Pragmatic Quotient Types in Coq

Cyril Cohen

Pragmatic Quotient Types in Coq

Cyril Cohen

Department of Computer Science and Engineering
University of Gothenburg
cyril.cohen@gu.se

Abstract. In intensional type theory, it is not always possible to form the quotient of a type by an equivalence relation. However, quotients are extremely useful when formalizing mathematics, especially in algebra. We provide a Coq library with a pragmatic approach in two complementary components. First, we provide a framework to work with quotient types in an axiomatic manner. Second, we program construction mechanisms for some specific cases where it is possible to build a quotient type. This library was helpful in implementing the types of rational fractions, multivariate polynomials, field extensions and real algebraic numbers.

Keywords: Quotient types, Formalization of mathematics, Coq

Introduction

In set-based mathematics, given some base set S and an equivalence $\equiv$, one may see the quotient $(S \,/\, \equiv)$ as the partition $\{\pi(x) \mid x \in S\}$ of S into the sets $\pi(x) \mathrel{\hat{=}} \{y \in S \mid x \equiv y\}$, which are called equivalence classes.

We distinguish several uses of quotients in the literature. On the one hand, we have structuring quotients, where the quotient can often be equipped with more structure than the base set. For instance, the quotient of pairs of integers to get rational numbers can be equipped with a field structure. Similarly, a quotient of the free algebra of terms generated by constants, variables, sums and products gives multivariate polynomials (*i.e.* polynomials with arbitrarily many variables). This kind of quotient is often left implicit in mathematical papers.

On the other hand, we have algebraic quotients, for which we can transfer the structure from the base set to the quotient. For instance, the quotient of a group by a normal subgroup or the quotient of a ring by an ideal belong to this category. For this kind of quotient, the structure on the base set and on the quotient set matter and the canonical surjection onto the quotient is a morphism for this structure.

In type theory, there are two known options to represent the notion of quotient. The first option is to consider quotients of setoids. A setoid is a type with an equivalence relation called setoid equality [1]. Now, quotienting a setoid amounts to changing the setoid equality to a broader one. However, we still consider elements from the base type, *i.e.* the type underlying both the base setoid

and the quotient setoid. This point of view is more the study of equivalence relations than the study of quotients. Moreover, although rewriting with setoid equality is supported by the system [2], it is still not as practical nor efficient as rewriting with Leibniz equality, because it must check the context (of the term to rewrite) commutes with the setoid equality.

The second option, and the one we focus on is to forge a quotient type, *i.e.* a type where each element represents one and only one equivalence class of the base type. This point of view leads to study the quotient as a new type, on which equality is the standard (Leibniz) equality of the system. In this framework, the equivalence that led to the quotient type is not a primitive notion. In Coq, the problem with quotient types is that there is no general way of forming them, without axioms [3].

In this paper, we do not focus on the theoretical problem of existence of quotients, but on the way to use them. We first describe our definition of a quotient interface in Section 1. We also show in which way it captures the desired properties of quotients and how we instrumented type inference to help the user to be concise. Surprisingly, this interface does not rely on an equivalence relation in its axiomatization, so we explain how we recover quotients by an equivalence relation from it in Section 2. We provide in Section 3 examples of applications of quotient types to rational fractions, multivariate polynomials, field extensions and real algebraic numbers. In Section 4, we compare our interface to previous designs of quotient types.

The Coq code for this framework and the examples are available at the following address: http://perso.crans.org/cohen/work/quotients/. We use the SSReflect tactic language [4] and the mathematical components project libraries [5].

1 A framework for quotients

Definition 1 (Quotient type).

A type `Q` *is a* quotient type *of a base type* `T` *if there are two functions* (`pi : T -> Q`) *and* (`repr : Q -> T`) *and a proof* `reprK`[1] *that* `pi` *is a left inverse for* `repr`*, i.e.* `forall x, pi (repr x) = x` *(see Figure 1).*

The function `pi` *is called the* canonical surjection *and we call the function* `repr` *the* representative function.

1.1 A small interface

The interface for quotients has a field for the quotient type, a field for the representative function `repr`, a field for the canonical surjection `pi` and a field for the axiom `reprK`, which says the representative function is a section of the canonical surjection. The definition of the quotient interface is split into two

[1] The name `reprK` comes from a standard convention in the SSReflect library to use the suffix "K" for cancellation lemmas.

