Notes on Natural Numbers

Apiros3

First Version : August 20, 2025
Last Update : – –, 2025

Contents

1 Introduction

Natural numbers act quite often as a very useful tool as a first read on new concepts. Here I attempt to write a list of concepts I've encountered, and how $\mathbb{N}$ acts as a useful tool in giving a concrete example of what is happening, before the idea is further abstracted.

Stuff here is mainly based on the QIIT paper by Ambrus Kaposi.

2 Quotient Inductive Inductive Types

2.1 Type Theory of Signatures

We consider a QIIT with four sorts, $\mathsf{Ty}\ \Gamma$ denotes the well-formed types with free variables in Γ. An element of $\mathsf{Sub}\ \Gamma\ \Delta$ is a list of terms where it contains one term for each type in Δ and each of these have free variables in Γ. An element of $\mathsf{Tm}\ \Gamma\ A$ is a term of type A with free variables in Γ

Con	:	Set	contexts
Ty	:	$\mathsf{Con} \to \mathsf{Set}$	types
Sub	:	$\mathsf{Con} \to \mathsf{Con} \to \mathsf{Set}$	substitutions
Tm	:	$(\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}$	terms

2.1.1 Substitution Calculus

The following constructors give the substitution calculus part of the syntax.

$\cdot$	:	Con	empty context
$- \triangleright -$	:	$(\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Con}$	context extension
$-[-]$	:	$\mathsf{Ty}\,\Delta \to \mathsf{Sub}\,\Gamma\,\Delta \to \mathsf{Ty}\,\Gamma$	substitution of types
id	:	$\mathsf{Sub}\,\Gamma\,\Gamma$	identity substitution
$- \circ -$	:	$\mathsf{Sub}\,\Theta\,\Delta \to \mathsf{Sub}\,\Gamma\,\Theta \to \mathsf{Sub}\,\Gamma\,\Delta$	composition
ε	:	$\mathsf{Sub}\,\Gamma\,\cdot$	empty substitution
$-, -$	:	$(\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to \mathsf{Tm}\,\Gamma\,A[\sigma] \to \mathsf{Sub}\,\Gamma\,(\Delta \triangleright A)$	substitution extension
π_1	:	$\mathsf{Sub}\,\Gamma\,(\Delta \triangleright A) \to \mathsf{Sub}\,\Gamma\,\Delta$	first projection
π_2	:	$(\sigma : \mathsf{Sub}\,\Gamma\,(\Delta \triangleright A)) \to \mathsf{Tm}\,\Gamma\,A[\pi_1\sigma]$	second projection
$-[-]$	:	$\mathsf{Tm}\,\Delta\,A \to (\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to \mathsf{Tm}\,\Gamma\,A[\sigma]$	substitution of terms
[id]	:	$A[\mathrm{id}] = A$	
$[\circ]$	:	$A[\sigma \circ \delta] = A[\sigma][\delta]$	
ass	:	$(\sigma \circ \delta) \circ \nu = \sigma \circ (\delta \circ \nu)$	
idl	:	$\mathrm{id} \circ \sigma = \sigma$	
idr	:	$\sigma \circ \mathrm{id} = \sigma$	
$\cdot\eta$	:	$\{\sigma : \mathsf{Sub}\,\Gamma\,\cdot\} \to \sigma = \varepsilon$	
$\triangleright\beta_1$	:	$\pi_1(\sigma, t) = \sigma$	
$\triangleright\beta_2$	:	$\pi_2(\sigma, t) = t$	
$\triangleright\eta$	:	$(\pi_1\sigma, \pi_2\sigma) = \sigma$	
$,\circ$	:	$(\sigma, t) \circ \delta = (\sigma \circ \delta,\ t[\delta])$	

The main syntax is captioned, while the laws are given explicit names and types. The operations are what one would expect, including the β-reduction and η-expansion rules.

2.2 Algebras

We give the notion of algebras by induction on the syntax of the theory of Signatures. We have:

$$\begin{aligned}(\Gamma : \mathsf{Con})^A &: \mathsf{Set}\\ (A : \mathsf{Ty}\,\Gamma)^A &: \Gamma^A \to \mathsf{Set}\\ (\sigma : \mathsf{Sub}\,\Gamma\,\Delta)^A &: \Gamma^A \to \Delta^A\\ (t : \mathsf{Tm}\,\Gamma\,A)^A &: (\gamma : \Gamma^A) \to A^A\,\gamma\end{aligned}$$

3 MLTT

3.1 In Functional Programming

In Haskell, creating an object that behaves in a way that we understand the natural numbers is straightforward. Simply,

```
data Nat where
    Zero  :: Nat
  | Succ  :: Nat → Nat
```

or isomorphically,

```
data Nat where
    Zero  :: ()  → Nat
  | Succ  :: Nat → Nat
```

