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 Type Theory

2.1 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
```

2.2 As an Inductive-Inductive Type

Metatheoretically, in the context of Type Theory, we have

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

2.2.1 Universes, El, and Inductive-Inductive Types

Definition 2.2.1. *A* ***Tarski-Style Universe*** *is a type* `U : Type` *whose elements are codes (names) for small types.*

Definition 2.2.2. *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.

Definition 2.2.3. *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.*

2.2.2 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

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

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,

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

2.3 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

3 Other Prerequisites

3.1 MLTT

3.2 Terms, Types, and Prop

3.3 First Order Logic to Type Theory

3.4 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]
```

3.5 Interpretation

3.5.1 Set Interpretation

Definition 3.5.1. *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 $[\![A]\!]$: `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 $[\![\Gamma]\!] : \texttt{Set}$, we expect $[\![A]\!] : [\![\Gamma]\!] \to \texttt{Set}$. 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

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

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

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

We can then type something like

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

Then we have

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

Noting that $[\![\diamond, \texttt{Nat}]\!] = \top \times \mathbb{N}$ and $[\![\texttt{Vec}_{\texttt{Bool}} \emptyset]\!]\gamma = \mathbb{V}\mathrm{ec}_{\mathbb{2}}([\![\emptyset]\!]\ \gamma)$, this simplifies to

$$[\![\texttt{trues @ } \emptyset]\!] : (\gamma : \top \times \mathbb{N}) \to \mathbb{V}\mathrm{ec}_{\mathbb{2}}([\![\emptyset]\!]\ \gamma)$$

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

$$[\![\texttt{trues @ } \emptyset]\!] : ((tt, n) : \top \times \mathbb{N}) \to \mathbb{V}\mathrm{ec}_{\mathbb{2}}\ n$$

3.5.2 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 $\Gamma^{\mathrm{D}} : \Gamma^{\mathrm{A}} \to \texttt{Set}$. A type $A : \texttt{Ty}\ \Gamma$ becomes a predicate that depends on a witness of Γ^{D}, which explictly is $A^{\mathrm{D}}\ \{\gamma : \Gamma^{\mathrm{A}}\} : \Gamma^{\mathrm{D}}\ \gamma \to A^{\mathrm{A}}\ \gamma \to \texttt{Set}$. Finally, a term $t : \texttt{Tm}\ \Gamma\ A$ 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
```

3.5.3 Motive / Method Interpretation

3.5.4 Modified Dependent Interpretation

3.5.5 Fiber?

Definition 3.5.2. *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 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. $\Sigma(A : Set)(A \to I) \simeq (I \to Set)$