First Year PhD Annual Report

LI Nuo

University of Nottingham

Table of Contents

Abstract. 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 type is formalised by a given type with one of its equivalence relations. However in intensional Type Theory the quotient type is still unavailable. The introduction of quotient type will enable us to define real numbers, functionals, and some other sets which are not definable in current implementations of intensional Type Theory(), such as Agda which is also a theorem prover. Also we can still benefit from the interaction of types within some quotients without using quotient types. Because some base types are simpler to deal with or have better features while what we want to use is the new type, for instance the integers represented by a pair of natural numbers is easier to handled compared to the normal form integers. We undertake this project to conduct a research on the implementation of quotients in Agda. This report aims at introducing quotient types and investigate some related work on this topic. I will also give some explanations to some results which has been done by Altenkirch, Thomas and me in [2] and show some instances in this report as a complement to it.

1 Introduction

In mathematics, a quotient represents the result of division. The notion of quotient is extended to other abstracted branches of mathematics. For example, we have quotient set, quotient group, quotient space, quotient category etc. They all define similar operations.

In everyday life, we can find quotient in our daily life. When you take a picture by a digital camera, the real scence is divided into each pixel on the picture, or we can say the real scence within a certain area is identified. The digital picture is just the quotient. Since things are equated with respect to certain rules, quotient is usually mentioned with information hiding or losing. In computer science, we also see quotient everywhere. Users are only concerned the extensional use of softwares rather than the intensional implementation of them, therefore softwares doing the same tasks are the same to them even they are programmed in two different languages. When doing programming, different classes implemented the same *interface* are extensionally equal.

In this report, I will mainly discuss quotient type in type theory, which can be seen as the interpretation of quotient set in set theory although they are fundamentally different. Since set theory is more familiar to some of you we will start from introducing qotient sets.

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 $\sim$,

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

Dividing a natural number a, by another number b, can be inductively defined. For example, natural numbers divided by 3 can be simulated by the following process. Assume 0 divided by any positive number gives 0 and remainder is 0. the quotient number can be obtained from the cardinality of the last equivalence class of set $\underline{a}$ under congruence modulo 3, that is the equivalence class whose elements share the remainder 2. The remainder can be obtained by adding one to the remainder of the previous number in the set of $0, 1, 2$.

There are many mathematics notions can be constructed in the form of quotient sets. Some are more natural to construct by quotients, such as integers modulo some number and rational numbers as fractions. Integers modulo some number n is the set of equivalence classes of all integers $\mathbb{Z}$ under the equivalence relation which equates two elements with the same remainders after divided by n. Pairs of integers $\mathbb{Z} \times \mathbb{Z}$ can be used to represent rational numbers, but eahc rational number can be represented by infinitely many internally different pairs. In other words, the set of rational numbers is the set of equivalence classes of $\mathbb{Z} \times \mathbb{Z}$ under the equivalence relation that equates different forms of one rational number. Some other sets like integers and real numbers are also intrinsically quotients. Integers can be interpreted as pairs of natural numbers $\mathbb{N} \times \mathbb{N}$ and real numbers can be represented by cauchy sequences of rational numbers. They can be encoded in Type Theory as we will discussed in detail later.

1.2 Type Theory

The theory of types was first introduced by Russell [20] as an alternative to naive set theory. After that, mathematicians or computer scientists have developed a number of variation of type theory. The type theory in this discourse is the one developed by Per Martin-Löf [13, 14] and it 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[18].

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 term. However in extensional Type Theory, they are not distinguished so that definitional equality is undecidable. Since type checking only depends on definitional equality[1], it is decidable and terminates in intensional Type Theory but not in extensional Type Theory.

Type theory can also serve as a programming language in which the evaluation of well-typed program always terminates [17]. 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, Because of the identification of types and propositions, it is not only a programming language but also a theorem prover. Moreover we are able to verify our Agda programs in the itself. It has a bundle of good features like pattern matching, unicode input, implicit aruguments etc [5]. Since this project is based on Martin-Löf type theory, it is ideal to implement our definitions and verify propositions in Agda.