3.2 As an Inductive-Inductive Type

Metatheoretically, in the context of Type Theory, we have

```
Nat : Type
zero : Nat
succ : Nat → Nat
```

Definition 3.2.1. *An **Inductive-Inductive Type** simultaneously introduces a type* `A : Type` *and an* `A`*-indexed family* `B : A → Type` *both by induction, where constructors of* `A` *can mention* `B` *and constructors of* `B` *can mention* `A` *in strictly positive ways.*

3.2.1 Nat as an Algebra

The following three element context is the signature for natural numbers. It has one sort and two operators:

$$\Delta :\equiv (Nat : \mathtt{U}, zero : \mathtt{El}\ Nat, suc : Nat \Rightarrow \mathtt{El}\ Nat)$$

Writing $-^A$ for the set of algebras given a signature, we can compute

$$\Delta^{\mathrm{A}} \equiv (N : \mathtt{Set}) \times N \times (N \to N)$$

We write the initial algebra by $\mathrm{con}_\Delta : \Delta^{\mathrm{A}}$. Explicitly, we have

$$\mathrm{con}_\Delta \equiv (\mathtt{Tm}\ \Delta\ (\mathtt{El}\ Nat),\ zero,\ \lambda t.suc\ \mathtt{@}\ t)$$

The dependent algebra over an algebra (N, z, s), written $-^{\mathrm{D}}$, consists of the proof-relevant predicate over N, a witness of the predicate at z and a proof that s respects the predicate. Thus,

$$\Delta^{\mathrm{D}}(N, z, s) \equiv (N^{\mathrm{D}} : N \to \mathtt{Set} \times N^{\mathrm{D}}\ z \times ((x : N)\ \to N^{\mathrm{D}}\ x \to N^{\mathrm{D}}\ (s\ x)))$$

3.3 Metatheory to Algebra

A signature on `Nat` looks something like

```
Nat  : Set
zero : Nat
succ : Nat → Nat
```

Then we can define algebras for this signature:

```
Record Nat_A : Type := {
    N : Set;
    z : A;
    s : A → A
}
```

We can realize this signature as an inductive type in the usual way, which generates the initial algebra:

```
Inductive nat : Set :=
    | O : nat
    | S : nat → nat.

Definition I : Nat_A := {|
    N := nat;
    z := O;
    s := S
|}
```

The displayed `Nat`-algebra over I is a family over the carrier set together with display evidence matching the operators. Thus, we can write

```
Record Nat_D (Γ : Nat_A) : Type := {
    N_D : N Γ → Set;
    z_D : N_D (z Γ);
    s_D : forall n, N_D n → N_D (s Γ n)
}.
```

Then, we can give the induction principle:

```
Fixpoint indNat {Γ : Nat_A} (Γ_D : Nat_D Γ) (n : N Γ) : N_D Γ_D n :=
    match n with
        | O    ⇒ z_D Γ_D
        | S n' ⇒ s_D Γ_D n' (indNat Γ_D n')
```

Omitting the explicit algebra for readibility, up to abuse of notation, we have

```
Variable Γ : Nat_A.
Variable Γ_D : Nat_D Γ.

Record Nat_D : Type := {
    N_D : N → Set;
    z_D : N_D z;
    s_D : forall n, N_D n → N_D (s n)
}.

Fixpoint indNat (n : N) : N_D n :=
    match n with
        | O    ⇒ z_D
        | S n' ⇒ s_D n' (indNat n')
```

which gives us the standard induction on `Nat` for an arbitrary `Γ : Nat_A`, given that we can provide an instance for the corresponding `Nat_D Γ`.

For example, consider

4 Other Prerequisites

4.1 MLTT

4.1.1 Dependent Types

Definition 4.1.1. *A **dependent type** is a type that can mention / depend on a term.*

If $A : \texttt{Type}$ and we have a type family $B : A \to \texttt{Type}$, then for every $a : A$, we have a fiber $B(a)$. Generally, we have dependent Π-types (dependent functions) and Σ-types (dependent pairs).

A Π-type with notation $\Pi_{x:A}B(x)$ is the type of functions that given a $x : A$, returns a result in the corresponding fiber $B(x)$. If B doesn't depend on x, then $\Pi_{x:A}B \simeq A \to B$. This comes along with formation,

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma, x : A \vdash B : \texttt{Type}}{\Gamma \vdash \Pi(x : A).B : \texttt{Type}}$$

Introduction (pairing),

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : \Pi(x : A).B}$$

Elimination (application),

$$\frac{\Gamma \vdash f : \Pi(x : A).B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[a/x]}$$

Computation (β),

$$(\lambda x.b)a \equiv b[a/x]$$

Often alongside the η-principle (uniqueness) at a judgement or propositional level, though not included in vanilla MLTT.

