Quotients dans le CCI

Loïc Pottier

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Quotients dans le CCI

Loïc Pottier

N° 4053

Novembre 2000

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4053--FR

INRIA
SOPHIA ANTIPOLIS

Quotients dans le CCI

Loïc Pottier

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lemme

Rapport de recherche n° 4053 — Novembre 2000 — 16 pages

Résumé : On étudie plusieurs voies pour formaliser les quotients dans le Calcul de Constructions Inductive: les quotients naïfs (en utilisant les travaux de [LW99] et [BB96], on montre qu´ils conduisent à une incohérence), les quotients décidables, les quotients classiques de la théorie des types (comme ceux étudiés dans [Hof95]), et finallement on propose une notion de quotients fonctionnels qui résolvent les problèmes des précédentes notions, et semblent satisfaisants théoriquement et pratiquement. Pour chaque notion introduite au cours du papier, on donne la traduction correspondante dans le système `Coq`[pro99].

Mots-clés : théorie des types, quotients, Coq, CCI

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Quotients in the CIC

Abstract: To formalize quotients sets in the Calculus of Inductive Constructions, we study several ways: naive quotients (using works of [LW99] and [BB96] we show that they lead to a contradiction), decidable quotients, classical quotients of type theory (as studied in [Hof95]), and finally propose a notion of functionnal quotients which solves problems of previous notions of quotient, and seems to be convenient theoretically and practically. For all introduced notions during the paper, we give the corresponding traduction in the system `Coq`[pro99].

Key-words: type theory, quotients, Coq, CIC

1 Introduction

We explore some ways to introduce quotients into the Calculus of Inductive Constructions (CIC)[CH88] [pro99], without extending its formalism. We will not undertake here an exhaustive theoretical study in type theory of the notion of quotient and connected notions (extentionnality, subtypes), not more than we will give a foundation to these notions by syntactic models. This work was partially carried out by several authors, and the interested reader can consult [Hof95], [Bou97] or [Bar95], for example.

Our goal is to show that with reasonable assumptions in mathematical practice, the notion of quotient, which hardly coexists with constructive notions, can however be adapted in order to be usable and enough powerful to develop reasonably advanced mathematics in the formalism of the CIC.

In order to avoid possible ambiguities, our development will be translated on the fly in the system Coq[1].

This paper is organized as follow.

We show in the section 2 that if one naively translates the notion of quotient into type theory (" naive quotients "), one gets to a contradiction. We use for that works of [LW99] and [BB96].

We then study in the section 3 a restriction of the naive quotients, the " decidable quotients ", which avoids their inconsistencies, and which is sufficiently powerful to formalize basic constructions like subtypes, parts of a set, canonical forms or functional spaces. But this restriction seems too strong, and in this case consistency is far from being obvious.

In the section 4, we are interested in the " traditional " quotients of the literature, in particular those studied by [Hof95], who shows their consistency by syntactic models. As noticed by M.Hofmann in the same reference, it will be seen that these quotients cannot treate an elementary construction of quotients of functional spaces of [Bou70].

In the section 5, we complete the traditional quotients to propose a notion of " functional quotients " which behaves well with spaces of functions. Although we do not show it here, it seems that the preceding syntactic models are still appropriate, then ensuring the consistency of these functional quotients.

We show as an example how they can be used in `Coq` to build subtypes, with a rather concise syntax.

Lastly, we conclude, comparing this work with other approaches using only setoids.

2 Naive quotients

It is shown here that a naive transposition of the mathematical notion of quotient in the CIC leads to an incoherent situation.

1. definitions, axioms and theorems, but not the proofs, which are hard to understand in `Coq` - scripts or λ-terms -. The complete `Coq` code can however be consulted by the interested reader at the following URL: `ftp://ftp-sop.inria.fr/lemme/Loic.Pottier/quotients.tar.gz`.

2.1 Definition

The mathematical practice wants that a quotient set is given by a set and an equivalence relation: its elements are the equivalence classes of the relation. One has the canonical surjection which maps an element to its equivalence class. Although it is not explicitly formulated in for example [Bou70], it is also natural, when one speaks about an equivalence class, to give oneself an element of this class. Indeed, one seldom says "let c be an equivalence class ", but rather "let $\bar{x}$ be a class" where x is an element of this class. Just like it is natural in mathematics, when one has a surjective mapping f, to be given, if one needs some, a section s of this map (i.e. $f\ o\ s = id$). In the case of quotients, this leads to be given a function which maps each equivalence class to one of its representatives.

