Investigation into definable quotient types

Li Nuo

May 5, 2011

1 Background

Quotient generally means the the result of the division. More abstractly, it represents the result of the partition of sets based on certain equivalence relation on them. Then can both written as A/B, both of them means divide A by B equally. Similarly, in type theory, have quotient types. Quotient types are created by dividing types by an equivalence relation.

However in Agda which is the type theory we will talk about cannot form axiomatic quotient types. But we can still implement the quotients as setoids containing the carrier the equivalence relation and the proof of it. We call them definable quotient types and I will present some examples from my definition for numbers in Agda [1] to illustrate the ideas.

2 Quotient definition of Integers

The integers were invented to represent all the result of subtraction between natural numbers. Therefore it is naturally to define a pair of natural numbers to be the result of the subtraction. Hence we can use a pair of natural nubmers as carrier $\mathbb{Z}_0$.

```
ℤ₀ = ℕ × ℕ
```

When `n1 - n2 = n3 - n4` we can say the quotients `(n1, n2)` and `(n3, n4)` represent the same integer. We can transform the equation to be `n1 + n4 = n3 + n2` so that it becomes a valid judgemental equality between natura numbers. Therefore we can define the equivalence relation of quotient integer as below,

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

With the euquivalence relation, the set of all pairs of natural numbers are divided into equivalence classes. For each equivalence class we need a representative. We can define a normalisation function to normalise the pair of natural numbers until either of the natural numbers becomes zero.

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

For example, $(3, 2)$ can be normalised to $(2, 1)$, then to $(1, 0)$.

As soon as we have the normalisation functions, we can use another more general way to define equivalence relation, namely just identify their normal form.

```
_~_   : Rel ℤ₀ zero
x ~ y = [x] ≡ [y]
```

The $[\_]$ is an endomap in the set $\mathbb{Z}_0$, and the resulting subset is actually isomorphic to the set of integers. However since we do not distinguish the types of the original form and the normal form, we lose the information that it has been normalised. Therefore we can define the type of the result to be the set of integers.

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

Then this is a retraction function for the normalisation function and we call it denormalisation function.

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

Firstly we need to prove ~ is actually an equivalence relation.

Reflexivity

```
zrefl : Reflexive _~_
zrefl {x+, x-} = refl
```

Symmetry

```
zsym : Symmetric _~_
zsym {x+, x-} {y+, y-} = sym
```

Transitivity

```
_>~<_ : Transitive _~_
_>~<_ {x+, x-} {y+, y-} {z+, z-} x=y y=z =
  cancel-+-left (y+ ℕ+ y-) (ℕ.exchange₁ y+ y- x+ z- >≡<
  (y=z += x=y) >≡< ℕ.exchange₂ z+ y- y+ x-)
```

~ isEquivalence relation

```
_~_isEquivalence : IsEquivalence _~_
_~_isEquivalence = record
  {refl  = zrefl
  ;sym   = zsym
  ;trans = _>~<_
  }
```

Now we can prove that the $\mathbb{Z}_0$ and its equivalence relation ~ form a setoid.
($\mathbb{Z}_0$, ~) is a setoid

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

3 Rational numbers

The quotient definition of rational number is more natural to understand and the normalisation is also commonly used in regular mathematics. We just use one integer and one natural number to represent a rational number. The reason is because it is hard to exclude the invalid denominator if we use integers, so I choose the natural numbers to represent positive natural number which are one bigger.

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

and this is the equivalence relation for it

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

Reflexivity

```
qrefl : Reflexive _~_
qrefl {n /suc d} = refl
```

symmetry

```
qsym : Symmetric _~_
qsym {a /suc ad} {b /suc bd} = sym
```

transitivity

```
qtrans : Transitive _~_
qtrans {a /suc ad} {b /suc bd} {c /suc cd} a=b b=c with ℤ.0? b
qtrans {a /suc ad} {◦ (+ 0) /suc bd} {c /suc cd} a=b b=c | yes refl =
   ℤ.solve0' (+ suc bd) {a} (λ ()) a=b 0 ~
   ℤ.solve0' (+ suc bd) {c} (λ ()) ⟨ b=c ⟩
qtrans {a /suc ad} {b /suc bd} {c /suc cd} a=b b=c | no ¬p =
   ℤ.l-integrity (b ℤ* (+ suc bd)) (ℤ.nz* b (+ suc bd) ¬p (λ ())) (
   ℤ.*-exchange₁ b (+ suc bd) a (+ suc cd) >≡<
   (ℤ.*-cong b=c a=b) >≡<
   ℤ.*-exchange₂ c (+ suc bd) b (+ suc ad))
```

~ *isEquivalence relation*

```
isEquivalenceℚ₀ : IsEquivalence _~_
isEquivalenceℚ₀ = record
   {refl = qrefl
   ;sym = qsym
   ;trans = qtrans
   }
```

Then it is natural to form the setoid
(ℚ₀, ~) is a setoid

```
ℚ₀setoid : Setoid _ _
ℚ₀setoid = record {
   Carrier = ℚ₀
   ; _≈_ = _~_
   ; isEquivalence = isEquivalenceℚ₀
   }
```

However these definition are just setoid and to form a quotient type, we need more structure. For definable quotient types, we need a representative of each equivalence class, we may have a set which is isomorphic to the set of equivalence classes, namely the normal form of the quotient type. Moreover, If we abstract the structure, we can prove some general properites for definable quotient types.

