Definable quotients in intensional type theory

Li Nuo

June 30, 2014

Abstract

In Martin-Löf type theory, given a setoid i.e. a set with an equivalence relation on it, a corresponding quotient set is expected. However since quotient types is not available in intensional type theory, it is not derivable straightforwardly . In this paper, we consider the case of quotient sets which are *definable* via a normalisation function. We define a number of algebraic structure of quotients which corresponds to coequalizer in category theory. In some cases, e.g. the definable quotients of integers and rational numbers, the setoid representation is simpler to reason about but the set one is better for display. The conversions between them provide us a flexible approach to utilize the quotients such that we can benefit from both representations.

1 Introduction

In intensional type theory [8], quotient types are unavailable and we use setoids [1] instead. Setoids are just sets together with an equivalence relation. However, the disadvantage of using setoids is that we have to define the same operations and types again on setoids e.g. we need lists as an operation on setoids, and it will also result in a lot of non-canonical problems. Moreover, setoids are not safe in the sense that any consumer of a setoid may access the underlying representation. Actually in many cases the quotient sets are themselves definable, and can be related with setoids via a normalisation function. This paper mainly investigates these cases with an emphasis on the application of quotients.

Given a setoid, a corresponding quotient set is not necessarily formed by quotient types. An example is the set of integers. It can be seen as a quotient set on the setoid given by pairs of natural numbers $\mathbb{Z}_0 = \mathbb{N} \times \mathbb{N}$, where the equivalence relation identifies pairs representing the same difference, that is $(a, b) \sim (c, d)$ iff $a + d = c + b$. However a setoid is unnecessary, if we interpret it as a natural number with a sign and a set can be defined inductively. We can now define operations like addition and multiplication and show algebraic properties, such as verifying that the structure is a ring. However, this is quite complicated and uses many unnecessary case distinctions. E.g. the proving of distributivity within this setting is not satisfactory since too many cases has

to be proved from scratch. We found it is often easier to define the operations on the setoid and the required algebraic properties are direct consequences of the semiring structure of the natural numbers. Another common example is the set of rational numbers. It makes more sense to treat a rational number as an equivalence class of fractions which can be reduced to the same one. The benefits of being a quotient is more clear. The coprimarity has to been kept and it complicates defining operations and proving properties. In these cases, the property of being a quotient is not revealed in the definition.

In such a situation, both the set interpretation and the setoid one have some nice features. Hence we propose to use both the setoid and the associated set, but to use the setoid structure to define operations on the quotient set and to reason about it. In the present paper we introduce the formal framework to define the the quotients via normalisation function in Type Theory. It can be seen a "manual construction" of quotient types, in other words, instead of creating a type immediately given a setoid, we prove another given type *is* the quotient. The quotient structure provides conversions between base types and "definable quotient types". For example functions which respect the equivalence relation can be lifted. It can alleviate repetitive work on definable quotient types, because the base types usually have a list of available functions and properties to use. In some cases the base types are simpler to manipulate, which give us a shortcut to handle the quotient types. In the present paper we also give a categorical interpretation of our definitions. We also exhibit the advantages of applying definable quotients using the examples of integers and rational numbers.

1.1 Related Work

Quotient types were introduced by Mendler in [6] and subsequently investigated in Hofmann's PhD dissertation [4]. An extensive investigation of setoids can be found in [1]. Maietti considers extensions of both intensional and extensional type theory by quotient types [5]. Courtieu considers an extension of CIC (an intensional type theory) by *normalized types* corresponding to our definable quotients [2]. Nogin describes a modular implementation of quotient types in NuPRL (an extensional type theory) [7].

2 Quotients

We define several algebraic structures for quotients based on setoids. A *prequotient* gives a basic ingredients for later constructions. We give two equivalent definitions of *quotients*, one has a dependent eliminator and the other given by Hofmann added a non-dependent eliminator and an induction principle. A quotient is *exact* [1] if exactly one element of quotient set represents one equivalence class. We also prove that as long as we have propositional univalence, all *quotients* are *exact*. Given an embedding function which selects canonical choice for

[1] Exact quotient is just *effective quotient* in [5]

each equivalence class, a prequotient becomes a *definable quotient* which is even stronger than the notion exact quotient. These concepts and some equivalences and conversions among them has been implemented in Agda, and presented in appendix.

We start from the definition of *setoid* which provides the setoid interpretation of the "quotient".

