First Year PhD Annual Report

Li Nuo

September 21, 2011

Contents

Abstract

In set theory, given a set equipped with an equivalence relation, one can form its quotient set, that is the set of equivalence classes. Reinterpreting this notion in type theory, the implementation of quotient set is called quotient type. However in the quotient type is still unavailable in type theories like intensional Type Theory. Quotients are very common in mathematics and computer science. The introduction of quotients could be very helpful. Some quotient types are less effective to define than being based on their base types and equivalence relations, such as the set of integers, even some others are impossible to define without quotients in current implementations of intensional Type Theory such as the set of real numbers. Quotient types are often more difficult to reason than their base types. Therefore it is more convenient to manipulate the base types and lift the operations and propositions using the properties of quotients. Therefore we conduct a research project on the implementation of quotients in intensional Type Theory. The work of this project will be divided into several phases. This report aims at introducing the basic notions in the project like type theory and quotient types, discuss some work related to this topic and conclude some result of the first phase. The results done by Altenkirch, Anberrée and me in [3] will be explained with a few instances of quotients.

1 Introduction

In mathematics, a quotient represents the result of division. The notion of quotient is extended to other more abstract branches of mathematics. For example, we have quotient set, quotient group, quotient space, quotient category etc. They are all defined similarly. Some group of objects are divided by some equivalence relation and the group of the equivalent classes is the quotient sets or other algebraic structure.

Actually, we can find quotients in our daily life. When you take a picture by a digital camera, the real scene is divided into each pixel on the picture, in other words, the real scene within a certain area is equated. The digital picture which is the set of the pixel is just the quotient of the real scene. Since things are equated with respect to certain rules, quotient is usually related to information hiding or information losing. Quotients also exist in computer science. Users are usually concerned the extensional use of softwares rather than the intensional implementation of them, different implementations of softwares doing the same tasks are treated as the same to them even though some of them are programmed in different languages. Another example is the application of *interface* in JAVA. All different classes implementing the same interface A are extensionally equal and are treated as the same when we call some object of A.

In this report, I will mainly discuss the quotients in type theory which are usually named as quotient types. Although set theory and type theory have different fundamentals, they have many similar notions like product and disjoint union. In this case, quotient type is also an interpretation of quotient set in set

theory which may be more familiar to some of you. Because of this reason, we start from quotient set in set theory and then move on to type theory.

1.1 Quotient Set

The division of sets is different from division of numbers. We divide a given set into small groups according to a given equivalence relation and the quotient is the set of these groups.

Formally, given a set A and an equivalence relation $\sim$ on A, the equivalence class for each $a \in A$ is,

$$[a] = \{b \in A \mid b \sim a\}$$

The quotient set denoted as $\mathrm{A} / \sim$ is the set of equivalence classes of $\sim$,

$$\mathrm{A} / \sim = \{[a] \in \wp(A) \mid a \in A\}$$

There are many mathematical notions which can be constructed as quotient sets from some other sets. Some are more natural to come up with, such as integers modulo some number n is the quotient set constructed by quotienting the set of all integers $\mathbb{Z}$ with the congruence relation which equates two integers sharing the same remainders when divided by n. The set of rational numbers $\mathbb{Q}$ is defined as the set of numbers which can be expressed as fractions, but different fractions like $\frac{1}{2}$ and $\frac{2}{4}$ can be reduced to the same rational number. In another word, $\mathbb{Q}$ can be constructed by quotienting the set of pairs of integers, while the second is non-zero integer, with the equivalence relation which equates fractions sharing the same irreducible forms. A less common example is the set of integers $\mathbb{Z}$, which can also been obtained from quotienting the set of pairs of natural numbers $\mathbb{N} \times \mathbb{N}$ which represent integers as the result of subtraction between two natural numbers within each pair. Furthermore real numbers can be represented by Cauchy sequences of rational numbers, hence the set of real numbers $\mathbb{R}$ is the quotient set of the set of Cauchy sequences of rational numbers with the equivalence relation that the distance between two sequences converges to zero. There are more examples of quotient sets, but the main topic of this report is quotients in *type theory*.

1.2 Type Theory

The theory of types was first introduced by Russell [21] as an alternative to naive set theory. After that, mathematicians or computer scientists have developed a number of variants of type theory. The type theory in this discourse is the one developed by Per Martin-Löf [14, 15] which is also called intuitionistic type theory. It is based on the Curry-Howard isomorphism between propositions and the types of its proofs such that it can served as a formalisation of mathematics. For detailed introduction, please refer to[19]. In this report, Type Theory specially indicates Martin-Löf type theory.

Per Martin-Löf proposed both an intensional and an extensional variants of his intuitionistic type theory. The distinction between them is whether definitional equality is distinguished with propositional equality. In intensional Type Theory, definitional equality exists between two intensionally identical objects, but propositional equality is a type which requires proof terms. Any thing is only definitionally equal to itself and all terms that can be normalised to it which means that definitional equality is decidable in intensional Type Theory. Therefore type checking which depends on definitional equality is decidable as well [1] . The propositional equality we use in intensional Type Theory is written as $Id(A, a, b)$ [18] which is also called *intensional equality*. In Agda, an implementation of intensional Type Theory, it is redefined as $\mathsf{a} \equiv \mathsf{b}$ with the type Aimplicitly, and this set has an unique element refl only if a and b are definitionally equal. However in extensional Type Theory, they are not distinguished so that if we have $p : Eq(A, a, b)$ which is called extensional equality, a and b are definitional equal. Terms which have different normal forms may be definitional equal or not. In other words, definitional equality is undecidable and type checking become undecidable as well. Altenkirch and McBride [2] introduce a variant of extensional Type Theory called *Observational Type Theory* in which definitional equality is decidable and propositional equality is extensional.

