First Year PhD Annual Report

Li Nuo

November 25, 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,quotient sets are called quotient types. However quotient types are still unavailable in Intensional Type Theory which is a very important type theory. Quotients are very common in mathematics and computer science, and thus the introduction of quotients could be very helpful. Some types are less effective to define from scratch than being defined as the quotients of some other types and their equivalence relations, such as the set of integers. Even some sets are impossible to define without being based on quotients such as the set of real numbers. Sometimes, quotient types are more difficult to reason about than their base types. We can achieve more convenience by manipulating base types and then lifting the operators and propositions according to the relation between quotient types and base types. Therefore it is worthwhile for us to 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 introduces the basic notions in my project on implementing quotients in type theory, such as type theory setoids, and quotient types, reviews some work related to this topic and concludes with some results of the first phase. The results done by Altenkirch, Anberrée and I in [4] will be explained with a few instances of quotients.

1 Introduction

In mathematics, the result of division is called quotient. Similar to product, the notion of a quotient is also extended to other more abstract branches of mathematics. For example, we have quotient sets, quotient groups, quotient spaces and quotient categories. They are all defined in this way: some collection of objects is partitioned by some equivalence relation and the set of the equivalent classes is the quotient set which usually bears some algebraic structure inherited from the structure of the original collection of objects.

Quotients are also common in our daily life. When you take a picture with a digital camera, the real scene is divided into pixels on the pictures, and a red point and a blue point on the real scene are indistinguishable on the photo if they are represented by the same pixel. The digital picture which is the set of pixels is just the quotient of the real scene. Since different things are equated with respect to certain rules, a quotient is usually related to information hiding or information losing.

Quotients also exist in computer science. Users are usually concerned with 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 an *interface* in Java. The objects of different classes implementing the same interface A should be treated equally when we create new objects of type A, even though they are not really the same.

However in some type theories, the construction of quotients remains problematic. In this project, we are going to explore this topic in the intensional variant of Martin-Löf type theory (See Section 2.2) which serves as the basis of some useful functional programming languages or theorem provers such as Coq and Agda. This report aims at introducing some basic notions used in this project, reviewing related works and explaining some results that have been done.

The structure of this report is as follows:

Section 2 introduces some basic concepts such as type theory and quotient types along with the discussion of the problems that arise when implementing quotient types. To make them easier to understand, it will start with quotients in set theory which may be more familiar to many people.

Section 3 reviews some related works. Some of them discuss implementing quotient types in other type theories, some of them introduce similar notions. There are also some works done in this area but still require further extensions. These constitute the basis of this project.

Section 4 gives more detailed objectives and a plan of how to achieve them.

Section 5 explains some results that have been done by the author or by Altenkirch, Anberrée and the author in [4].

Section 6 concludes this report along with the discussion of future works.

In Appendix, I will list the full versions of some codes within the text which are less relevant. All the codes are written in Agda which will be introduced later. The readers who are familiar with Agda or interested in the codes could look up them there. To make the codes more readable, I eliminate some unnecessary parts such as the universe polymorphism.

2 Background

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

2.1 Quotient Sets

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

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 A $/ \sim$ is the set of equivalence classes of $\sim$,

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

There are many mathematical notions which can be constructed as quotient sets. For example, the 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. For another example, 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 other words, $\mathbb{Q}$ 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*.

2.2 Type Theory

The theory of types was first introduced by Russell [26] as an alternative to naive set theory. Since then, mathematicians and 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 [18, 19] 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 a detailed introduction, refer to [23].

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 from propositional equality. In Intensional Type Theory, definitional equality exists between two definitionally identical objects, but propositional equality is a type which requires proof terms. Anything 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 that depends on definitional equality is decidable as well [2] . The type for propositional equality in Intensional Type Theory is usually written as $Id(A, a, b)$ (which is also called *intensional equality* [22]). In Agda[24], an implementation of Intensional Type Theory, it is written as $\mathsf{a} \equiv \mathsf{b}$ with the type A often kept implicit. This set has an unique element refl only if a and b are definitionally equal and is uninhabited if not. However in Extensional Type Theory, the two kinds of equalities are not distinguished, so if we have $p : Eq(A, a, b)$ (which is called extensional equality [22]), a and b are definitionally equal. It means that terms which have different normal forms may be definitionally equal. In other words, definitional equality is undecidable and type checking becomes undecidable as well. However Altenkirch and McBride have introduced a variant of Extensional Type Theory called *Observational Type Theory* [3] 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 a well-typed program always terminates [22]. There are various implementations based on different type theories, such as NuPRL, LEGO, Coq, Epigram and Agda. Agda is one of the most recent implementations of intensional version of Martin-Löf type theory. It is a dependently typed programming language, we can write program specifications as types. As we have seen, 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 package of useful features such as pattern matching, unicode input, and implicit arguments [7], but it does not have tactics and consequently its proofs are less readable than implementations that do. Since this project is based on Martin-Löf type theory, it is a good choice to implement our definitions and verify our theorems and properties in Agda. For a detailed introduction of Agda, refer to [24].

