Pragmatic Quotient Types in Coq

Cyril Cohen

University of Gothenburg
cyril.cohen@gu.se

July 24, 2013

Examples of quotient

- $\mathbb{Z} = \mathbb{N} * \mathbb{N}/\texttt{equivZ}$

  ```
  equivZ x y := (x.1 + y.2 = y.1 + x.2)
  ```

- $\mathbb{Q} = \mathbb{Z} * \mathbb{Z}^*/\texttt{equivQ}$

  ```
  equivQ x y := (x.1 * y.2 = y.1 * x.2)
  ```

- $F = D * D^*/\texttt{equivF}$

  ```
  equivF m n := (x.1 * y.2 = y.1 * x.2)
  ```

- $\mathrm{Setoid}/\equiv$

Original motivation for this work:

algebraic numbers for Feit Thompson.

Why forming a quotient type?

Context: Intensional Type Theory (here Coq)

Main goal: get Leibniz equality in Type Theory.

Because it is substitutive in any context:

```
eq_rect :
  forall (A : Type) (x : A) (P : A -> Type),
         P x -> forall y : A, x = y -> P y.
```

What is this work about?

It is **not** about

- category theory,
- adding quotient types to the meta theory of Coq,
- giving a general axiomatization of quotients.

It is about a framework **usable in practice** for our applications (mainly discrete algebra), **without modifying** Coq.

An abstraction layer

Operations and properties on the Quotient.

```
addZC : forall x y z : Z,
  x +Z (y +Z z) = (x +Z y) +Z z
```

```
addNNC : forall m n p : N * N,
  equivZ (m +N*N (n +N*N p)) ((m +N*N n) +N*N p)
```

(where `equivZ x y := (x.1 + y.2 = y.1 + x.2)`)

Low level operations and properties.

An abstraction layer

Operations and properties on the Quotient.

```
addZC : forall x y z : Z,
  x +Z (y +Z z) = (x +Z y) +Z z
```

```
addNNC : forall m n p : N * N,
  m +N*N (n +N*N p) = (m +N*N n) +N*N p %[mod Z]
```

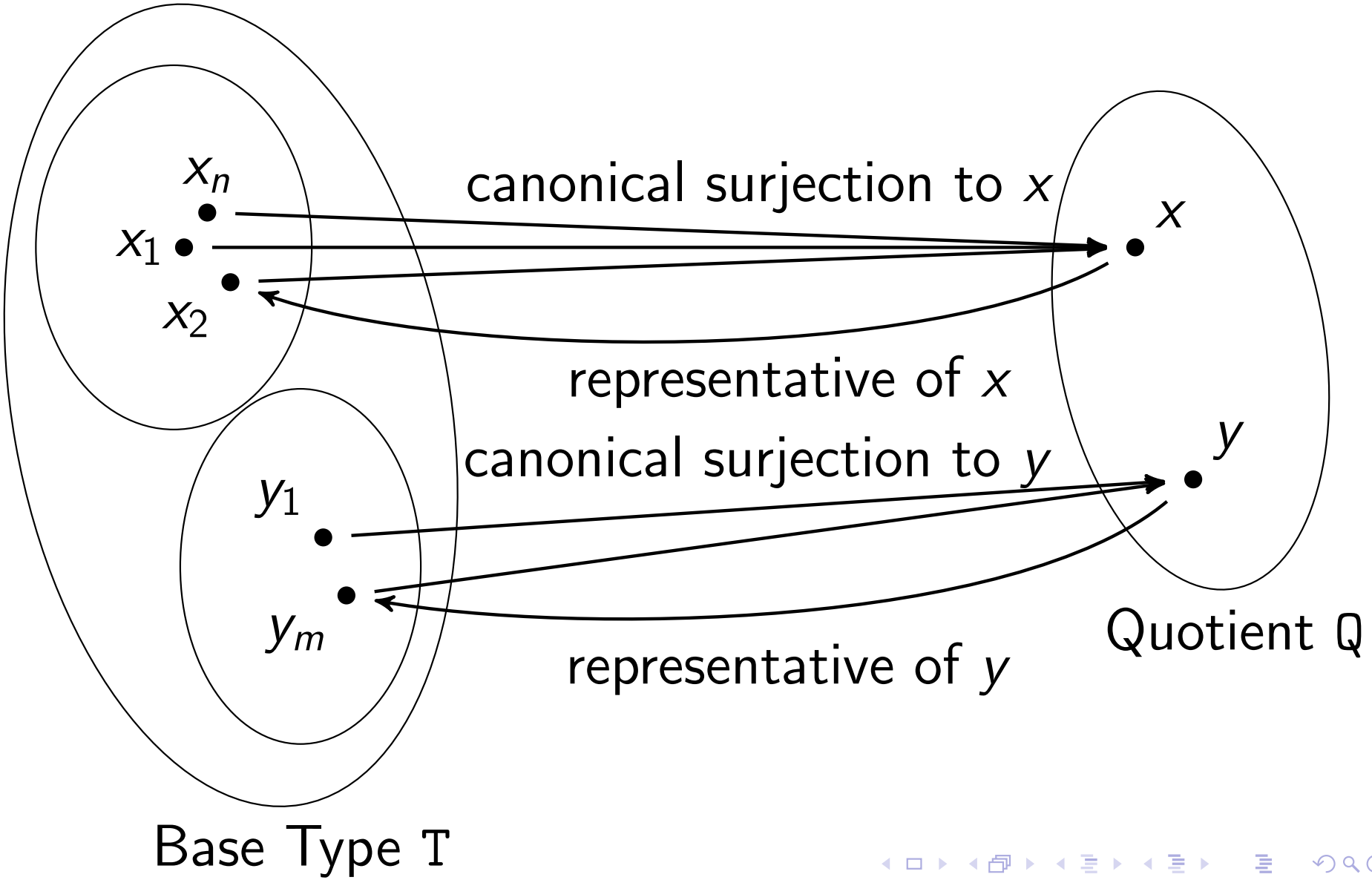
Low level operations and properties.