Definition 1. *Setoid. A setoid is formed by*

1. *a set* A,
2. *a relation* $\sim\colon A \to A \to \mathbf{Prop}$, *and*
3. *it is an equivalence, i.e. it is reflexive, symmetric and transitive.*

Given a setoid and a function respects this equivalence (note: not necessary to be a normalisation function) we obtain a *prequotient*.

Definition 2. *Prequotient. Given a setoid* $(A, \sim)$, *a* prequotient $(Q, [\_], \text{sound})$ *over that setoid consists in*

1. *a set* Q,
2. *a function* $[\_] : A \to Q$,
3. *a proof* sound *that the function* $[\_]$ *is compatible with the relation* $\sim$, *that is*
$$\text{sound}\colon (a, b : A) \to a \sim b \to [a] = [b],$$

Roughly speaking, 1 corresponds to the formation rule, 2 and 3 correspond to the introduction rule. The function $[\_]$ is intended to be the *normalisation* function with respect to the equivalence relation, however it is not enough to determine it now. It is observable that given any non-empty set Q with any constant function fit in this definition.

To complete a *quotient*, we also need the elimination rule and computation rule.

Definition 3. *Quotient. A prequotient* $(Q, [\_], \text{sound})$ *is a quotient if we also have*

4. *for any* $B : Q \to \mathbf{Set}$, *an eliminator*
$$\begin{aligned}\text{qelim}_B \ :\ & (f\colon (a : A) \to B\,[a]) \\ & \to ((p : a \sim b) \to f\,a \simeq_{\text{sound}\ p} f\,b) \\ & \to ((q : Q) \to B\,q)\end{aligned}$$
such that qelim-β : $\text{qelim}_B\ f\,p\,[a] \equiv f\,a$.

This eliminator is also called the dependent lifting function, since it lifts a function well-defined on $(A, \sim)$ to a function defined on Q.

An alternative equivalent definition given by Martin Hofmann has a non-dependent eliminator and an induction principle.

Definition 4. *Quotient (Hofmann's). A prequotient* $(Q, [\_], \text{sound})$ *is a quotient (Hofmann's) if we also have*

$$\text{lift} : (f : A \to B) \to (\forall a, b \to a \sim b \to f\ a \equiv f\ b) \to (Q \to B)$$

together with an induction principle. Suppose B *is a predicate,*

$$\text{qind} : ((a : A) \to B\,[a]) \to ((q : Q) \to B\,q)$$

Finally, such a quotient is exact if additionally we have the exact property.

Definition 5. *Exact quotient. A quotient is* exact *(or efficient) if exactly one equivalence class corresponds to an element of* Q*, i.e. given* $a, b : A$*, if they are "normalised" to the same element of* Q *(*$[a] \equiv [b]$*), they must be in the same equivalence class.*

$exact : (\forall a, b : A) \to [a] \equiv [b] \to a \sim b$

Remark 1. *This is indeed provable if we have univalence axiom for propositions. The proof can be found in A.*

Finally we have a strongest definition *Definable quotient* which is more practical and give rise to an exact quotient.

Definition 6. *Definable quotient. Given a setoid* $(A, \sim)$*, a* definable quotient *is a prequotient* $(Q, [\_], \text{sound})$ *with*

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

Intuitively speaking, if we regard Q as the set of the equivalence classes, then the embedding function emb actually selects one representative in A for each equivalent class.

A definable quotient proves to be *exact* A.

Compared to the two definitions of quotients above, the definable quotient structure provides a more flexible way to eliminate quotients via conversions $[\_]$ and emb, hence it is more useful. But it only applies to the setoids which have canonical a choice for each equivalence class. Luckily, in many cases we have such a choice function. For instance, the setoid integers and setoid rational numbers as we will discuss below.

Remark 2. *In fact, the embedding function can be seen as the lifted version of the real normalisation function with respect to the equivalence relation* $\sim$*,*

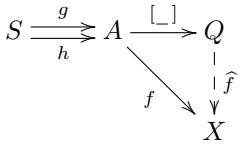
$$\begin{aligned} \mathit{emb}_0 &: A \to A \\ \mathit{embsound} &: (a : A) \to \mathit{emba} \sim a \\ \mathit{oneForEach} &: (a, b : A) \to a \sim b \to \text{emb}\,a = \text{emb}\,b \end{aligned}$$

In the other way around, emb_0 *can also be obtained by composing*$[\_]$ *with* emb.

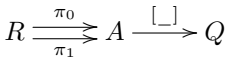
2.1 Coequalizer – quotients in category theory

Categorically speaking, quotients corresponds to coequalizers.

