SOME EXAMPLES OF HIGHER INDUCTIVE TYPE

Peixin YOU
peixin.you@polytechnique.edu

ABSTRACT

Homotopy type theory is a theory axiomatize equality and also links type theory with geometry. In HoTT, types are interpreted as spaces. In HoTT we could generate the idea of inductive type to higher inductive by allowing define constructor to construct equalities. Here, we define and study higher inductive types in order to present some well-known and useful types and implement it in cubical Agda. We define integers as a HIT and show that it forms an Abelian group when equipped with addition. We also introduce two different definitions of the notion of free group on a set and show that they are equivalent. Finally, study two HIT definitions of symmetric groups: the "natural" one (which corresponds to the standard presentation by transpositions) and one which correspond to some "normal form" for symmetries, called Lehmer codes. We construct a map from the later to the former, and conjecture that it is part of an equivalence.

Keywords Homotopy Type Theory · Logic · Higher Inductive Type

1 Introduction

Type Theory is a branch of mathematical symbolic logic, which derives its name from the fact that it formalizes not only mathematical terms - such as a veriable x or a function f - and operations on them, but also formalizes the idea that each such term is of some definite type, for instance that the type $\mathbb{N}$ of natural number $x : \mathbb{N}$ is different from the type $\mathbb{N} \to \mathbb{N}$ of a function $f : \mathbb{N} \to \mathbb{N}$ between natural numbers. In type theory, a type can also be considered as a proposition. For example, we could consider a type A which expresses the proposition that 2 is even. Then $a : A$ can be regarded as the fact that a is a proof of A. Similarly, if we consider the type corresponding to falsity, we do not expect that it can be proved, so that it should be empty.

Sometimes, the type will depend on some term. For instance, the type which expresses the fact that a natural number n is even, depends on n. This follows our intuition, since the proof of n is even depends on n. We will denote a dependent type as $B : A \to \mathcal{U}$. For example, in the "is even" case we have $\texttt{isEven} : \mathbb{N} \to \mathcal{U}$. We could also write a dependent function like this: for a given type A and a dependent type $B : A \to \mathcal{U}$ we could also denote this as $\prod_{x:A} B(x) : \mathcal{U}$. When type $B(x)$ is treated as proposition, we could also regard this Π as a universal quantification ($\forall$). For example, $\prod_{n:\mathbb{N}} \texttt{isEven}(2n)$ means for all $n : \mathbb{N}, 2n$ is even. When B is not dependent on A, $\Pi_{x:A} B$ is exactly $A \to B$. As dependent function type is a generalization of function type, it is also very useful to generate the product types, which is exactly the Cartesian product $A \times B$, to make the second component depends on the first component. This is called a dependent pair type or Σ-type, because in set theory it corresponds to an indexed sum (in the sense of coproduct or disjoint union) over a given type. When you treat $\Sigma_{a:A} B(a) : \mathcal{U}$ as proposition, you could regard Σ as there exists. For example, $\Sigma_{x:\mathbb{N}} \texttt{isEven}(x)$ means there exists a natural number x such that x is even. As we have seen, all the proposition will also be treated as a type in type theory, so is the identity type $x = y$. The identity type is very important in Martin-Löf Type Theory (MLTT) which is the incarnation of equality. That is if we have $x, y : A$, then $\texttt{Id}_A(x, y)$ is the type of proofs that x and y equal. In this paper we will use $x = y$ or $x =_A y$ instead of $\texttt{Id}_A(x, y)$.

Homotopy type theory (HoTT) keeps and extends this idea of equality with Voevodsky's univalence principle. Not only we consider if two items equal or not, but also how or in which way they are equal i.e. the path between x and y. This extension highly enrich the structure in identity type. More precisely, HoTT interprets type theory from homotopical perspective. In HoTT, we regard the types as "spaces" or higher groupoids, and the logical constructions as homotopy-invariant constructions on these spaces. For instance, in HoTT we think of $a : A$ as: "a is a point of space A", every function $f : A \to B$ in HoTT is regarded as a continuous map from space A to space B. Similarly, we could

regard the type family $B : A \to \mathcal{U}$ as a fibration with base space A with $B(a)$ being the fiber over a and with $\Sigma_{a:A}B(a)$ being the total space of the fibration, with first projection $\Sigma_{a:A}B(a) \to A$. The key idea of this interpretation is that the logical notion of identity $a = b$ of two objects $a, b : A$ can be understood as the existence of a path $p : a \sim b$. As in fibration (type family in our case) $B : A \to \mathcal{U}$, we have the transport operation lifts path in the base type A to functions between the fibers, that is, $\texttt{transport}_B : \Pi_{x,y:A} x =_A y \to B(x) \to B(y)$. For $u : B(x)$ and $v : B(y)$, we write $u =_P v$ for the lifted path over p given by transport function. The functorial cation on sections $f : \Pi_{x:A}B(x)$ of the fibration is given by $\texttt{subst}_f : \Pi_{x,y:A}(p : x = y) \to f(x) =_p f(y)$