Under the Curry-Howard correspondence, $\Pi_{x:A}B(x)$ corresponds to $\forall x : A.B(x)$. That is, a term of this type is a function that given a witness $x : A$, produces a proof of $B(x)$.

A Σ-type with notation $\Sigma_{x:A}B(x)$ is the type of pairs where the second component's type depends on the first. An element is (a, b) with $a : A$ and $b : B(a)$. If B does not depend on x, then $\Sigma_{x:A}B \simeq A \times B$. Similar to Π-types, this comes with formation,

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma, x : A \vdash B : \texttt{Type}}{\Gamma \vdash \Sigma(x : A).B : \texttt{Type}}$$

Introduction (pairing),

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \Sigma(x : A).B}$$

Elimination (projection / dependent eliminator),

$$\texttt{fst} : (\Sigma(x : A).B) \to A \qquad \texttt{snd} : \prod_{p:\Sigma(x:A).B} B(\texttt{fst}\ p)$$

In particular, for $p : (\Sigma(x : A).B)$, we have $\texttt{fst}\ p : A$ and $\texttt{snd}\ p : B(\texttt{fst}\ p)$. Then we have computation (β),

$$\texttt{fst}(a, b) \equiv a \qquad \texttt{snd}(a, b) \equiv b$$

Under the curry-howard isomorphism, $\Sigma_{x:A}B(x)$ corresponds to $\exists x : A.B(x)$ constructively. That is, a witness a together with a proof $B(a)$.

In the context of sets, given a family $B : A \to \texttt{Set}$, we get

- $\Pi_{x\in A}B_x$: choice functions picking $b_x \in B_x$ for each $x \in A$
- $\Sigma_{x\in A}B_x$: disjoint union $\biguplus_{x\in A}\{x\} \times B_x$

Categorically, for a map $f : X \to A$, pulling back families along f has adjoints:

- Right adjoint Π_f : dependent product (universal quantification)
- Left adjoint Σ_f : dependent sum (existential / pushforward)

Some intuitive examples:

- $\texttt{Vec}(A, n) : \texttt{Type}$ depends on $n : \mathbb{N}$
  - Head has type $\Pi_{n:\mathbb{N}}\Pi_{v:\texttt{Vec}(A,\texttt{suc}\ n)}A$
  - A value with its length lives in $\Sigma_{n:\mathbb{N}}\texttt{Vec}(A, n)$
- A sorted list with a proof it is sorted would have the type $\Sigma_{\ell:\texttt{List}\ A}\ \texttt{Sorted}(\ell)$

4.1.2 Universes

We use the notion of *types of types* to avoid size paradoxes (i.e. Russell's paradox). Hence we use a predicative ladder

$$\texttt{Type}_0 : \texttt{Type}_1 : \texttt{Type}_2 : \cdots$$

usually along with a notion of cumulativity, such that if $A : \texttt{Type}_i$ then $A : \texttt{Type}_{i+1}$. Then by universe polymorphism, we can write level-generic code.

We have (generally) two universe styles, the Russell-style and Tarski-style. Over a Russell-style universe, judgements are of the form $\texttt{A} : \texttt{Type}_i$, such that type formers return types directly:

$$\frac{A : \texttt{Type}_i \qquad (x : A) \vdash B : \texttt{Type}_j}{\prod(x : A).B : \texttt{Type}_{\max(i,j)}}$$

In that sense, the universe acts like a set of types, and it's members are types. Hence the saying "types are elements".

Contrary to this, over a Tarski-style universe, the universe is instead a *type of codes* $\texttt{U}_i : \texttt{Type}_{i+1}$ plus some decoding $\texttt{El}_i : \texttt{U}_i : \texttt{Type}_i$. Closure under type formers is then given by *codes*. We make this formal:

Definition 4.1.2. *A **Tarski-Style Universe** is a type* `U : Type` *whose elements are codes (names) for small types.*

Definition 4.1.3. *An **element**, written* `El` *is a function that turns a code into the type it names. Explicitly, we have* `El : U → Type` *such that if* `u : U` *is a code, then* `El u` *is the actual type it denotes.*

In a Tarski Universe, we inductively define what types of codes exist, and recursively say what that means as an element of `Type` via `El` (so to be more precise, a Tarski Style Universe is a pair `(U, El)`). The `El` ensures that the constructors of the datatype are strictly positive. For example, we can consider the following toy example:

```
data U : Type where
    uUnit  : U
    uSigma : (a : U) → (El a → U) → U
    uPi    : (a : U) → (El a → U) → U
```

Defines a small universe explicitly, then we define

```
El : U → Type
El uUnit        = Unit
El (uSigma a b) = Σ (x : El a), El (b x)
El (uPi a b)    = (x : El a) → El (b x)
```

which gives concrete meaning to `uSigma` and `uPi`. We write `El u1 = Unit` for simplicity.