Definition 7. *Coequalizer. Given two morphisms $g, h : S \to A$, a* coequalizer *of g and h is a morphism $[\_] : A \to Q$ such that for any $f : A \to X$ satisfying $f \circ g = f \circ h$, there exists a unique $\widehat{f}$ such that*

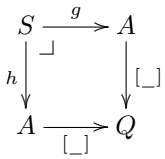


Indeed a prequotient corresponds to a diagram of morphisms



and eliminator corresponds to the universal property which makes it the colimit i.e. coequalizer.

A coequalizer is *effective* (or exact) if



and it is *split* if the morphism $[\_]$ is a split epi, that is if it has a right inverse $\mathrm{emb} : Q \to A$. That is to say, the definable quotient just corresponds to the notion *split coequalizer*.

3 The set of integers

Usually we can describe an integer as a natural number with a positive or negative sign in front:

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

However this gives two different constructors for 0 which is considered harmful. We lose canonicity and it will lead to unnecessary troubles.

A better representation provides a separate symbol for 0 but $+suc\ 0$ for 1, $-suc\ 0$ for -1. This does not sacrifice canonicity, although the denotations are not natural.

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

Take into account the embedding of natural numbers into integers, it makes sense to combine the positive's integers with zeros. Actually it is better pragmatically. In principle, definition with less cases is preferable because pattern matching grows with exponentially with the number of cases.

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

Although it is not symmetric, it is simple and canonical and it is our final choice.

3.1 The setoid of Integers

The introduction of negative numbers is believed to denote the results of subtraction of a larger number from a smaller one. Generalising this interpretation, integers is used to represent the results of subtraction of any natural number from another. Therefore integers can be represented by pairs of natural numbers

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

for example, from the equation $1 - 4 = -3$, we learn that $(1, 4)$ gives rise to -3.

Also different pairs can result in the same integer. For any two pairs of natural numbers (n_1, n_2) and (n_3, n_4), we know they represent the same integer if

$$n_1 - n_2 = n_3 - n_4$$

but technically this does not work because the "subtraction" defined for natural numbers only returns zero if the pair is for negative number. We only need to modify the equation a bit,

$$n_1 + n_4 = n_3 + n_2$$

This gives rise to an equivalence relation (reflexive, symmetric and transitive needs to prove separately). Combined with the carrier $\mathbb{Z}_0$, it forms a setoid interpretation of integers.

```
ℤ-Setoid : Setoid
ℤ-Setoid = record
```

```
{ Carrier    = ℤ₀
; _≈_        = _∼_
; isEquivalence = _∼_.isEquivalence
}
```

3.2 The definable quotient of integers

The basic ingredients for the definable quotient of integers have been given. One essential component of the quotient structure which relates the base type and quotient type is a normalisation function which can be recursively defined as follows:

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

In this case, we choose to define the embedding function before proving the soundness property.

It is plausible to find a canonical choice in each equivalence classes. In fact the definition of [_] already gives the answer in the first two cases.

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

To complete the definition of definable quotient, there are several properties to prove including sound, complete, stable and exact.

Because we have shown that exact quotient is equivalent to definable quotient, the lifting functions (dependent and non-dependent) are also derivable.

3.3 The application of definable quotients

Usually the definable quotient structure is useful when the base type (or carrier) is easier to handle, in other words, it is more convenient to use the setoid interpretation in defining operations and proving properties. In the case of integers, Since there is one case for he setoid representation, it generates less cases. Although for simple operations it does not simply too much, it makes a big different in proving complicated theorems like distributivity.

We will introduce the way we apply definable quotient in practical use.

Operations With two functions converting the two encodings, it is enough to lift all functions. It is possible to lift operations uniformly by mixing the normalisation and embedding functions. For example, given an unary operator

```
lift₁ : (op : ℤ₀ → ℤ₀) → ℤ → ℤ
```

```
lift₁ op = [_] ∘ op ∘ ⌜_⌝
```

This can be generalised to n-ary operators.

As long as we implement an operation of integers, it is safe to lift the setoid version. However it is clear that not all operations on the base types should be lifted and they do not make sense. Hence it is better to verify if the operation is well-defined on the setoid:

$$a \sim b \to op\ a \sim op\ b$$

Most of the operations for setoid integers can be concluded from expression rewriting equations. For example to define addition, we only need to transform the expression $(a_1 - b_1) + (a_2 - b_2) = (a_1 + a_2) - (b_1) - (b_2)$ so that we only do valid operations on natural numbers ($+$ or $*$) and minus is going to be replaced by pairing operation.