To move from set theory to type theory, the similarities and differences should be made clear. Although type theory has some similarities to set theory, their foundations are different. Types play a similar role to sets and they are also called sets in many situations. However we can only create elements after we declare their types, while in set theory elements exist there before we have sets. For example, we have the 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. Therefore this approach to construct quotients in set theory has some problems in type theory. In fact, Voevodsky constructs quotients using this approach in Homotopy Type Theory using Coq [27] but here we mainly discuss how to reinterpret quotient sets in the current settings of Intensional Type Theory (e.g. Agda).

2.3 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 Σ-Types and the functions can be formed by Π-Types [23].

However, in Intensional Type Theory a general approach to construct quotient types is still unavailable.

Alternatively, in Intensional Type Theory, we have *setoids* which contain all the ingredients of quotients as follows,

Definition 2.1. A setoid $(A, \sim) : \mathbf{Set}_1$ is a set [1] $A : \mathbf{Set}$ equipped with an equivalence relation $\sim : A \to A \to \mathbf{Prop}$.

Here we assume **Set** means type, and for any P: **Prop**, it has at most one element, namely we can get proof irrelevance for propositions which has type **Prop**. In Agda, we define a setoid as

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

It contains Carrier for an underlying set, _≈_ [2] for a binary relation on Carrier and a proof that it is an equivalence relation.

We can use setoids to represent quotients, just as we can represent 4 by the pair $(8, 2)$. However there are several problems if we use this approach. Firstly Setoid are different from sets so that we have to redefine all the operations on Set. An interesting problem is how to represent quotients if the base type A is already a setoid. Setoids are also unsafe because we have access to the underlying sets [4]. It is better that if the base type A is of type **Set** then the type of quotient derived from it and its equivalence relation should also be of type **Set**, just as if we divide 8 by 2 we prefer 4 than $(8, 2)$. From mathematical perspective, we can find the structure of the base object is usually the same as the structure of the resultant quotient object. So what could be a quotient type?

Here we firstly describe what quotient types are. Given a setoid $(A, \sim) : \mathbf{Set}_1$, a type $Q : \mathbf{Set}$ is called the quotient type of this setoid, if we can prove it implements the quotient set $A/\sim$, no matter how we construct it.

For instance, since integers represent the result of subtraction of any pairs of natural numbers, we can represent integers by the setoid $(\mathbb{N} \times \mathbb{N}, \sim)$ where $\sim$ is defined as (more details in section 5.1 on page 12)

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

[1] Setoid could be universe polymorphic.

[2] _ mark the spaces for the explicit arguments in non-prefix operators

$\sim$ can also be proved to be an equivalence relation. However the set of integers $\mathbb{Z}$: **Set** can also be constructed as

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

The type $\mathbb{Z}$: **Set** is just the quotient type corresponding to the setoid $(\mathbb{N} \times \mathbb{N}, \sim)$.

Quotient types have uses beyond encoding mathematical quotients. 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 [4], propositions quotiented by logical equivalence relation $\iff$ or 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 $a \sim b \overset{\text{def}}{=} f\,a \equiv f\,b$. Based on this setoid $(A, \sim)$ we can form a quotient. Indeed any type 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 topics of this project.

One feasible approach in current setting of Intensional Type Theory is to manually construct the quotient type as we do in the example of the set of integer above, and prove it is the required quotient type. We can form a quotient using the quotient interfaces introduced in [4], which require the necessary proofs for some type Q : **Set** to be the quotient type of some setoid $(A, \sim)$. These proofs are also the basic properties of quotients, so 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 [25] . Although it has some drawbacks, it is feasible without extending Intensional Type Theory and it provides some convenience in practice. There have been some results on this [4], which I will discuss them in section 5.1.

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 definitionally equal. Nevertheless there are still some problems which we discuss in the literature review.