Tarski Style Universes aid in internalizing a general schema for inductive-inductive types, as this universe encode allowed constructors, indices, parameters, etc, and from which one can interpret the actual data types as initial algebras.

Over a Russell Style Universe, types are elements of $\texttt{Type}_\ell$, but there is no data that can be inspected. In plain MLTT, we can define a function like

$$F : \texttt{Type}_\ell \to \texttt{Type}_\ell \qquad F\ A = A \times A$$

but cannot define a function

$$G : \texttt{Type}_\ell \to \mathbb{N}$$

by "pattern matching" on A. That is, there is no eliminator that reveals the structure of an arbitrary $A : \texttt{Type}$. In contrast, over a Tarski style universe, as `U` is an inductive family of codes, we can do recursive pattern matching / induction.

As a toy model, we can a Russell style identity function which probably looks like:

```
id : Π (A : Type U), A → A
id A x = x
```

whereas under a Tarski style it would look like

```
id : Π (A : U), El A → El A
id A x = x
```

4.2 Terms, Types, and Prop

4.3 First Order Logic to Type Theory

4.4 Intrinsic and Extrinsic Representation

In general, an extrinsic representation is one which has some independent separation between the object and a property/proof. In contrast, an intrinsic representation will carry that property inside that definition. For instance, the code

```
xs : List A
p : length xs = 5
```

using `List` makes the property that `xs` has length 5 becomes extrinsic. In contrast, if we just had some

```
xs : Vec A 5
```

Then the length of the list is an intrinsic property that is embedded within the construction itself.

Lambda calculus has an extrinsic representation where terms are made by an abstract syntax tree, and then we give an external typing judgement to rule out unwanted terms (like Ω).

We give a toy example of this. First consider the raw syntax of types, contexts, variables, and terms:

```
A ::=  A ⇒ A | Bool
Γ ::= ◇ | Γ, A
x ::= ∅ | suc x
t ::= x | lam x | t @ t | true | false | ite t t t
```

This clearly contains unwanted terms like `true @ false`, which we remove by a typing judgement, in our case the following:

In contrast we can make this intrinsic by defining terms to be dependent on the context and type like

```
data Var : Con → Ty → Set where
    ∅   : Var (Γ, A) A
    suc : Var Γ A → Var (Γ, B) A

data Tm : Con → Ty → Set where
    var : Var Γ A → Tm Γ A
    lam : Tm (Γ, A) B → Tm Γ (A → B)
    _@_ : Tm Γ (A → B) → Tm Γ A → Tm Γ B
```

If we want β-reduction, we can just quotient by the equality

```
β : lam t @ u = t [id, u]
```

4.5 Interpretation

4.5.1 Set Interpretation

Definition 4.5.1. *The **set interpretation** is the interpretation into the set model. The operation "$[\![-]\!]$" gives the set interpretation, given a signature and a context/term.*

The set interpretation of types is what one would expect, where $[\![A]\!] : \texttt{Set}$ and we have the standard rules like

$$\begin{aligned}
[\![A \Rightarrow B]\!] &= [\![A]\!] \to [\![B]\!] \\
[\![\texttt{Prod}\ A\ B]\!] &= [\![A]\!] \times [\![B]\!] \\
[\![\texttt{Bool}]\!] &= \mathbb{2} \\
[\![\texttt{Nat}]\!] &= \mathbb{N} \\
[\![\texttt{Unit}]\!] &= \mathbb{1}
\end{aligned}$$

Then, the set interpretation of contexts is also straightforward, recursively defined as

$$[\![\Gamma]\!] : \texttt{Set} \qquad \begin{cases} [\![\diamond]\!] = \mathbb{1} \\ [\![\Gamma, A]\!] = [\![\Gamma]\!] \times [\![A]\!] \end{cases}$$

Finally, the set interpretation of terms depends on the context and returns the type based on it, so when we have $\Gamma \vdash t : A$, we should have

$$[\![t]\!] : [\![\Gamma]\!] \to [\![A]\!]$$

As a sanity check, consider $\Gamma, A \vdash \emptyset : A$. We get what we expect,

$$[\![\emptyset]\!] : [\![\Gamma]\!] \times [\![A]\!] \to [\![A]\!] \qquad [\![\emptyset]\!] = \text{snd}$$

Now recall the inference rule

$$\frac{\Gamma \vdash x : A}{\Gamma, B \vdash \texttt{suc}\ x : A}$$

So we have

$$[\![\texttt{suc}\ x]\!] : [\![\Gamma]\!] \times [\![B]\!] \to [\![A]\!]$$

given

$$[\![x]\!] : [\![\Gamma]\!] \to [\![A]\!]$$

Thus the only reasonable function we have is

$$[\![\texttt{suc}\ x]\!] = [\![x]\!] \circ \text{fst}$$

As another example, consider the abstraction rule

$$\frac{\Gamma, A \vdash t : B}{\Gamma \vdash \texttt{lam}\ t : A \Rightarrow B}$$