The addition of integers can be defined in one line

```
_+_ : ℤ₀ → ℤ₀ → ℤ₀
(x+ , x-) + (y+ , y-) = (x+ ℕ+ y+) , (x- ℕ+ y-)
```

We can easily verify it well-defined but it is not necessary because it has been implied in the mathematical reasoning before implementing it.

Properties Following axioms and theorems we learnt about integers, we can prove the plus, multiplication is commutative and associative, minus is inverse of plus etc. All these properties about plus, minus and multiplication forms a commutative ring of integers [2]. Like for natural numbers, axioms in classic mathematics are theorems to prove in constructive mathematics. The only axioms for integers are contained in the definition.

One of the important motivations of using setoid integers is that the setoid definition reduces the complexity as we have shown in the previous part, but it will be much more evident when proving properties of the ring of integers.

In practice, for the set definition of integers, most of basic operations and simple theorems are not unbearably complicated to deal with. However, as we mentioned before, the number of cases grows exponentially when case splitting is unavoidable. Although it is possible to prove lemmas which covers several cases and reduce number of cases, it often does not reduce complexity in total. An important case which is extremely difficult to conduct in practice is the proving of distributivity.

An efficient utilization of quotient structure: the proving of distributivity At first when we attempted to define the ring of integers, we were stuck in proving the the distributivity of $*$ over $+$. As an example we will show the attempt of left distributivity $x \times (y + z) = x \times y + x \times z$.

[2] can cite somewhere else http://www.millersville.edu/ bikenaga/number-theory/ring-of-integers/ring-of-integers.html

To simplify it, we define the multiplication in an arithmetic way instead of pattern matching.

```
_ℤ*_ : ℤ → ℤ → ℤ
i ℤ* j = sign i S* sign j ◃ | i | ℕ* | j |
```

Of course it is not rational to split cases into $2 * 2 * 2$. The first idea is to apply the left distributivity law of natural numbers when they are non-negative. To utilize the it more, it can be generalised to the cases when all three variables have the same signs. Moreover when only y and z have the same symbol, it is still plausible. The following is a partial definition (Note: DistributesOver^l means that the distributivity of the first operators over the second one) and three lemmas are given after it

```
dist^l   : _ℤ*_ DistributesOver^l _ℤ+_
dist^l x y z with sign y S≟ sign z
dist^l x y z    | yes p
   rewrite p
      | lem1 y z p
      | lem2 y z p =
   trans (cong (λ n → sign x S* sign z ◃ n)
   (ℕdist^l | x | | y | (| z |)))
   (lem3 (| x | ℕ* | y |) (| x | ℕ* | z |) _)
dist^l x y z | no ¬p = ...
```

To prove these simpler cases we need three lemmas,

```
lem1 : ∀ x y → sign x ≡ sign y → | x ℤ+ y | ≡ | x | ℕ+ | y |
lem1 (-suc x) (-suc y) e = cong suc (sym (m+1+n≡1+m+n x y))
lem1 (-suc x) (+ y) ()
lem1 (+ x) (-suc y) ()
lem1 (+ x) (+ y) e = refl

lem2 : ∀ x y → sign x ≡ sign y → sign (x ℤ+ y) ≡ sign y
lem2 (-suc x) ( -suc y) e = refl
lem2 (-suc x) (+ y) ()
lem2 (+ x) (-suc y) ()
lem2 (+ x) (+ y) e = refl


lem3 : ∀ x y s → s ◃ (x ℕ+ y) ≡ (s ◃ x) ℤ+ (s ◃ y)
lem3 0 0 s = refl
lem3 0 (suc y) s = sym (ℤ-id-l _)
lem3 (suc x) y s = trans (h s (x ℕ+ y)) (
      trans (cong (λ n → (s ◃ suc 0) ℤ+ n) (lem3 x y s)) (
```

```
    trans (sym (ℤ-+-assoc (s ◃ suc 0) (s ◃ x) (s ◃ y))) (
    cong (λ n → n ℤ+ (s ◃ y)) (sym (h s x)))))
where
h : ∀ s y → s ◃ suc y ≡ (s ◃ (suc 0)) ℤ+ (s ◃ y)
h s 0 = sym (ℤ-id-r _)
h Sign.- (suc y) = refl
h Sign.+ (suc y) = refl
```

In the second category of cases, if y and z have different signs, it is impossible to apply the left distributivity law for natural numbers. Intuitively speaking, there is no rule to turn expressions like $x * (y - z)$ into expression which only contains natural numbers, therefore we have to prove it from scratch. Although it is possible to prove finally, it is not the best solution we want.