Let us transpose all that in type theory: being given a type A and an equivalence relation R on this type, one gives oneself a type Q, two functions $class$ and $choice$, with respective types $A \to Q$ and $Q \to A$, such that Q with Leibnitz equality[2] is isomorphic to A with R:

$$\forall x : Q,\ class(choice(x)) == x$$

$$\forall x : A,\ choice(class(x))\ R\ x$$

$$\forall x,y : A,\ x\ R\ y \Rightarrow class(x) == class(y)$$

In Coq, this becomes:

```
Record  type_quotient [A : Type; R : (Relation A);
                       Prf_equiv : (Equivalence A R)] : Type :=  {
   quo:> Type;
   class: A -> quo;
   choice: quo -> A;
   quo_comp: (x, y : A) (R x y) -> (class x) == (class y);
   quo_idset: (x : A)  (R (choice (class x)) x);
   quo_idQ: (x : quo)  (class (choice x)) == x}.

Axiom naive_quotient:
   (A : Type) (R : (Relation A)) (Prf_equiv : (Equivalence A R))
   (type_quotient Prf_equiv).
```

Addition of this axiom[3], which makes it possible to create new types, is not without consequences. We will see that it can introduce an inconsistency. First of all, we are in position to prove some alternatives of the excluded middle. The method which will follow is drawn from [LW99] (which takes as a starting point a proof of Goodman and Myhill), according to a suggestion of B.Werner.

2.2 Excluded middle in the sorts *Prop*, *Set* and *Type*

Let us prove first the excluded middle in the sort *Prop* of Coq:

$$\forall P : Prop,\ P \vee \neg P$$

2. noted ==
3. that we will call *axiom of the naive quotients*

Let P be a proposition. On the type $bool = \{true, false\}$, one defines a relation R by $x\ R\ y \Leftrightarrow x == y \vee P$. R is clearly an equivalence relation, with only one class if P is true, and two if not. Now let Q be the quotient of *bool* by R, *class* and *choice* the associated maps. It is seen that then $choice(class(true)) == choice(class(false))$ iff P is true. As in *bool* we can decide if two elements are equal, we thus obtains $P \vee \neg P$.

In Coq this proof becomes:

```
Variable P:Prop.
Definition eqb := [x, y : bool]  x == y \/ P.
Lemma eqb_equiv: (Equivalence bool eqb).
Definition B := (naive_quotient eqb_equiv).
Lemma eqb_dec: (x, y : bool)  {x == y}+{~ x == y}.
Lemma middle: P \/ ~ P.
Case (eqb_dec (choice (class B true)) (choice (class B false))).
...Qed.
```

With the same method, we can show in Coq:

```
Lemma middle_dec:(P:Prop) {P}+{~ P}.
```

which means that every proposition is decidable (since the result is in the sort *Set*, constructive in the CIC).

In the same way, if one adapts the definition of the inductive type *sumbool* of Coq to the sort $Type$, one obtains an excluded middle whose result is in $Type$.

2.3 Proof-irrelevance and inconsistency

In [BB96], it is shown that in the sort *Prop* the excluded middle together with the choice make it possible to show *proof-irrelevance*, in other words that all the proofs of a proposition are equal:

$$\forall P : Prop, \forall p,q : P,\ p == q$$

Thanks to the excluded middle in *Set* obtained by the axiom of the naive quotients: `middle_dec:(P:Prop) {P}+{ P}`, and thanks to the fact that the sort *Set* is also impredicative we will be able to adapt their proof to the sort *Set*.