Type theory is fundamentally different to set theory. There is no set-theoretical set and we have to declare types before we have any object because objects or terms cannot live without their types.

1.3 Quotient Types

In intensional Type Theory, many notions from set theory and propositional logic can be reinterpreted easily. For instances, the product of sets can be formalised by $\Sigma - Type$ and the functions can be formalised by $\Pi - Type$ [18]. However the reinterpretation of quotients in Type Theory is still a problematic issue and quotient types are still unavailable.

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

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

We can use setoids to represent quotients but they are not sets. If we use the setoid $(\mathbb{N} \times \mathbb{N}, \sim)$ for integers, we have to redefine set-based operations and it is unsafe [2]. Moreover to define quotients based on this setoid, such as rational numbers, is impossible. Therefore it is not a good solution.

Quotient types should enable users to implement quotients in intensional Type Theory which means the quotient types based on the setoid should have type **Set**. Moreover it should be universe polymorphic.

In this report, I will use the symbol $A/\sim$ for the quotient based on a given setoid $(A, \sim)$. To make the difference between setoids and quotient types clear, we use an analogy, $8 \div 2 = 4$. The number 4 is the quotient because $4 \times 2 = 8$, and we cannot recover the dividend and the divisor from the quotient 4 or manipulate 8 or 2 separately. Similarly, setoids contain pairs of dividend and divisor, but quotient sets do not include all the information from the setoids. Futhermore one set can be the quotient sets of several different setoids.

intensional Type Theory has to be extended if we want to define quotient types. Alternatively, some quotients are definable such that we can construct it from scratch and prove it is the quotient based on a given setoid [2]. For instances in [19], the normal form integers, reducible rational numbers are definable and proved to construct quotients with respect to the corresponding setoids, such

that we can treat them as the quotient types. The quotient interfaces introduced in [2] do not provide the access to the underlying setoids but include a set of properties which serve as not only proofs but also tools. I will explain the quotient interface and discuss some examples later. The advantages of this idea are, it is feasible in current setting of intensional Type Theory, and we achieve some convenience from constructing quotients. The disadvantages are, it only works for the quotient types which are definable in intensional Type Theory and the quotient types and functions are constructed manually rather than derived automatically. Hence the lifting of functions and predicates is a bit complicated as it requires proofs that these functions or predicates respect the equivalence relation[10]. Moreover, if the quotient sets are not definable, for instance the set of real numbers $\mathbb{R}$, even though $\mathbb{R}$ can be seen as the quotient set of the Cauchy sequences of rational numbers $\mathbb{Q}$ and the equivalence relation that two sequences converge to zero.

Quotient types is not only a tool to implement quotients in mathematics. It is based on Martin-Löf type theory, therefore some notions in Type Theory or in programming languages are actually quotients. For example partiality monad divided by a weak similarity ignoring finite delays [2], propositions divided by $\iff$ and the set of extensionally equal functions. Also set-theoretical finite sets can be implemented as the quotient of lists in Type Theory.

Furthermore from any given 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 $a \sim b \stackrel{\text{def}}{=} f\, a \equiv f\, b$. From this setoid we can form a quotient.

Actually all types can be seen as quotient types. The types without specified equivalence relation can be seen as the quotient of itself by the trivial equivalence relation, namely the definitional equality.

1.4 The relation between equality and quotient types

As we have discussed before, we distinguish definitional equality with propositional equality in intensional Type Theory. The equivalence relation of quotients is propositional equality which is non-trivial despite the case that it is the same with definitional equality. Hence the type-checking which depends on definitional equality does not respect the equivalence relation.