It is much simpler to prove the distributivity for setoid representations. In fact with the help of ring solver, it can be proved automatically. All these definitions of operators only involves operators for natural numbers which forms a commutative semiring for natural numbers. This implies that these expressions of setoid integers which only involves plus, minus and multiplication, can be turned into equation of natural numbers. The theorems are also simply equations of natural numbers which are automatically solvable.

Remark 3. *A* ring solver *is an automatic equation checker for rings e.g. the ring of integers. It is implemented based on the theory described in "Proving Equalities in a Commutative Ring Done Right in Coq" by Grégoire and Mahboubi [3].*

Therefore to prove the distributivity, the simplest way is to use semiring solver for natural numbers.

```
dist^l :  _*_ DistributesOver^l _+_
dist^l (a , b) (c , d) (e , f) = solve 6
   (λ a b c d e f → a :* (c :+ e) :+ b :* (d :+ f) :+
      (a :* d :+ b :* c :+ (a :* f :+ b :* e))
      :=
      a :* c :+ b :* d :+ (a :* e :+ b :* f) :+
      (a :* (d :+ f) :+ b :* (c :+ e))) refl a b c d e f
```

It is not the simplest way to use ring solver since we have to feed the type (i.e. the equation) to the solver which can be automatically figured out by "reflection". It helps us quote the type of the goal so that we can define a function that automatically do it without explicitly providing the equations. There is already some work done by van der Walt [9]. It can be seen as an analogy of the "ring" tactic from Coq.

Use the ring solver for natural numbers we can prove all theorems required for ring solver for integers. However the process of proof terms generation and type checking take very long time in practice. It may heavily slow the type

checking although the optimization of Agda already shows a big improvement in this technical efficiency issue. As these theorems are going to be used quite often, it is reasonable to manually construct the proof terms to improve efficiency of library code, sacrificing some conveniences.

Luckily it is still much simpler than the ones for the set of integers $\mathbb{Z}$. First there is only one case of integer and as we know the equations is indeed equations of natural numbers which can be proved using only the properties in the commutative semiring of natural numbers. There is no need to prove some properties for $\mathbb{Z}$ from scratch like in the proving of distributivity.

```
dist-lem^l : ∀ a b c d e f →
  a ℕ* (c ℕ+ e) ℕ+ b ℕ* (d ℕ+ f) ≡
  (a ℕ* c ℕ+ b ℕ* d) ℕ+ (a ℕ* e ℕ+ b ℕ* f)
dist-lem^l a b c d e f = trans
  (cong₂ _ℕ+_ (ℕdist^l a c e) (ℕdist^l b d f))
  (swap23 (a ℕ* c) (a ℕ* e) (b ℕ* d) (b ℕ* f))

dist^l : _ℤ₀*_ DistributesOver^l _ℤ₀+_
dist^l (a , b) (c , d) (e , f) =
  cong₂ _ℕ+_ (dist-lem^l a b c d e f)
  (sym (dist-lem^l a b d c f e))
```

It only needs one special lemma which can be proved by applying distributivity laws for natural numbers. The swap23 is a commonly used equation rewriting lemma

$(m + n) + (p + q) = (m + p) + (n + q)$

After all, the application of quotient structure in the integer case provides us a general approach to define functions and prove theorems when the base types are simpler to deal with. Adopting this approach, both the appearance – the set representation of integers and the underlying – the convenience of manipulating setoid representation of integers, are maintained. In the case of rational numbers, the base type is also simpler to deal with.

4 Rational numbers

In Agda standard library, the set of rational numbers is defined as fractions whose numerator and denominator are coprime. It ensures the canonicity of representations, but it complicates the manipulation of rational numbers since the coprimarity has to be kept for each intermediate step of calculations. From experience of calculating fractions in arithmetic, we usually do not reduce fractions until we are going to show a simplified answer, and all the calculation can be carried out correctly without reducing. Actually from a discussion on Agda mailing list [3], a better way to manipulate the rational numbers is required. A

[3] http://comments.gmane.org/gmane.comp.lang.agda/6372

flexibility way to choose from unreduced fractions or reduced ones in operations can be fulfilled by a definable quotient of rational numbers. The unreduced fractions are mainly used for underlying calculations and property proving because it ignores a lot of reducing steps and coprimarity checking, and the reduced fractions are accessible when displaying rational numbers. It simplifies defining functions and proving properties related to rational numbers. Moreover, we believe the efficiency of programs involving rational numbers calculation is improved due to the removal of reducing , even though some people claim that the unreduced numbers are too large to make it efficient.