Type theory can also serve as a programming language in which the evaluation of well-typed program always terminates [18]. There are a few implementation based on different type theories, such as NuPRL, Coq and Agda. Agda is one of the most recent implementation of intensional version of Martin-Löf type theory. As we have seen that Martin-Löf type theory is based on the Curry-Howard isomorphism, types are identified with propositions and programs or terms are identified with proofs. Therefore it is not only a programming language but also a theorem prover which allows user to verify Agda programs in itself. Compared to other implementations, it has a bundle of good features like pattern matching, unicode input, implicit arguments etc [6] but it does not have tactics such that the proofs are less readable. Since this project is based on Martin-Löf type theory, it is a good choice to implement our definitions and verify propositions in Agda.

Although type theory has some similarities to set theory, they are fundamentally different. Types play a similar role to sets and are also called sets in many situations. However we can only create elements after we declare their types, while in set theory elements are there before we have sets. For example, we have type $\mathbb{N}$ for natural numbers corresponding to the set of natural numbers in set theory. In set theory, 2 is a shared element of the set of natural numbers and the set of integers. While in type theory, $\mathbb{N}$ provides us two constructors $zero : \mathbb{N}$ and $suc : \mathbb{N} \to \mathbb{N}$, and 2 can be constructed as $suc\,(suc\ zero)$ which is of type $\mathbb{N}$ and does not have any other types like $\mathbb{Z}$. Because different sets may contain the same elements, we have the subset relation such that we can construct equivalence classes and quotient set. In type theory we have to give constructors for any type before we can construct elements which is different to the situation in set theory that elements exist before we construct quotient sets.

The approach in set theory fails here. So how can we reinterpret quotient sets in type theory?

1.3 Quotient Types

Following the correspondence between sets and types, many notions from set theory can be reinterpreted in type theory. The product of sets can be formed by $\Sigma - Type$ and the functions can be formed by $\Pi - Type$ [19]. However in intensional Type Theory quotient types are still unavailable and it is a problematic issue to interpret quotients.

Alternatively, in intensional Type Theory we have the ingredients of quotients as follows,

Definition 1.1. A setoid $(A, \sim) : \mathbf{Set}_1$ is a set $A : \mathbf{Set}$ equipped with an equivalence relation $\sim : A \to A \to \mathbf{Prop}$.

Here we assume **Set** means type and for any $p\,q :$ **Prop**, $p = q$. In Agda, we define a setoid as

```
record Setoid : Set₁ where
  infix 4 _≈_
  field
    Carrier : Set
    _≈_ : Carrier → Carrier → Set
    isEquivalence : IsEquivalence _≈_
```

Setoid could be universe polymorphic.

We can use setoids to represent quotients. However setoids is different from sets so that we have to redefine all the operations on sets and it is unsafe [3]. Another interesting problem is how to represent quotients if the A is already a setoid. It means that it is better that if the base type A is of type **Set** then the type of quotient should also be of type **Set**. It should be universe polymorphic as well. From mathematical perspective, we also found the structure of the base object is always the same as the structure of the result quotient object. So what should be a quotient type?

Definition 1.2. Given a setoid $(A, \sim) : \mathbf{Set}_1$, there is a type $Q : \mathbf{Set}$ which is an implementation of the quotient set $A/\sim$, Q is called the quotient type of $(A, \sim)$

For example, given a setoid $(\mathbb{N} \times \mathbb{N}, \sim)$ where $\sim$ is defined as

$$(a, b) \sim (c, d) \stackrel{\text{def}}{=} a + d \equiv c + b$$

and $\sim$ is proved to be an equivalence relation, the quotient type corresponding to this setoid is just the set of integers $\mathbb{Z} :$ **Set**.

Quotient types can be very useful. The encoding of quotients in mathematics is not all what quotient types can do. It is a type theoretical notion which

means some notions in Type Theory or in programming languages can also be treated as quotient types. For example partiality monad divided by a weak similarity ignoring finite delays [3], propositions divided by $\Longleftrightarrow$ and the set of extensionally equal functions. Also set-theoretical finite sets can be implemented as the quotient of lists in Type Theory. Furthermore given any function $f : A \to B$, we obtain an equivalence relation $\sim : A \to A \to \mathbf{Prop}$ called *kernel* of f which is defined as $a \sim b \stackrel{\text{def}}{=} f\, a \equiv f\, b$. Based on this setoid $(A, \sim)$ we can form a quotient. Indeed all types can be seen as quotient types of itself with the intensional equality $\equiv$.

In the definition of quotient types, we do not provide an approach to construct them from given setoids. Indeed how to obtain a quotient type of a given setoid is one of the main topic of this project.

One feasible approach in current setting of intensional Type Theory is to manually construct the quotient type and prove it is the required quotient type. For instances in [20], the normal form integers, irreducible rational numbers are definable and proved to construct quotients with respect to the corresponding setoids. The quotient interfaces introduced in [3] requires the necessary properties for some type Q to be the quotient type of some setoid $(A, \sim)$. Since they are basics of quotients, we can use them to lift operations and prove some general theorems. However, this approach is inefficient because the quotient types and the properties have to be manually figured out rather than automatically derived. Furthermore, some quotients like real numbers are undefinable even though we can define the base type and equivalence relation for them [20] . Although it has some drawbacks, it is feasible without extending intensional Type Theory and it provides some convenience in practice. There has been some results on this [3]. I will discuss them later.

The ideal approach should be an axiomatised type former for quotient types. It means that we have to extend intensional Type Theory with the introduction rules and elimination rules of quotient types. However there are many problems arising, for example the constructors for quotient types, the definitional equality of quotient types etc.

Quotient types can be seen as the result of replacing the equivalence relation of given types. This operation does not work in intensional Type Theory, but it seems easier to manage in extensional Type Theory where propositional equal terms are also definitional equal. Nevertheless there are still some problems which we discuss in the literature review.