There are also two different propositions expressing the equailty between two elements in Type Theory [17]. Both of them require the types of two elements are definitionally equal. One is intensional equality written as $Id(A, a, b)$ and it is inhabited only we have a proof showing a and b are definitionally equal. The other is extensional equality written as $Eq(A, a, b)$, the elements do not depend on an element of A and the largest difference is if $Eq(A, a, b)$ is inhabited, then a converts to b and vice versa. The latter one will make type-checking undecidable so we usually use the first one which is available in intensional Type Theory. For example in Agda, it is redefined as $a \equiv b$ with the type A implicitly, and it has an unique element $refl$.

Intensional equality is enough for many quotients. However if we want quotient types corresponding to the sets of functions, the intensional equalities of

functions are not inhabited [1]. This is because equated functions can have different definitions and they are possibly to reduced to different normal forms. Hence in intensional Type Theory where we can form propositional equality from definitional equality, the propositional equality of functions is not inhabited and it requires the functional extensionality i.e. the proposition that if two functions are pointwise equal then they are propositional equal which is not inhabited in original intensional Type Theory. It can be expressed as follow,

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

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

However it is not an easy problem to extending intensional Type Theory. If we just 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 a eliminator of intensional equality. Because of that, the equality and type checking become undecidable. Fortunately Altenkirch investigates this issue and gives a solution in [1]. He proposed an extension of intensional Type Theory by a universe of propositions **Prop**. All proofs of a the same proposition are definitionally equal, namely it is proof irrelevant. 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.

Most of the topics concerning quotient types are closely related to equality. One of the main issues of quotient types 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 the extensional Type Theory as we will discussed later.

1.5 Literature Review

In [6], Mendler et al. have considered building new types from a given type by 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 quotient types. They also discuss the problems arised when defining functions on the new type. We can illustrate this problem with a simple example. Assume the base set is A and the new equality relation is E, then the new type can be represented by $A//E$. If we want to define a function $f : A//E \to Bool$, Assume we have two different elements in A, $a, b\colon A$ such that $E\,a\,b$ but $f\,a \neq f\,b$, then it becomes inconsistent since $E\,a\,b$ implies a converts to b, then $f\,a = f\,b$ which contradicts with the assumption $f\,a \neq f\,b$. Therefore even in extensional Type Theory, the definition of functions on the quotient types are not so simple. The functions have to respect the equivalence relation, namely

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

then f is well-defined on the new type. We call it *sound* in [2] and this project.

After the introduction of quotient types, Mendler futher investigates this topics from a categorical perspective in [15]. 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.

Hofmann proposed in his PhD thesis [9] three models for quotient types. 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 trivial reflection equality. It is similar to NuPRL. While in [10] 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 [11].

Nogin [16] considers a modular approach to axiomatizing the same quotient types also in NuPRL. He also discusses a few complicated problems about quotient types despite the ease of constructing new types from base types. For example, since the the equality is extensional, we can not recover the witness of the equality. So he suggests to include more axioms to conceptualise quotients. He decomposes the concept of quotient type into several more primitive concepts such that the quotient types can be formalised based on these concepts and can be handled much simpler.

Homeier [12] 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 [8] 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. Normalised types are proper subsets of quotient types,

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

However not all quotient types have normal forms. Therefore it only solves part of the problem.

Similarly, Barthe and Geunvers [3] also proposes *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. The congruence types can be treated as an alternative to pattern matching introduced in [7]. Hence it aims at solving problems in term rewriting systems rather than simply implementing quotient types.

2 Aims and Objectives of the Project

The objective of this project is to investigate and explore the ways to implement quotient types in Type Theory, especially in intensional one since type checking always terminates. As we have seen quotients are quite useful in implementing mathematical objects and programming datatypes, it will be very helpful if we can define quotients in thoerem prover like Agda. Also to implement some other undefinable quotients such as Real numbers, it is an unvoidable issue to implement the idea of quotient.

The current aim is to implement some definable quotients, use the quotient interfaces for them and study their benefits. We also need to do research on the different definitions of quotients.

Next we need to investigate undefinable quotients such as the real numbers and partiality monads and prove why they are undefinable. The key different characters between definable and undefinable quotients will be studied.

3 Theoretical Methods

Part of our work is implemented in Agda, which is a dependent typed programming language and mainly used as a theorem prover.

