Les types quotients en *Coq*

Cyril Cohen

INRIA Saclay–Île-de-France
Laboratoire d'Informatique de l'École polytechnique,
91128 Palaiseau CEDEX, France
cyril.cohen@inria.fr

RÉSUMÉ. *La formalisation des structures quotients en théorie des types est un problème délicat. On propose ici une nouvelle solution pratique pour manipuler ces structures dans le Calcul des Constructions Inductives. En particulier, on propose une méthode systématique pour les quotients constructifs basés sur des types munis d'une égalité décidable. Dans ce cadre, on retrouve le confort d'utilisation d'un formalisme extensionnel comme la théorie des ensembles, en particulier grâce à l'usage d'outils de méta-programmation, disponibles en Coq. De plus on se base sur les bibliothèques développées dans l'extension* SSReflect*, qui proposent à la fois une formalisation de la théorie de ces types à l'égalité décidable, et un support pour les structures algébriques qui sont à la base des exemples de structures quotients étudiées.*

ABSTRACT. *Formalising quotients structures in type theory is a delicate problem. Here I suggest a new practical solution in order to manipulate these structures in the Calculus of Inductive Constructions. More precisely, I suggest a systematic method to build constructive quotients based on types with decidable equality. In this framework, I get back the facilities of an extensionnal formalism like set theory, thanks, among other things, to the use of meta-programming utilities provided by the Coq proof assistant. Moreover, I base my work over libraries developped with the* SSReflect *extension, which grants me a formalisation of type with decidable equality theory, along with a support for algebraic structures over which my examples are based.*

MOTS-CLÉS : *coq, ssreflect, théorie des types, quotient, calcul des constructions inductives, mathématiques constructives*

KEYWORDS: *coq, ssreflect, type theory, quotient, calculus of inductive constructions, constructive mathematics*

1. Introduction

Une relation d'équivalence permet d'identifier des objets appartenant à une même classe. Par exemple, la relation dont les classes sont tous les singletons est l'égalité "syntaxique". Mais on peut imaginer des relations arbitrairement fines ou grossières. Ainsi, toutes les personnes dont le numéro de Sécurité Sociale commence par le même chiffre ont le même sexe, celles qui ont les trois mêmes premiers chiffres ont même sexe et même année de naissance, etc. La structure quotient associée à une telle relation est l'ensemble des classes pour cette relation : il y en a deux pour la relation "avoir le même premier chiffre dans le numéro de Sécurité Sociale", beaucoup plus pour celle qui compare les trois premiers chiffres. Les structures quotients sont une construction omniprésente, quoique souvent implicite dans la littérature mathématique, au point qu'elle est fréquemment notée avec un symbole $=$, qui peut se trouver ainsi lourdement surchargé. L'égalité des fractions, qui identifie deux fractions qui ont le même représentant irréductible est ainsi en fait une structure quotient sur $\mathbb{Z} \times \mathbb{Z}^*$. Mais il est très fréquent que deux objets soient implicitement identifiés dans une structure quotient beaucoup plus subtile : quotients de groupes, quotients d'idéaux,...

Toutefois, lorsqu'on s'intéresse à la formalisation de théories mathématiques, ces passages au quotients doivent être gérés explicitement. Dans la théorie des ensembles qui sous-tend l'essentiel de la littérature mathématique non formalisée, la construction et la manipulation de telles structures est naturelle, et ne pose pas de problème technique particulier. Par contre, dans le contexte de théories des types dites intentionnelles, la construction de structures quotients devient plus problématique, du fait du statut radicalement différent de l'égalité. Le rôle central des structures quotient a donné lieu à de nombreux travaux visant à explorer les solutions envisageables dans un cadre de formalisation basé sur une théorie des types. Néanmoins, dans l'état actuel des théories des types implémentées, ce problème ne semble pas avoir trouvé de solution satisfaisante dans tous les contextes [AMS07].