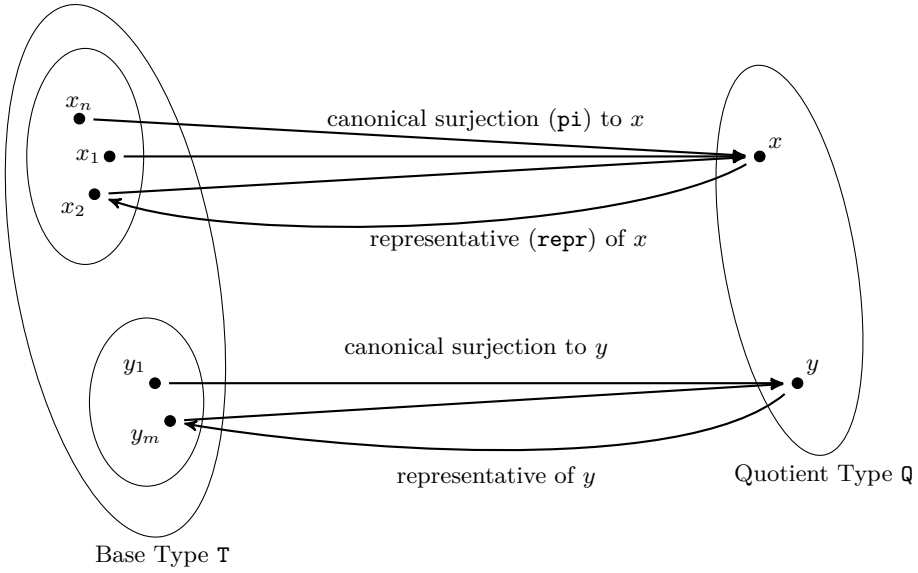


Fig. 1. Quotients without equivalence relation

parts, in the same way the interfaces from the SSReflect algebraic hierarchy are [6], in order to improve modularity.

```
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; (* quotient type *)
  quot_class : quot_class_of T quot_sort (* quotient class *)
}
```

Definition 2 (Quotient structure). *An instance of the quotient interface is called a* quotient structure.

Example 1. Let us define the datatype `int` of integers as the quotient of pairs of natural numbers by the diagonal. In other words, integers are the quotient of $\mathbb{N} \times \mathbb{N}$ by the equivalence relation

$$((n_1, n_2) \equiv (m_1, m_2)) \quad \hat{=} \quad (n_1 + m_2 = m_1 + n_2).$$

Now, we explicitly define the type of canonical representatives: pairs of natural numbers such that one of them is zero. For example, the integer zero is represented by $(0, 0)$, one by $(1, 0)$ and minus one by $(0, 1)$.

```
Definition int_axiom (z : nat * nat) := (z.1 == 0) || (z.2 == 0).
Definition int := {z | int_axiom z}.
```

We then define the particular instances `reprZ` of `repr` and `piZ` of `pi` and we show that `reprZ` is indeed a section of `piZ`:

```
Definition reprZ (n : int) : nat * nat := val n.

Lemma sub_int_axiom x y : int_axiom (x - y, y - x).

Definition piZ (x : nat * nat) : int :=
  exist _ (x.1 - x.2, x.2 - x.1) (sub_int_axiom _ _).

Lemma reprZK x : piZ (reprZ x) = x.
```

where `val` is the first projection of a Σ-type and where `exist` is the constructor of a Σ-type.

Now, we pack together `reprZ`, `piZ` and `reprZK` into the quotient class, and in the quotient structure `int_quotType`.

```
Definition int_quotClass := QuotClass reprZK.
Definition int_quotType := QuotType int int_quotClass.
```

We created a data type `int` which is the candidate for being a quotient, and a structure `int_quotType` which packs `int` together with the evidence that it is a quotient.

Remark 1. For various reasons we do not detail here, we selected none of the implementation of `int` from this paper to define the integer datatype `int` from SSReflect library [7]. Our examples use `int` only for the sake of simplicity and we provide a compilation of the ones about `int` in the file `quotint.v`. However, this framework is used in practice for more complicated instances (see Section 3).

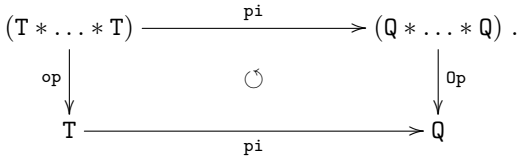
1.2 Recovering an equivalence and lifting properties

The existence of a quotient type `Q` with its quotient structure `qT` over the base type `T` induces naturally an equivalence over `T`: two elements `(x y : T)` are equivalent if `(pi qT x = pi qT y)`. We mimic the notation of the SSReflect library for equivalence in modular arithmetic on natural numbers, which gives us the following notation for equivalence of `x` and `y` of type `T` modulo the quotient structure `qT`:

```
Notation "x = y %[mod qT]" := (pi qT x = pi qT y).
```

We say an operator on `T` (*i.e.* a function which takes its arguments in `T` and outputs an element of `T`) is compatible with the quotient if given two lists of arguments which are pairwise equivalent, the two outputs are also equivalent. In other words, an operator is compatible with the quotient if it is constant on each equivalence class, up to equivalence.