Then we have that

$$[\![\texttt{lam}\ t]\!] : [\![\Gamma]\!] \to [\![A \Rightarrow B]\!] = [\![\Gamma]\!] \to [\![A]\!] \to [\![B]\!]$$

along with

$$[\![t]\!] : [\![\Gamma]\!] \times [\![A]\!] \to [\![B]\!]$$

Hence

$$[\![\texttt{lam}\ t]\!] = \text{curry}_2\ [\![t]\!]$$

where the curry_2 function is one which currys a function with a 2-tuple input. More intuitively,

$$[\![\texttt{lam}\ t]\!]\gamma = \lambda\alpha.[\![t]\!](\gamma, \alpha)$$

In a dependent type theory setting, types can depend on the context. Hence, while $[\![\Gamma]\!] : \mathtt{Set}$, we expect $[\![A]\!] : [\![\Gamma]\!] \to \mathtt{Set}$. Terms hence depend on a context and returns a type that depends on the input context. Explicitly, we have $[\![t]\!] : (\gamma : [\![\Gamma]\!]) \to [\![A]\!]\gamma$.

Consider the usual example

$$\mathtt{Vec}_{\mathtt{Bool}} : \mathtt{Nat} \to \mathtt{Type}$$

alongisde a function `trues` that given a natural number returns a vector of that length with only trues:

$$\mathtt{trues} : (n : \mathtt{Nat}) \to \mathtt{Vec}_{\mathtt{Bool}}\ n$$

We can then type something like

$$\diamond, \mathtt{Nat} \vdash \mathtt{trues}\ @\ \emptyset : \mathtt{Vec}_{\mathtt{Bool}}\emptyset$$

Then we have

$$[\![\mathtt{trues}\ @\ \emptyset]\!] : (\gamma : [\![\diamond, \mathtt{Nat}]\!]) \to [\![\mathtt{Vec}_{\mathtt{Bool}}\emptyset]\!]\gamma$$

Noting that $[\![\diamond, \mathtt{Nat}]\!] = \top \times \mathbb{N}$ and $[\![\mathtt{Vec}_{\mathtt{Bool}}\emptyset]\!]\gamma = \mathbb{V}\mathrm{ec}_2([\![\emptyset]\!]\ \gamma)$, this simplifies to

$$[\![\mathtt{trues}\ @\ \emptyset]\!] : (\gamma : \top \times \mathbb{N}) \to \mathbb{V}\mathrm{ec}_2([\![\emptyset]\!]\ \gamma)$$

Rewriting $\gamma := (tt, n)$ and noting that $[\![\emptyset]\!] = \mathrm{snd}$, we have

$$[\![\mathtt{trues}\ @\ \emptyset]\!] : ((tt, n) : \top \times \mathbb{N}) \to \mathbb{V}\mathrm{ec}_2\ n$$

4.5.2 Display Algebra / Dependent Algebra

Given an algebra, the display algebra is the family version of it that lives over it, such that it can model dependent elimination.

Here, a context is interpreted as a predicate $\Gamma^{\mathrm{D}} : \Gamma^{\mathrm{A}} \to \mathtt{Set}$. A type $A : \mathtt{Ty}\ \Gamma$ becomes a predicate that depends on a witness of Γ^{D}, which explictly is $A^{\mathrm{D}}\ \{\gamma : \Gamma^{\mathrm{A}}\} : \Gamma^{\mathrm{D}}\ \gamma \to A^{\mathrm{A}}\ \gamma \to \mathtt{Set}$. Finally, a term $t : \mathtt{Tm}\ \Gamma\ A$ is interpreted as $t^{\mathrm{D}}\ \{\gamma : \Gamma^{\mathrm{A}}\} : (\gamma^{\mathrm{D}} : \Gamma^{\mathrm{D}}\ \gamma) \to A^{\mathrm{D}}\ \gamma^{\mathrm{D}}\ (t^{\mathrm{A}}\ \gamma)$. Intuitively, t^{A} says what the value of t is in the base algebra, and t^{D} says how to lift t into the display algebra (that is, given a display evidence γ^{D} for the context, it produces a display evidence that t has type A, or equivalently a witness in the fiber $A^{\mathrm{D}}\ \gamma^{\mathrm{D}}\ (t^{\mathrm{A}}\ \gamma)$).