TO DO: I have to write some other things here. Agda has been introduced before It has dependent type so that we can use Curry-Howard correspondence between types and propositions. Since propositions can be represented as types, its type checker can verify the proof.

In this project the work will be proved in Agda and also verified in Agda since it is a good choice of intensional Type Theory.

4 Results and Discussion

4.1 Definitions

Currently, we have done some work on the framework of quotient. We have submitted a paper [2] for APLAS 2011. It is about the definable quotients and some undefinable quotients. Here we only talk about the quotient set, but it is universal polymorphic.

To associate a setoid $(A, \sim)$ with a set Q, we have several definitions as in [2], I will not present it again but explain some ideas behind them.

Given a setoid $(A, \sim)$, we denote the set of equivalence classes as $A/\sim$ and the normalisation function is $[\cdot]_\sim : A \to A/\sim$, assigning each elements to the set it is belonging to. Hence we have

Proposition 2. $\forall\, a\,, b : A, a \sim b \iff [a]_\sim = [b]_\sim$

And the normalisation function is surjective, hence we assume classically,

Proposition 3. $\forall\, e :\ A/\sim, \exists\, a : A, [a]_\sim = e$

Namely, the normalisation function is split,

Proposition 4. $\exists\, s :\ \mathrm{A}/\sim \to A,\ [\,\cdot\,]_\sim \circ s = 1_{\mathrm{A}/\sim}$

Since then

$$[\,\cdot\,]_\sim \circ (s \circ [\,\cdot\,]_\sim) = [\,\cdot\,]_\sim \circ 1_{\mathrm{A}/\sim}$$

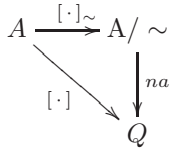
And with Proposition 2, we can prove that

Proposition 5. $\forall\, a : A, (s \circ [\,\cdot\,]_\sim)\, a \sim a$

Some of them are only classically true. However, we worked in intensional Type Theory which is constructive. What we do is to associate a given set Q to the setoid or the quotient set. Given a function $[\,\cdot\,]\colon A \to Q$,

$$\mathrm{sound}\colon (a, b : A) \to a \sim b \to [a] = [b]$$

is a property which means that from the images of the elements from the same equivalence class are identical, namely $[\,\cdot\,]$ respects the equivalence relation. It is also equivalent to say that there is a naming function, $na :\ \mathrm{A}/\sim \to Q$, such that the following diagram commutes,



And na can be constructed as $[\,\cdot\,] \circ s$. We can prove this diagram is commute as,

$$na \circ [\,\cdot\,]_\sim = [\,\cdot\,] \circ s \circ [\,\cdot\,]_\sim$$

Apply Proposition 5 and Proposition 2, we know,

$$\forall\, a : A, (na \circ [\,\cdot\,]_\sim)\, a = ([\,\cdot\,] \circ s \circ [\,\cdot\,]_\sim)\, a = [\,\cdot\,] \circ ((s \circ [\,\cdot\,]_\sim)\, a) = [\,\cdot\,]\, a$$

Extensionally, we proved that the diagram commute.

However, with this property, we cannot confirm Q is the required quotient set, we only construct a prequotient. To complete a quotient, we require one eliminator for every $B\colon Q \to \mathbf{Set}$,

$$\begin{aligned}\mathrm{qelim}_B\colon\ & (f\colon (a : A) \to B\,[a]) \\ & \to ((p : a \sim b) \to f\, a \simeq_{\mathrm{sound}\ p} f\, b) \\ & \to ((q : Q) \to B\, q)\end{aligned}$$

such that qelim-β: $\mathrm{qelim}_B\, f\, p\, [a] \equiv f a$.