First of all the axiom of choice becomes given by following projections of the type *sig* of Coq:

```
Definition ex_element :=
   [A : Set] [P : A -> Prop]  (proj1_sig A [x : A]  (P x)).
Definition ex_proof := [A : Set] [P : A -> Prop]
          [h : {x : A | (P x)}](proj2_sig A [x : A]  (P x) h).
```

What is essential in the proof of Barbanera and Berardi is the fact that *Prop* is impredicative, which makes it possible to build there a version of Russell's paradox. Initially, using $middle_dec$, one defines a function $IfThenElse$ on the propositions:

```
Section ifthenelse.
Variable A:Prop.
Variable C:Set.
```

```
Variables c1, c2 : C.
Definition IfAux: {x : C | x == c1 /\ A \/ x == c2 /\ ~ A}.
...Defined.
Definition IfThenElse :=
   (!ex_element C [x : C]  x == c1 /\ A \/ x == c2 /\ ~ A IfAux).
End ifthenelse.
```

Now let *Ensemble* be a set, *ensemble*1 and *ensemble*2 be two of its elements. We will show that they are equal. The central step consists in building a version of Russell's paradox, in fact a set U which will contain the set of its parts (power set). The following construction *PowerSet* plays the role of the set of parts:

```
Variable Ensemble:Set.
Variables element1, element2 : Ensemble.
Definition PowerSet := [A : Set] A -> Ensemble.
```

The set U, which maps to its power set:

```
Definition U := (X : Set)  (PowerSet X).
Definition projU : U -> (PowerSet U) := [u : U]  (u U).
```

Let us note that the term $(u\ U)$ exists because the sort *Set* is impredicative in the CIC: U is itself a set. The reverse injection is more difficult to define. One starts by defining a notion of retraction:

```
Definition Id := [A : Set] [x : A]  x.
Definition comp :=
   [A, B, C : Set] [h : B -> A] [g : A -> B] [x : A]  (h (g x)).

Definition e_p_pair :=
   [A, B : Set] [g : A -> B] [h : B -> A]  (!comp A B A h g) == (!Id A).
```

The following property[4] states that if $X == U$ then $(Powerset\ X)$ is a retraction of $(Powerset\ U)$, which is false from an intuitionist point of view, but which is possible here using the excluded middle:

```
Definition t': (X : Set) {g : (PowerSet X) -> (PowerSet U)
      & {h : (PowerSet U) -> (PowerSet X) | X == U -> (e_p_pair g h)}}.
```

The two functions of this retraction, composed, give the sought injection of $(Powerset\ U)$ in U:

```
Definition phi :=
   [X : Set]
   (projS1 (PowerSet X) -> (PowerSet U) [g : (PowerSet X) -> (PowerSet U)]
      {h : (PowerSet U) -> (PowerSet X) | X == U -> (e_p_pair g h)} (t' X)).
Definition psi := [X : Set](proj1_sig ?
```

4. to prove it,in order to be able to make a rewriting on objects of the sort *Set*, it is necessary to add the following axiom: `Axiom eqT_rec : (A : Type) (x : A) (P : A ->Set) (P x) ->(y : A) x == y ->(P y)`. This constant could be defined in Coq: `eqT_rec = [A:Type; x:A; P:(A->Set); f:(P x); y:A; e:(x==y)] <P>Cases e of refl_eqT =>f end` . Indeed *eqT* is defined as an inductive type with only one constructor, and thus the elimination of this type could be authorized for the sort *Set* as it is for the sort *Prop* (it could certainly also be authorized for the sort *Type*). But it is not the case, primarily to preserve the method of extraction of Coq: the fact of adding the constant *eqT_rec* to the CIC (or of accepting its definition well) does not introduce inconsistency into the CIC itself ([fol00])

```
     [h] X == U -> (e_p_pair (!phi X) h)
      (projS2 ? [g]  {h : (PowerSet U) -> (PowerSet X)
                     | X == U -> (e_p_pair g h)}     (t' X))).
Definition injU := [f : (PowerSet U)] [X : Set]  (!psi X (!phi U f)).
```

We verify that the power set of U can be seen as a subset of U, which is the basis of Russel's paradox:

```
Lemma injUprojU: (!comp (PowerSet U) U (PowerSet U) projU injU)
                  == (!Id (PowerSet U)).
Theorem PU_is_retract_U: {g : (PowerSet U) -> U& {h : U -> (PowerSet U) |
                          (e_p_pair g h)}}.
```