1.4 Functional extensionality and quotient types

In intensional Type Theory intensional propositional equality $Id(A, a, b)$ is inhabited if and only if a and b are definitional equal terms,

```
data Id (A : Set) : A → A → Set where
  refl : (a : A) → Id A a a
```

but the intensional propositional equalities of functions are not inhabited [1]. When talking about the equality of functions, they are usually treated exten-

sionally as black boxes. If two functions pointwise generate the same outputs for the same inputs, they are equivalent even though their inside definitions may differ. This is called functional extensionality which is not inhabited in original intensional Type Theory and can be expressed as following,

given two types A and B, and two functions f, $g : A \to B$,

$$Ext = \forall\, x\colon A, fx = gx \to f = g$$

If seems that we just need to add the constant Ext to intensional Type Theory. However postulate something new may cause inconsistence. If we postulate Ext, then the theory is said to be not adequate which means it is possible to define irreducible terms. It can be easily verified in Agda through formalising a non-canonical term of natural number by an eliminator of intensional equality. Firstly we postulate Ext

```
postulate
  Ext : {A : Set} {B : A → Set} {f g : (x : A) → B x}
    → ((x : A) → Id (B x) (f x) (g x))
    → Id ((x : A) → B x) f g
```

And the eliminator of the intensional equality is

```
J : (A : Set) (P : (a b : A) → Id A a b → Set)
  → ((a : A) → P a a (refl a))
  → (a b : A) (p : Id A a b) → P a b p
J A P m .b b (refl .b) = m b
```

Finally we can construct a irreducible term of natural number as

```
irr : ℕ
irr = J (ℕ → ℕ) (λ a b x → ℕ) (λ a → 0) (λ x → x) (λ x → x) (Ext refl)
```

Because of the inconsistency, the definitional equality and type checking become undecidable.

Altenkirch investigates this issue and gives a solution in [1]. He propose an extension of intensional Type Theory by a universe of propositions **Prop** in which all proofs of same propositions are definitionally equal, namely it is proof irrelevant. At the same time, a setoid model where types are interpreted by a type and an equivalence relation acts as the metatheory and η-rules for Π-types and Σ-types hold in the metatheory. The extended type theory generated from the metatheory is decidable and adequate, Ext is inhabited and it permits large elimination. Within this type theory, introduction of quotient types is straightforward. The set of functions are naturally quotient types, the hidden information is the definition of the functions and the equivalence relation is the functional extensionality.

There are more problems concerning quotient types and most of them are related to equality. One of the main problems is how to lift the functions for base types to the ones for quotient types. Only functions respects the equivalence

relation can be lifted. Even in extensional Type Theory, the implementation of quotient types does not stop at replacing equality of the types. We will discussed these later.

1.5 Literature Review

In [7], Mendler et al. have firstly considered building new types from a given type using a quotient operator //. Their work is based on an implementation of extensional Type Theory, NuPRL. In NuPRL, every type comes with its own equality relation, so the quotient operator can be seen as a way of redefining equality in a type. But it is not all about building new types. They also discuss the problems arises from defining functions on the new type. The problem can be illustrated using a simple example.

Assume the base type is A and the new equality relation is E, the new type can be formed as $A//E$. If we want to define a function $f : A//E \to Bool$, Assume we have two $a, b : A$ such that $E\,a\,b$ but $f\,a \neq f\,b$, then it becomes inconsistent since $E\,a\,b$ implies a converts to b in extensional Type Theory, the left hand side $f\,a$ can be converted to $f\,b$ namely $f\,b \neq f\,b$ which is contradicted with the equality reflection rule. Only a function respects the equivalence relation, namely

$$\forall\, a\, b : A, E\,a\,b \to f\,a = f\,b$$

f is said to be well-defined on the new type. We call it *sound* in [3] and this project.

After the introduction of quotient types, Mendler further investigates this topic from a categorical perspective in [16]. He use the correspondence between quotient types in Martin-Löf type theory and coequalizers in a category of types to define a notion called *squash types* which is further discussed by Nogin.

To adding quotient types to Martin-Löf type theory, Hofmann proposes three models for quotient types in his PhD thesis [10]. The first one is a setoid model for quotient types. In this model all types are attached with partial equivalence relations, namely all types are setoids rather than sets. Types without specific equivalence relation can be translated as setoids with the basic intensional equality. It looks a bit like extensional Type Theory. The second one is groupoid model which solves some problems but it is not definable in intensional Type Theory. He also proposes a third model to combine the advantages of the first two models, but it also has some disadvantages. Later in [11] he gives a simple model in which we have type dependency only at the propositional level, he also shows that extensional Type Theory is conservative over intensional Type Theory extended with quotient types and a universe [12].

Nogin [17] considers a modular approach to axiomatizing the same quotient types in NuPRL as well. Despite the ease of constructing new types from base types, he also discusses some problems about quotient types. For example, since the the equality is extensional, we can not recover the witness of the equality. He suggests to include more axioms to conceptualise quotients. He decomposes

the formalisation of quotient type into several smaller primitives such that they can be handled much simpler.

Homeier [13] axiomatises quotient types in Higher Order Logic (HOL), which is also a theorem prover. He creates a tool package to construct quotient types as a conservative extension of HOL such that users are able to define new types in HOL. Next he defines the normalisation functions and proves several properties of these. Finally he discussed the issues when quotienting on the aggregate types such as lists and pairs.

Courtieu [9] shows an extension of Calculus of Inductive Constructions with *Normalised Types* which are similar to quotient types, but equivalence relations are replaced by normalisation functions. However not all quotient types have normal forms. Normalised types are proper subsets of quotient types, because we can easily recover a quotient type from a normalised type as below

$$(A, Q, [\,\cdot\,] \colon A \to Q) \Rightarrow (A, \lambda\, a\, b \to [a] = [b])$$

Barthe and Geunvers [4] also proposes a new notion called *congruence types*, which is also a special class of quotient types, in which the base type are inductively defined and with a set of reduction rules called the term-rewriting system. The idea behind it is the β-equivalence is replaced by a set of β-conversion rules. Congruence types can be treated as an alternative to pattern matching introduced in [8]. The main purpose of introduction of congruence types is to solve problems in term rewriting systems rather than to implement quotient types.