With this eliminator we can lift a function which takes in $a : A$ but the result is dependent on the $[a] : Q$ and identifies more than $[\,\cdot\,]$, namely for the elements in the same equivalence class by $[\,\cdot\,]$, the result produced by f is the same. Combining with this function,

However this is not a exact quotient, since it is unnecessary for na function to assign one *name* $(q : Q)$ to each equivalence class $(e : \mathrm{A}/\sim)$. It is very inefficient to define a too general quotient. Therefore we need a property to make a quotient exact,

$$\mathrm{exact} : \forall\, a\, b : A\,,\, [a] = [b] \to a \sim b$$

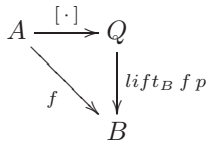
equivalently, we have the property that na is injective,

$$\begin{aligned}\forall\, e\, f : \mathrm{A}/\sim, na\, e = na\, f \Rightarrow ([\,\cdot\,] \circ s)\, e = ([\,\cdot\,] \circ s)\, f \\ \Rightarrow [s\, e] = [s\, f] \Rightarrow s\, e \sim s\, f \Rightarrow [s\, e]_\sim = [s\, f]_\sim \Rightarrow e = f\end{aligned}$$

The alternative definition of quotient with non-dependent eliminator introduced in [9], and consists of,

$$\begin{aligned}&\mathrm{lift}_B \colon (f \colon A \to B) \to (\forall a, b \cdot a \sim b \to f\, a \equiv f\, b) \to (Q \to B)\\ &\text{lift-}\beta \colon \mathrm{lift}_B\, f\, p\, [a] \equiv f a\end{aligned}$$

for any $B \colon \mathbf{Set}$, which can lift a function f which respects the equivalence relation and the following diagram commute with respect to lift-β,



In this definition we also need an introduction principle if B in the dependent eliminator is a predicate on Q,

$$\mathrm{qind}_P \colon ((a : A) \to P\,[a]) \to ((q : Q) \to P\, q)$$

The quotient with dependent eliminator and the one with non-dependent eliminator are actually equivalent. We prove this by formalise one by another in Agda. It is quite trivial to generate the non-dependent version from dependent version since lift_B and qind_P are both special cases of the dependent eliminator. However to recover dependent eliminator, it is a little complicated. We need a function *indep* to transform the dependent $a : A \to B\,[a]$ into the non-dependent $A \to \Sigma\, Q\, B$ which is defined as $\mathit{indep}\, f\, a \mapsto [a]\,, f\, a$. Then we can use non-dependent eliminator to lift $\mathit{indep}\, f$ and the projection of the second component is the same as dependent function. You can check the detailed Agda proof in the Appendix.

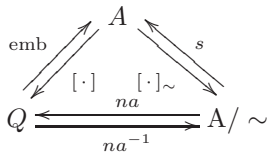
When the quotient type is definable and we want the target type Q is just the quotient type, which means

$$Q \cong \mathrm{A}/\sim$$

Therefore, to constructively define isomorphism in intensional Type Theory, we not only need na, but also the inverse function of it. So the definable quotients in [2] is the prequotient with

$$\begin{aligned} \mathrm{emb} &: Q \to A \\ \mathrm{complete} &: (a : A) \to \mathrm{emb}\,[a] \sim a \\ \mathrm{stable} &: (q : Q) \to [\mathrm{emb}\ q] \equiv q \end{aligned}$$

emb is the embedding function which choose one representative element for each equivalence class. Hence the following diagram needs to commute,



Such that $na^{-1} = [\,\cdot\,]_\sim \circ \mathrm{emb}$ is the inverse function of na.

$$\begin{aligned} & na^{-1} \circ na = 1_{\mathrm{A}/\sim} \Rightarrow [\,\cdot\,]_\sim \circ \mathrm{emb} \circ [\,\cdot\,] \circ s = 1_{\mathrm{A}/\sim} \\ & \Rightarrow \forall a \colon A, a \sim (s \circ [\,\cdot\,]_\sim)\, a \sim (s \circ 1_{\mathrm{A}/\sim} \circ [\,\cdot\,]_\sim)\, a \\ & \sim (s \circ ([\,\cdot\,]_\sim \circ \mathrm{emb} \circ [\,\cdot\,] \circ s) \circ [\,\cdot\,]_\sim)\, a \sim \mathrm{emb}\,[a] \end{aligned}$$