In HoTT, now we regard an equality as a path, and an equality between equalities as a homotopy between paths. But in this case, we don't have the unique identity principle (UIP) , i.e. $p, q : a = b \to p = q$, anymore. For propositions, we can think of a proposition as either being

- a point meaning that it is true, or
- empty, meaning that it is false

In particular, when it is true, we only allow for one point. In both cases (true or false), we can remark that a proposition is such that any two points x and y are related by a path $x = y$: this property holds by definition when the proposition is true and is vacuously true when the proposition is empty. This remind us could define the predicate `isProp` which expresses the fact that a type is a *proposition* by

```
isProp : ∀ {i} → Type i → Type i
isProp A = (x y : A) → x ≡ y
```

After considering propositions, the next types of interest are *sets*. In a set, two points x and y are either in the same connected component, in which case they are equal in a unique way, or they are in distinct components, in which case they are not equal. The corresponding predicate is defined as

```
isSet : ∀ {i} → Type i → Type i
isSet A = (x y : A) (p q : x ≡ y) → p ≡ q
```

In this paper, we choose to work in the type-theoretic language of the HoTT book [1]. We also implement all of our proof by using cubical Agda which support cubical type theory, so that we could implement higher inductive type and the theory in HoTT. Before going into higher inductive types, we will first give a brief introduction about inductive types. The intuition behind those is that they list all the possible constructors to build the elements in this type. For instance, for the natural numbers $\mathbb{N}$, there are two ways to construct the elements of $\mathbb{N}$, first zero is an element of $\mathbb{N}$, second, if n is natural number, the successor of n is also a natural. This suggest us to define $\mathbb{N}$ as follows:

```
data Nat : Set where
  zero : Nat
  suc  : (n : Nat) → Nat
```

As we saw before, all the constructors defining an inductive type A should have A as target. In *higher* inductive types, we also allow constructors to build equalities between the elements of the type. For example the higher inductive type describing a circle is

```
data S₁ : Set where
  base : S₁
  loop : base ≡ base
```

Based on those higher inductive types, we develop here tree main constributions:

- With HITs, we are able to define the quotient of a type by a relation, by using the idea of [2]. We add another higher constructor that eliminates all of the higher-dimensional structure from the quotient type, in other words, we set truncate the type. Once quotients introduced, we can define integers as $\mathbb{N} \times \mathbb{N}$ quotient under the following relation $(a_1, b_1) \sim (a_2, b_2)$ when $a_1 + b_2 = b_1 + a_2$. It is interesting to give the arithmetic on integer in this setting.
- Using this equality type for equations, it is also a challenging problem to define (higher) algebraic structures. In this paper, we are interested in formalizing free groups in two different ways and showing the equivalence between two definitions. The most usual way to define by using "symbolic calculation". For example, the free group on two symbols x and y contains words such as xy, yx^{-1}, x^3y^2, xyx with some axiom such as $aa^{-1} = 1$. Intuitively, we could write xy, xyx^{-1} aslist $[x, y], [x, y, x^{-1}]$ and we have a equivalence relation, $[x, x^{-1}] \sim []$. Then the free group should be as same as the list quotient by that relation.

- Then we will consider `Fin(n)` which is the finite set with n elements. In order to show that `Fin(n)` $\cong$ `Fin(n)` is equivalent to $\mathtt{Sym}_n$, we will first show that `Fin(n)` $\cong$ `Fin(n)` is equivalent to the normal form and also $\mathtt{Sym}_n$ is equivalent to the normal form. This said that evry permutation can be written as the composition of transitive. In the end of this paper, we give a presentation for `Fin(n)` $\cong$ `Fin(n)` by using the Lehmer Code and gave a function from Lehmer Code to symmetry group and we will show the inverse in the future by using the idea mentioned in [3] section 2.1.