Equality modulo

```
Notation "x = y %[mod Q]" := \pi_Q x = \pi_Q y.
```

where `\pi_Q` is the canonical surjection in the quotient type `Q`.

Quotient primitives



Quotient interface

```
Record quot_class_of (T Q : Type) := QuotClass{
  repr : Q -> T;
  pi : T -> Q;
  reprK : forall x : Q, pi (repr x) = x
}.
Record quotType (T : Type) := QuotType {
  quot_sort :> Type;
  quot_class : quot_class_of T quot_sort
}
```

An instance of the quotient interface is called a *quotient structure*. (Altenkirch et al.: definable quotient)

Quotient theory vs. Quotient construction

Outline:

1. Quotient theory
   (embedding, lifting and automated translation)
2. Quotient construction
   (on choice types, on discrete algebraic structures)
3. Applications

Quotient theory vs. Quotient construction

Outline:

1. Quotient theory
   (embedding, lifting and automated translation)
2. Quotient construction
   (on choice types, on discrete algebraic structures)
3. Applications

Morphism lifting

Possible definition:

```
x +_Z y := \pi (repr x +_N*N repr y)
x <=_Z y := (repr x <=_N*N repr y)
```

Possible specification:

```
forall m n, \pi (m +_N*N n) = \pi m +_Z \pi n
forall m n, (\pi m <=_Z \pi n) = (m <=_N*N n)
```

(morphism properties)

Morphism lifting

Possible definition:

```
x +_Z y := \pi (repr x +_N*N repr y)
x <=_Z y := (repr x <=_N*N repr y)
```

Possible specification:

```
forall m n, \pi (m +_N*N n) = \pi m +_Z \pi n
forall m n, (\pi m <=_Z \pi n) = (m <=_N*N n)
```

(morphism properties)

```
forall m n, (m = n %[mod Z]) = (equivZ m n)
```

An abstraction layer

Operations and properties on the Quotient.

```
addZC : forall x y z : Z,
  x +Z (y +Z z) = (x +Z y) +Z z
```

```
addNNC : forall m n p : N * N,
  m +N*N (n +N*N p) = (m +N*N n) +N*N p %[mod Z]
```

Low level operations and properties.

Translation from Q to T

1. elimintation principle:

```
quotW : (forall a : T, P (\pi a)) ->
        forall x : Q, P x.
```

2. pushing \pi outwards,

```
\pi m + (\pi n + \pi p) =
(\pi m + \pi n) + \pi p
```

```
\pi (m + (n + p)) = \pi ((m + n) + p).
```

Translation from Q to T

1. elimintation principle:

```
quotW : (forall a : T, P (\pi a)) ->
        forall x : Q, P x.
```

2. pushing \pi outwards,

```
\pi m + (\pi n + \pi p) =
(\pi m + \pi n) + \pi p
```

```
m + (n + p) = (m + n) + p %[mod Z].
```

Automated translation

```
Record equal_to Q u := EqualTo
  {equal_val : Q; _ : u = equal_val}.
```

(`equal_to u`) is the type of all elements of `Q` that are equal to (`u : Q`).