2.4 Functional extensionality and quotient types

As we have mentioned before, in Intensional Type Theory propositional equality $Id(A, a, b)$ is inhabited if and only if a and b are definitionally equal terms. The Agda definition could be written as

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

However the equality of functions are not only judged by definitions. Functions are usually viewed extensionally as black boxes. If two functions pointwise generate the same outputs for the same inputs, they are equivalent even though their definitions may differ. This is called functional extensionality which is not inhabited [2] in original Intensional Type Theory and can be expressed as following,

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

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

The problem seems easy to solve by just adding a constant $ext : Ext$ to Intensional Type Theory as following codes in Agda

```
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
```

However, postulating something could lead to inconsistence. If we postulate Ext, then theory is no longer adequate, which means it is possible to define irreducible terms. It can be easily verified in Agda through formalising a non-canonical term for a natural number by an eliminator of intensional equality.

Using the eliminator J [3] of the Id A a b :

```
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
```

we can construct an irreducible term of natural number as

```
irr : ℕ
irr = J (ℕ → ℕ) (λ f g P → ℕ) (λ f → 0) (λ x → x) (λ x → x) (Ext refl)
```

With this term, we can construct irreducible terms of any type A by a mapping f : ℕ → A. This will destroy some good features of Intensional Type Theory since it could leads to nonterminating programs.

[3] It is originally used by Martin-Löf [22] and a good explanation could be found in [17]

Altenkirch investigates this issue and gives a solution in [2]. He proposes an extension of Intensional Type Theory by a universe of propositions **Prop** in which all proofs of same propositions are definitionally equal, namely the theory 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 (defining a dependent type by recursion). 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 respecting 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 discuss these in next section.

3 Literature Review

In [8], 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 problems that arise from defining functions on the new type which can be illustrated using a simple example.

Assume the base type is A and the new equivalence relation is E, the new type can be formed as $A//E$.

When we want to define a function $f : A//E \rightarrow Bool$, $f\,a \neq f\,b$ may exists for $a, b : A$ such that $E\,a\,b$. This will lead to inconsistency since $E\,a\,b$ implies a converts to b in Extensional Type Theory, hence the left hand side $f\,a$ can be converted to $f\,b$, namely we get $f\,b \neq f\,b$ which is contradicted with the equality reflection rule.

Therefore a function is said to be well-defined [8] on the new type only if it respects the equivalence relation E, namely

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

We call this *soundness* property in [4].

After the introduction of quotient types, Mendler further investigates this topic from a categorical perspective in [20]. He uses 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 [21].

To add quotient types to Martin-Löf type theory, Hofmann proposes three models for quotient types in his PhD thesis [12]. 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 a specific equivalence relation can be seen as setoids with the basic intensional equality. This is similar to Extensional Type Theory in some sense. 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 [13] 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 [14].

Nogin [21] 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 equality is extensional, we cannot recover the witness of the equality. He suggests including 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 [16] 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 [10] 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 Geuvers [5] also propose 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 the pattern matching introduced in [9]. The main purpose of introducing congruence types is to solve problems in term rewriting systems rather than to implement quotient types.

Barthe and Capretta [**?**] compare different ways to setoids in type theory. The setoid is classified as partial setoid or total setoid depending on whether the equality relation is reflexive or not. They also consider obtain quotients with different kinds of setoids, especially the ones from partial setoids are difficult to define because the lack of reflexivity.

Abbott, Altenkirch et al. [1] provides the basis for programming with quotient datatypes polymorphically based on their works on containers which are datatypes whose instances are collections of objects, such as arrays, trees and so on. Generalising the notion of container, they define quotient containers as the containers quotiented by a collection of isomorphisms on the positions within the containers.

Voevodsky [27] implements quotients in Coq based on a set of axioms of Homotopy Type Theory. It is based on the groupoid model for Intensional Type Theory where isomorphisms are equalities. He firstly implement equivalence class and use it to implement quotients which is an analogy to the construction of quotient sets in set theory.

4 Aims and Objectives of the Project

As we have seen, quotients can enable defining or constructing various kinds of mathematical notions or programming datatypes, thus the introduction of quotient types will be quite beneficial in theorem provers and programming languages based on Type Theory.

The objective of this project is to investigate and explore the ways of implementing quotients in Martin-Löf type theory, especially in intensional variant where type checking always terminates.

The project will be undertaken step by step. Firstly, we should make the basic notions clear, for example what are 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 to work in the 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 [4].