2 Aims and Objectives of the Project

The objective of this project is to investigate and explore the ways of implementing quotients in Martin-Löf type theory, especially in intensional one where type checking always terminates. As we have seen quotients can enable defining various kinds of mathematical notions or programming datatypes, the introduction of quotient types will be quite beneficial in theorem provers and programming languages based on Type Theory.

The ultimate aim is to extend intensional Type Theory such that all quotients can be defined and handled easily and correctly without losing consistency and good features of intensional Type Theory.

The project will be undertaken step by step. Firstly, we should make the basic notions clear, for example what is quotients and if we want quotients in Type Theory what kind of problems need to be solved. We also need to do research on related works on this topic as much as possible. The second step is working in current setting of intensional Type Theory, investigating some definable quotients, and building the module structure of quotients. The module structure and some research on definable quotients has been done in [3].

Next we need to investigate some undefinable quotients such as the set of real numbers $\mathbb{R}$ and partiality monads and prove why they are undefinable. The key different characters between definable and undefinable quotients will be studied. You can find the proof of why $\mathbb{R}$ is undefinable in [3] as well.

The development of framework of quotient types in intensional Type Theory is one of the major objective. We need to propose a set of rules to axiomatise quotient types in intensional Type Theory. To test our approach with a few typical quotients to explore its potential benefits. The correctness of axiomatisation and the consistency of extended intensional Type Theory require formal proofs.

A possible intermediate result is to extend the addition of extensionality in intensional Type Theory [**?**]. The conservativity of intensional Type Theory with quotient types over extensional Type Theory could be proved following the work in [**?**].

Finally, we will summarise the approach of defining quotient types in intensional Type Theory, the benefits of it and the application of it into a PhD Thesis.

3 Theoretical Methods

To do this research, we need to review several related work, compare the existing approaches in different implementations of Martin-Löf type theory and try to figure out the best approach by testing it in real cases.

As we have mentioned before, Agda is a good implementation of intensional Type Theory. Conducting this research in Agda will be very efficient and useful. We can verify our proofs in it and try to apply quotient types to a lot of practical examples.

We also need to advertise our work, get feedbacks from the users and improve our approaches such that they are more applicable and easier to use.

4 Results and Discussion

4.1 Definitions

Currently, we are on the first stage and there are some progression on definable quotients in intensional Type Theory [3]. Here I will present some necessary knowledge from that paper.

During the first stage, the aim is to explore the potential to define quotients in current setting of intensional Type Theory.

Given a setoid $(A, \sim)$, we know what is a quotient type but we can not define it from the setoid because there is no axiomatised quotient types. If means that we can just prove some type is quotient type of given setoid. Therefore the only way to introduce possible quotient types $Q : \mathbf{Set}$ is to define it by ourselves. With defined Q and $(A, \sim)$, we are required construct some structures of quotients in [3] which consists of a set of essential properties of quotients.

Here I will explain these structures by using the example of integers in Agda. All integers are the result of subtraction between two natural numbers. Therefore we can use a pair of natural numbers in a subtraction expression to represent the resulting integer. For example, $1 - 4 = -3$ says that the pair of natural

numbers $(1, 4)$ represents the integer -3. Assuming we have the necessary definitions of natural numbers, the base type of this quotient is

$$\mathbb{Z}_0 = \mathbb{N} \times \mathbb{N}$$

Mathematically we know that for any two pairs of natural numbers (n_1, n_2) and (n_3, n_4),

$$n_1 - n_2 = n_3 - n_4 \iff n_1 + n_4 = n_3 + n_2$$

Because the results of subtraction are the same, we can infer that the two pairs represent the same integer, so the equivalence relation $\sim$ for $\mathbb{Z}_0$ could be defined as

```
_∼_ : Rel ℤ₀ zero
(x+, x-) ∼ (y+, y-) = (x+ ℕ+ y-) ≡ (y+ ℕ+ x-)
```

Here $\equiv$ is the propositional equality. Of course we must prove $\sim$ is an equivalence relation then we can define the setoid $(\mathbb{Z}_0, \sim)$ in Agda as

```
ℤ-Setoid : Setoid
ℤ-Setoid = record
  { Carrier = ℤ₀
  ; _≈_ = _∼_
  ; isEquivalence = _∼_isEquivalence
  }
```

In set theory, we can immediately derive the quotient set from this setoid which is the set of integers $\mathbb{Z}$, but in current setting of intensional Type Theory, we need to define $\mathbb{Z}$ as follows

```
data ℤ : Set where
  +_ : (n : ℕ) → ℤ
  -suc_ : (n : ℕ) → ℤ
```

This is called normal form or canonical form of integers.

The next step is to prove that it is the quotient type of the setoid $(\mathbb{Z}_0, \sim)$. To relate the setoid and the potential quotient type, we need to provide a mapping function from the base type $\mathbb{Z}_0$ to the target type $\mathbb{Z}$ which should be the normalisation function

```
[_]           : ℤ₀ → ℤ
[m, 0]        = + m
[0, ℕ.suc n]  = -suc n
[ℕ.suc m, ℕ.suc n] = [m, n]
```

The first property to prove is the *sound* property,

```
sound : ∀ {x y} → x ∼ y → [x] ≡ [y]
sound {x} {y} x∼y = ⌞ compl >∼< x∼y >∼< compl' ⌟
```

The normalised results of two propositional equal elements of $\mathbb{Z}_0$ should be the same. With this property, we are able to form a prequotient which is defined as

```
record PreQu (S : Setoid) : Set₁ where
  constructor
    Q:_ [ ] :_sound:_
  private
    A = Carrier S
    _∼_ = _≈_ S
  field
    Q   : Set
    [_] : A → Q
    sound : ∀ {a b : A} → a ∼ b → [a] ≡ [b]
```

and the prequotient of integers is,

```
ℤ-PreQu : PreQu ℤ-Setoid
ℤ-PreQu = record
  {Q      = ℤ
  ;[_]    = [_]
  ;sound  = sound
  }
```