When an operator `op` on `T` is compatible with the quotient, it has a *lifting*, which means there exists an operator `Op` on the quotient type `Q` such that following diagram commutes:



The canonical surjection is a morphism for this operator.

For example, a binary operator `(Op : Q -> Q -> Q)` is a lifting for the binary operator `(op : T -> T -> T)` with regard to the quotient structure `qT` as soon as the canonical surjection `(pi qT : T -> Q)` is a morphism for this operator:

```
forall x y, pi qT (op x y) = Op (pi qT x) (pi qT y) :> Q
```

which can be re-expressed in a standardized form for morphisms of binary operators in SSReflect:

```
{morph (pi qT) : x y / op x y >-> Op x y}
```

Example 2. Let us define the add operation on `int` as the lifting of the point-wise addition on pairs of natural numbers.

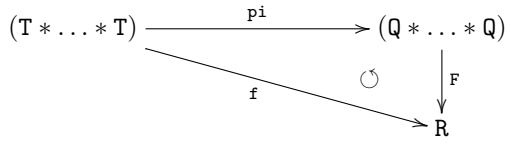
```
Definition add x y := (x.1 + y.1, x.2 + y.2).
Definition addz X Y := \pi_int (add (repr X) (repr Y)).
Lemma addz_compat : {morph \pi_int : x y / add x y >-> addz x y}.
```

Where the statement of `addz_compat` can be read as follows:

```
forall x y, \pi_int (add x y) = addz (\pi_int x) (\pi_int y).
```

and where `\pi_int` stands for `(pi int_quotType)` (see Example 4).

Similarly, given an arbitrary type `R`, we say that a function with values in `R` is compatible with the quotient if it is constant on each equivalence class. When a function `f` with arguments in `T` and values in `R` is compatible with the quotient, it has a *lifting*, which means there exists an operator `F` with arguments in the quotient type `Q` and values in `R` such that the following diagram commutes:



The canonical surjection is a morphism for this function.

For example, a binary function `(F : Q -> Q -> R)` is a lifting of a binary function `(f : T -> T -> R)` if:

```
forall x y, (f x y) = F (pi qT x) (pi qT y) :> R
```

or in a standardized form for morphisms of relations in SSReflect:

```
{mono (pi qT) : x y / f x y >-> F x y}
```

1.3 Inference of quotient structures

We recall that given a base type `T`, we say `Q` is a quotient type for `T` if there is a quotient structure `qT` and if the projection `(quot_sort qT)` is `Q`. In practice, given `x` in `Q` we want to be able to write `(repr x)`, but such a statement would be ill-typed.

Remark 2. The problem comes from the fact `repr` has an implicit argument which must have type `quotType`. The expanded form for `(repr x)` is `(@repr` $?_{\texttt{quotType}}$ `x)`, where `x` must have type `(quot_sort` $?_{\texttt{quotType}}$`)`. But if `x` has type `Q`, the type inference algorithm encounters the unification problem

$$(\texttt{quot\_sort}\ ?_{\texttt{quotType}}) \quad \equiv \quad \texttt{Q}$$

which it cannot solve without a hint, although we guess a solution is `qT`.

However, it is possible to make Coq type this statement anyway, by providing the information that the quotient structure `qT` is a *canonical structure* [8] for the quotient type `Q`. Registering a structure as canonical provide the unification mechanism a solution for the kind of equations we encounter.

Example 3. We make `int_quotType` the canonical quotient structure for the quotient type `int` by using the following Coq vernacular command:

```
Canonical Structure int_quotType.
```

Now, given `x` of type `int`, the system typechecks `(repr x)` as an element of `(nat * nat)`, as expected.

Since a quotient structure is canonically attached to every quotient type, we may also simplify the use of `pi`. Indeed, for now, `pi` has the following type.

```
forall (T : Type) (qT : quotType T), T -> quot_sort qT
```

Hence, `(pi qT)` has type `T -> Q`, but it is not possible to use `(pi Q)` to refer to this function. To circumvent this problem we provide a notation `\pi_Q` which gives exactly `(pi qT)` where `qT` is the canonical quotient structure attached to `Q` if it exists (otherwise, the notation fails). This notation uses a phantom type to let the system infer `qT` automatically [7].

Example 4. The canonical surjection function `\pi_int` from pairs of naturals to integers is in fact `(@pi int_quotType)` and has type `(nat * nat -> int)`.

We also adapted the notation for equivalence modulo quotient, so that we can provide `Q` instead of `qT`, as follows:

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

1.4 Automatic rewriting

When an operation (or a function) is compatible with the quotient, we can forge the lifting by hand by composing the initial operation with `pi` and `repr`. In this case the canonical surjection is indeed a morphism for the operator (cf the example of `addz` in Section 1.2).