Morphism specification

```
Lemma eq_to_addZ (m n : N * N)
  (x : equal_to (\pi m))
  (y : equal_to (\pi n)) :
  equal_to (\pi (m +_{N*N} n)) :=
EqualTo (x +_Z y)
        (_ : \pi (m +_{N*N} n) = x +_Z y).
```

Term inference

Example of term translation using

```
Lemma equalE (Q : Type) (u : Q)
    (m : equal_to u) : equal_val m = u.
```

```
on \pi m + (\pi n + \pi p).
```

Term inference

Example of term translation using

```
Lemma equalE (Q : Type) (u : Q)
    (m : equal_to u) : equal_val m = u.
```

on `\pi m + (\pi n + \pi p)`.

Unification problem

$$\texttt{\textbackslash pi m + (\textbackslash pi n + \textbackslash pi p)} \equiv \texttt{equal\_val} \; ?_{(\texttt{equal\_to ?})}$$

Unification flow

$$\texttt{\textbackslash pi m + (\textbackslash pi n + \textbackslash pi p)} \equiv \texttt{equal\_val ?}_{\texttt{(equal\_to ?)}}$$

Unification flow

$$\texttt{\textbackslash pi m + (\textbackslash pi n + \textbackslash pi p)} \equiv \texttt{equal\_val ?}_{\texttt{(equal\_to ?)}}$$

Canonical solution:

```
equal_to_addZ ?m ?n (?x : equal_to (\pi ?m))
                    (?y : equal_to (\pi ?n)) :
equal_to (\pi (?m + ?n))
```

Unification flow

```
\pi m + (\pi n + \pi p) ≡
 equal_val eq_to_addZ (equal_to (\pi (?m + ?n)))
```

Canonical solution:

```
equal_to_addZ ?m ?n (?x : equal_to (\pi ?m))
                    (?y : equal_to (\pi ?n)) :
equal_to (\pi (?m + ?n))
```

Unification flow

$$\texttt{\textbackslash pi m + (\textbackslash pi n + \textbackslash pi p)} \equiv \texttt{equal\_val eq\_to\_addZ}_{\texttt{(equal\_to (\textbackslash pi (?m + ?n)))}}$$

$$\texttt{\textbackslash pi m} \equiv \texttt{equal\_val ?x}_{\texttt{(equal\_to ?)}}$$
$$\texttt{\textbackslash pi n + \textbackslash pi p} \equiv \texttt{equal\_val ?y}_{\texttt{(equal\_to ?)}}$$

Unification flow

$$\texttt{\textbackslash pi m + (\textbackslash pi n + \textbackslash pi p)} \equiv \texttt{equal\_val eq\_to\_addZ}_{\texttt{(equal\_to (\textbackslash pi (m + ?n)))}}$$

$$\texttt{\textbackslash pi m} \equiv \texttt{equal\_val eq\_to\_pi}_{\texttt{(equal\_to (\textbackslash pi m))}}$$
$$\texttt{\textbackslash pi n + \textbackslash pi p} \equiv \texttt{equal\_val ?y}_{\texttt{(equal\_to ?)}}$$

Unification flow

```
\pi m + (\pi n + \pi p) ≡
 equal_val eq_to_addZ (equal_to (\pi (m + (?  + ?))))
```

```
\pi m ≡ equal_val eq_to_pi(equal_to (\pi m))
\pi n + \pi p ≡
 equal_val eq_to_addZ (equal_to (\pi (?  + ?)))
```

Unification flow

```
\pi m + (\pi n + \pi p) ≡
 equal_val eq_to_addZ (equal_to (\pi (m + (?  + ?))))
```

```
\pi m ≡ equal_val eq_to_pi(equal_to (\pi m))
\pi n + \pi p ≡
 equal_val eq_to_addZ (equal_to (\pi (?  + ?)))
```

```
\pi n ≡ equal_val ?(equal_to ?)
\pi p ≡ equal_val ?(equal_to ?)
```

Unification flow

```
\pi m + (\pi n + \pi p) ≡
 equal_val eq_to_addZ (equal_to (\pi (m + (n + p))))
```

```
\pi m ≡ equal_val eq_to_pi(equal_to (\pi m))
\pi n + \pi p ≡
 equal_val eq_to_addZ (equal_to (\pi (n + p)))
```

```
\pi n ≡ equal_val eq_to_pi(equal_to (\pi n))
\pi p ≡ equal_val eq_to_pi(equal_to (\pi p))
```

Term inference

Example of term translation using

```
Lemma equalE (Q : Type) (u : Q)
    (m : equal_to u) : equal_val m = u.
```

on `\pi m + (\pi n + \pi p)`.

Unification problem

$$\texttt{\textbackslash pi m + (\textbackslash pi n + \textbackslash pi p)} \equiv \texttt{equal\_val ?}_{\texttt{(equal\_to ?)}}$$

Solution

$$\texttt{\textbackslash pi m + (\textbackslash pi n + \textbackslash pi p)} \equiv \texttt{equal\_val eq\_to\_addZ}\ _{\texttt{(equal\_to (\textbackslash pi (m + (n + p))))}}$$

When not use quotients?