So we need this property called completeness which ensures the correctness of emb. Also,

$$na \circ na^{-1} = 1_Q \Rightarrow [\,\cdot\,] \circ s \circ [\,\cdot\,]_\sim \circ \mathrm{emb} = [\,\cdot\,] \circ \mathrm{emb} = 1_Q \Rightarrow \forall\, q : Q, [\mathrm{emb}\ q] = q$$

is needed called the stable property which ensures $[\,\cdot\,]$ is surjective, Hence it is *normalisation* function and the Q is the quotient type without redundance.

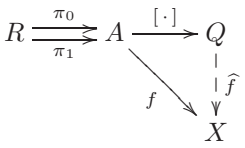
With these two properties, we can conclude that $[\,\cdot\,]_\sim \circ \mathrm{emb}$ is the inverse function of na , hence Q is isomorphic to $\mathrm{A}/\sim$. We can use it as the quotient type.

In category theory, coequalizers are the generalization of quotients. We assume

$R = \Sigma a, b : A, a \sim b$ are the pairs of equivalent elements in A

$\pi_0\,, \pi_1 \colon R \to A$ are the projection functions for R

$[\,\cdot\,] \colon A \to Q$ satisfies that sound: $\forall\, a, b \colon A, a \sim b \to [a] = [b]$



Since

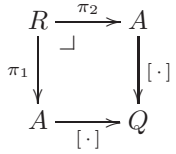
1. $(Q\,, [\,\cdot\,])$ fulfils that $[\,\cdot\,] \circ \pi_0 = [\,\cdot\,] \circ \pi_1$, we can acquire this from applying the $\pi_0\, r, \pi_1\, r$ for all r to sound.
2. Given any $(X, f\colon A \to X)$, there exists a unique $\hat{f}$, such that the diagram above commutes. From the definition of quotients, we can use the eliminator to lift f , namely $\hat{f} = \text{lift}\, f$, and the β-law simply implies the diagram commutes. The uniqueness can be proved as follows

$$\forall\, g : Q \to X, g \circ [\,\cdot\,] = f \Rightarrow \forall\, a : A, g\,[a] = f\, a = \text{lift}\, f\, q\,[a] \Rightarrow g = \text{lift}\, f\, q$$

These two parts proved from quotients exactly define a coequalizer. Also we can prove $[\,\cdot\,]$ is an epimorphism

$$\begin{aligned}&\forall\, g_1, g_2 : Q \to Z, g_1 \circ [\,\cdot\,] = g_2 \circ [\,\cdot\,]\\ &\Rightarrow \forall\, q : Q, g_1\, q = \text{lift}\,(g_1 \circ [\,\cdot\,])\, q = \text{lift}\,(g_2 \circ [\,\cdot\,])\, q = g_2\, q \Rightarrow g_1 = g_2\end{aligned}$$

Also the exact quotient is equivalent to the exact coequalizer,



1. This diagram commutes
   $(\forall\, r : R, \pi_1\, r \sim \pi_2\, r \Rightarrow [\pi_1\, r] = [\pi_2\, r]) \Rightarrow [\,\cdot\,] \circ \pi_1 = [\,\cdot\,] \circ \pi_2$
2.

$$\begin{aligned}&\forall (Z, z_1\colon Z \to A, z_2\colon Z \to A),\, [\,\cdot\,] \circ z_1 = [\,\cdot\,] \circ z_2\\ &\Rightarrow (\exists\, u : Z \to R, \pi_1 \circ u = z_1 \wedge \pi_2 \circ u = z\\ &\wedge\ \forall\, u'\colon Z \to R, \pi_1 \circ u' = z_1 \wedge \pi_2 \circ u' = z_2\\ &\Rightarrow u = u')\end{aligned}$$