To form quotients we have several different definitions as written in [3],

1. *Quotient with a dependent eliminator*

```
record Qu {S : Setoid} (PQ : PreQu S) : Set₁ where
  constructor
    qelim:_qelim-β:_
  private
    A      = Carrier S
    _∼_    = _≈_ S
    Q      = Q' PQ
    [_]    = nf PQ
    sound : ∀ {a b : A} → (a ∼ b) → [a] ≡ [b]
    sound  = sound' PQ
  field
    qelim : {B : Q → Set}
          → (f : (a : A) → B [a])
          → ((a b : A) → (p : a ∼ b)
            → subst B (sound p) (f a) ≡ f b)
          → (q : Q) → B q
    qelim-β : ∀ {B a f} q → qelim {B} f q [a] ≡ f a
```

2. *Exact (or efficient) quotient*

```
record QuE {S : Setoid} {PQ : PreQu S} (QU : Qu PQ) : Set₁ where
  constructor
    exact:_
  private
    A = Carrier S
    _∼_ = _≈_ S
    [_] = nf PQ
  field
    exact : ∀ {a b : A} → [a] ≡ [b] → a ∼ b
```

3. *Quotient with a non-dependent eliminator and induction principle*

```
record QuH {S : Setoid} (PQ : PreQu S) : Set₁ where
  constructor
    lift:_lift-β:_qind:_
  private
    A       = Carrier S
    _∼_     = _≈_ S
    Q       = Q' PQ
    [_]     = nf PQ
  field
    lift    : {B : Set}
            → (f : A → B)
            → ((a b : A) → (a ∼ b) → f a ≡ f b)
            → Q → B
    lift-β  : ∀ {B a f q} → lift {B} f q [a] ≡ f a
    qind    : (P : Q → Set)
            → (∀ x → (p p' : P x) → p ≡ p')
            → (∀ a → P [a])
            → (∀ x → P x)
```

4. *Definable quotient*

```
record QuD {S : Setoid} (PQ : PreQu S) : Set₁ where
  constructor
    emb:_complete:_stable:_
  private
    A       = Carrier S
    _∼_     = _≈_ S
    Q       = Q' PQ
    [_]     = nf PQ
  field
    emb     : Q → A
```

```
complete : ∀ a → emb [a] ∼ a
stable : ∀ q → [emb q] ≡ q
```

We have proved that the first and third definitions are equivalent and the last one is the most strongest definition which can generate any other from it [3].

For integers, it is natural to define a function to choose a representative for each element in $\mathbb{Z}$,

```
⌜_⌝      : ℤ → ℤ₀
⌜ + n ⌝ = n, 0
⌜ -suc n ⌝ = 0, ℕ.suc n
```

Now we need to prove $\ulcorner \_ \urcorner$ is the required embedding function, namely it is the inverse function of $[\_]$. Firstly $\ulcorner \_ \urcorner$ is left inverse of $[\_]$,

```
compl                : ∀ {n} → ⌜ [n] ⌝ ∼ n
compl {x, 0}         = refl
compl {0, nsuc y}    = refl
compl {nsuc x, nsuc y} = compl {x, y} >∼< ⟨ ℕ.sm+n≡m+sn x y ⟩
```

This is called the *complete* property.

Secondly $\ulcorner \_ \urcorner$ is right inverse of $[\_]$,

```
stable          : ∀ {n} → [⌜ n ⌝] ≡ n
stable {+ n}    = refl
stable {-suc n} = refl
```

This is called the *stable* property.

Now we can form the definable quotient structure with the prequotient we have,

```
ℤ-QuD : QuD ℤ-PreQu
ℤ-QuD = record
  {emb      = ⌜_⌝
  ;complete = λ z → compl
  ;stable   = λ z → stable
  }
```

Now we have the mapping between the base type $\mathbb{Z}_0$ and the target type $\mathbb{Z}$, and have proved that $[\_]$ is a normalisation function.

We can obtain the dependent and non-dependent eliminators by translate the definable quotient into other definitions,

```
ℤ-Qu  = QuD→Qu ℤ-QuD
ℤ-QuE = QuD→QuE {_} {_} {ℤ-Qu} ℤ-QuD
ℤ-QuH = QuD→QuH ℤ-QuD
```

It is not easy to find what benefit we obtain from constructing quotients. The real benefit is generated from the interaction between setoids and quotient types. Firstly the setoid definitions are usually more simpler than the normal definitions. In the case of integers, the normal form have two constructors. For propositions with only one argument, sometimes we have to prove them for both cases in the canonical definition. With the increasing number of arguments in propositions, the number of cases we need to prove would increase exponentially. A real case is when trying to prove the distributivity of multiplication over addition for integers, there are too many cases which makes the proving cumbersome and we can hardly save any effort from any theorems we proved. However for the setoid definition of integers, a proposition can be converted into another proposition on natural numbers which is much convenient to prove because we do not need to prove case by case and we have a bundle of theorems for natural numbers. For example,

```
dist^r : _*_ DistributesOver^r _+_
dist^r (a, b) (c, d) (e, f) =
  ℕ.dist-lem^r a b c d e f +=
  ⟨ ℕ.dist-lem^r b a c d e f ⟩
```

Moreover, as we have constructed the semiring of natural numbers, it is even simpler to use an automatic prover *ring solver* to prove simple equation of natural numbers.