To finish Russel's paradox, it is enough to build, using the preceding retraction, the part of U made from elements of U which are not contained in themselves:

```
Definition belongs := [u, v : U]  ((projU v) u) == element1.
Definition RussellClass := [u : U]
    (IfThenElse ~(belongs u u) element1 element2).
Definition r : U := (injU RussellClass).
Lemma proof_irrelevance_set: element2 == element1.
Generalize injUprojU.
...Case (middle_dec (belongs r r))....Qed.
```

We thus proved $\forall A : Set, \forall x,y : A, x == y$. In the CIC, this leads to an inconsistency:

```
Lemma incoherence:False:=
<False> Cases (<[b:bool](if b then True else False)>
         Cases (proof_irrelevance_set bool true false) of refl_eqT => I end)
         of end.
```

In conclusion, the axiom of the naive quotients, helped of the constant eqT_rec, lead to a contradiction in the sort Set.

Let use note that because we use the impredicativity of Set, in the sort $Type$ the proof does not work any more, because the Russel's paradox cannot be formalized.

2.4 Computable canonical form

The existence of naive quotients leads to the existence of an effective canonical form for any equivalence relation, computed by the function $f = choice\ o\ class$. We have $x\ R\ y \Leftrightarrow f(x) == f(y)$. We can deduce from this that if Leibnitz equality is decidable on E, then every equivalence relation on E is also decidable.

2.5 Conclusion

To introduce quotients without precautions into the CIC thus leads to excluded middles in the three sorts $Prop$, Set and $Type$. In the sort $Prop$ that leads to proof-irrelevance, which is mathematically reasonable. On the other hand, in the sort Set, one obtains an inconsistency.

A solution would be to eliminate the sort *Set* from the CIC. It is tempting, more especially as its principal justification seems to be the current method of extraction of `Coq`. Other methods of extraction exist which are not based on *Set*, for example [Pot00].

Lastly, the existence of naive quotients provides a function computing canonical forms to any relation of equivalence, and makes it decidable as soon as the Leibnitz equality is. Which is exaggerated.

3 Decidable quotients

3.1 Definition

An inexpensive way and which seems reasonable to limit the naive quotients to avoid the nuisances that they cause is to require that when the Leibnitz equality is decidable, the equivalence relation by which one quotients is also decidable (one will call this condition *condition of decidability*):

```
Definition is_decidable:= [E: Type] [R: (Relation E)]
(x, y: E) (R x y)+  (R x y).
```

We define in `Coq` decidable quotients by the same definition as for naive quotients, except that we add the condition of decidability as a parameter. Let us note that the existence of naive quotients implies that of decidable quotients, since then an equivalence relation is decidable as soon as the Leibnitz equality is. Opposite not being obviously true.

Proofs of excluded middle inspired from the proof of Goodman and Myhill do not work now any more: on *bool*, one will be able to quotient by the relation $x\ R\ y \Leftrightarrow x == y \vee P$ only if P is decidable (and it is precisely what one obtained with the naive quotients). One avoids the preceding nuisances then, mainly the inconsistency in *Set*. Good news. It remains to see whether the restriction brought is not too constraining. One will see now that several significant cases in practice can be treated with the decidable quotients.

Non decidable Leibnitz equality. If the Leibnitz equality of a type E is not decidable, then one can quotient by any equivalence relation on E, the condition of decidablity being satisfied.

Computable canonical forms. Let us suppose that on a type E the relation R is defined by

$$x\ R\ y \Leftrightarrow f(x) == f(y)$$

where f is a function from E to E. The function f then calculates a canonical form for R for each element of E. In this case the condition of decidability is easily verified, and we can thus build a quotient. One can notice that the existence of a decidable quotient for an equivalence relation leads to the existence of an effective canonical form, as that was the case with the naive quotients. These two notions, canonical forms and condition of decidability are thus in fact equivalent from the point of view of the decidable quotients.

Relation equivalent to equality. If the equivalence relation implies the Leibnitz equality, the condition of decidability is true. This is the case when one has a property of extentionnality, for example in the three following cases.

Parts of a set. A part of a type E is a predicate on E. Two parts having the same elements are equal:$(\forall x : E,\ P(x) \Leftrightarrow Q(x)) \Rightarrow P == Q$, which is a form of extentionnality on the predicates:[5]. One can then build the type of the parts of a type E, like quotient of the type $E \to Prop$ by eqP.