4 The general structure of definable quotient types

I will use the interfaces written by Thomas Amberree in this part. We need to first establish the quotient signature.

```
record QuSig (S : Setoid zero zero) : Set₁ where
  field
    Q    : Set
    [_]  : Carrier S → Q
    sound : ∀ {a b : Carrier S} → (_≈_ S a b) → [ a ] ≡ [ b ]
```

In this type signature, for certain setoid we have a type represent the set of the normal form, a normalisation function, and the proof that two elements in the same equivalence class normalised to the same form. With soundness, we can say normalisation is a function if we treat S as the set of equivalence classes.

However, there is no surjective requirements for the map in this signature. It means that the set of equivalence classes are not isomorphic to the set Q.

Actually, we can use the same type for Carrier S and Q. For example, for Setoid $\mathbb{Z}_0, \sim$, we can build a quotient siganature by giving $\mathbb{Z}_0$ and the endomap normalisation function.

Now, using the quotient signature if we can prove that any function of type S → B respects the equivalence relation, then we can lift it to be a function of type Q → B. Of course we need to prove that it is lift function. With the lift function we have the first definition of quotient.

```
record Qu {S : Setoid zero zero} (QS : QuSig S) : Set₁ where
  private S = Carrier S
    _~_  = _≈_ S
    Q      = Q QS
    [_]    = [_] QS
    sound   : ∀ {a b : S₀} → (a ~ b) → [ a ] ≡ [ b ]
    sound    = sound QS
  field
    lift    : {B : Q → Set}
            → (f : (a : S) → (B [ a ]))
            → ((a a' : S) → (p : a ∼ a') → subst B (sound p) (f a) ≡ f a')
            → (c : Q) → B c
    liftok  : ∀ {B a f q} → lift {B} f q [ a ]    ≡ f a
    liftIrr : ∀ {B a f q q'} → lift {B} f q [ a ] ≡ lift
    {B} f q' [ a ]
```

In my opinion the proof irrelevance of lift operations are unecessary since lifeok implies it.

However, there can be more than one equivalence classes normalised to the same form. Therefore the normal form do not fully contain the information of quotient type. If we can prove the completeness **Nuo:** I don't think this is completeness, it is only injective, namely two elements normalised to the same form must be in the same equivalence class. they belong to the same equivalence class. Hence the normalisation is injective from the set of equivalence classes to the set of normal forms.

```
record QuE {S : Setoid zero zero} {QS : QuSig S} (QU : Qu QS) : Set₁ where
  private S  = Carrier S
    _∼_    = _≈_ S
    Q      = Q QS
    [_]    = [_] QS
    sound  : ∀ {a b : S} → (a ∼ b) → [ a ] ≡ [ b ]
    sound  = sound QS
  field
    complete : ∀ {a b : S} → [ a ] ≡ [ b ] → a ∼ b
```

Even if we prove the normalisation to be injective, we still not require it to be surjective. Then Q may have some redundance. Therefore we need more efficient quotient type.

In Nf we have a embedding function used to choose a representative for each equivalence class. The proof of stability shows that emb is a section of normalisation function. Since all elements in Q can be the result of the normalisation, it must be surjective. The proof compl shows that the representative is in the same equivalence class hence we can prove the completeness as well. In this definition of quotient, the set of all equivalence classes are in fact isomorphic to the set Q.

```
record Nf {S : Setoid zero zero} (QS : QuSig S) : Set₁ where
  private S = Carrier S
```

```
    _~_  =  _≈_ S
    Q      =  Q QS
    [_]  =  [_] QS
  field
    emb    : Q → S
    compl  : ∀ a → emb [ a ] ~ a
    stable : ∀ x → [ emb x ] ≡ x
```

We can easily establish the function transforming the Nf to QuE, since completeness can be derived from compl.

```
nf2quE : {S : Setoid zero zero} → {QS : QuSig S} → {QU : Qu QS} → (Nf QS) → (QuE QU
nf2quE {S} {QS} {QU} nf =
  record {
    complete  =  λ {a} {b} [ a ] ≡ [ b ] →
                       ⟨ compl a ⟩ ▶ subst (λ x → x ~ b) (emb ⋆ ⟨ [ a ] ≡ [ b ] ⟩) (compl b)
    }
      where
      private S  =  Carrier S
        _~_    =  _≈_ S
        Q        =  Q QS
        [_]      =  [_] QS
        emb      =  emb nf
        compl    =  compl nf
        ⟨_⟩      :  Symmetric _~_
        ⟨_⟩      =  symmetric S
        _▶_      :  Transitive _~_
        _▶_      =  transitive S
```

We can also define non-dependent lift version of quotients. We need to prove quotient induction when we have uniqueness of proof for certain proposition dependent on Q.