Then we want to show elementary properties on the lifting, and those can often be derived from the properties of the initial operation. Thanks to the compatibility lemma, it is easy to go back and forth between the operation and its lifting by making `pi` and the operation commute.

Example 5. We show that zero is a neutral element on the right for addition:

```
Definition zeroz := \pi_int (0, 0).
Lemma add0z x : addz zeroz x = x.
```

Then, by rewriting backwards with `reprK`, the statement of `add0z` is equivalent to:

```
addz (\pi_int (0, 0)) (\pi_int (repr x)) = (\pi_int (repr x))
```

which can be solved by rewriting backwards with `addz_compat`.

However, when faced with more complex expressions involving lifted operators, it becomes more complicated to control where rewriting must happen. In order to save the user from the need to use a chain of rewriting rules of the form `[pattern]op_compat`, we introduce an automated mechanism to globally turn an expression on the quotient into an expression on the base type. For this, an operation `Op` (respectively a function `F`) which is a lifting has to be recognized automatically. We must register in some way that the value associated with `(Op (\pi_Q x)(\pi_Q y))` is `(\pi_Q (op x y))` (respectively, that the value associated with `(F (\pi_Q x)(\pi_Q y))` is `(f x y)`). For this purpose, we define a structure `equal_to u`, the type of all elements of `Q` that are equal to `(u : Q)`:

```
Record equal_to Q u := EqualTo {equal_val : Q; _ : u = equal_val}.
Notation "{pi a}" := (equal_to (\pi a)) : quotient_scope.
Lemma piE (Q : Type) (u : Q) (m : equal_to u) : equal_val m = u.
```

The type parameter `u` of the structure is the translation to infer, while the content of the field `equal_val` is the information present in the goal. We also introduce a notation `{pi a}` for the type of all elements of `Q` which are equal to `(\pi a)`. Rewriting with lemma `piE`, will trigger an instantiation of the left hand side of the equality, which will infer a value for `m`, and thus for its type, which contains `u`.

To declare an instance for an operator `op`, we must provide a lifted operator `Op` and a proof that given two elements that are the canonical surjections of `x` and `y`, it returns a value which is the canonical surjection of `(op x y)`.

```
Canonical Structure op_equal_to (x y : T) (qT : quotType T)
    (X : {pi x}) (Y : {pi y}) : {pi (op x y)} :=
  EqualTo (Op X Y) (_ : pi qT (op x y) = Op X Y).
```

We declare that any term of the form `(\pi x)` has a trivial `{pi x}` structure (where both `u` and `equal_val` are `(\pi x)`):

```
Canonical Structure equal_to_pi T (qT : quotType T)
  (x : T) : {pi x} := EqualTo (\pi x) (erefl _).
```

Example 6. For the addition in `int`, we can write:

```
Lemma addz_pi (x y : nat * nat) (X : {pi x}) (Y : {pi y}) :
  \pi_int (add x y) = addz (equal_val X) (equal_val Y).

Canonical Structure addz_equal_to (x y : nat * nat)
   (X : {pi x}) (Y : {pi y}) : {pi (add x y)} :=
  EqualTo (addz (equal_val X) (equal_val Y)) (addz_pi X Y).
```

By declaring `addz_equal_to` as canonical, we can now use `piE` to rewrite an expression of the form `(addz` $\tilde{x}$ $\tilde{y}$`))` into `(\pi_int (add x y))`, where $\tilde{x}$ and $\tilde{y}$ are arbitrarily complicated expressions that can be canonically recognized as elements of respectively `{pi x}` and `{pi y}`, i.e. as being canonically equal to respectively `(\pi_int x)` and `(\pi_int y)`.

In the examples above, we defined by hand the quotient `int` of pairs of natural numbers `nat * nat`. Moreover, the equivalence was left implicit in the code. We could expect a generic construction of this quotient by the equivalence relation mentioned in the first example. We now deal with this deficiency.

1.5 Recovering structure

This interface for quotient does not require the base type or the quotient type to be discrete (i.e. to have decidable equality), a choice type (see Section 2.1) or an algebraic structure. However, in the special case where the base type is discrete (respectively a choice type), we provide a mechanism to get the decidable equality structure (respectively the choice structure) on the quotient type.
Preserving structure through quotienting is more difficult to do for algebraic structures, but we provide such a support for quotients by an ideal, which we do not detail in this paper for the sake of space. Given a ring `R` with an ideal `I`, one can form the quotient `{ideal_quot I}` which has again a ring structure. Moreover, in that case, the canonical surjection `pi` is a ring morphism.