In section 2 we define the add and minus on integer and show that the integer is an Abelian group with addition. In section 3 we give two different definition of free group on a set A. We also show that these two definitions are equivalent and the free group of a set is still a set. In section 4 We generalize the Natural number to groupoid and give a representation of it. Actually, we find that the groupoid structure of natural number is symmetry group, then the main goal is to represent the symmetry group by normal forms.

2 Arithmetic on integers

We start by giving the definition of integers. Let's first see how we define quotient in HITs. Let A be a set and $R : A \times A \to$ Prop a family of mere propositions then the higher inductive type A/R generated by

- A function $q : A \to A/R$, we will denote by `[_]` $: (a : A) \to A/R$ in Agda
- For each $a, b : A$ such that $R(a, b)$ an equality $q(a) = q(b)$, we will denote by `eq/` $: (a, b : A) \to R(a, b) \to [a] \equiv [b]$ in Agda
- The truncation constructor : for all $x, y : A/R$ and $p, q : x = y$ we have $p = q$ we will denote by `squash/` $: (x, y : A/R) \to (p, q : x \equiv y) \to p \equiv q$ in Agda

For instance, we could define the integers $\mathbb{Z}$ as a set-quotient $\mathbb{N} \times \mathbb{N}/\sim$ where $\sim$ is defined by $(a, b) \sim (c, d) :\equiv (a + d = b + c)$.

Before defining the addition on $\mathbb{Z}$ we want to show a lemma before:

Lemma 2.1. *For any set B, precomposing with $q : A \to A/R$ yields an equivalence*

$$(A/R \to B) \simeq \left(\sum_{(f:A\to B)} \prod_{(a,b:A)} R(a, b) \to (f(a) = f(b)) \right)$$

Proof. The quasi-inverse of $- \circ q$ is just the recursion principle for A/R which is very useful in the following chapter. That is, given $f : A \to B$ such that

$$\prod_{a,b:A} R(a, b) \to (f(a) = f(b))$$

We define $\hat{f} : A/R \to B$ by $\hat{f}(q(a)) :\equiv f(a)$, from this definition we can directly get that $(f \mapsto \hat{f})$ is a right inverse to $(- \circ q)$. Now we want to find the left inverse, i.e. we want to show that for any $g : A/R \to B$ and $x : A/R$ we have $g(x) = \widehat{g \circ q}(x)$. Since we know that q is subjective, so we have there exists a such that $q(a) = x$, so $g(x) = g(q(a)) = \widehat{g \circ q}(q(a)) = \widehat{g \circ q}(x)$ □

By applying recursion principle, we could get the following lemma:

```
rec2 : ∀ {ℓ} {A : Type ℓ} {R : A → A → Type ℓ}
       {B : Type ℓ} (Bset : isSet B)
       (f : A → A → B) (feql : (a b c : A) (r : R a b) → f a c ≡ f b c)
                       (feqr : (a b c : A) (r : R b c) → f a b ≡ f a c)
    → A / R → A / R → B
```

Now we try to define the addition on integers $\_ + \_ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, for instance, if $u, v \in \mathbb{Z}$ are represented by (a, b) and (c, d), respectively, then $u + v$ is represented by $(a + c, b + d)$, by using rec2.

So that $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N} \to \mathbb{Z}$ can be defined as $f(a, b) = q(a +\mathbb{N}' b)$. Before using rec2, there still to condition needs to be checked:

```
+-right-congruence : ∀ x y z → rel y z → rel (x +ℕ' y) (x +ℕ' z)
+-left-congruence : ∀ x y z → rel x y → rel (x +ℕ' z) (y +ℕ' z)
```

By using these two lemma, we could easily get feql and feqr, so the definition of _ + _ is the following:

```
_+_ : ℤ → ℤ → ℤ
_+_ = rec2 ℤ-isSet (λ x y → [ x +ℕ' y ]) feql feqr
  where
    feql : (a b c : ℕ × ℕ) (r : rel a b) → [ a +ℕ' c ] ≡ [ b +ℕ' c ]
    feqr : (a b c : ℕ × ℕ) (r : rel b c) → [ a +ℕ' b ] ≡ [ a +ℕ' c ]
```

It's not difficult to define _ − _ by defining $-\mathbb{Z}'\_$ as the inverse of _ + _

```
-ℤ'_  : ℤ → ℤ
_-_ : ℤ → ℤ → ℤ
```

For proving $(\mathbb{Z}, +, q(0,0))$ is an Abelian group, we now introduce the elimination into proposition.