4.1 The setoid of rational numbers: unreduced fractions

The setoid representation of rational numbers – unreduced fractions are more common in mathematics. Usually a fraction denoted as $\frac{m}{n}$ consists of an integers m called *numerator* and a non-zero integer n called *denominator*. In type theory, since the data types have different complexity, we have a decision to make in the choice of best definition like what we did for integers. Alternatively keeping the numerator integer, the denominator can be a positive natural number such that the sign of rational number is kept in the numerator solely and there is no need to add a restriction to exclude 0 which is not so easy for the set of integer.

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

Technically there is no need to define a new type for positive natural numbers $\mathbb{N}^+$, so we use $\mathbb{N}$ which we can name it as *denominator-1* or implement it as follows

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

such that $2/suc2$ stands for $\frac{2}{3}$.

Remark 4. *It is also reasonable to define it with a pair of natural numbers with a sign defined separately. However this also complicates calculation somehow.*

The rational number represented by a fraction $\frac{m}{n}$ is indeed the result of division $m \div n$. Therefore two different fractions can represent the same rational numbers. In mathematics, to judge the equality of two fractions, it is easier to conduct the following conversion,

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

Hence the equivalence relation can be defined as,

```
_*_ : ℤ → ℕ → ℤ
(+ x) * d = + (x ℕ* d)
(-suc x) * 0 = + 0
```

```
(-suc x) * suc d = -suc (x ℕ+ suc x ℕ* d)

_∼_     : Rel ℚ₀ _
(n1 /suc d1) ∼ (n2 /suc d2) =    n1 * suc d2 ≡ n2 * suc d1
```

4.2 The set definition of rational numbers: reduced fractions

The reduced fractions are canonical representations of rational numbers. It is a subset of fractions, so we only need to add a restriction that the numerator and denominator is coprime,

$$\mathbb{Q} = \Sigma((n, d) : \mathbb{Z} \times \mathbb{N}).\mathsf{coprime}\ n\ (d + 1)$$

It is implemented as follows (also available in standard library),

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

  denominator : ℤ
  denominator = + suc denominator-1

  coprime : Coprime numerator denominator
  coprime = toWitness isCoprime
```

4.3 The definable quotient of rational numbers

From the definition, we already have a setoid which contains unreduced fractions $\mathbb{Q}_0$ as base type and an equivalence relation on it. Compare with the set definition, it is clear that normalisation from $\mathbb{Q}_0 \to \mathbb{Q}$ is just reducing functions.

To implement reducing process, we can utilize the library code about *great common divisor (gcd)*. The function gcd calculates the divisor and a data type GCD containing information from calculating the greatest common divisor which can be converted to another data type GCD′ [4] which contains the three essential results we need, i.e. the new numerator, new denominator and the proof term that they are coprime. Combing the library functions and types with a self-defined function GCD′→ℚ which constructs the reduced fraction we need, the normalisation function is implemented as follows,

```
[_] : ℚ₀ → ℚ
```

[4] http://www.cse.chalmers.se/ nad/repos/lib/src/Data/Nat/Coprimality.agda

```
[ (+ 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)
```

and the embedding function is trivial, because we only need to forget the coprime proof in the normal form,

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

Similar to the case of integers, complete the definable quotient, we also need to prove the sound, complete, stable and exact properties. There is no need to show the proof of them since the normalisation function is implemented using an approach which has been tested in mathematics. Also we can apply the definable quotient structures in helping form the field of rational numbers. After embedding the natural numbers into integers, any theorems using only the operations in the field $\mathbb{Q}$, are turned out to be propositions of integers which can be solved using ring solver or theorems proven for integers.

5 Conclusion

In this paper, we have shown that there are some quotients definable via a normalsation function. We present some quotient structures and prove some equivalences and conversions among the them. In fact they provide us conversions between base types and quotient types such that we can usually benefit from the ease from base types while still keeping the canonicity and a better display from the quotient types. To show the application of definable quotients, we used two examples, the set of integers and the set of rational numbers. Some concrete cases are also presented to show how to uniformly lift operations and theorems from base types, and how can we benefit from the utilization of definable quotient structures.

References

[1] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, 2003.

[2] Pierre Courtieu. **Normalized types**. In *Proceedings of Computer Science Logic, 15th International Workshop*, volume 2142 of *Lecture Notes in Computer Science*, pages 554–569. Springer, 2001.

[3] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in coq. In *Theorem Proving in Higher Order Logics*, pages 98–113. Springer, 2005.

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

[5] M. Maietti. About effective quotients in constructive type theory. *Types for Proofs and Programs*, pages 166–178, 1999.

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

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

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

[9] Paul van der Walt and Wouter Swierstra. Engineering Proof by Reflection in Agda. In Ralf Hinze, editor, *Implementation and Application of Functional Languages - 24th International Symposium*, volume 8241 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 2012.

A Appendix