Le travail décrit dans cet article se place dans le contexte du projet de formalisation de la preuve du théorème de Feit-Thompson [FT63], mené dans l'équipe *Composants Mathématiques* du *Centre*

commun INRIA - Microsoft Research. Cette formalisation a pour but de valider les recherches menées dans cette équipe pour trouver des méthodes de développements de bibliothèques formalisées modulaires et réutilisables. En effet la preuve publiée du théorème de Feit-Thompson est à la fois significativement volumineuse mais surtout fait appel à une combinaison de concepts variés (algèbre linéaire, théorie des groupes finis, théorie des modules, corps finis, extensions de corps...) et de résultats avancés. Une particularité des bibliothèques développées dans ce projet est qu'elles s'appuie sur des types pour lesquels l'égalité (de Leibniz) est décidable. De façon cruciale, il y a donc unicité des preuves d'égalité pour les éléments d'un tel type.

Si la construction de quotient sur des types finis peut se résoudre de façon complètement intentionnelle [GMR+07], la construction de structures quotient comme les nombres algébriques ou les quotients d'idéaux requièrent une approche complètement différente. Les bibliothèques et le langage SSReflect[GM08] permettent de retrouver le confort d'un cadre classique, alors qu'on raisonne dans un formalisme intuitionniste, mais sur un fragment admettant le tiers-exclu. De même, le travail proposé ici vise à retrouver le confort d'un cadre intentionnel, alors qu'on travaille dans une théorie extensionnelle, mais avec des relations d'équivalence ayant la propriété d'unicité des preuves.

Après un bref retour sur les travaux menés sur les quotients en théorie des types, on présentera les techniques de méta-programmation disponibles dans le système *Coq*[BC04], et qui sont des outils clefs de la manipulation des quotients. On présentera ensuite un méthode générique de formalisation pour les quotients constructifs sur des types à égalité décidable. Enfin on donnera quelques exemples de constructions utilisant cette méthode.

2. Le problème et les solutions existantes

Le problème

La définition usuelle de quotient en théorie des ensembles est la suivante :
Soit E un ensemble et $\sim$ une relation d'équivalence sur E.

– Soit $x \in E$, on définit la classe de x et on note $\hat{x}$ l'ensemble $\{y \in E \mid x \sim y\}$.

– On définit ensuite le quotient de E par $\sim$ et on note $E/\sim$ l'ensemble $\{\hat{x} \mid x \in E\}$

Cette définition parle de relations, élément central de la théorie des ensembles. L'égalité y est fondamentalement extensionnelle, il n'existe donc aucun problème pour identifier $\hat{x}$ et $\hat{y}$ lorsque $x \sim y$: l'égalité $\hat{x} = \hat{y}$ est vraie.

La théorie des types place le calcul au centre du formalisme, ainsi, à la base se trouvent les fonctions ; les relations en sont vues comme un cas particulier. Une relation d'équivalence `R` sur un type `T` est une fonction `R : T → T → Prop` qui vérifie les axiomes de réflexivité, de symétrie et de transitivité. On peut alors éventuellement voir la classe équivalence de `x : T` comme le prédicat `R x : T → Prop`. Mais lorsque `R x y` est habité, on devrait pouvoir identifier `R x` et `R y` ...

L'égalité dont on dispose est syntaxique (modulo conversion), or les deux termes ainsi formés sont en général syntaxiquement différents, même si ce sont des fonctions qui prennent la même valeur en tout point. En effet, on considère `R x` et `R y` comme des programmes, et ce n'est pas parce qu'ils coïncident sur leurs entrées qu'ils sont identiques : par exemple le tri bulle et le quicksort font la même chose, mais sont syntaxiquement différents.

On parle alors d'égalité intentionnelle, alors que l'égalité classique en mathématique est extensionnelle.

Nous allons aborder les différentes solutions existantes, la première permettant de rajouter artificiellement de l'extensionnalité, et la deuxième permettant de l'encoder. La troisième sera une solution intermédiaire.