Theorem 2.2 (ElimProp)**.** *Given a set-quotient A/R and a dependent type $B : A/R \to \mathcal{U}$ satisfy $\forall x : A/R, B(x)$ is proposition. Given a function $g : (a : A) \to B([a])$ then there exists a unique function $f : (x : A/R) \to B(x)$ satisfy for all $a : A, g(a) = f([a])$*

Having elimination from proposition, we could begin to prove that $(\mathbb{Z}, +, q(0,0))$ is an Abelian group.

Theorem 2.3. *$q(0,0)$ is the unit element of $(\mathbb{Z}, +, q(0,0))$*

Proof. In order to use theorem2.2, we need to check if for all $a : \mathbb{N} \times \mathbb{N}, [a] + [(0,0)] = [a]$ is Prop. This is because since $\mathbb{Z}$ is set, so for all $p, q : [a] + [(0,0)] = [a]$ we have $p = q$. It is the same for for all $a : \mathbb{N} \times \mathbb{N}, [(0,0)] + [a] = [a]$ is Prop. Then let's apply theorem 2.2. So now we only need to show that when $x = [a]$, $[a] + [(0,0)] = [a]$ and $[(0,0)] + [a] = [a]$. Both of them could directly get from zero is the unit element of natural number. □

Using the similar idea we could easily check the following theorems.

Theorem 2.4. *$-\mathbb{Z}'a$ is the inverse of a*

Theorem 2.5. *For all $a, b, c : \mathbb{Z}, (a + b) + c = a + (b + c)$*

Then we have

Theorem 2.6. *$(\mathbb{Z}, +, q(0,0))$ is an Abelian group*

3 Free Group

Formally speaking, let A be a set we could define a higher inductive type $F(a)$ with the following generators.

Definition 3.1 (FG)**.** Let A be a set, we define the HIT $F(A)$ with the following point and path constructors:

$$\begin{aligned} \eta &: A \to F(A) \\ m &: F(A) \times F(A) \to F(A) \\ e &: F(A) \\ i &: F(A) \to F(A) \end{aligned} \qquad \begin{aligned} \texttt{assoc} &: (x,y,z : F(A)) \to m(x, m(y,z)) = m(m(x,y), z) \\ \texttt{unitl} &: (x : F(A)) \to m(x,e) = x \\ \texttt{unitr} &: (x : F(A)) \to m(e,x) = x \\ \texttt{invl} &: (x : F(A)) \to m(x, i(x)) = e \\ \texttt{invr} &: (x : F(A)) \to m(i(x), x) = e \\ \texttt{trunc} &: (x,y : F(A)) \to (p,q : x = y) \to x = y \end{aligned}$$

The first constructor says that A maps to $F(A)$. The next three give $F(A)$ the operations of a group: multiplication, an identity element, and inversion. The three constructors after that assert the axioms of a group: associativity, unitality, and inverses. Finally, the last constructor asserts that $F(A)$ is a set. Therefore $F(A)$ is a group, and also is a free group on A. For completeness, we state the induction principle for FG in appendix **??**. Using the induction principle and computation rules, we can prove the following theorem When eliminating to propositions, the induction principal can be simplified.

Lemma 3.1 (Propositional induction principle)**.** *Let $P : F(A) \to \mathcal{U}$ be a type family with the following data*

- $\forall x \to \texttt{isProp}(P(x))$
- $\forall x \to P(\eta(x))$ *and* $P(e^*)$

- $\forall x, y \to P(m^*(x, y))$, $\forall x \to P(x) \to P(i^*(x))$

There is a unique function $f : (x : F(A)) \to P(x)$ satisfying the appropriate computation rules.

Theorem 3.2. *$F(A)$ is the free group on A. In other words, for any group G, composition with $\eta : A \to F(A)$ determines an equivalence*

$$\hom_{\text{Group}}(F(A), G) \simeq (A \to G)$$

where $\hom_{\text{Group}}(-,-)$ denotes the set of group homomorphisms between two groups.

Of course, it is sometimes also useful to have a concrete description of free algebraic structures. In the case of free groups, we can provide one, using quotients. Consider List $(\mathbb{B} \times A)$, where in $\mathbb{B} \times A$ we write $(1, a)$ as a, and $(1, a)$ as a^{-1}. The elements of $\texttt{List}(\mathbb{B} \times A)$ are words for the free group on A.