Quotient are not a replacement for setoids.

Example: associate polynomials.

```
Definition eqp p q := (p %| q) && (q %| p).
Notation "p %= q" := (eqp p q).
```

- Compatible with division `(_ %| _)` and `(_ %= _)`,
- compatible with multiplication,

When not use quotients?

Quotient are not a replacement for setoids.

Example: associate polynomials.

```
Definition eqp p q := (p %| q) && (q %| p).
Notation "p %= q" := (eqp p q).
```

- Compatible with division `(_ %| _)` and `(_ %= _)`,
- compatible with multiplication,
- **not compatible** with addition.

Quotient theory vs. Quotient construction

Outline:

1. Quotient theory
   (embedding, lifting and automated translation)
2. Quotient construction
   (on choice types, on discrete algebraic structures)
3. Applications

In an ideal system

An impredicative definition.

```
Definition is_class (C : T -> Prop) : Prop :=
  exists x, forall y, C y <-> R x y.

Definition Q : Type := {C : pred T | is_class C}
```

One element by equivalence class?

- First projection: functional extensionality.
- Second projection: proof irrelevance (for `Prop`).

Back to Coq $\leq$ 8.4, without axioms

Particular case:

- a decidable equivalence `equiv : T -> T -> bool`

and

- a countable type,

This suffices to select a unique element in each equivalence class.

Back to Coq $\leq$ 8.4, without axioms

Particular case:

- a decidable equivalence `equiv : T -> T -> bool`

and either

- a countable type,
- a choice type

This suffices to select a unique element in each equivalence class.

Back to Coq $\leq$ 8.4, without axioms

Particular case:

- a decidable equivalence `equiv : T -> T -> bool`

and either

- a countable type,
- a choice type
- a type encodable to a choice type.

This suffices to select a unique element in each equivalence class.

Quotient theory vs. Quotient construction

Outline:

1. Quotient theory
   (embedding, lifting and automated translation)
2. Quotient construction
   (on choice types, on discrete algebraic structures)
3. Applications

Applications

- Fraction field: $(D * D^*)/(\lambda x, y.\ x_1 y_2 \equiv y_1 x_2)$,
- Real algebraic numbers:

$$\left(\sum_{x:\mathbb{R}} \sum_{P:\mathbb{Q}[X]} (P(x) \equiv_{\mathbb{R}} 0) \right) / \equiv_{\mathbb{R}} .$$

- Multivariate polynomials
- Elliptic curves (Bartzia and Strub)
- Field extensions (O'Connor)
- Countable algebraic closure (Gonthier)

Related work

In intensional type theory:

- Hoffman (PhD Thesis),
- Chicli et al. (mathematical quotients in TT),
- Courtieu (normalized types),
- Voevodsky's UF and HoTT (cf HoTT book),
- Altenkirch et al. (definable quotients).

Elsewhere (Isabelle/HOL, ...)

Conclusion

- A framework to both use and construct quotients.
- Deals with quotients in discrete algebra.
- Successfully used in the Mathematial Components project.

Future work:

- Generalize to encompass quotients without `repr`.
- Elimination of quotient and `\pi` in one go?

Thank you for your attention.

HIT Quotients

From the HoTT book (homotopytypetheory.org/book/): the higher inductive type A/R generated by

- A function $q : A \to A/R$;
- For each $a, b : A$ such that $R(a, b)$, an equality $q(a) = q(b)$; and
- The 0-truncation constructor: for all $x, y : A/R$ and $r, s : x = y$, we have $r = s$.

Choice types

If a type `T` has a choice structure, there exists an operator

```
xchoose : forall P : T -> bool,
    (exists y : T, P y) -> T.
```

which given a proof of `exists y, P y`
returns an element `z`, such that `z` is the same if `xchoose` is given a proof of `exists y, Q y` when `P` and `Q` are logically equivalent.

Xchoosing a canonical representative

```
Lemma equiv_exists (x : T) :
  exists y, (equiv x) y.
```

`(equiv x)` is the equivalence class of `x`.

```
Definition canon (x : T) :=
  xchoose (equiv_exists x).
```

Choice of a unique element in `(equiv x)`.

Construction of the quotient

```
Record equiv_quot := EquivQuot {
  erepr : T;
  erepr_canon : canon erepr == erepr
}.
```

Choice types (sordid details)

Georges Gonthier's definition:

```
Record Choice.mixin_of T := Choice.Mixin {
  find : pred T -> nat -> option T;
  _ : forall P n x,
        find P n = Some x -> P x;
  _ : forall P : pred T,
        (exists x, P x) -> exists n, find P n;
  _ : forall P Q : pred T,
        P =1 Q -> find P =1 find Q
}.
```