2.1. *Les types quotients*

Une des solutions les plus directes est l'introduction de types quotients, ils ont été étudiés entre autres par M. Hofmann dans [Hof95b] et par L. Chicli dans [Chi03] et [CPS03].

```
Record type_quotient (E:Type) (R:rel E) (p:
    equivalence R) := {
  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 R
     f) (x:E),
                ((quot_lift F f H) o class) x = f x;
  quo_surj : ∀(c:quo), ∃x:E, c = class x
}.
```

Pour ajouter les types quotients au calcul des constructions, il ne reste plus qu'à énoncer un axiome concernant leur existence :

```
Axiom quotient: ∀(E:Type) (R:rel E) (p:equivalence R)
    , (type_quotient p).
```

Cependant, en posant un axiome, on admet l'existence d'un terme (que l'on ne peut pas construire) qui habite le type, ce qui a principalement deux inconvénients.

Premièrement, on perd l'assurance de la cohérence de notre théorie (il faudrait valider la nouvelle théorie en y trouvant un modèle dans lequel l'axiome serait satisfait).

Deuxièmement, on ne peut donc pas calculer à l'aide de ce terme : cette définition n'est pas *constructive*. La seule façon de manipuler un type quotient est d'y appliquer les règles de réécriture qui le caractérisent.

2.2. *Les Sétoïdes*

À l'autre bout de la chaîne se trouve la solution des *Sétoïdes*, étudiés par M. Hofmann dans [Hof95a] et par Gilles Barthe et al dans [BCP03]. Cette technique ne nécessite pas d'axiome ni de modification du calcul.

Un *Sétoïde* est un type muni d'une relation d'équivalence (appelée *égalité Sétoïde*). Mais comment utiliser cette relation d'équivalence pour réécrire ? D'ailleurs, cette *égalité Sétoïde* n'est licite que si le contexte est stable par la relation.

La solution a ce problème est apportée par de la méta-programmation : on demande au système de mémoriser la relation et on lui indique quels sont les fonctions qui sont stables (les morphismes), ainsi que les preuves de corrections. Enfin, c'est le système qui se charge de recomposer la réécriture *Sétoïde* à chaque fois que l'utilisateur le demande, en assemblant des théorèmes élémentaires découlant des preuves de correction qu'on lui a donné plus tôt.

En pratique, on déclare une égalité *Sétoïde* A_{eq} sur un type A par la commande vernaculaire `Add Parametric Relation`. On aura à rentrer les preuves de réflexivité, symétrie et transitivité.

À partir de là, on peut déclarer quelles sont les fonctions qui stabilisent cette relation d'équivalence, grâce à la commande `Add Parametric Morphism`. Le système en demandera la preuve également.

REMARQUE. —
cf le manuel de référence de *Coq* v8.2 pour plus de détails sur la déclaration de *Sétoïdes*.

Une fois ces opérations effectuées, un `rewrite` fait tout le travail à la place de l'utilisateur pour réécrire un élément de A et un autre, dans un cadre licite.

Cette vision des choses est une transposition directe de la notion ensembliste : on prend un type et une relation dessus, et on tire une notion d'égalité un peu plus extensionnelle. La puissance de cette technique est toute entière contenue dans les tactiques de réécriture.

2.3. *Les types normalisés*

Une solution intermédiaire a été proposée par Pierre Courtieu dans sa thèse [Cou01].

On dit qu'un quotient $E/\mathcal{R}$ est calculatoire lorsqu'il existe une fonction $\mathrm{nf} : E \to E$ telle que

$$\forall x, y \in E,\ x\ \mathcal{R}\ y \Leftrightarrow \mathrm{nf}(x) = \mathrm{nf}(y)$$

Les types normalisés étendent le CCI pour formaliser la notion de quotient calculatoire :

– on rajoute les types de la forme $\mathbf{Norm}(A, \mathrm{nf})$ de types associés à leur fonction de normalisation

– leurs habitants : $\mathbf{Class}(A, \mathrm{nf}, x)$

– et une élimination adaptée

REMARQUE. —
Cette formalisation nécessite d'implémenter une extension du calcul.
On peut prouver que ce formalisme est équivalent à la *Proof-Irrelevance*.

2.4. *Panorama*

En passant des types quotients aux *Sétoïdes*, on passe d'une théorie où l'on a rajouté de l'extensionnalité, à une théorie qui reste intentionnelle mais où l'on a dû cacher le sucre syntaxique, l'encodage qui permet de simuler un peu d'extensionnalité.