2 Quotient by an equivalence relation

Until now we have shown a quotient interface with no equivalence in its signature, and a notion of equivalence which is defined from the quotient. Now, we

explain how to get the quotient structure of a type by an equivalence relation. Given a type `T` and an equivalence relation `equiv`, we have to find a data type representing the quotient *i.e.* such that each element is an equivalence class.

A natural candidate to represent equivalence classes is the Σ-type of predicates that characterize a class. The elements of a given equivalence class are characterized by a predicate `P` that satisfies the following `is_class` property:

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

Thus, we could define the quotient as follows.

```
Definition quotient := {P : T -> Prop | is_class P}.
```

However, because Coq equality is intensional, two predicates which are extensionally equal (*i.e.* equal on every input) may not be intensionally equal, and also, the proof that a given predicate is a class is not unique either. Hence, in order for `quotient` to have only one element by equivalence class, it would suffice to have both propositional extensionality and proof irrelevance for the sort `Prop`.

However, this is not enough to make `quotient` a quotient type. We have enough information to build the canonical surjection `pi`, but we do not know how to select a representative element for each class.

2.1 Quotient of a choice structure

If a type `T` has a choice structure [6], 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.

Given a base type `T` equipped with a choice structure and a decidable equivalence relation `(equiv : T -> T -> bool)`, it becomes possible to build a quotient type. The construction is slightly more complicated than above.

For each class we can choose an element `x` in a canonical fashion, using the following `canon` function:

```
Lemma equiv_exists (x : T) : exists y, (equiv x) y.
Proof. by exists x; apply: equiv_refl. Qed.

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

We recall that `xchoose` takes a proof of existence of an element satisfying a predicate (here the predicate `(equiv x)`) and returns a witness which is unique, in the sense that two extensionally equal predicates lead to the same witness. This happens for example with the two predicates `(equiv x)` and `(equiv y)` when `x` and `y` are equivalent: the choice function will return the same element `z`

which will be equivalent both to `x` and `y`. Such a canonical element is a unique representative for its class.

Hence, the type formed with canonical elements can represent the quotient.

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

Indeed, thanks to Boolean proof irrelevance [9], the second field (`erepr_canon`) of such a structure is unique up to equality, which makes this Σ-type a sub-type.

The representative function is trivial as it is exactly the projection `erepr` on the first field of the Σ-type `equiv_quot`. However, more work is needed to build the canonical surjection. Indeed we first need to prove that `canon` is idempotent.

```
Lemma canon_id (x : T) : canon (canon x) == canon x.
Definition epi (x : T) := EquivQuot (canon x) (canon_id x).
```

Finally, we need to prove that the canonical surjection `epi` cancels the representative `erepr`:

```
Lemma ereprK (u : equiv_quot T) : epi (erepr u) = u.
```

The proof of `ereprK` relies on the proof irrelevance of Boolean predicates.

Proof. Two elements of `equiv_quot` are equal if and only if their first projection `erepr` are equal, because the second field `erepr_canon` of `equiv_quot` is a Boolean equality, and has only one proof. Thanks to this `(epi (erepr u))` and `u` are equal if and only if `(erepr (epi (erepr u))` is equal to `(erepr u)`. But by definition of `epi`, `(erepr (epi (erepr u))` is equal to `(canon (erepr u))`, and thanks to the property `(erepr_canon u)`, we get that `(canon (erepr u))` is equal to `(erepr u)`, which concludes the proof.

We then package everything into a quotient structure:

```
Definition equiv_quotClass := QuotClass ereprK
Canonical Structure equiv_quotType := QuotType equiv_quot
    equiv_quotClass.
```

We declare this structure as canonical, so that any quotient by an equivalence relation can be recognized as a canonical construction of quotient type.

However, we omitted to mention that the proof of `canon_id` and hence the code of `epi` requires a proof that `equiv` is indeed an equivalence relation. In order to avoid to add unbundled side conditions ensuring `equiv` is an equivalence relation, we define an interface for equivalence relations which coerces to binary relations:

```
Structure equiv_rel := EquivRelPack {
  equiv_fun :> rel T;
          _ : reflexive equiv
          _ : symmetric equiv
          _ : transitive equiv
}.
```

The whole development about `equiv_quot` takes `(equiv : equiv_rel T)` as a parameter.

Remark 3. Given an equivalence relation `equiv` on a choice type `T`, we introduced the notation `{eq_quot equiv}` to create a quotient by inferring both the equivalence structure of `equiv` and the choice structure of `T` and applying the latter construction.

We recall that we defined the equivalence induced by the quotient by saying `x` and `y` are equivalent if `(\pi_Q x = \pi_Q y)`, where `Q` is the quotient type. We refine the former notation for equivalence modulo `Q` to specialize it to quotients by equivalence, as follows.