```
module Quotient where
```

Prequotient
Given a setoid, we can turn it into a pre-quotient, corresponds to fork (or cofork) in category theory.

```
record pre-Quotient (S : Setoid) : Set₁ where


  field
    Q    : Set
    [_] : A → Q
    [_]= : [_] respects _~_
    QisSet : isSet Q
```

Quotient
A prequotient with a dependent eliminator.

```
record Quotient {S : Setoid}(PQ : pre-Quotient S) : Set₁ where
  open pre-Quotient PQ
  field
    qelim    : {B : Q → Set}
             → (f : (a : A) → B [ a ])
             → (∀ {a a'} → (p : a ~ a')
             → subst B [ p ]= (f a) ≡ f a')
             → (q : Q) → B q
    qelim-β : ∀ {B a f}(resp : (∀ {a a'} → (p : a ~ a')
             → subst B [ p ]= (f a) ≡ f a'))
             → qelim {B} f resp [ a ] ≡ f a
```

Quotient (Hofmann's)
A prequotient with a non-dependent eliminator (lifting).

```
record Hof-Quotient {S : Setoid}
  (PQ : pre-Quotient S) : Set₁ where
  open pre-Quotient PQ
  field
    lift     : {B : Set}
             → (f : A → B)
             → f respects _~_
             → Q → B
```

```
    lift-β : ∀ {B a f}(resp : f respects _~_)
           → lift {B} f resp [ a ] ≡ f a

    qind   : ∀ (P : Q → Set)
           → (∀{x} → isProp (P x))
           → (∀ a → P [ a ])
           → (∀ x → P x)
```

Exact quotient

```
record exact-Quotient {S : Setoid}
   (PQ : pre-Quotient S) : Set₁ where
   open pre-Quotient PQ
   field
      Qu    : Quotient PQ
      exact : ∀ {a b : A} → [ a ] ≡ [ b ] → a ~ b
```

Definable quotient

```
record def-Quotient {S : Setoid}
         (PQ : pre-Quotient S) : Set₁ where
   open pre-Quotient PQ
   field
      emb      : Q → A
      complete : ∀ a → emb [ a ] ~ a
      stable   : ∀ q → [ emb q ] ≡ q
```

Proof : Definable quotients are exact.

```
   exact : ∀{a b} → [ a ] ≡ [ b ] → a ~ b
   exact {a} {b} p =
      ~-trans (~-sym (complete a))
      (~-trans (subst (λ x →
      emb [ a ] ~ emb x)
      p ~-refl) (complete b))
```

Equivalences and conversions among the quotient structures
Hofmann's quotient is equivalent to *Quotient*.

```
Hof-Quotient→Quotient : {S : Setoid} → {PQ : pre-Quotient S}
   → (Hof-Quotient PQ) → (Quotient PQ)
Hof-Quotient→Quotient {S} {PQ} QuH =
   record
      { qelim    = λ {B} f resp
      → proj₁ (qelim' f resp)
```

```
  ; qelim-β = λ {B} {a} {f} resp
  → proj₂ (qelim' f resp)
  }
where
  open pre-Quotient PQ
  open Hof-Quotient QuH

  qelim' : {B : Q → Set}
    → (f : (a : A) → B [ a ])
    → (∀ {a a'} → (p : a ~ a') 
    → subst B [ p ]= (f a) ≡ f a')
    → Σ[ f^ : ((q : Q) → B q) ]
          (∀ {a} → f^ [ a ] ≡ f a)
  qelim' {B} f resp =   f^ , f^-β
where
```

The dependent lifting is defined to be the second projection of a non-dependent lifting functions as follows:

```
f₀ : A → Σ Q B
f₀ a = [ a ] , f a

resp₀ : f₀ respects _~_
resp₀ p = Σeq [ p ]= (resp p)


f' : Q → Σ Q B
f' = lift f₀ resp₀

id' : Q → Q
id' = proj₁ ∘ f'