The rest we have to do is to lift the properties proved for setoid definition to the ones for canonical definition. We can easily lift n-ary operators defined for $\mathbb{Z}_0$ to the ones for $\mathbb{Z}$ by

```
liftOp : ∀ n → Op n ℤ₀ → Op n ℤ
liftOp 0 op = [op]
liftOp (ℕ.suc n) op = λ x → liftOp n (op ⌜ x ⌝)
```

However, this lift function is unsafe because some operations on $\mathbb{N} \times \mathbb{N}$ do not make sense by applying this function. It is similar to the situation when defining functions on types with replaced equality in extensional Type Theory. The solution is to lift functions which respects the equivalence relation. I only define the two most commonly used safe lifting functions

```
liftOp1safe : (f : Op 1 ℤ₀) →
  (∀ {a b} → a ∼ b → f a ∼ f b) →
    Op₁ ℤ
liftOp1safe f cong = λ n → [f ⌜ n ⌝]
liftOp2safe : (op : Op 2 ℤ₀) →
  (∀ {a b c d} → a ∼ b → c ∼ d →
    op a c ∼ op b d) →
  Op₂ ℤ
liftOp2safe _op_ cong = λ m n → [⌜ m ⌝ op ⌜ n ⌝]
```

Then we can obtain the β-laws which are very useful,

```
liftOp1-β : (f : Op 1 ℤ₀) →
  (cong : ∀ {a b} → a ∼ b → f a ∼ f b) →
  ∀ n → liftOp1safe f cong [n] ≡ [f n]
liftOp1-β f cong n = sound (cong compl)
liftOp2-β : (op : Op 2 ℤ₀) →
  (cong : ∀ {a b c d} → a ∼ b → c ∼ d →
    op a c ∼ op b d) →
  ∀ m n → liftOp2safe op cong [m] [n] ≡ [op m n]
liftOp2-β op cong m n = sound (cong compl compl)
```

Now we can lift the negation easily

```
-_ : Op 1 ℤ
-_ = liftOp1safe ℤ₀.-_ ℤ₀.⁻¹-cong
```

and the β-laws for negation can be proved as

```
-β : ∀ a → - [a] ≡ [ℤ₀- a]
-β = liftOp1-β ℤ₀-_ ℤ₀.⁻¹-cong
```

When trying to prove theorems for canonical integers, we can lift proved properties for the setoid integers, such as commutativity of any binary operations,

```
liftComm : ∀ {op : Op 2 ℤ₀} →
  P.Commutative _∼_ op →
  Commutative (liftOp 2 op)
liftComm {op} comm x y = sound $ comm ⌜ x ⌝ ⌜ y ⌝
```

The generalised lifting function for commutativity is also one of the derived theorem of quotients as it only uses sound and $\ulcorner\_\urcorner$ which are part of the quotients. then we can lift the commutativity of addition and multiplication,

```
+-comm : Commutative _+_
+-comm = liftComm ℤ₀.+-comm
*-comm : Commutative _*_
*-comm = liftComm ℤ₀.*-comm
```

It is also much simpler to prove the complicated distributivity of multiplication over addition,

```
distʳ : _*_ DistributesOverʳ _+_
distʳ a b c = sound $ ℤ₀.*-cong (compl {⌜ b ⌝ ℤ₀+ ⌜ c ⌝}) zrefl >∼<
  ℤ₀.distʳ ⌜ a ⌝ ⌜ b ⌝ ⌜ c ⌝ >∼<
  ℤ₀.+-cong compl' compl'
```

There is no need to use pattern matching, namely prove the propositions inductively. The simplicity of the proof is achieved by applying the quotient

properties such as sound, we can translate or convert the proposition into the corresponding proposition for setoid integers. We can further translating the propositions for setoid integers into some easier propositions for natural numbers. The connections between canonical integers and natural numbers is built by the definition of quotient.

We can also lift a structure of properties such as monoid,

```
liftId : ∀ {op : Op 2 ℤ₀} (e : ℤ) →
  P.Identity _∼_ ⌜ e ⌝ op →
  Identity e (liftOp 2 op)
liftId e (idl, idr) = (λ x → sound (idl ⌜ x ⌝) >≡< stable),
  (λ x → sound (idr ⌜ x ⌝) >≡< stable)
liftAssoc : ∀ {op : Op 2 ℤ₀} (cong : Cong2 op) →
  P.Associative _∼_ op →
  Associative (liftOp2safe op cong)
liftAssoc {op} cong assoc a b c = sound $ cong (compl {op ⌜ a ⌝ ⌜ b
  ⌝}) zrefl >∼< assoc ⌜ a ⌝ ⌜ b ⌝ ⌜ c ⌝ >∼< cong zrefl compl'
liftMonoid : {op : Op 2 ℤ₀} {e : ℤ} (cong : Cong2 op) →
  IsMonoid _∼_ op ⌜ e ⌝ →
  IsMonoid _≡_ (liftOp 2 op) e
liftMonoid {op} {e} cong im = record
  {isSemigroup = record
    {isEquivalence = isEquivalence
    ;assoc = liftAssoc cong (IsMonoid.assoc im)
    ;•-cong = cong₂ (liftOp 2 op)
    }
  ;identity = liftId {op} e (IsMonoid.identity im)
  }
```

These lift functions for operators and properties can be generalised even further such that they can be applied to all quotients. They are all derived theorems for quotients which can save a lot of work for us. We can reuse them in the next example, the set of rational numbers $\mathbb{Q}$.

4.2 Rational numbers

The quotient of rational numbers is better known than the previous quotient. We usually write two integers m and n (n is not zero) in fractional form $\frac{m}{n}$ to represent a rational number. Alternatively we can use an integer and a positive natural number such that it is simpler to exclude 0 in the denominator. Two fractions are equal if they are reduced to the same irreducible term. If the numerator and denominator of a fraction are coprime, it is said to be an irreducible fraction. Based on this observation, it is naturally to form a definable quotient, where the base type is

$$\mathbb{Q}_0 = \mathbb{Z} \times \mathbb{N}$$

The integer is *numerator* and the natural number is *denominator-1*. This approach avoids invalid fractions from construction.