Theorem 3.3. *Let A be a set, and let $FL(A)$ be the set-quotient of $\texttt{List}(\mathbb{B} \times A)$ by the following relations.*

$$(\cdots, a_1, a_2, a_2^{-1}, a_3, \cdots) = (\cdots, a_1, a_3, \cdots)$$
$$(\cdots, a_1, a_2^{-1}, a_2, a_3, \cdots) = (\cdots, a_1, a_3, \cdots)$$

Then $FL(A)$ is also the free group on the set A

We show that $FL(A)$ is a group with the empty list `nil` as unit and the binary multiplication given by the concatenation operation _ + _ defined below

Definition 3.2. The concatenation operation $\_ + \_ : FL(A) \to FL(A) \to FL(A)$ is defined by recursion on the first argument.

$$\texttt{nil} + ys :\equiv ys$$
$$(x :: xs) + ys :\equiv x :: (xs + ys)$$

by using the `rec2` mentioned above, we could check this is well defined.

We could also define the inverse in a similar way:

Definition 3.3. The concatenation operation $-\_ : FL(A) \to FL(A)$ is defined by recursion on the argument.

$$\texttt{nil} :\equiv \texttt{nil}$$
$$x :: xs :\equiv (-xs) + ([\neg x])$$

where $\neg x$ is defined as below:

$$\neg x : (\mathbb{B} \times A) \to (\mathbb{B} \times A)$$
$$\neg(0, a) :\equiv (1, a), \neg(1, a) :\equiv (0, a)$$

by using the rec2 mentioned above, we could check this is well defined.

To show that $FL(A)$ is a group, we need to prove a few identities about elements in $FL(A)$. Since $FL(A)$ is quotient type, so we could use the propositional induction principle from 2.2.

Lemma 3.4. *The concatenation operation $\_ + \_$ is associative and $\texttt{nil}$ is a left and right unit.*

- *For all $xs, ys, zs : FL(A)$, have $(xs + ys) + zs = xs + (ys + zs)$*
- *For all $xs : FL(A)$, we have $\texttt{nil} + xs = xs$*
- *For all $xs : FL(A)$, we have $xs + \texttt{nil} = xs$*

We further need to establish that the $-\_$ give exactly the inverse of the elements in $FL(A)$. Similar, we use the 2.2, so we only need to do the calculation on list.

Lemma 3.5. *The concatenation operation $-\_$ give the left and right inverse of $\_ + \_$:*

- *For all $xs : FL(A)$, we have $xs + (-xs) = \texttt{nil}$*
- *For all $xs : FL(A)$, we have $(-xs) + xs = \texttt{nil}$*

Theorem 3.6. *Given a set A, $FL(A)$ is a group*

Theorem 3.7. *Given a set A, F(A) ≅ FL(A)*

4 Groupoid structure of natural numbers

For a given natural number n, we can regard n as a list of successors with length n. In natural number,s for all $p, q : m = n, p = q$, now we want to study the groupoid structure of natural numbers. Formally speaking, we construct the groupoid structure of natural numbers (`Bij`) by the following constructors:

Definition 4.1 (Bij)**.** We define the HIT `Bij` with the following point and path constructors: first are two point constructors, `zero:Bij` and `suc:Bij` $\to$ `Bij`.

The path constructor `swap` : $n : \mathbb{N} \to$ `suc (suc n) = suc (suc n)` three restrictions of swap:

- Swapping the same two elements twice should be the same
- When swapping two different pair of elements, the order of swapping should not matter.
- There are two equivalent ways of swapping the first and last elements in a sequence of three elements

Intuitively, the element in `Bij` is the bijection between finite set with n elements. Now we want to show the equivalent between $\mathtt{Sym}_n$ and `Fin n` $\cong$ `Fin n` so we can give the presentation of `Bij`.

First let's give a formal definition of `Fin`.

Definition 4.2 (Fin)**.** The type family Fin : $\mathbb{N} \to \mathcal{U}$ is the type of finite sets indexed by their cardinality. It is defined as follow:

$$\mathrm{Fin}_n :\equiv \sum_{k:\mathbb{N}} k < n$$

The main idea of proving $\mathrm{Sym}_n \cong$ `Aut(Fin(n))` is first showing that both `Aut(Fin(n))` and $\mathtt{Sym}_n$ are equivalent to `LehmerCode(n)`. Lehmer Code are known in mathematics and in particular in combinatorics, it is a particular way to encode each possible permutation of a sequence of n numbers. We could define Lehmer Code inductive on n:

Definition 4.3 (Lehmer Code : $\mathbb{N} \to \mathcal{U}$)**.**

$$\begin{aligned} \texttt{Fin (1)} &: \texttt{LehmerCode(0)} \\ \texttt{Fin (suc n)} \times \texttt{LehmerCode(n)} &: \texttt{LehmerCode(suc n)} \end{aligned}$$

We first try to show that `Fin (n)` $\cong$ `LehmerCode(n)`. In order to prove `Fin (n)` $\cong$ `LehmerCode(n)` we still need following definition

Definition 4.4 (FinExcept)**.** The type family `FinE` : `Fin(n)` $\to \mathcal{U}$ is the sub type of `Fin(n)` which picks out all elements in `Fin(n)` except the one in the argument:

$$\mathtt{FinE}(i) :\equiv \sum_{j:\mathtt{Fin}\ \_} \neg(i \equiv j)$$

Since FinE is a sub type of `Fin(n)`, so we could check that FinE is a set. We also have

Lemma 4.1. *For all* i : *`Fin(suc n)` $\to$ `FinE(i)` $\cong$ `Fin(n)`*

Theorem 4.2. *For all* n, *`Aut`*$(\mathtt{Fin(n)}) \cong$ *`LehmerCode n`*

Proof. Let's prove by induction. When $n = 0$, we have both `LehmerCode 0` and `Aut(Fin(0))` are contractible, so they are equivalent.

When $n =$ `suc m`, we have

$$\begin{aligned} \texttt{Aut(Fin(suc m))} &\cong \sum_{k:\texttt{Fin (suc m)}} (\texttt{FinE (0, 0<suc m)} \cong \texttt{FinE k}) \\ &\cong \texttt{Fin (suc m)} \times \texttt{Aut(Fin m)} \\ &\cong \texttt{Fin (suc m)} \times \texttt{LehmerCode m} \\ &\cong \texttt{LehmerCode n} \end{aligned}$$

□

Before we show the equivalence between $\mathtt{Sym}_n$ and `LehmerCode(n)`, we need to define a function $n \downarrow k$ inductive on the second argument:

Definition 4.5. The function $\_ \downarrow \_ : \mathbb{N} \to \mathbb{N} \to \texttt{List } \mathbb{N}$ defined as follow:

$$n \downarrow 0 = \texttt{nil}$$
$$n \downarrow \texttt{(suc k)} = (k + n) :: (n \downarrow k)$$

When we try to decode a Lehmer Code to get the correspondence permutation, we notice that, the number r at position k in lehmer code describes how many inversions the element k has. Thus we need to perform r many adjacent transpositions to get to the desired position, which is given by $(\texttt{suc n} - r) \downarrow r$

So the following fuction $h : n : \mathbb{N} \to \texttt{LehmerCode(n)} \to \texttt{List (Fin suc n)}$ will give an equivalence between `LehmerCode` and $\texttt{Sym}_n$.

Definition 4.6. The function $h : \mathbb{N} \to \texttt{LehmerCode(n)} \to \texttt{List(Fin (suc n))}$ defined as follow:

$$h(0)(\texttt{Fin(1)}) = \texttt{nil}$$
$$h(\texttt{suc } n)((r, l)) = h(n)(l) + ((\texttt{suc } n - r) \downarrow r)$$

So now we have $\texttt{Sym}_n \cong \texttt{LehmerCode(n)} \cong \texttt{Aut(Fin(n))}$.

5 Conclusion and future work

HoTT is a very new field, there are lots of stuff could be done. It also enriches the structure that type theory can describe, for example, quotient, symmetry group and some other HITs structure. Sometimes the normalization of Agda is also not good enough, even if something is really easy to implement by hand we will still meet some problem when implement it in Agda. In the future, I hope I can improve the normalization algorithm and also auto prover for cubical Agda.

Because of the way its implement HoTT in Agda, if we try to define a binary operation on a quotient type by pattern matching we have to fill a hyper cube which is really a huge work. We notice that we can first fix the first argument then we only need to define the operation on the second argument. In this case, we only need to fill a cube, which will be much easier. In the future, I wish to explore more higher algebra structure for example a higher presentation of monotone map.

Acknowledgements

This article was done under the supervision of Prof. Samuel Mimram.

References

[1] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[2] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: A dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[3] Yves Lafont. Towards an algebraic theory of boolean circuits. *Journal of Pure and Applied Algebra*, 184(2):257–310, 2003.