Extentionality of functions. We suppose now that two functions whose values are equal in each point, are equal:$(\forall x, f(x) == g(x)) \Rightarrow f == g$. One can then form the quotient of $E \to F$.

Subtypes. We can introduce subtypes as quotients. Being given a type E and a predicate on E, one can consider the subtype of E made up of the elements of E which verify P. A way of representing this set is to take the couples (x,p) where x is in E and p is a proof of $P(x)$. Naturally one will identify two couples with the same left component. The condition of decidability does not have any reason to be verified for all E and P. A way which seems reasonable to make it possible to carry out is to suppose proof-irrelevance $\forall P : Prop,\ \forall p,q : P,\ p == q$. In this case one can show the condition of decidability. We then have a notion of subtype of a type, as quotient by identification of proofs.

3.2 Conclusion

We have seen that the naive quotients lead to an inconsistency of the CIC (modulo the addition of eqT_rec). The decidable quotients avoid this problem, and have a certain power of representation (parts of a set, extentionnality of functions, canonical forms, subtypes). But we did not give any model for them, which would prove their consistency. *A priori*, the axiom of the decidable quotients could cause, following the example of the axiom of the naive quotients, an inconsistency of the CIC. In addition, with the decidable quotients, one can quotient in a type with decidable equality only by a decidable equivalence relation, which can be too restrictive.

4 "Classical" quotients

We will be interested now in the notion of quotient studied by M.Hofmann in [Hof95]. It has the advantage of having a model (that of the *setoids*), and to be rather categorical. Let E be a type and R an equivalence relation on E, we give ourselves a type Q, a function $class$ of type $E \to Q$, compatible with R. We give also a way to lift in Q any function from E to any type which is constant on the equivalence classes of R. Finally, to ensure that $class$ is surjective, we give an induction principle on Q which says that to show that a property is true for all elements of Q, it is enough to show it for all the images of elements of E by the function $class$. In `Coq`, that gives:

```
Record  type_quotient [E : Type; R : (Relation E);
                Prf_equiv : (Equivalence E R)] : Type :=  {
   quo:> Type;
   class: E -> quo;
```

5. this assumption is made for example in the theory `SETS` of the basic library of `Coq`.

```
   quo_comp: (x, y : E) (R x y) -> (class x) == (class y);
   quo_lift: (F : Type) (f : E -> F) ((x, y : E) (R x y)
             -> (f x) == (f y)) -> quo -> F;
   quo_lift_prop: (F : Type) (f : E -> F)
                  (H : (x, y : E) (R x y) -> (f x) == (f y))
                  (x : E)(!quo_lift F f H (class x)) == (f x);
   quo_ind: (P : quo -> Prop) ((x : E)  (P (class x)))
             -> (y : quo)  (P y)}.
Axiom classic_quotient:(E : Type) (R : (Relation E))
   (Prf_equiv : (Equivalence E R)) (type_quotient Prf_equiv).
```

This notion of quotient is not however appropriate for the functional sets. One cannot indeed build the natural isomorphism which exists between the sets $E \to \frac{F}{R}$ and $\frac{E->F}{S}$, where $f\ S\ g \Leftrightarrow \forall x,\ f(x)\ R\ g(x)$, and that neither in a direction nor in the other [Hof95]. To build the canonical bijection b from $\frac{E->F}{S}$ in $E \to \frac{F}{R}$ supposes to lift the function $f \mapsto class(F)\ o\ f$. But one has no possibility to show that it is compatible with the relation R, because this requires a form of extentionality [6]. To build its inverse or to show that it exists is even more problematic: that supposes to be able to build a function from E to F from of a function of E in $\frac{F}{R}$, therefore to have a section of the function $class(F)$, which amounts giving itself naive quotients, which one saw that they are incoherent. To turn over the problem in all the directions, one realizes that the extentionality of functions is impossible to circumvent to build the bijection b. It also allows, with the induction principle *quo_ind*, to show its injectivity. But it still misses something to show its surjectivity: `(f : E ->(quotient F)) (exT? [g : (quotient Q)] f == (b g))`. We need an induction principle on the functions which go into the quotient. More generaly, we see that it is not enough to characterize the functions which start from a quotient, it is also necessary to characterize those which arrives there. These reflexions lead us to the following notion of quotient.