For instance, given as an algebraic signature (metatheory), we have

```
Bool : Set
true : Bool
false : Bool
```

An algebra for this signature can be seen as

```
Record Bool_alg : Type := {
    A : Set;
    true_alg : A;
    false_alg : A
}.
```

We can think of the inductive `Bool` type as

```
Inductive Bool_I : Set
    | true_I : Bool_I
    | false_I : Bool_I.
```

which we can then package as

```
Definition I : Bool_alg := {|
    A := Bool;
    true_alg := true;
    false_alg := false
|}.
```

Then this induces a displayed algebra

```
Bool• : Bool_I → Set
true• : Bool• true_I
false• : Bool• false_I
```

which gives the induction principle on bool:

```
ind_I : (b : Bool_I) → Bool• b
```

4.5.3 Morphism

An algebra homomorphism is given by a function between sets which respects operators, with propositional equality being denoted $=$.

We write $-^{\mathrm{M}}$ to represent morphisms. If Δ is an algebra, we have $\Delta^{\mathrm{M}} : \Delta^{\mathrm{A}} \to \Delta^{\mathrm{A}} \to \texttt{Set}$. We interpret contexts as a relation $\Gamma^{\mathrm{M}} : \Gamma^{\mathrm{A}} \to \Gamma^{\mathrm{A}} \to \texttt{Set}$.

4.5.4 Modified Dependent Interpretation

4.5.5 Fiber?

Definition 4.5.2. *A **fiber** of an element y under a function f is the preimage of the singleton set $\{y\}$.*

In the scope of these notes, it is the set that sits over a base element in the family. Explicitly, if $B : A \to \texttt{Set}$ is a family, then for each $a : A$, the fiber of B over a is just $B(a)$.

In the context of this subsection, given an inductive type W, the base algebra gives an element $w : W$, given a dependent family $P : W \to \texttt{Type}$ with constructor cases (inductive hypotheses), the display algebra gives the fiber $P(w)$, and the display evidence is an element of that fiber. For instance, for $P : \mathbb{N} \to \texttt{Set}$, the fiber over 3 is $P(3)$, and a proof $p : P(3)$ is the display evidence living in that fiber.

4.6 Interpretation, Categorically

4.6.1 Signatures / Endofunctors

A signature Σ for an inductive object induces an endofunctor $F : \mathcal{C} \to \mathcal{C}$.

We outline some simple examples:

- Naturals

Naturals come with shapes $S = 1 + 1$ corresponding to zero and succ. The positions are given by $P(\mathsf{zero}) = \emptyset$ and $P(\mathsf{succ}) = 1$, which gives the functor $F(X) = 1 + X$, alongisde an induced initial algebra `Nat` with `zero : Nat` and `suc : Nat -> Nat`. As usual, we can write this out explicitly:

```
data Nat : Set where
    zero :: Nat
  | succ :: Nat → Nat
```

- Lists over some A

Given a fixed $A : \mathsf{Type}$, we have shapes $S = 1 + A$ corresponding to nil and $\mathsf{cons}\ a$. Positions are then given by $P(\mathsf{nil}) = \emptyset$ and $P(\mathsf{cons}\ a) = 1$, which gives the functor $F(X) = 1 + A \times X$, giving initial algebra

```
data List (A : Set) : Set where
    nil  :: List A
    cons :: A → List A → List A
```

- Binary Trees with B-laveled leaves

Given a fixed $B : \mathsf{Type}$, we have shapes $S = B + 1$ corresponding to leaf and node (functionally we can view this like $S \simeq B \uplus 1$ and `inl b` to mean leaf with label b and `inr *` to mean the binary node). We have positions $P(\mathsf{inl}\ b) = \emptyset$ and $P(\mathsf{inr}\ *) = 2$.

This gives a polynomial functor

$$F(X) = B + X \times X$$

Alongside the initial algebra

```
data Tree (B : Set) : Set where
    leaf : B → Tree B
    node : Tree B → Tree B → Tree B
```

```
record Container : Set1 where
    field Shape : Set
          Pos   : Shape → Set
```

4.7 Equivalences, Choice

TODO: ext, eta, strength, choice?

5 Todo

1. Free monoid over A is just a List
2. Quotienting representation without quotients (eg. integer, lambda calculus)
3. Decidability (some things can't be done, say CL-logic, real numbers, no normal form (else halting))
4. $\Sigma(A : Set)(A \to I) \simeq (I \to Set)$
5. locally cartesian closed categories
6. Dependent algebra can capture equality as well, useful for structural induction (consider the leaf counting function and node counting function)