Appendix

```
-ℤ'_  : ℤ → ℤ
-ℤ' [ a ] = [ snd a , fst a ]
-ℤ' eq/ a b r i = eq/ (snd a , fst a) (snd b , fst b) (tmp a b r) i
     where
       tmp : ∀ a b → fst a +ℕ snd b ≡ fst b +ℕ snd a → snd a +ℕ fst b ≡ snd b +ℕ fst a
       tmp a b r =
           snd a +ℕ fst b ≡⟨ ℕ.+-comm (snd a) (fst b) ⟩
           fst b +ℕ snd a ≡⟨ sym r ⟩
```

```
           fst a +ℕ snd b ≡⟨ ℕ.+-comm (fst a) (snd b) ⟩
           snd b +ℕ fst a
-ℤ' squash/ a a₁ p q i i₁ = squash/ (-ℤ' a) (-ℤ' a₁)
                                   (cong (λ x → -ℤ' x) p)
                                   (cong (λ x → -ℤ' x) q) i i₁

_-_ : ℤ → ℤ → ℤ
a - b = a + (-ℤ' b)

record Group A : Type₀ where
  field
    set : isSet A

    _∘_  : A → A → A
    -_   : A → A
    ε    : A

    unit-l : ∀ x     → ε ∘ x ≡ x
    unit-r : ∀ x     → x ∘ ε ≡ x
    inv-r  : ∀ x     → x ∘ (- x) ≡ ε
    inv-l  : ∀ x     → (- x) ∘ x ≡ ε
    assoc  : ∀ x y z → (x ∘ y) ∘ z ≡ x ∘ (y ∘ z)

open Group {{...}}

data HITGro A : Type where
  ⟨_⟩   : A → HITGro A
  :ε:   : HITGro A
  _:∘:_  : HITGro A → HITGro A → HITGro A
  :-:_  : HITGro A → HITGro A

  :unit-l: : ∀ x      → :ε: :∘: x ≡ x
  :unit-r: : ∀ x      → x :∘: :ε: ≡ x
  :inv-r:  : ∀ x      → x :∘: (:-: x) ≡ :ε:
  :inv-l:  : ∀ x      → (:-: x) :∘: x ≡ :ε:
  :assoc:  : ∀ x y z  → (x :∘: y) :∘: z ≡ x :∘: (y :∘: z)

  trunc    : isSet (HITGro A)

freeGroup : ∀ A → Group (HITGro A)
freeGroup A = record
  { set = trunc
  ; _∘_ = _:∘:_
  ; -_ = :-:_
  ; ε = :ε:
  ; unit-l = :unit-l:
  ; unit-r = :unit-r:
  ; inv-l = :inv-l:
  ; inv-r = :inv-r:
  ; assoc = :assoc:
  }

elimFGProp : {A : Type₀} → (P : HITGro A → Type) → (∀ x → isProp (P x))
             → (∀ x → P ⟨ x ⟩) → P :ε: → (∀ x y → P x → P y → P (x :∘: y))
             → (∀ x → P x → P (:-: x)) → ∀ x → P x
elimFGProp P PIsProp P⟨_⟩ Pe P∘ Pinv = go
  where
    go : ∀ x → P x
    go ⟨ x ⟩ = P⟨_⟩ x
```

```
    go :ε: = Pe
    go (x :∘: y) = P∘ x y (go x) (go y)
    go (:-: x) = Pinv x (go x)
    go (:unit-l: x i) = isProp→PathP (λ j → PIsProp (:unit-l: x j))
                                     (P∘ _ _ Pe (go x)) (go x) i
    go (:unit-r: x i) = isProp→PathP (λ j → PIsProp (:unit-r: x j))
                                     (P∘ _ _ (go x) Pe) (go x) i
    go (:inv-r: x i) = isProp→PathP (λ j → PIsProp (:inv-r: x j))
                                     (P∘ _ _ (go x) (Pinv _ (go x))) (Pe) i
    go (:inv-l: x i) = isProp→PathP (λ j → PIsProp (:inv-l: x j))
                                     (P∘ _ _ (Pinv _ (go x)) (go x)) (Pe) i
    go (:assoc: x y z i) = isProp→PathP (λ j → PIsProp (:assoc: x y z j))
                                     (P∘ _ _ (P∘ _ _ (go x) (go y)) (go z))
                                     (P∘ _ _ (go x) (P∘ _ _ (go y) (go z))) i
    go (trunc x y p q i j) = isOfHLevel→isOfHLevelDep 2 (λ a → isProp→isSet (PIsProp a))