5 Functional quotients

We start by supposing the extentionality of functions, which is natural in mathematical practice:

```
Axiom Extentionality :(E,F:Type) (f, g : E -> F)
       ((x : E)  (f x) == (g x)) -> f == g.
```

More, a slightly weaker form of extentionality [7] can be derived from quotients of [Hof95], showing that extentionnality is deeply associated with quotients.

One can now identify (for the Leibnitz equality) two equal functions in each point.

Let E be a type and R an equivalence relation on E. The quotient of E by R is given as a type Q, with a function *class* from E to Q such that $x\ R\ y \Leftrightarrow class(x) == class(y)$.

As before, we give a way of lifting any function f from E to F compatible with R in a function $\hat{f}$ from Q to F such that $f == \hat{f}\ o\ class$. We finally give an induction principle,

6. this is $(\forall x,\ f(x) == g(x)) \Rightarrow \lambda x\ class(f(x)) == \lambda x\ class(g(x))$
7. $(\forall x,\ f(x) == g(x)) \Rightarrow (\lambda x\ f(x)) == (\lambda x\ g(x))$

more powerful than the precedent:

$$\forall G,\ \forall P : (G \to Q) \to Prop, (\forall g : G \to E, P(class\ o\ g)) \Rightarrow \forall f : G \to Q, P(f)$$

In Coq, that gives:

```
Record  type_quotient [E : Type; R : (Relation E);
                       Prf_equiv : (Equivalence E R)] : Type :=  {
   quo: Type;
   class: E -> quo;
   quo_comp: (x, y : E) (R x y) -> (class x) == (class y);
   quo_comp_rev: (x, y : E) (class x) == (class y) -> (R x y);
   quo_lift: (F : Type) (f : E -> F) (compatible R f) -> quo -> F;
   quo_lift_prop: (F : Type) (f : E -> F) (H : (compatible E f))
                  (comp (quo_lift F f H) class) == f;
   quo_ind_fun_left: (G : Type) (P : (G -> quo) -> Prop)
        ((f : G -> E)  (P (comp class f))) -> (f : G -> quo)  (P f)}.

Axiom quotient:(E : Type)(R : (Relation E))
     (Prf_equiv : (Equivalence E R)) (type_quotient E).
```

5.1 Some consequences

The induction principle $quo\_ind\_fun\_left$ makes it possible to characterize the functions which arrive in a quotient: they factorize all with the function $class$ [8].

It is possible to show the induction principle of the section 4, i.e. the quotient does not have other elements than the images of the function $class$:

```
Variable E:Type.
Variable R:(Relation E).
Variable Prf_equiv:(Equivalence E R).

Definition Q := (quotient  Prf_equiv).

Lemma quo_ind: (P : Q -> Prop) ((x : E)  (P (class x)))
               -> (y : Q)  (P y).
```

Moreover one can show that the functions which start from the quotient are all liftings of compatible functions:

```
Lemma quo_ind_fun_right: (F : Type) (P : (Q -> F) -> Prop)
    ((f : E -> F) (H : (compatible R f))  (P (quo_lift H)))
    -> (f : Q -> F)  (P f).
```

5.2 The problematic example is solved

Finally the example against which butted the traditional quotients is solved here, as one will show it now.

8. but it is impossible to build this factorization, if not there would be a section of the function $class$.

Let E and F be two types, R an equivalence relation on F, and S the equivalence relation on $E \to F$ defined by $f\ S\ g \Leftrightarrow \forall X,\ f(x)\ R\ g(x)$.

We build the function b from $\frac{E \to F}{S}$ to $E \to \frac{F}{R}$ as the lifting of the function $f \mapsto class(F)\ o\ f$, once shown that it is compatible with the relation S (what uses the axiom of extentionnality). To show that b is injective also works thanks to the extentionnality. The surjectivity of b is obtained with the principle $quo\_ind\_fun\_left$ and again the extentionnality.

This example is in a certain way generic. It shows the necessary compatibility between two operations on types: on the one hand formation of functions, and on the other hand formation of quotients. In the form of equation on types it results in:

$$E \to \frac{F}{R} = \frac{E -> F}{E -> R}$$

where $E \to R$ is the relation S.