3. Outils de méta-programmation

Le système *Coq* dispose d'un mécanisme d'inférence de type, qui nous permet de laisser des termes incomplets dans les types qu'on lui donne. Il essaye alors normalement de boucher ces "trous" par unifications.

Exemple 1 : inférence de type

Dans l'énoncé,

```
∀x, x = 0
```

il y a en fait deux arguments implicites : le type de `x` et le type sur lequel porte l'égalité. En représentant les implicites cela donnerait :

```
∀x : ?, eq ? x 0
```

Le mécanisme d'unification lui permet alors d'inférer que les deux implicites sont en fait des `nat`.

Il existe actuellement en *Coq* deux mécanismes permettant au système de retrouver de l'information que l'unification ne lui permet pas d'inférer : les *Structures Canoniques* qui interviennent au coeur de l'algorithme d'unification, et les *TypeClasses* qui donnent une autre politique pour boucher ces "trous".

3.1. *Les TypeClasses*

Existantes en Haskell, elles ont été ajoutés à la dernière version de *Coq* par Mathieu Sozeau [SO08].

Pour les utiliser, on doit définir une classe : un inductif qui va avoir un statut particulier. On pourra déclarer des instances de cet inductif qui serviront à remplir une base. Pour utiliser des *TypeClasses*, le type sur lequel l'inférence va s'effectuer doit se trouver en paramètre de classe.

Exemple 2 : Surcharge d'un opérateur

```
Class Additive (T:Type) := { add : T →T →T }.
Notation "x + y" := (add x y).

Instance AddNat : Additive nat := {| add := plus |}.
Instance AddBool : Additive bool := {| add := xorb |}.

Check 1+ 2. > 1 + 2 : nat
Check true+ false. > true + false : bool
```

Exemple 3 : Surcharge de preuves

On veut formaliser la notion de groupe :

```
Class Group T := {
  op : T →T →T;
  neutre : T;
  inv : T →T;

  associativity : ∀x y z, op (op x y) z = op x (op y z);
  neutrePr : ∀x, op neutre x = x;
  neutrePl : ∀x, op x neutre = x;
  invPr : ∀x, op (inv x) x = neutre;
  invPl : ∀x, op x (inv x) = neutre
}.
```

On pourra alors déclarer des instances sous la forme

```
Program Instance Z2 : Group bool := {|
  op := xorb;
  neutre := false;
  inv := fun x ⇒ x
|}.
```

On peut alors utiliser `op`, `neutre`, `inv` et tous les lemmes les concernant, de manière générique.

REMARQUE. —
Les *Sétoïdes* sont actuellement implémentée à l'aide de *TypeClasses* (cf section 2.2, et [SO08]).

3.2. *Les Structures Canoniques*

Elles sont dans le système *Coq* depuis plus longtemps que les *TypeClasses*, introduites par Amokrane Saibi [Saï98] en même temps que les coercions implicites.

Elles sont massivement utilisées par l'équipe *Composants Mathématiques* (cf [GMR+07], [BGOP08] et [GGMR09]).

Exemple 4 : Surcharge de preuves par *Structure Canonique*

```
Structure Group := mkGroup {
  T :> Type;

  op : T →T →T;
  neutre : T;
  inv : T →T;

  associativity : ∀x y z, op (op x y) z = op x (op y z);
```

```
  neutrePr : ∀x, op neutre x = x;
  neutrePl : ∀x, op x neutre = x;
  invPr : ∀x, op (inv x) x = neutre;
  invPl : ∀x, op x (inv x) = neutre
}.
```

On pourra alors déclarer une instance

```
Canonical Structure Z2 : mkGroup bool xorb false (fun x ⇒ x)
    xorb_assoc xorb_false_r xorb_false_l xorb_nilpotent xorb_nilpotent.
```

Si on laisse vide le premier paramètre de op, le système pourra retrouver tout seul de quelle structure il s'agit, si elle est canonique.

4. L'*Interface Quotient*

Le but de ma démarche a été de ne pas sortir du calcul des constructions inductives et de fabriquer des outils génériques pour construire des types quotients sans ajouter d'instructions vernaculaires supplémentaires au langage.