   We can construct the unique function as $u\, x \mapsto z_1\, x\,, z_2\, x$, but we need to prove $z_1\, x \sim z_2\, x$ from exact property of quotient,
   $[\,\cdot\,] \circ z_1 = [\,\cdot\,] \circ z_2 \Rightarrow \forall\, x : Z, ([z_1\, x] = [z_2\, x] \Rightarrow z_1\, x \sim z_2\, x)$
   u is the function which makes the diagram commutes,

$$\forall\, x : Z, (\pi_1 \circ u)\, x = z_1\, x$$

$$\forall\, x : Z, (\pi_2 \circ u)\, x = z_2\, x$$

u is unique,

$$\begin{aligned}&\forall\, u' \colon Z \to R, \pi_1 \circ u' = z_1 \wedge \pi_2 \circ u' = z_2\\ &\Rightarrow \forall\, x : Z, u'\, x = z_1\, x\,, z_2\, x = u\, x \Rightarrow u' = u\end{aligned}$$

5 Examples

We have already define the basic requirements to create quotients in intensional Type Theory, I will then present some concrete examples in [19] to illustrate these ideas. They are implemented in Agda.

5.1 Integers

All the result of subtraction between natural numbers are integers. Therefore it is naturally to define a pair of natural numbers to represent integers. Hence the base type of the quotient is

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

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

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

since the pair of integers represent the same result of subtraction, they define the same integer. Hence we can define an equivalence relation for $\mathbb{N} \times \mathbb{N}$ as

$$(n_1, n_2) \sim (n_3, n_4) = n_1 + n_4 \equiv n_3 + n_2$$

Here $\equiv$ is the propositional equality. so that the $\mathbb{Z}_0/\sim$ is the quotient integer. Integer is also definable in intensional Type Theory as $\mathbb{N} + \mathbb{N}$ where we define two constructors

$(n \colon \mathbb{N}) \Rightarrow +\, n \colon \mathbb{Z}$

$(n \colon \mathbb{N}) \Rightarrow -\, \mathrm{suc}\ n \colon \mathbb{Z}$

Firstly to construct the prequotient based on the setoid $(\mathbb{Z}_0, \sim)$, we need to define the $[\,\cdot\,] \colon \mathbb{Z}_0 \to \mathbb{Z}$ as

$$\begin{aligned}[(a, 0)] &= +\, a\\ [(0, \mathrm{suc}\ b)] &= -\, \mathrm{suc}\ b\\ [(\mathrm{suc}\ a, \mathrm{suc}\ b)] &= [(a, b)]\end{aligned}$$

and prove *sound*. Then we define emb function and prove all the required properties for definable quotients

$$\begin{aligned} \text{emb}\,(+a) &= (a, 0) \\ \text{emb}(-\,\text{suc}\;b) &= (0, b+1) \end{aligned}$$

We have done these in Agda [19]. The quotients here are not just something relate the setoid with the quotient type, we use lift functions to define functions trivially and use properties to transform the proof term for the setoid to the quotient type. For instance, the addition of the setoid is defined as

$$(a, b) +_0 (a', b') = (a + a', b + b')$$

We can then define use the eliminator to lift the operator, or just define a lift function for binary operators,

$$\text{lift} * z_1\, z_2 = [\text{emb}\; z\mathit{text}_1 * \text{emb}\; z_2]$$

or a more general lift function for n-ary operators,

$$\begin{aligned} \text{lift}'\; 0\, op &= [op] \\ \text{lift}'\,(\text{suc}\; n)\, op &= \lambda\, x \to \text{lift}'\, n\,(op\,(\text{emb}\, x)) \end{aligned}$$

Then we don't need to define the addition of integers by several cases.

$$+ = \text{lift} +_0$$

If we lift the operators in this way, we have to prove it respects the equivalence relation later. The main benefits from the quotients arise in proving properties. Because for normal form integers, we have two cases for each argument. The number of cases will expand exponentially if we can not combine cases. The proof of distributivity of multiplication over addition is so cumbersome that it is hard to write and read. However, we could lift the proof for the setoid integers so that we could prove it in one case. This convenience is due to the simplicity of the proof for the setoid $(\mathbb{Z}_0, \sim)$.