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. A proof of why $\mathbb{R}$ is undefinable is also given in [4].

The development of framework of quotient types in Intensional Type Theory is one of the major objectives. 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. It is better if we could constructed quotients in a general way and the quotient types have useful properties that facilitating programming and reasoning. The correctness of axiomatisation and the consistency of extended Intensional Type Theory require formal proofs.

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

Finally, we will summarise all these works, including background information, literature review, defining quotient types in Intensional Type Theory, the benefits of it and the application of it into a PhD Thesis.

To do this research, we need to review and compare the existing approaches in different implementations of Martin-Löf type theory, and try to determine 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 useful as 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 feedback from the users, and improve our approaches such that they are more applicable and easier to use.

5 Results and Discussion

5.1 Definitions

Currently, we are on the first stage and there is some progression on definable quotients in Intensional Type Theory [4]. 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 cannot define it from the setoid because there are no axiomatised quotient types. We can only prove some type is a quotient type of a given setoid. Therefore the only way to introduce possible quotient types $Q :$ **Set** is to define it by ourselves. With defined Q and $(A, \sim)$, we are required construct some structures of quotients in [4] 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
(n1, n2) ∼ (n3, n4)  =  (n1 + n4) ≡ (n3 ℕ+ n2)
```

Here $\equiv$ is 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[4]

[4] the proof _∼_isEquivalence is omitted here

```
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, suc n] = -suc n
[suc m, suc n] = [m, n]
```

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

```
sound : ∀ {x y} → x ∼ y → [x] ≡ [y]
```

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 [4],

1. *Quotient with a dependent eliminator*

```
record Qu {S : Setoid} (PQ : PreQu S) : Set₁ where
  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
  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
  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 [4].

For integers, it is natural to define a function to choose a representative for each element in ℤ,

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

Now we need to prove ⌜_⌝ is the required embedding function, namely it is the inverse function of [_].

Firstly ⌜_⌝ is left inverse of [_],

```
compl : ∀ {n} → ⌜ [n] ⌝ ∼ n
```

This is called the *complete* property.

Secondly ⌜_⌝ is right inverse of [_],

```
stable : ∀ {n} → [⌜ n ⌝] ≡ n
```

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 {z}
  ;stable = λ z → stable {z}
  }
```

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 translating 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
```

We can benefit from the interaction between setoids and quotient types in a number of ways.

Firstly the setoid definitions are usually 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: the large amount of cases 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,

```
distʳ : _*_ DistributesOverʳ _+_
distʳ (a, b) (c, d) (e, f) = ℕ.dist-lemʳ a b c d e f += ⟨ ℕ.dist-lemʳ b a c d e f ⟩
```

Moreover, as we have constructed the semiring of natural numbers, it is even simpler to use the 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 (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 when 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

```
liftOp1s : (f : Op₁ ℤ₀) → (∀ {a b} → a ∼ b → f a ∼ f b) → Op₁ ℤ
liftOp1s f cong = λ n → [f ⌜ n ⌝]
liftOp2s : (* : Op₂ ℤ₀) → (∀ {a b c d} → a ∼ b → c ∼ d → * a c ∼ * b d) → Op₂ ℤ
liftOp2s _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]
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]
```

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 ⌜_⌝ which are part of the quotients. We can then lift the commutativity of addition and multiplication,

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

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

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

There is no need to use pattern matching, namely prove the propositions inductively. At first, I tried to prove distributivity by case analysis, and I found it is especially difficult and the proof is too long and it looks like (I omit to write the long proof for each clause):

```
distʳ : _*_ DistributesOverʳ _+_
distʳ (+ n) (+ n') (+ n0) = ...
distʳ (+ n) (+ n') (-suc n0) = ...
distʳ (+ n) (-suc n') (+ n0) = ...
distʳ (+ n) (-suc n') (-suc n0) = ...
distʳ (-suc n) (+ n') (+ n0) = ...
distʳ (-suc n) (+ n') (-suc n0) = ...
distʳ (-suc n) (-suc n') (+ n0) = ...
distʳ (-suc n) (-suc n') (-suc n0) = ...
```

Even though it is provable in this way, it is not the best choice to prove something like distributivity by induction.