In Agda, to make the terms more meaningful we define it as

```
data ℚ₀ : Set where
  _/suc_ : (n : ℤ) → (d : ℕ) → ℚ₀
```

In mathematics, to judge the equality of two fractions, it is easier to conduct the following conversion,

$$\frac{a}{b} = \frac{c}{d} \iff a \times d = c \times b$$

Therefore the equivalence relation can be defined as,

```
_∼_ : Rel ℚ₀ zero
n1 /suc d1 ∼ n2 /suc d2 = n1 ℤ*ℕ suc d2 ≡ n2 ℤ*ℕ suc d1
```

The normal form of rational numbers, namely the quotient type in this quotient is the set of irreducible fractions. We only need to add a restriction that the numerator and denominator is coprime,

$$\mathbb{Q} = \Sigma(n\colon \mathbb{Z})\Sigma(d\colon \mathbb{N}), \text{Coprime } n\,(d+1)$$

We can encode it using record type in Agda,

```
record ℚ : Set where
  field
    numerator : ℤ
    denominator-1 : ℕ
    isCoprime : True (C.coprime? | numerator | (suc denominator-1))
```

The normalisation function is an implementation of the reducing process, the gcd function which calculates the greatest common divisor can help us reduce the fraction and give us the proof of coprime,

```
⌊_⌋ : ℚ₀ → ℚ
⌊(+ 0) /suc d⌋ = ℤ.+_ 0 ÷ 1
⌊(+ (suc n)) /suc d⌋ with gcd (suc n) (suc d)
⌊(+ suc n) /suc d⌋ | di , g = GCD′→ℚ (suc n) (suc d) di (λ ()) (C.gcd-gcd′ g)
⌊(-suc n) /suc d⌋ with gcd (suc n) (suc d)
... | di , g = - GCD′→ℚ (suc n) (suc d) di (λ ()) (C.gcd-gcd′ g)
```

The embedding function is simple. We only need to forget the coprime proof in the normal form,

```
⌜_⌝ : ℚ → ℚ₀
⌜ x ⌝ = (ℚ.numerator x) /suc (ℚ.denominator-1 x)
```

Similarly, we are able to construct the setoid, the prequotient and then the definable quotient of rational numbers. We can benefit from the ease of defining operators and proving theorems on setoids while still using the normal form of rational numbers which is safer. and the lifted operators and properties.

4.3 Real numbers

The previous quotient types are all definable in intensional Type Theory so that we can construct the definable quotients for them. However, there are some types undefinable in intensional Type Theory. The set of real numbers $\mathbb{R}$ has been proved to be undefinable in [3].

We have several choices to represent real numbers. We choose Cauchy sequences of rational numbers to represent real numbers [5].

$$\mathbb{R}_0 = \{s : \mathbb{N} \to \mathbb{Q} \mid \forall \varepsilon : \mathbb{Q}, \varepsilon > 0 \to \exists m : \mathbb{N}, \forall i : \mathbb{N}, i > m \to |s_i - s_m| < \varepsilon\}$$

We can implement it in Agda. First a sequence of elements of A can be represented by a function from $\mathbb{N}$ to A.

```
Seq : (A : Set) → Set
Seq A = ℕ → A
```

And a sequence of rational numbers converges to zero can be expressed as follows,

```
Converge : Seq ℚ₀ → Set
Converge f = ∀ (ϵ : ℚ₀*) → ∃ λ lb → ∀ m n →
  | (f (suc lb + m)) - (f (suc lb + n)) | <' ϵ
```

Now we can write the Cauchy sequence of rational numbers,

```
record ℝ₀ : Set where
  constructor f:_p:_
  field
    f : Seq ℚ₀
    p : Converge f
```

To complete the setoid for real numbers, an equivalence relation is required. In mathematics two Cauchy sequences $\mathbb{R}_0$ are said to be equal if their pointwise difference converges to zero,

$$r \sim s = \forall \varepsilon : \mathbb{Q}, \varepsilon > 0 \to \exists m : \mathbb{N}, \forall i : \mathbb{N}, i > m \to |r_i - s_i| < \varepsilon$$

The Agda version is

```
_Diff_on_ : Seq ℚ₀ → Seq ℚ₀ → Seq ℚ₀*
f Diff g on m = | f m - g m |
_~_ : Rel ℝ₀ zero
(f: f p: p) ~ (f: f' p: p') =
  ∀ (ϵ : ℚ₀*) → ∃ λ lb → ∀ i → (lb < i) → f Diff f' on i <' ϵ
```

In set theory we can construct quotient set $\mathbb{R}_0/\sim$. However since real numbers have no normal forms we can not define the quotient in intensional Type Theory. Hence the definable quotient definition does not work for it.The undefinability of the any type $\mathbb{R}$ which is the quotient type of the setoid $(\mathbb{R}_0, \sim)$ is proved by local continuity [3].

4.4 All epimorphisms are split epimorphisms

In addition we also proved that classically all epimorphisms are split epimorphisms.

A morphism e is an epimorphism if it is right-cancellative

```
Epi : {A B : Set} → (e : A → B) → (C : Set) → Set
Epi {A} {B} e C =
  ∀ (f g : B → C) →
  (∀ (a : A) → f (e a) ≡ g (e a)) →
  ∀ (b : B) → f b ≡ g b
```

If it has a right inverse it is called a split epi

```
Split : {A B : Set} → (e : A → B) → Set
Split {A} {B} e = ∃ λ (s : B → A) → ∀ b → e (s b) ≡ b
```

We assume the axioms of classical logic

```
postulate classic : (P : Set) → P ∨ (¬ P)
raa : {P : Set} → ¬ (¬ P) → P
raa {P} nnp with classic P
raa nnp | inl y = y
raa nnp | inr y with nnp y
... | ()
contrapositive : ∀ {P Q : Set} → (¬ Q → ¬ P) → P → Q
contrapositive nqnp p = raa (λ nq → nqnp nq p)
```