6 Categories

6.1 Basic Definitions

Definition 6.1.1. *A category $\mathcal{C}$ consists of the following data:*

- *A collection* $\text{ob}\ \mathcal{C}$ *of objects of* $\mathcal{C}$
- *For every* $x, y \in \text{ob}\ \mathcal{C}$ *a collection* $\text{Hom}_{\mathcal{C}}(x, y)$ *of morphisms*
- *For every* $x \in \text{ob}\ \mathcal{C}$*, the identity morphism* $\text{id}_x \in \text{Hom}_{\mathcal{C}}(x, x)$
- *For every* $x, y, z \in \text{ob}\ \mathcal{C}$*, the composition map*

$$\circ : \text{Hom}_{\mathcal{C}}(y, z) \times \text{Hom}_{\mathcal{C}}(x, y) \to \text{Hom}_{\mathcal{C}}(x, z)$$

These then must satisfy the following axioms:

- *For any two* $x, y \in \text{ob}\ \mathcal{C}$ *and any morphism* $f \in \text{Hom}_{\mathcal{C}}(x, y)$ *we have*

$$f \circ \text{id}_x = f \text{id}_y \circ f = f$$

- *Morphisms under composition are associative*

Notation 6.1.2. We write $x \in \mathcal{C}$ for $x \in ob\ \mathcal{C}$ and omit the subscript in *Hom* when the category is clear. We may sometimes write $\mathcal{C}(x, y)$ for $Hom_{\mathcal{C}}(x, y)$. We also write $Hom_{\mathcal{C}}(x)$ for $Hom_{\mathcal{C}}(x, x)$ and call these endomorphisms.

Definition 6.1.3. *Given a category* $\mathcal{C}$*, we have the* ***opposite category*** $\mathcal{C}^{\text{op}}$*, which has the same objects and* $\text{Hom}_{\mathcal{C}^{\text{op}}}(x, y) = \text{Hom}_{\mathcal{C}}(y, x)$*.*

6.1.1 Functor

Definition 6.1.4. *A functor* $F : \mathcal{C} \to \mathcal{D}$ *between two categories* $\mathcal{C}, \mathcal{D}$ *consists of the following data:*

- *a map* $F : \text{ob}\ \mathcal{C} \to \text{ob}\ \mathcal{D}$
- *For any two objects* $x, y \in \mathcal{C}$*, a map of sets* $F : \text{Hom}_{\mathcal{C}}(x, y) \to \text{Hom}_{\mathcal{D}}(F(x), F(y))$

Such that they satisfy

- *Unit: for any* $x \in \mathcal{C}$*,* $F(\text{id}_x) = \text{id}_{F(x)}$
- *For any objects* $x, y, z \in \mathcal{C}$ *and morphisms* $f \in \text{Hom}_{\mathcal{C}}(x, y)$ *and* $g \in \text{Hom}_{\mathcal{C}}(y, z)$*, we have*

$$F(g \circ f) = F(g) \circ F(f)$$

We say that a functor $F : \mathcal{C} \to \mathcal{D}$ *is* ***faithful*** *if the map* $\text{Hom}_{\mathcal{C}}(x, y) \to \text{Hom}_{\mathcal{D}}(F(x), F(y))$ *is injective for any objects* x *and* y*. We say that* F *is* ***full*** *if this map is surjective, and that it is* ***fully faithful*** *if it is both full and faithful.*

We say that F *is a* ***contravariant functor*** *from* $\mathcal{C}$ *to* $\mathcal{D}$ *if it is a functor* $\mathcal{C}^{\text{op}} \to D$*. A functor* $\mathcal{C}$ *to* $\mathcal{D}$ *is also referred to as a covariant functor.*

Note we use the same F to refer to the map between both objects and morphisms.

Definition 6.1.5. *A morphism $f \in \mathrm{Hom}(x, y)$ in a category is an **isomorphism** if there is a morphism $f^{-1} \in \mathrm{Hom}(y, x)$ such that $f^{-1} \circ f = \mathrm{id}_x$ and $f \circ f^{-1} = \mathrm{id}_y$. We also say that f is **invertible**. If two objects $x, y \in \mathcal{C}$ are isomorphic, we write $x \cong y$.*

Remark 6.1.6. As usual, inverses are unique. Suppose that $f \in Hom(x, y)$ and we have $g \circ f = id_x$ and $f \circ h = id_y$. Then,

$$g = g \circ id_y = g \circ (f \circ h) = (g \circ f) \circ h = id_x \circ h$$

Definition 6.1.7. *A category $\mathcal{C}$ is called a **groupoid** if every morphism is invertible. We say that a groupoid is **connected** if any two objects are isomorphic.*

Remark 6.1.8. Consider a groupoid with a single object. The data required to specify such a groupoid is the monoid of endomorphisms in which every object has an inverse. This is just a group.