Le cadre dans lequel on se place est plus spécifique, car on considère des types muni d'une procédure de décision de l'égalité. Quand ce n'est pas le cas, il est de toute façon vain d'essayer d'obtenir une procédure de construction du quotient. Les preuves d'égalités de deux éléments d'un tel type sont alors toutes prouvablement égales : on a localement obtenu la *Proof-Irrelevance*.

Cette procédure de décision de l'égalité est notée ≡, c'est une fonction qui retourne un booléen. On utilise régulièrement la coercion triviale de `bool` vers `Prop` pour voir les objets `A≡B` de type `bool` comme des propositions `A≡B = true` de type `Prop` [1].

L'idée développée ici a beaucoup de points commun avec l'approche des *types normalisés* (cf section 2.3). En effet, on se restreint aux cas où l'on peut construire effectivement le quotient. Cependant grâce à la *Proof-Irrelevance* sur l'égalité, les problèmes disparaissent.

Pour cacher tous les encodages et avoir ainsi des opérations génériques, on a besoin d'un peu de méta-programmation. Par tradition

1. formulation équivalente à `A = B`, par choix de ≡.

de l'équipe *Composants Mathématiques*, ce sont les *Structures Canoniques* qui ont été utilisées.

Cette fois ci, on ne doit pas définir une fonction qui prend une relation d'équivalence et retourne un objet tout construit, mais on doit spécifier ce dernier. On laissera à l'utilisateur le soin de construire le quotient à sa convenance et dans un second temps de lui faire correspondre la spécification ainsi qu'une relation d'équivalence.

4.1. *Spécification*

Construction

Voici la définition de l'interface. Il s'agit d'un inductif `quotType` à un seul constructeur `QuotType`, et dont les trois premiers projecteurs sont nommés :

– `quot_sort` : le type qui va représenter le quotient

– `repr` : la fonction de sélection d'un représentant canonique dans `T` d'un élément de `Q`.

– `pi` la surjection canonique de `T` dans `Q`.

```
Record quotType (T : Type) := QuotType {
  quot_sort :> Type;
  repr : quot_sort → T;
  pi : T → quot_sort;
  _ : ∀ x : quot_sort, pi (repr x) = x
}.
```

Pour l'utiliser, l'utilisateur doit construire :

– le type `Q` "à la main" qui servira de `quot_sort`

– deux fonctions `reprQ : Q → T` et `piQ : T → Q`

– une preuve `pi_reprQK` de `∀ x : T, piQ (reprQ x)= x`

On peut alors munir `Q` d'une *Structure Canonique* `qT` de quotient (cf exemple 5)

Exemple 5 : Définition de $\mathbb{Z}$

```
Definition axiom (z : nat*nat) := (z.1 ≡0) || (z.2 ≡0).
Definition relative := { z | axiom z }.
Definition Relative z (pz : axiom z) : relative := exist axiom _ pz.

Lemma one_diff_eq0 : ∀x y, ((x-y) ≡0) || ((y-x) ≡0).
Proof. (* ... *) Qed.

Definition pi_relative (x : nat*nat) := @Relative ((x.1-x.2),(x.2-x
    .1)) (one_diff_eq0 _ _).
Definition repr_relative := projT1.
Lemma pi_repr_relativeK : ∀x, pi_relative (repr_relative x) = x.
Proof. (* ... *) Qed.

Canonical Structure relative_qT := Eval hnf in QuotType
    pi_repr_relativeK.
```

REMARQUE. —
On attire l'attention sur les notations `z.1` et `z.2` spécifiques à SSREFLECT qui signifient respectivement `fst z` et `snd z`.

Cette *Structure Canonique* permet, dans la suite, de donner moins d'information au système : ainsi on arrive a utiliser `repr` sans préciser l'*Interface Quotient* `qT` (l'argument étant de type `Q`, on arrive a "inférer canoniquement" `qT`).

On arrive également à définir **la notation** `\pi_Q` qui permet de ne préciser que le type de base `Q` : le système retrouve tout seul qu'il s'agit en fait de `pi qT`.