We also need one of the De Morgan's law in classical logic

```
postulate DeMorgan : ∀ {A : Set} {P : A → Set} →
¬ (∀ (x : A) → P x) → ∃ λ (x : A) → ¬ P x
```

What we need to prove is

```
Epi→Split : {A B : Set} → (e : A → B) → Set₁
Epi→Split e = ((C : Set) → Epi e C) → Split e
```

Because we have classical theorems, it is equivalent to prove the contrapositive of Epi→Split. To make the steps clear, we decompose the complicated proof. In order, we postulate the following things first

```
postulate A B : Set
postulate e : A → B
postulate ¬split : ¬ Split e
```

Now from the assumption that e is not a split, we can find an element $b : B$ which is not the image of any element $a : A$ under e

```
¬surj : ∃ λ b → ¬ (∃ λ (a : A) → (e a ≡ b))
¬surj = DeMorgan (λ x → ¬split ((λ b → proj₁ (x b)), λ b → (proj₂ (x b))))
b = proj₁ ¬surj
ignore : ∀ (a : A) → ¬ (e a ≡ b)
ignore a eq = proj₂ ¬surj (a, eq)
```

We can define a constant function

```
f : B → Bool
f x = false
```

and postulate a function to decide whether $x : B$ is equal to b. The reason to postulate it is we do not know the constructor of b and we are sure that if B is definable in Agda, the intensional equality must be decidable,

```
postulate g : B → Bool
postulate gb : g b ≡ true
postulate gb' : ∀ b' → ¬ (b' ≡ b) → false ≡ g b'
```

Finally we can prove e is not an epi

```
¬epiBool : ¬ Epi e Bool
¬epiBool epi with assoc (epi f g (λ a → gb' (e a) (ignore a)) b) gb
... | ()
¬epi : ¬ ((C : Set) → Epi e C)
¬epi epi = ¬epiBool (epi Bool)
```

This proposition can only been applied to definable types A B in intensional Type Theory and is only proved to be true with classic axioms.Therefore it does not make sense for the epimorphism from $\mathbb{R}_0$ to $\mathbb{R}$.

5 Conclusion

In the first phase of the project of quotient types in intensional Type Theory, we investigate the quotients which are definable in current setting of intensional Type Theory. The quotient types are separately defined and then proved to be correct by forming definable quotients of some setoids. The properties contained in the quotient structure is very helpful in lifting functions and propositions for setoids to quotient types. This approach provides us an alternative choice to define functions and prove propositions. It is probably simpler to define functions on setoids and we can reuses proved theorems for the setoids in many cases. However it is a little complicated to build the quotients and is only applicable to quotients which are definable in intensional Type Theory.

In the next phase we will focus on the undefinable quotients and extending intensional Type Theory with axiomatic quotient types. To implement undefinable quotients, a new type former with the essential introduction and elimination

rules is unavoidable. Although the quotient structures only works for definable quotients, it can be a good guidance to axiomatise the quotient types. We can extend the work in [1] and find an approach to extend intensional Type Theory without losing nice features such as termination and decidable type checking. Another possible task is to investigate the conservativity of intensional Type Theory with quotient types over extensional Type Theory.

References

[1] Thorsten Altenkirch. Extensional equality in intensional type theory. In *14th Symposium on Logic in Computer Science*, pages 412 – 420, 1999.

[2] Thorsten Altenkirch. Should extensional type theory be considered harmful? Talk given at the Workshop on Trends in Constructive Mathematics, 2006.

[3] Thorsten Altenkirch, Thomas Anberrée, and Nuo Li. Definable quotients in type theory. 2011.

[4] Gilles Barthe and Herman Geuvers. Congruence types. In *Proceedings of CSL'95*, pages 36–51. Springer-Verlag, 1996.

[5] Errett Bishop and Douglas Bridges. *Constructive Analysis*. Springer, New York, 1985.

[6] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda - a functional language with dependent types. In *Theorem Proving in Higher Order Logics*, pages 73–78, 2009.

[7] Robert L. Constable, Stuart F. Allen, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, Scott F. Smith, James T. Sasaki, and S. F. Smith. Implementing mathematics with the nuprl proof development system, 1986.

[8] Thierry Coquand. Pattern matching with dependent types. In *Types for Proofs and Programs*, 1992.

[9] Pierre Courtieu. **Normalized types**. In *Proceedings of CSL2001*, volume 2142 of *Lecture Notes in Computer Science*, 2001.

[10] Martin Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, School of Informatics., 1995.

[11] Martin Hofmann. A simple model for quotient types. In *Proceedings of TLCA'95, volume 902 of Lecture Notes in Computer Science*, pages 216–234. Springer, 1995.

[12] Martin Hofmann. Conservativity of equality reflection over intensional type theory. In *Selected papers from the International Workshop on Types for Proofs and Programs*, TYPES '95, pages 153–164, London, UK, 1996. Springer-Verlag.

[13] Peter V. Homeier. Quotient types. In *In TPHOLs 2001: Supplemental Proceedings*, page 0046, 2001.

[14] Per Martin-Löf. A theory of types. Technical report, University of Stockholm, 1971.

[15] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI, Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science*, volume 104, pages 153 – 175. Elsevier, 1982.

[16] N.P. Mendler. Quotient types via coequalizers in martin-lof type theory. In *Proceedings of the Logical Frameworks Workshop*, pages 349–361, 1990.

[17] Aleksey Nogin. Quotient types: A modular approach. In *ITU-T Recommendation H.324*, pages 263–280. Springer-Verlag, 2002.

[18] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory: an introduction.* Clarendon Press, New York, NY, USA, 1990.

[19] Bengt Nordström, Kent Petersson, and Jan M. Smith. volume 5, chapter Martin-Löf's type theory. Oxford University Press, 10 2000.

[20] Li Nuo. Representing numbers in agda. 2010.

[21] Bertrand Russell. *The Principles of Mathematics.* Cambridge University Press, Cambridge, 1903.