```
Notation "x = y %[mod_eq equiv]" := (x = y %[mod {eq_quot equiv}]).
```

In the present situation, it seems natural that this induced equivalence coincides with the equivalence by which we quotiented.

```
Lemma eqmodP x y : reflect (x = y %[mod_eq equiv]) (equiv x y).
```

Example 7. Let us redefine once again `int` as the quotient of $\mathbb{N} \times \mathbb{N}$ by the equivalence relation $((n_1, n_2) \equiv (m_1, m_2))$ defined by $(n_1 + m_2 = m_1 + n_2)$.

In this second version, we directly perform the quotient by the relation, so we first define the equivalence relation.

```
Definition equivnn (x y : nat * nat) := x.1 + y.2 == y.1 + x.2.

Lemma equivnn_refl : reflexive equivnn.
Lemma equivnn_sym : symmetric equivnn.
Lemma equivnn_trans : transitive equivnn.

Canonical Structure equivnn_equiv : equiv_rel (nat * nat) :=
  EquivRel equivnn equivnn_refl equivnn_sym equivnn_trans.
```

Then `int` is just the quotient by this equivalence relation.

```
Definition int := {eq_quot equivnn}.
```

This type can be equipped with a quotient structure by repackaging the quotient class of `equiv_quotType equivnn_equiv` together with `int`.

2.2 Quotient of type with an explicit encoding to a choice type

In Section 3.4, we quotient by a decidable equivalence a type which is not a choice type, but has an explicit encoding to a choice type. Let us first define what we mean by explicit encoding, and then show how to adapt the construction of the quotient.

Definition 3 (Explicit encoding).
We say a type `T` with a equivalence relation `equivT` is explicitly encodable to a type `C` if there exists two functions `(T2C : T -> C)` and `(C2T : C -> T)` such that the following coding property *holds:*

```
forall x : T, equivT (C2T (T2C x)) x.
```

The function `T2C` *is called the* encoding *function because it codes an element of* `T` *into* `C`*. Conversely, the function* `C2T` *is called the* decoding *function.*

Remark 4. Here `T` can be seen as a setoid, and the coding property can be interpreted as: `C2T` is a left inverse of `T2C` in the setoid `T`. It expresses that encoded elements can be decoded properly.

There is no notion of equivalence on the coding type `C` yet, but we can provide one using the equivalence induced by `T`. Thus, we define `equivC` by composing `equivT` with `C2T`.

```
Definition equivC x y := equivT (C2T x) (C2T y).
```

When `equivT` is a Boolean relation, so is `equivC`. When `C` is a choice type and `equivC` is a decidable equivalence on this choice type, we can reproduce the exact same construction of quotient as in Section 2.1, so that we get `ereprC : equiv_quot -> C`, `epiC: C -> equiv_quot` and a proof of cancellation `ereprCK`. Now we can compose these operators with `T2C` and `C2T`.

```
Definition ereprT (x : equiv_quotient) : T := C2T (ereprC x).
Definition epiT (x : T) : equiv_quotient := epiC (T2C x).
```

And we can prove the cancellation lemma `ereprTK` using `ereprCK` and the coding property.

```
Lemma ereprTK (x : equiv_quotient) : epiT (ereprT x) = x.
```

Finally, we have everything we need to create a quotient type, like in Section 2.1.

3 Applications

3.1 Rational fractions

Given an integral domain D, i.e. a ring where $ab = 0 \Leftrightarrow a = 0 \vee b = 0$. One can build a field of rational fractions of D by quotienting $\{(x, y) \in D \times D \mid y \neq 0\}$ by the equivalence relation defined by:

$$(x, y) \equiv (x', y') := xy' = yx'$$

For example, rational numbers $\mathbb{Q}$ and the polynomial fractions $\mathbb{Q}(X)$ can be obtained through this construction.

In Coq, we first formalize the type `ratio` of pairs in a discrete ring, where the second element is non zero, and the above relation. Then, we prove it is an equivalence and quotient by it:

```
Inductive ratio := mkRatio { frac :> R * R; _ : frac.2 != 0 }.
Definition equivf x y := equivf_def (x.1 * y.2 == x.2 * y.1).
Canonical Structure equivf_equiv : equiv_rel.
Definition type := {eq_quot equivf}.
```

Remark 5. First restricting the domain to "ratios" and then quotienting is our way to deal with the partiality of the equivalence relation between pairs.

We then recover the decidability of the equality and the choice structure for free, through the quotient.[2] However, the ring and field structures cannot be derived automatically from the ring structure of $R \times R$ and have to be proven separately.

This development on rational fractions has been used in a construction of elliptic curves by Bartzia and Strub [10].

3.2 Multivariate polynomials

The goal is to have a type to represent polynomials with an arbitrary number of indeterminates. From this we could start formalizing the theory of symmetric polynomials. Possible applications are another Coq proof of the Fundamental Theorem of Algebra [11] or the study of generic polynomials in Galois Theory.

We provide this construction for a discrete ring of coefficients R and a countable type of indeterminates. We form the quotient of the free algebra generated by constants in R, addition, multiplication and indeterminates (X_n), which we quotient by the relation defined by $t_1 \equiv t_2$ if t_1 and t_2 represent the same univariate polynomial in $(\dots(R[X_1])[X_2]\dots)[X_n]$ where n is big enough [12].

In order to know if two terms represent the same univariate polynomial in $(\dots(R[X_1])[X_2]\dots)[X_n]$ with n fixed, we iterate the univariate polynomial construction `{poly R}` from the SSReflect library.

3.3 Field extensions

The proof of the Feit-Thompson Theorem relies on Galois theory and on a construction of the algebraic closure for countable fields (including finite fields). Both these constructions involve building a field extension through quotienting, but in two different ways. Although we did not develop the formal proof of these constructions, we briefly mention how quotients were used.

Galois theory studies the intermediate fields between a base field F and a given ambient field extension L. In this context, we build the field extension of F generated by the root $z \in L$ of a polynomials $P \in F[X]$ of degree n. It amounts in fact to building a subfield $F[z]$ of dimension n between F and L. This is done by quotienting the space $K_n[X]$ of polynomials with coefficients in K of degree at most n by the equivalence relation defined by $P \equiv Q := (P(z) = Q(z))$.

To build the algebraic closure of a *countable* field F, one can iterate the construction of a field extension by all irreducible polynomials of $F[X]$. Each iteration consists in building a maximal ideal and quotienting by it, which happens to be constructive because we have countably many polynomials.

[2] As there are two possible equality operators available (one obtained from the equality of the base type, the other from the equivalence relation), we must be very careful to let the user choose the one he wants, once and for all.

3.4 Real algebraic numbers

Finally, we used a quotient with an explicit encoding to a choice type (see Section 2.2) to build the real closed field of real algebraic numbers [13,7].

4 Related work on quotient types

Given a base type `(T : Type)` and an equivalence `(equiv : T -> T -> Prop)`, the Coq interface below is due to Laurent Chicli, Loïc Pottier and Carlos Simpson [3], following studies from Martin Hofmann [14]. It sums up the desired properties of a the quotient type: its existence, a surjection from the base type `T` to it, and a way to lift to the quotient functions that are compatible with the equivalence `equiv`.

```
Record type_quotient (T : Type) (equiv : T -> T -> Prop)
  (Hequiv : equivalence equiv) := {
  quo :> Type;
  class :> T -> quo;
  quo_comp : forall (x y : T), equiv x y -> class x = class y;
  quo_comp_rev : forall (x y : T), class x = class y -> equiv x y;
  quo_lift : forall (R : Type) (f : T -> R),
      compatible equiv f -> quo -> R;
  quo_lift_prop :
      forall (R : Type) (f : T -> R) (Hf : compatible equiv f),
      forall (x : T), (quot_lift Hf \o class) x = f x;
  quo_surj : forall (c : quo), exists x : T, c = class x
}.
```

where `\o` is the infix notation for functional composition and where `equivalence` and `compatible` are predicates meaning respectively that a relation is an equivalence (reflexive, symmetric and transitive) and that a function is constant on each equivalence class. We believe[3] they are defined as below:

```
Definition equivalence (T : Type) (equiv : T -> T -> Prop) :=
  reflexive equiv /\ symmetric equiv /\ transitive equiv.
Definition compatible (T R : Type) (equiv : T -> T -> Prop)
   (f : T -> R) := forall x y : T, equiv x y -> f x = f y.
```

Once this `type_quotient` defined, they [3] add the existence of the quotient as an axiom.

```
Axiom quotient : forall (T : Type) (equiv : T -> T -> Prop)
   (p:equivalence R), (type_quotient p).
```

Although this axiom is not provable in the type theory of Coq, its consistency with the Calculus of Constructions has been proved in [14]. The construction of this interface was made in order to study the type theory of Coq augmented

[3] No definitions for `equivalence` or `compatible` are explicitly given in [3].

with quotient types. This is not our objective at all. First, we want to keep the theory of Coq without modification, so quotient types do not exist in general. Second, we create an interface to provide practical tools to handle quotients types that do exist.

The reader may notice that here the field `quo` plays the role of our `quot_sort` and `class` the role of `pi` of our interface. The combination of `repr` and `reprK` is a skolemized version of `quo_surj`.

Remark 6. This is not exactly the case, because `quot_surj` is a `Prop` existential, which unlike existentials in `Type` cannot be extracted to a function `repr` which has the property `reprK`. This was already observed [3] in the study of the consistency of Coq with variants of `type_quotient`.

However, the parameters about `equiv` and properties about the lifting of morphism disappear completely in our interface, because they can all be encoded as explained in Section 1.2.

Example 8. For example, `quo_lift` can be encoded like this:

```
Definition new_quo_lift (T R : Type) (qT : quotType T)
  (f : T -> R) (x : Q) := f (repr x)
```

Note that the precondition `(compatible equiv f)` was not needed to define the lifting `new_quo_lift`. Only the property `quo_lift_prop` still needs the precondition.

Our approach can also be compared to Normalized Types [15]. The function `pi` can be seen as a user defined normalization function inside Coq.

Conclusion

This framework for quotient types is not a substitute for setoids and especially not for setoid rewriting. Indeed, it is not designed to make it easy to rewrite modulo equivalence, but to rewrite directly using equality inside the quotient type. Quotient types can mimic to some extend the behaviour of quotients in set-based mathematics.

This framework has already been useful in various non trivial examples. It handles quotients by ideals in the sense of the SSReflect library. Also, a natural continuation would be to study quotients of vector spaces, algebras and modules.

Also, in the SSReflect library, quotients of finite groups [16] are handled separately, taking advantage of the support for finite types. Maybe they could benefit from a connection with this more generic form of quotient.

Finally, it seems that this framework could work to quotient lambda-terms modulo alpha-equivalence. I did not attempt to do this construction, but it seems worth a try.

Acknowledgements

I would like to thank early experimenters of my framework, namely Pierre-Yves Strub and Russell O'Connor, who dared using it in real life formalizations and reported the problems they encountered. I also thank Georges Gonthier, for even earlier discussions on quotient types, thank to which I initially learned my way through the use of Canonical Structures in the SSReflect library. I also wish to thank the anonymous reviewers for their constructive feedback.

References

1. Barthe, G., Capretta, V., Pons, O.: Setoids in type theory. Journal of Functional Programming **13**(2) (2003) 261–293 Special Issue on Logical Frameworks and Metalanguages.
2. Sozeau, M.: A New Look at Generalized Rewriting in Type Theory. Journal of Formalized Reasoning **2**(1) (December 2009) 41–62
3. Chicli, L., Pottier, L., Simpson, C.: Mathematical quotients and quotient types in Coq. In Geuvers, H., Wiedijk, F., eds.: Types for Proofs and Programs. Number 2646 in LNCS, Springer (2003) 95–107
4. Gonthier, G., Mahboubi, A., Tassi, E.: A small scale reflection extension for the Coq system. INRIA Technical report, http://hal.inria.fr/inria-00258384.
5. Project, T.M.C.: SSReflect extension and libraries. http://www.msr-inria.inria.fr/Projects/math-components/index_html
6. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging Mathematical Structures. In Berghofer, S., Nipkow, T., Urban, C., Wenzel, M., eds.: Theorem Proving in Higher Order Logics, TPHOLs 2009 proceedings. Volume 5674 of Lecture Notes in Computer Science., Springer (2009) 327–342
7. Cohen, C.: Formalized algebraic numbers: construction and first order theory. PhD thesis, École polytechnique (2012)
8. Saibi, A.: Typing algorithm in type theory with inheritance. In: Principles of Programming Languages, POPL 1997 proceedings. (1997) 292–301
9. Hedberg, M.: A coherence theorem for Martin-Löf's type theory. Journal of Functional Programming (1998) 4–8
10. Bartzia, I., Strub, P.Y.: A formalization of elliptic curves. http://pierre-yves.strub.nu/research/ec/ (2011)
11. Cohen, C., Coquand, T.: A constructive version of laplace's proof on the existence of complex roots. Journal of Algebra **381**(0) (May 2013) 110 – 115
12. Cohen, C.: Reasoning about big enough numbers in Coq. http://perso.crans.org/cohen/work/bigenough/ (2012)
13. Cohen, C.: Construction of real algebraic numbers in Coq. In Beringer, L., Felty, A., eds.: ITP - 3rd International Conference on Interactive Theorem Proving - 2012, Princeton, United States, Springer (August 2012)
14. Hofmann, M.: Extensional concepts in intensional type theory. Phd thesis, University of Edinburgh (1995)
15. Courtieu, P.: Normalized types. In: Proceedings of CSL2001. Volume 2142 of Lecture Notes in Computer Science. (2001)
16. Gonthier, G., Mahboubi, A., Rideau, L., Tassi, E., Théry, L.: A Modular Formalisation of Finite Group Theory. Rapport de recherche RR-6156, INRIA (2007)