The simplicity of the short 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. Although all the lemmas may be as much as the long proofs by induction, they are more meaningful and can be reused. 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) → Identity _∼_ ⌜ e ⌝ op → Identity e (liftOp 2 op)
liftAssoc : ∀ {op : Op 2 ℤ₀} (cong : Cong2 op) → Associative _∼_ op →
  Associative (liftOp2safe op cong)
liftMonoid : {op : Op 2 ℤ₀} {e : ℤ} (cong : Cong2 op) → IsMonoid _∼_ op ⌜ e ⌝ →
  IsMonoid _≡_ (liftOp 2 op) e
```

These lift functions for operators and properties can be generalised even further such that they can be applied to all quotients that have similar algebraic structures. 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 $\mathbb{Q}$.

5.2 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
n1 /suc d1 ∼ n2 /suc d2 = n1 * suc d2 ≡ n2 * 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
```

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, the lifted operators and properties which are safer.

5.3 Real numbers

The previous quotient types are all definable in Intensional Type Theory, so 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 [4].

We have several choices to represent real numbers. We choose Cauchy sequences of rational numbers to represent real numbers [6].

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

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 in Appendix.

In set theory we can construct quotient set $\mathbb{R}_0/\sim$. However since real numbers have no normal forms we cannot define the quotient in Intensional Type Theory. Hence the definable quotient definition does not work for it. The undefinability of any type $\mathbb{R}$ which is the quotient type of the setoid $(\mathbb{R}_0, \sim)$ is proved by local continuity [4].

5.4 All epimorphisms are split epimorphisms

In addition we have also prove that classically all epimorphisms are split epimorphisms.

As we have mentioned above, Mendler [20] has investigated quotient types via coequalizers in category theory. We also explain the correspondence in [4]. The coequalizer q which corresponds to the function [_] in our quotient structures is an epimorphism.