We could have taken this equation between types to complete the quotients of the section 4. But on the one hand that would not have been sufficient for lack of extentionnality, and on the other hand it had been exaggerated: it is not reasonable to identify two isomorphic types (in any case this is not done in mathematics).

5.3 Application in Coq: subtypes

Functional quotients works well in the system Coq. As an example we will build a notion of subtype. Thanks to the mechanism of coercions of Coq, we will obtain in this case a rather clear and not very verbose syntax, close to the mathematical practice. We also use the notion of setoid existing in the library of Coq to represent the sets provided with an equivalence relation. Thanks to coercions, one can see a setoid as a type (its support), or as a binary relation (its equivalence relation), or as a proof that its relation is an equivalence relation.

The definition of the functional quotients adapts as follows:

```
Require Export Setoid.
Implicit Arguments On.
 Coercion Equal : Setoid >-> Relation.

Record  type_quotient [E : Setoid] : Type :=  {
   quo:> Type;
   class:> E -> quo;
   quo_comp: (x, y : E) (Equal x y) -> (class x) == (class y);
   quo_comp_rev: (x, y : E) (class x) == (class y) -> (Equal x y);
   quo_lift: (F : Type) (f : E -> F) (compatible E f) -> quo -> F;
   quo_lift_prop: (F : Type) (f : E -> F) (H : (compatible E f))
                  (comp (quo_lift F f H) class) == f;
   quo_ind_fun_left: (G : Type) (P : (G -> quo) -> Prop)
                     ((f : G -> E)  (P (comp class f)))
                     -> (f : G -> quo)  (P f)}.
```

```
Axiom quotient:(E : Setoid)  (type_quotient E).

 Coercion quotient : Setoid >-> type_quotient.
```

Three coercions are defined here for the quotients. The first, *quo*, allows quotient to be used as a type. The second, *class*, makes it possible to see a quotient as a function (its function *class*). Lastly, the third, *quotient*, makes it possible to see a *Setoid* as a quotient.

Let us define now subtypes.

We start with a definition which can seem stupid:

```
Record  TYPE  : Type :=  {
   TYPE_Type:> Type}.
```

This is purely technical, to allow for example a function f of type $F \to E$ where E is a type variable, to be used as a coercion, which is impossible in Coq with such a type ending with a variable. Because if E is rather of type $TYPE$ (what is equivalent, in fact), then the type of f is $F \to (TYPE\_Type\ E)$, and f can become a coercion arriving in the class[9] $TYPE\_Type$.

We now define the subtype of the elements of a type E which verify a predicate P. We start from the type whose elements are dependent couples (x,p), where x is in E and p is a proof of $P(x)$. We identify two couples which have the same first component, which gives an equivalence relation, and, therefore, a setoid. It remains to take the quotient of it: we have a function which with any type and with any predicate associates a type which is not different from the required subtype.

```
Section Definition_of_subtypes.
Variable E:TYPE.
Variable P:E -> Prop.

Record  subtype_carrier  : Type :=  {
   elt:> E;
   prf: (P elt)}.
Definition subtype_eq := [x, y : subtype_carrier]  (elt x) == (elt y).
Definition subtype_setoid: Setoid.
Apply (!Build_Setoid subtype_carrier subtype_eq) .
...
Defined.
Definition subtype := (quo (quotient subtype_setoid)).
End Definition_of_subtypes.
```

Let us note that the function *elt* is a coercion: it makes it possible to see a couple (x,p) as an element of E. Let us carry out some basic constructions with this notion of subtype:

```
Section Tools_for_subtypes.
Variable E:TYPE.
Variable P:E -> Prop.

Lemma elt_compatible: (compatible (subtype_setoid P) (!elt E P)).
```

9. with the definition of classes in coercions of Coq

Canonical injection from the subtype into the type:

```
Definition subtype_inj : (subtype P) -> E := (quo_lift elt_compatible).
```

The restriction of a function of domain E to the defined subtype:

```
Definition restrict := [F : Type] [f : E -> F]  (comp f subtype_inj).

End Tools_for_subtypes.
 Coercion subtype_inj : subtype >-> TYPE_Type.
```

We can write and show easily the following properties:

```
Section Tools_for_subtypes2.
Variable E:TYPE.
Variable P:E -> Prop.

Lemma subtype_prop: (c : (subtype P))  (P c).
```