5.2 Rational numbers

The quotients of rational numbers is more natural to understand and the normalisation function is also commonly used in regular mathematics. Generally we can use a pair of integers to represent rational numbers. However, it is complicated to exclude 0 in the denominator. For simplicity, we just use one integer for

numerator and one natural number for *denominator-1* to represent a rational number to avoid the invalid cases from construction.

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

The equivalence relation is

$$(n_1, d_1) \sim (n_2, d_2) = n_1 \times (d_2 + 1) \equiv n_2 \times (d_1 + 1)$$

The normal form of rational numbers can just be defined by adding a condition that the numerator and denominator are coprime.

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

Since there are a set of *gcd* (great common divisor) functions in Agda, it is possible to define the normalisation functions (See Appendix). emb function can be trivially defined by forgetting coprime proof.

5.3 Real numbers as cauchy sequences

We can represent real numbers as cauchy sequences of rational numbers [4].

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

And we define the equivalence relation of two sequences by the proposition that their pointwise difference converges to 0.

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

Then $\mathbb{R}_0/\sim$ is the quotient set of real numbers. However it is undefinable because real numbers have no normal forms. Therefore we cannot use the definable quotient interface for it. The undefinability is proved in [2]. Nevertheless, we could easily embedding rational numbers as the cauchy sequences of all the same rational numbers. But for irrational numbers, there is no such an uniform way to generate a sequences.

6 Conclusion

Currently we investigate the possible quotient definitions in intensional Type Theory and present some examples and benefits from the definable quotients. For definable quotients, it provides an alternative choice to define functions or prove propositions which reuses things and could be simpler in most cases. However, to solve the problems arose from undefinable quotients, a new type former may be unavoidable.

TO DO: future extension.

One of the future work is to investigate the conservativity of intensional Type Theory and quotient types over extensional Type Theory. Another one

is to extend the work in [1] and find an approach to extend intensional Type Theory without losing nice features, termination and decidable type checking.

If we axiomize quotient types in intensional Type Theory, then every type can be seen as quotient type, when the default equivalence relation is just reflection equality.

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, Thomas Anberrée, and Nuo Li. Definable quotients in type theory. 2011.
3. Gilles Barthe and Herman Geuvers. Congruence types. In *Proceedings of CSL'95*, pages 36–51. Springer-Verlag, 1996.
4. Errett Bishop and Douglas Bridges. *Constructive Analysis*. Springer, New York, 1985.
5. 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.
6. 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.
7. Thierry Coquand. Pattern matching with dependent types. In *Types for Proofs and Programs*, 1992.
8. Pierre Courtieu. **Normalized types**. In *Proceedings of CSL2001*, volume 2142 of *Lecture Notes in Computer Science*, 2001.
9. Martin Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, School of Informatics., 1995.
10. 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.
11. 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.
12. Peter V. Homeier. Quotient types. In *In TPHOLs 2001: Supplemental Proceedings*, page 0046, 2001.
13. Per Martin-Löf. A theory of types. Technical report, University of Stockholm, 1971.
14. Per Martin-Lf. 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.
15. N.P. Mendler. Quotient types via coequalizers in martin-lof type theory. In *Proceedings of the Logical Frameworks Workshop*, pages 349–361, 1990.
16. Aleksey Nogin. Quotient types: A modular approach. In *ITU-T Recommendation H.324*, pages 263–280. Springer-Verlag, 2002.
17. Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory: an introduction*. Clarendon Press, New York, NY, USA, 1990.
18. Bengt Nordström, Kent Petersson, and Jan M. Smith. volume 5, chapter Martin-Löf's type theory. Oxford University Press, 10 2000.
19. Li Nuo. Representing numbers in agda. 2010.
20. Bertrand Russell. *The Principles of Mathematics*. Cambridge University Press, Cambridge, 1903.