According to the definition of 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)
```

In Intensional Type Theory, if the quotient types are undefinable, we can not construct the normalisation function [_], and the proposition is only proved to be true with classical axioms. Therefore it does not make sense for the epimorphism from $\mathbb{R}_0$ to $\mathbb{R}$.

6 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. Given some setoids, the corresponding quotient types are separately defined and then proved to be correct by forming definable quotients structure with the setoids. This approach provides us an alternative choice to define functions and prove propositions. The properties contained in the quotient structure are very helpful in lifting functions and propositions for setoids to quotient types. From the examples we have discussed, comparing manipulating quotient types, it is probably simpler to define functions on setoids and then we can lift functions and properties when they respect the equivalence relation. However it is a little complicated to build the quotients and is only applicable to quotients which are definable in current setting of Intensional Type Theory.

6.1 Future work

In next phase we will focus on the undefinable quotients (e.g. the set of real numbers) and to implement undefinable quotients, extending Intensional Type Theory with axiomatised quotient types is unavoidable. Although the definable quotients are limited, it is still a good guide when axiomatising the quotient types. The other quotient structures could also be applied to axiomatised quotient types. To axiomatise quotient types we can refer to Martin Hofmann's work in [12]. we can implement and then extend his work in Agda. We need to axiomatise the formation, introduction, elimination rules for quotient types. After that we should be able to define lifting function for quotient types. The properties of quotient types should be proved and we should define some typical examples of undefinable quotients and also definable ones. Moreover the other extensional concepts such as proof irrelevance, functional extensionality and propositional extensionality should be present.

Additional future work could be to give a detailed proof of the conservativity of Extensional Type Theory over Intensional Type Theory with axiomatised quotient types extending the work in [14]. It means that the same types are inhabited if they make sense in that type theory. To prove it, as he said, it is enough that the type former of quotient types admits an action on propositional isomorphisms. He also mentioned that he has shown this in [12] by adding quotient types and a universe, but the approach to axiomatise quotient types is different. The proof of conservativity is non-constructive because of the utilization of set-theoretic quotienting and choice of representatives.

One area work is to extend the setoid model as metatheory constructed by Altenkirch in [2] to quotient types, and to find an approach to extend Intensional Type

Theory without losing nice features such as termination and decidable type checking. The basic idea of setoid model is to interpret all types as types with equivalent relation, and since we have the proof-irrelevant universe **Prop**, the identity proof is unique. The extention of Intensional Type Theory with proof-irrelevant propositions and η-rules as metatheory should be proved decidable, consistent and adequate. Decidability of the type theory can be proved if definitional equality is decidable as we discussed above. Consistency means there is no contradiction in a type theory, and it should be provable using strong normalisation and Church-Rosser theorem. Adequacy which has been mentioned above can be proved if there are no closed terms of type $\mathbb{N}$ that are not reducible to numerals. As in [2], the model employed is called *categories with families* which is introduced by Dybjer and Hofmann [11, 15]. To introduce the objective type theory, Altenkirch uses an approach that is different to commonly used syntactical approach. He define a model inside the metatheory and verify it is also decidable, consistent and adequate. We should follow a similar approach to extend the model with quotient types. Recently Agda has new a new feature called *Dependent irrelevant function types* and it allows us to define the eliminator for the squash type and it should be helpful for us to implement the proof-irrelevant propositions in Agda.

Some other models, such as groupoid model could also be investigated if we do not have proof irrelevance. Voevodsky's construction of quotients in Homotopy Type Theory can be found in[27]. Since in Homotopy Type Theory, two isomorphic objects are equal, the implementation of equivalent classes should be possible and then the quotient types are natural to define. However there are some problems of the definition: since the encoding is impredicative, the size problem will be present; the properties of quotients are not provable. Nevertheless we can try to learn from his construction and find out how it can fit into our work.

References

[1] Michael Abott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Constructing polymorphic programs with Quotient Types. In *7th International Conference on Mathematics of Program Construction (MPC 2004)*, 2004.

[2] Thorsten Altenkirch. Extensional Equality in Intensional Type Theory. In *14th Symposium on Logic in Computer Science*, pages 412 – 420, 1999.

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

[4] Thorsten Altenkirch, Thomas Anberrée, and Nuo Li. Definable Quotients in Type Theory. 2011.

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

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

[7] 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.

[8] 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.

[9] Thierry Coquand. Pattern Matching with Dependent Types. In *Types for Proofs and Programs*, 1992.

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

[11] Peter Dybjer. Internal Type Theory. In *Lecture Notes in Computer Science*, pages 120–134. Springer, 1996.

[12] Martin Hofmann. *Extensional concepts in Intensional Type Theory.* PhD thesis, School of Informatics., 1995.

[13] 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.

[14] 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.

[15] Martin Hofmann. Syntax and Semantics of Dependent Types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.

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

[17] Nicolai Kraus. Equality in the Dependently Typed Lambda Calculus: An Introduction to Homotopy Type Theory. `http://www.cs.nott.ac.uk/~ngk/karlsruhe.pdf`, October 2011.

[18] Per Martin-Löf. A Theory of Types. Technical report, University of Stockholm, 1971.

[19] 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.

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

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

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

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

[24] Ulf Norell. Dependently typed programming in Agda. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.

[25] Li Nuo. Representing numbers in Agda. 2010.

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

[27] Vladimir Voevodsky. Generalities on hSet - Coq library hSet. `http://www.math.ias.edu/~vladimir/Foundations_library/hSet.html`.

A Appendix

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


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


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


dis2ˡ : ∀ a b c d e f → a * (b + c) + d * (e + f) ≡ (a * b + a * c) + (d * e + d * f)
dis2ˡ a b c d e f = distˡ a b c += distˡ d e f
ex : ∀ a b c → a + (b + c) ≡ b + (a + c)
ex a b c = ⟨ +-assoc a b c ⟩ >≡< +-comm a b ⋆+ c >≡< +-assoc b a c
exchange₃ : ∀ m n p q → (m + n) + (p + q) ≡ (m + p) + (n + q)
exchange₃ m n p q = +-assoc m n (p + q) >≡<
  m +⋆ (ex n p q) >≡<
  ⟨ +-assoc m p (n + q) ⟩
dist-lemˡ : ∀ a b c d e f → a * (c + e) + b * (d + f) ≡ (a * c + b * d) + (a * e + b * f)
dist-lemˡ a b c d e f = dis2ˡ a c e b d f >≡< exchange₃ (a * c) (a * e) (b * d) (b * f)
distʳ : _*_ DistributesOverʳ _+_
distʳ (a, b) (c, d) (e, f) = ℕ.dist-lemʳ a b c d e f += ⟨ ℕ.dist-lemʳ b a c d e f ⟩
distʳ : _*_ DistributesOverʳ _+_
distʳ a b c = sound $ ℤ₀.*-cong (compl {⌜ b ⌝ ℤ₀+ ⌜ c ⌝}) zrefl >∼< ℤ₀.distʳ ⌜ a ⌝ ⌜ b ⌝ ⌜ c ⌝ >∼< ℤ₀.+-cong compl' compl'


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)


liftId : ∀ {op : Op 2 ℤ₀} (e : ℤ) → 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) → 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)
  }
```

```
[_] : 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)
```

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