We could write (Pc) because the canonical injection is a coercion.

```
Definition subtype_partial_inj: (x : E) (P x) -> (subtype P)
:=[x:E][H:(P x)]((quotient (subtype_setoid P)) (Build_subtype_carrier H)).
```

Here, the quotiented subtype (*quotient* (*subtype_setoid P*)) is used, by coercion, as its function *class*. The function *Build_subtype_carrier* is not different than the single constructor of the inductive type *subtype_carrier*. We show that the canonical injection is injective, that it has a retraction, and that one has a proof irrelevance for subtypes:

```
Lemma subtype_surj_inj: (c : (subtype P))
                        (subtype_partial_inj (subtype_prop c)) == c.
Lemma subtype_inj_is_injective: (x, y : (subtype P))
         (subtype_inj x) == (subtype_inj y) -> x == y.
Lemma subtype_partial_inj_pro: (x : E) (h : (P x))
         (subtype_inj (subtype_partial_inj h)) == x.
Lemma subtype_proof_irrelevance: (x : E) (h, h' : (P x))
    (subtype_partial_inj h) == (subtype_partial_inj h').
End Tools_for_subtypes2.
```

5.4 Consistency

The syntactic model of the setoids of [Hof95] is a natural candidate to be a model of the functional quotients. We will be satisfied here with this assumption, which requires confirmation.

6 Conclusion

The notion of quotient seems irreducible in mathematics. It still gives problem in type theory. With this study of various ways to introduce quotients, we hope to have shown that the CIC is, without extention of its language, and with addition of axioms which seems consistent, sufficient to formalize a notion of quotients at the same time close to the

mathematical practice and usable in a system such as `Coq`. Indeed, the quotients such as they are defined in the section 5 are compatible with the notion of function, which missed with the preceding notions. It remains however to give a proof of consistency to the axioms which define them. What should be obtained by taking as a starting point the work of [Hof95].

We will finish by noticing that if one wants to abstain from introducing quotients types in type theory, one is obliged to work with sets with equivalence relations, i.e. setoids. The developments are then rather tiresome. To be convinced, one can consult [Pot99] or [GPWZ00]. To work with setoids however makes it possible to form quotients of setoids easily: it is enough to change the equivalence relation. But this advantage seems small compared with the disadvantages. Indeed, since a quotient is not a type, one loses the construction of functions by simple abstraction (one must show that the functions which one defines on the setoids are compatible with the associated equivalence relations). One loses also the power of the rewritings by Leibnitz equality: one can change a subterm by an equivalent subterm only if all the functions of the context are compatible with the equivalence relations. It follows a heaviness in the proofs which proves to be expensive, even crippling or ridiculous from a mathematical point of view.

Références

[Bar95] G. Barthe. Extensions of pure type systems. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of TLCA'95*, volume 902, pages 16–31. Springer-Verlag, 1995.

[BB96] F. Barbanera and S. Berardi. Proof-Irrelevance out of Exluded-Middle and Choice in the Calculus of Constructions. *Journal of Functional Programming*, pages 519–525, 1996.

[Bou70] N. Bourbaki. *Théorie des ensembles*. 1970.

[Bou97] S. Boutin. *Réflexions sur les quotients*. PhD thesis, Thèse de doctorat, Université de Paris 7, 1997.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[fol00] Coq's folklore. «eqT_rec is safe». 2000.

[GPWZ00] H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. Squeleton for the proof development leading to the fundamental theorem of algebra. page `http://www.cs.kun.nl/ herman/FTA.mathproof.ps.gz`, 2000.

[Hof95] M. Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, PhD thesis, LFCS Edinburgh, 1995.

[LW99] S. Lacas and B. Werner. Which choices imply the Excluded Middle? *pre-print*, 1999.

[Pot99] L. Pottier. Basic notions of algebra. *The users' contributions of the Coq system*, page `http://pauillac.inria.fr/coq/contribs/algebra.tar.gz`, 1999.

[Pot00] L. Pottier. Extraction dans le CCI. *Rapport de recherche INRIA*, RR-4026, oct 2000.

[pro99] The Coq project. The coq proof assistant, reference manual. page `http://pauillac.inria.fr/coq/doc/main.html`, 1999.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399