                                     (go x) (go y) (cong go p)
                                     (cong go q) (trunc x y p q) i j

X : Type
X = Bool × A

FA : Type
FA = List X

rel-ex : FA → FA → Type
rel-ex s t = Σ[ u ∈ FA ] (Σ[ v ∈ FA ]
            (Σ[ x ∈ FA ] (((s ≡ (u ++ x ++ finv x ++ v)) × (t ≡ u ++ v)) ⊎
                          ((t ≡ (u ++ x ++ finv x ++ v)) × (s ≡ u ++ v)))))

FG : Type
FG = FA / rel-ex

data Bij : Type₁
ladd : ℕ → Bij → Bij
suc' : Bij → Bij

ladd zero n = n
ladd (suc k) n = suc' (ladd k n)

data Bij where
  zero : Bij
  suc  : Bij → Bij
  swap : (n : Bij) → suc' (suc' n) ≡ suc' (suc' n)
  inv  : (n : Bij) → swap n • swap n ≡ refl
  xchg : {k : ℕ} {n : Bij} →
         cong (ladd (suc (suc k))) (swap n) • swap (ladd k (suc' (suc' n))) ≡
         swap (ladd k (suc' (suc' n))) • cong (ladd (suc (suc k))) (swap n)
  yb   : {n : Bij} → swap (suc' n) • cong suc' (swap n) • swap (suc' n) ≡
                     cong suc' (swap n) • swap (suc' n) • cong suc' (swap n)
  gpd  : isGroupoid Bij

suc' = suc

_↓_ : (n : ℕ) → (k : ℕ) → List ℕ
n ↓ zero = []
n ↓ suc k = (k + n) :: (n ↓ k)

data _∼'_ {m : ℕ} : List (Fin (suc m)) → List (Fin (suc m)) → Type₀ where
    swap : {n : Fin (suc m)} {k : Fin (suc m)} → (suc (k .fst) < (n .fst))
```

```
                                                → (n :: k :: []) ∼' (k :: n :: [])
    braid : {n : Fin m} → (fsuc n :: fweak n :: fsuc n :: []) ∼' (fweak n :: fsuc n :: fweak n ::[])
    cancel : {n : Fin (suc m)} → (n :: n :: []) ∼' []

data _∼_ : {m : ℕ} → List (Fin m) → List (Fin m) → Type₀ where
    id : {m : ℕ} {l : List (Fin m)} → l ∼ l
    rel : {m : ℕ} {l1 l2 : List (Fin (suc m))} → l1 ∼' l2 →  l1 ∼ l2
    Symmetry : {m : ℕ} {l1 l2 : List (Fin m)} → (l1 ∼ l2) → l2 ∼ l1
    trans : {m : ℕ} {l1 l2 l3 : List (Fin m)} → (l1 ∼ l2) → (l2 ∼ l3) → l1 ∼ l3
    ++-hom : {m : ℕ} {l l' r r' : List (Fin m)} → (l ∼ l') → (r ∼ r') → (l ++ r) ∼ (l' ++ r')

data _~_ : List ℕ → List ℕ → Type₀ where
  cancel~ : {n : ℕ} → (l r m mf : List ℕ) → (defm : m ≡ l ++ n :: n :: r) →
                      (defmf : mf ≡ l ++ r) → (m ~ mf)
  swap~ : {n : ℕ} → {k : ℕ} → (suc k < n) →  (l r m mf : List ℕ) →
                              (defm : m ≡ l ++ n :: k :: r) →
                              (defmf : mf ≡ l ++ k :: n :: r) → (m ~ mf)
  long~ : {n : ℕ} → (k : ℕ) → (l r m mf : List ℕ) →
                              (defm : m ≡ l ++ (n ↓ (2 + k)) ++ (1 + k + n) :: r) →
                              (defmf : mf ≡ l ++ (k + n) :: (n ↓ (2 + k)) ++ r) → (m ~ mf)

data _~*_ : List ℕ → List ℕ → Type₀ where
  id : {m : List ℕ} → m ~* m
  trans~ : {m1 m2 m3 : List ℕ} → (m1 ~ m2) → (m2 ~* m3) → m1 ~* m3

Sym : (n : ℕ) → Type₀
Sym n = (List (Fin n)) / (_∼_ {n})

data LehmerCode : (n : ℕ) → Type₀ where
  [] : LehmerCode zero
  _::_ : ∀ {n} → Fin (suc n) → LehmerCode n → LehmerCode (suc n)
```