Nuo: Why we need qind?

```
record QuH {S : Setoid zero zero} (QS : QuSig S) : Set₁ where
  private S  =  Carrier S
    _~_   =  _≈_ S
    Q       =  Q QS
    [_]  =  [_] QS
    sound  :  ∀ {a b : S} → (a ~ b) → [ a ] ≡ [ b ]
    sound  =  sound QS
  field
```

```
liftH    : {B : Set}
  → (f : S → B)
  → ((a a' : S) → (a ∼ a') → (f a) ≡ f a')
  → Q → B
liftHok : ∀ {B a f q} → liftH {B} f q [ a ] ≡ f a
  -- quotient induction
qind : (P : Q → Set)
  → (∀ {x} → (p p' : P x) → p ≡ p')
  → (∀ {a} → P [ a ])
  → (∀ {x} → P x)
```

If we have the normal form definition, we can lift the function easily.

```
nf2qu : {S : Setoid zero zero} → {QS : QuSig S} → (Nf QS) → (Qu QS)
nf2qu {S} {QS} nf =
  record {
  lift     = λ {B} f q a- → subst B (stable0 a-) (f (emb0 a-));
  liftok   = λ {B} {a} {f} {q} →
  substIrr B (stable [ a ]) (sound (compl a)) (f (emb [ a ])) ▶ q _ _ (compl a);
  liftIrr  = refl
  }
  where S       = Carrier S
        _∼_   = _≈_ S
        [_]    = [_] QS
        sound  : ∀ {a b : S} → a ∼ b → [ a ] ≡ [ b ]
        sound  = sound QS
        compl  = compl nf
        stable = stable nf
        emb    = emb   nf
```

5 The properties of definable quotient types

Not only the predicate can be lifted, but also the operators can be lifted.

```
Op : ℕ → Set → Set
Op 0 = λ t → t
Op (suc n) = λ t → (t → Op n t)
record SetoidOp (St : Setoid zero zero) (n : ℕ) : Set1 where
  constructor §_§
  private
```

```
      S = Setoid.Carrier St
    field
      op : Op n S
  record QuotientOp {St : Setoid zero zero}
    {Qs : QuSig St} (nf : Nf Qs) (n : ℕ) : Set₁ where
    constructor §_§
    private
      Q = QuSig.Q Qs
    field
      op : Op n Q
  auxf : {S Q : Set} (n : ℕ) ([_] : S → Q) (emb : Q → S) →
    Op n S → Op n Q
  auxf zero [_] emb op = [op]
  auxf (suc n) [_] emb op = λ x → auxf n [_] emb (op (emb x))
  liftop : {S : Setoid zero zero} (n : ℕ) (Qs : QuSig S)
    (So : SetoidOp S n) (nf : Nf Qs) → QuotientOp nf n
  liftop n (Q, [_], sound) § op § (emb, compl, stable) = § auxf n
  [_] emb op §
```

We can lift operators of any order within the normal form definition of quotient type. According to this, lift the general properties are also possible.

6 The undefinable quotient types

All of integers, rational numbers and the congruence class of modulo have definable normal form or canonical form. However real numbers is not belonging to this group. It does not have normal forms. We can use cauchy sequences or signed digits to represent real numbers. They are obvious quotient sets isomorphic to the true real numbers, but we cannot use the interface introduced above. We can only postulate it in Agda. Moreover the equivalence relation is undecidable. For these kinds of quotient types, for which we do not have normal form, we call them axiomatised quotient types.

This could be a cauchy sequence used to represent real numbers (Simplified for readability),

```
  record cauℝ : Set where
    field
      f : ℕ → ℚ
      p : (n : ℕ) → ∀ (m : ℕ) → (n < m) → | (f m) - (f n) | < (1 / 2 ↑ n)
```

It contains a function to generate the sequence of numbers and a proposition that the sequence converges by bounded rate.

For example we can embedding rational numbers easily,

```
emb : ℚ₀ → ℝ
emb q = f: (λ _ → q) p: λ n m n<m → (s≤s z≤n) resp (abscanc q)
```

But for irrational numbers we have to use different ways to generate the sequences. For square root, we can use Taylor series.

7 Conclusion

Here, we only talk about definable quotient types within Agda. The quotient is a setoid and the elements in an euqivalence classes are not definitionally equal. However, if we axiomatize the type form of quotient type and let the Agda automatically normalise the carrier, then the definitional equality between different elements in same equivalence class will be present.

References

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