it then follows that is-prop$_{x=y}$, finalizing the proof. □

The metaphorical tower we've been investigating does keep going indefinitely. The levels of said tower are called *truncation levels*, and the types therein are labeled using indices starting at -2. That is, contractible types can also be referred to as -2-types, propositions can be referred to as -1-types, sets can be referred to as 0-types, and so on. Many properties generalize to arbitrary levels along the hierarchy. For instance, it is indeed the case that if A is a k-type, and we have an equivalence between A and B, then B is also a k-type. This is not difficult to prove once one establishes that equivalent types have equivalent path types (i.e. [3, Theorem 11.4.2]), although doing so without characterising equivalences further is an arduous task. The interested reader is recommended to check out [3, Chapter 12.4] for an introduction to general truncation levels, and [5, Chapter 4] for a more in-depth treatment of equivalences.

6 The univalence principle

Having briefly explored the idea of identity types, one might wonder — is there a way for us to identify types together? This is what the *univalence principle* (also known as the *univalence axiom*) offers us.

To set the stage, note that paths induce equivalences.

Lemma 18. Given a path $A =_{\mathcal{U}_l} B$, we have an equivalence $A \simeq B$. Formally, there is a map

$$\text{path-equiv}_{A,B} : A =_{\mathcal{U}_l} B \to A \simeq B.$$

Proof. We proceed by path induction. We have to construct an equivalence $A \simeq A$, which we have already constructed in lemma 8. □

We are now ready to state the univalence axiom:

Axiom (The univalence principle)**.** The map path-equiv is itself an equivalence. Formally, there exists an element

$$\text{univalence}_{A,B} : \text{is-equiv}(\text{path-equiv}_{A,B}).$$

While in Homotopy Type Theory as presented by [5] the univalence principle is an axiom (and thus cannot be computed with), [1] presents a specific flavour of Homotopy Type Theory named *Cubical Type Theory*, in which the univalence principle is no longer an axiom, and thus constructive. The interested reader is also recommended to check out [4], which formalizes some mathematics in cubical type theory via the Agda theorem prover.

An example consequence of univalence, as constructed by [4, `1Lab.Univalence`], is *equivalence induction*. Using the univalence principle, we can make the powerful idea of path induction work for equivalences.

Lemma 19 (Equivalence induction)**.** Given a base point $A : \mathcal{U}_l$ and a type family

$$P : \prod_{B:\mathcal{U}_l} A \simeq B \to \mathcal{U}_k,$$

it suffices to construct $P(A, \text{id-equiv})$ to conclude that $P(B, e)$ holds for any equivalence $e : A \simeq B$. Formally, there is a map

$$\text{ind}_{\simeq}(A, P) : P(A, \text{id-equiv}) \to \prod_{e:A\simeq B} P(B, e).$$

Equivalence induction makes proving things like the fact that equivalent types have equivalent path types (which we briefly mentioned at the end of section 5.3) trivial.

Univalence also unlocks a third way to arrive at the integers — taking the loop space of the circle type. Intuitively, the circle is the higher inductive type generated by a single point "base", and a path loop : base = base. This mirrors the classical notion that $\mathbb{Z}$ is the homotopy group of the circle. A more in depth introduction to the circle type is available in [3, Part III].

For a more detailed introduction to univalence, the interested reader is recommended to check out [3, Chapter 17] and [5, Chapter 2.10].

7 Conclusion

Having taken a brief tour through the world of type theory, from the rules of the formal system, to inductive & identity types and the integration of ideas from homotopy, our adventure comes to an end. We hope we've offered a glimpse into what Martin Löf's Type Theory and Homotopy Type Theory have to offer, and how the paradigm of proving things by construction works in practice.

Our through line of constructing the integers in two different ways stands to highlight some of the complexities one must take care of when defining non-trivial equivalences (although of course, there are tools one can use to standardize the process). Finally, our brief overview of univalence is meant to highlight the important role played by equivalences in the theory, as the primary characterization for paths between types.

While this thesis has hardly scratched the surface of the topic, the interested reader is encouraged to delve deeper into the subject by taking a look at both [3] and [5], as they both cover the material a lot more extensively.

References

[1] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. 2016.

[2] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Dissertation, Université de Paris VII, 1972.

[3] Egbert Rijke. *Introduction to Homotopy Type Theory*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2025.

[4] The 1Lab Development Team. The 1Lab, 2024. https://1lab.dev.

[5] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. 2013.