6.1.2 Natural Transformation

Definition 6.1.9. *Let $F, G : \mathcal{C} \to \mathcal{D}$ be two functors. A natural transformation $\eta : F \Rightarrow G$ consists of morphisms $\eta_x \in \mathrm{Hom}_{\mathcal{D}}(F(x), G(x))$ for every object $x \in \mathcal{C}$ such that the diagram*

$$\begin{array}{ccc} F(x) & \xrightarrow{F(f)} & F(y) \\ {\scriptstyle \eta_x}\downarrow & & \downarrow{\scriptstyle \eta_y} \\ G(x) & \xrightarrow[G(f)]{} & G(y) \end{array}$$

commutes for every morphism $f \in \mathrm{Hom}_{\mathcal{C}}(x, y)$.

*We say a natural transformation $\eta : F \Rightarrow G$ is a **natural isomorphism** if the morphisms η_x are isomorphisms for any $x \in \mathcal{C}$.*

Given two categories $\mathcal{C}$ and $\mathcal{D}$, one can construct a category $Fun(\mathcal{C}, \mathcal{D})$ of functors between $\mathcal{C}$ and $\mathcal{D}$. It's objects are functors $\mathcal{C} \to \mathcal{D}$ and morphisms are given by natural transformations. Then we can view natural isomorphisms as isomorphisms in the functor category.

Notation 6.1.10. We write natural transformations as

$$\mathcal{C} \overset{F}{\underset{G}{\Downarrow \eta}} \mathcal{D}$$

6.1.3 Examples to keep in mind

Example 6.1.11. Groups form a category **Grp** with morphisms by homomorphisms of groups. We can restrict groups to be abelian, which forms a category **Ab**.

We consider some functors:

- The forgetful functor from **Ab** to **Grp** is fully faithful.
- The abelianization $G \mapsto G/[G, G]$ gives a functor $\mathbf{Grp} \to \mathbf{Ab}$. It is neither full nor faithful.

Example 6.1.12. If k is a field, k-vector spaces form a category $\mathbf{Vect}_k$ with morphisms given by linear maps.

Example 6.1.13. (Small) categories form a category **Cat** where morphisms are given by functors.

Example 6.1.14. A set X can be regarded as a category $\mathcal{C}$ with $ob\,\mathcal{C} = X$ where $Hom_{\mathcal{C}}(x, y) = \emptyset$ for $x \neq y$ and $End_{\mathcal{C}}(x) = \{id_x\}$. These categories are called a **discrete category**.

Given a set and viewing it as a discrete category, this gives a fully faithful functor **Set** $\to$ **Cat**.

6.1.4 Products

Definition 6.1.15. *Let A and B be objects in a category $\mathcal{C}$. An A, B-**pairing** is a triple (P, p_1, p_2) where P is an object and $p_1 : P \to A$ with $p_2 : P \to B$.*

6.2 Set-theoretic issues

Definition 6.2.1. *A **Grothendieck universe** U is a set satisfying the following properties:*

- *If $x \in y$ and $y \in U$, then $x \in U$*
- *If $x \in U$ and $y \in U$ and $\{x, y\} \in U$*
- *If $x \in U$ then the power set of x is in U*
- *If $x \in U$ and $f : x \to U$ is a map, then $\bigcup_{i \in x} f(i) \in U$*

7 Other

7.1 Curry-Howard Correspondence

7.2 LEM?

The **Law of Excluded Middle (LEM)** is the schema that

$$\forall P.P \vee \neg P$$

Under the Curry-Howard correspondence, this is a type that is inhavited precisely when every proposition P is *decidable*.

7.2.1 Computability

Under Curry-Howard, a proof of Σ `x:A. P x` is a program that returns a concrete `x:A` plus evidence of `P x`. Similarly, a proof of `P` $\uplus\neg$ `P` is a program that decides `P`. Unrestricted LEM says $\forall$ `P. P` $\uplus\neg$ `P`. This would be a single, uniform decision procedure for every proposition `P`. Such a procedure cannot exist in a purely computable setting (as it would decide the halting problem, etc.). Hence, mainstream constructive type theories (like MLTT/Agda/Coq's core) do not validate LEM by computation.

When LEM is postulated at the computational level (`Type` / `Set`), we obtain a constant with no reduction rule. Then, we can write closed programs whose result depend on evaluating that constant, so the evaluation gets stuck. Consider:

```
postulate
    EM : (P : Set) → P ⊎ (P → ⊥)

coin : ℕ
coin with EM (0 ≡ 1)
... | inj₁ _ = 0
... | inj₂ _ = 1
```

Then, `coin` is a closed $\mathbb{N}$ but does not reduce to a numeral, as it is stuck on `EM`. Hence, even if normalization of existing terms is preserved, addition of postulates makes new closed terms non-caninical. Then, program extraction (in a weak sense) fails.

All in all, these have to do with proof relevance.

7.2.2 Inderivability

LEM is *not derivable* in constructive type theory and is *not refutable* either. Hence this axiom is an independent design-level choice.