P : Q → Set
P q = id' q ≡ q

f'-β : {a : A} → f' [ a ] ≡ [ a ] , f a
f'-β = lift-β _

f'-sound : ∀{a} → id' [ a ] ≡ [ a ]
f'-sound = cong proj₁ f'-β

f'-sound' : ∀ {q} → id' q ≡ q
f'-sound' {q} = qind P QisSet
  (λ _ → f'-sound) q

f'-sound2 : ∀ {a} →
```

```
    subst B f'-sound (proj₂ (f' [ a ])) ≡ f a
    f'-sound2 = cong-proj₂ _ _ f'-β

    f^ : (q : Q) → B q
    f^ q = subst B (f'-sound') (proj₂ (f' q))

    f^-β : ∀ {a} → f^ [ a ] ≡ f a
    f^-β {a} = trans (subIrr QisSet
      f'-sound' f'-sound) f'-sound2


Quotient→Hof-Quotient :
    {S : Setoid}{PQ : pre-Quotient S}
    → (Quotient PQ)
    → (Hof-Quotient PQ)
Quotient→Hof-Quotient {S} {PQ} QU =
    record
    { lift    = λ f resp
        → qelim f (resp' resp)
    ; lift-β = λ resp
        → qelim-β (resp' resp)
    ; qind = λ P isP f
            → qelim {P} f (λ _ → isP)
    }
    where
      open pre-Quotient PQ
      open Quotient QU

      resp' : {B : Set}{a a' : A}
        {f : A → B}
        (resp : f respects _~_)
        (p : a ~ a')
        → subst (λ _ → B) [ p ]⁼ (f a)
        ≡ f a'
      resp' resp p =
        trans (subIrr2 [ p ]⁼)
        (resp p)
```

Definable quotients is the strongest and gives rise to a *Quotient* (and ineed it is an *Exact quotients*).

```
def-Quotient→Quotient :
    {S : Setoid}{PQ : pre-Quotient S}
    → (def-Quotient PQ) → (Quotient PQ)
def-Quotient→Quotient {S} {PQ} QuD =
    record { qelim =
```

```
    λ {B} f resp q → subst B (stable q) (f (emb q))
    ; qelim-β =
    λ {B} {a} {f} resp →
    trans (subIrr QisSet (stable [ a ])
    [ complete a ]⁼) (resp (complete a))
  }
  where
  open pre-Quotient PQ
  open def-Quotient QuD
```

Remember that we have proven *Definable quotients* are exact.

```
def-Quotient→exact-Quotient :
  {S : Setoid}{PQ : pre-Quotient S}
  → def-Quotient PQ → exact-Quotient PQ
def-Quotient→exact-Quotient {S} {PQ} QuD =
  record { Qu = def-Quotient→Quotient QuD
    ; exact = exact
    }
  where
  open pre-Quotient PQ
  open def-Quotient QuD
```

The propositional univalence implies a quotient is always exact

```
Prp = Set

_⇔_ : (A B : Prp) → Prp
A ⇔ B = (A → B) × (B → A)
```

```
module PuImpEff
```

Assume we have the propositional univalence (the other part is trivial),

```
(PropUni₁ : ∀ {p q : Prp} → (p ⇔ q) → p ≡ q)
```

and a quotient (Note that we postulate the lifting function for $B : Set_1$ because for convenience we did not take into account the universe levels, but it is required in the proof here)

```
{S : Setoid}
{PQ : pre-Quotient S}
  where
```

```
open pre-Quotient PQ


  postulate
    lift₁ : {B : Set₁}    →
              (f : A → B) →
              (f respects _~_) →
                Q → B

  postulate
    lift-β₁ : ∀ {B a f}{resp : (f respects _~_)}
              → lift₁ {B} f resp [ a ]    ≡ f a

  coerce : {A B : Set} → A ≡ B → A → B
  coerce refl m = m

  exact : ∀ a a' → [ a ] ≡ [ a' ] → a ~ a'
  exact a a' p = coerce P^-β (~-refl {a})
    where
      P : A → Prp
      P x = a ~ x

      P-resp : P respects _~_
      P-resp {b} {b'} bb' =
            PropUni₁ ((λ ab → ~-trans ab bb') ,
            (λ ab' → ~-trans ab' (~-sym bb')))

      P^ : Q → Prp
      P^ = lift₁ P P-resp

      P^-β : P a ≡ P a'
      P^-β = trans (sym lift-β₁)
        (trans (cong P^ p) lift-β₁)
```