Élimination

On peut définir le schéma d'élimination générique suivant :

```
Lemma quotW : ∀(qT : quotType) P,
      (∀y:T, P (\pi_qT y)) →∀x:qT, P x.
```

Ce schéma d'élimination est fondamental car il permet d'effectuer une analyse de cas sur un type, en le regardant comme quotient d'un autre. On peut l'instancier

```
Definition QelimW := @quotW _ qT.
> QelimW : ∀P : qT →Type,
  (∀x : T, P (\pi_Q x)) →∀x : qT, P x
```

On dispose ainsi d'un lemme permettant de ramener un problème quantifié universellement par des variables du type quotient à un lemme quantifié sur leur type de base.

Exemple 6 : élimination dans $\mathbb{Z}$

Ainsi on pourra éliminer `x` dans `∀x : relative, Px` ce qui donnera `∀x : nat * nat, P (\pi_relative x)`.

Il en existe aussi une version plus forte qui permet de restreindre son étude aux représentants canoniques.

```
Definition QelimP := @quotP _ qT.
> QelimP : ∀P, (∀x : T, repr (\pi_Q) x) = x
             →P (\pi_Q x)) →∀x : qT, P x.
```

4.2. *Définition et utilisation de fonctions stables*

Définition et encodage

Les prédicats `\compat1_Q f` et `\compat2_Q f` expriment le fait qu'une fonction `f` respectivement unaire ou binaire soit stable par passage dans `Q`.

Les fonctions

```
> qT_op1 : ∀(T : eqType) (Q : quotType T) (S : Type)
       (op : T →S), \compat1_Q op →(Q →S)
> qT_op2 : ∀(T : eqType) (Q : quotType T) (S : Type)
       (op : T →T →S), \compat2_Q op →(Q →Q →S)
```

servent alors a construire le relèvement des fonctions stables grâce à la preuve de stabilité.
Ainsi, par exemple :

– Si l'on dispose d'une fonction `f : T →T →E`

– Si l'on peut trouver une preuve $\texttt{compat}_{\texttt{f}}$ de type `\compat2_Q f`

Alors on peut définir le relèvement $\hat{\texttt{f}}$: `Q →Q →E` de `f` de la manière suivante.

```
Definition f̂ := qT_op2 compat_f.
```

On dispose maintenant de la fonction relevée, qui est la factorisation par rapport à la surjection canonique de la fonction qu'on lui avait donné.

On dispose de bonnes propriétés dessus, notamment on sait la calculer et on sait l'éliminer (cf paragraphe *Réécriture*).

Exemple 7 : addition dans $\mathbb{Z}$

```
Lemma addz_compat : compat2_relative _
     (fun x y ⇒ \pi_relative (x.1 + y.1 , x.2 + y.2)).
Notation addz := (qT_op2 addz_compat).
```

Réécriture

On dispose des deux lemmes suivant tous deux regroupés sous la désignation `qTE`

```
Lemma qT_op1E : ∀S f compat_f x,
   @qT_op1 S f compat_f (π_Q x) = f x.
Lemma qT_op2E : ∀S f compat_f x y,
   @qT_op2 S f compat_f (π_Q x) (π_Q y) = f x y.
```

Ceci permet par exemple de substituer `f̂ (\pi_Q x)(\pi_Q y)` par `f x y`, dans tout contexte.

On peut également établir un lemme `weak_qT_op2E` qui permette de réécrire `f̂ ζ ξ` en `f (repr ζ) (repr ξ)`, mais cette dernière formulation est moins générale et fait perdre la possibilité de choisir un représentant quelconque, au profit du représentant canonique, qui n'est pas toujours le bon candidat.

4.3. *Quotient et relation d'équivalence*

Jusque là, on a parlé de quotient, mais on n'a pas mentionné de relation d'équivalence. Cela montre bien que l'on a abandonné le coté relationnel de la notion, au profit du côté fonctionnel.

Cependant, il faut à présent faire le lien. L'équivalence induite par l'*Interface Quotient* sur le type de base est ainsi définie :

```
Notation "x ≡y 'mod' Q" := (\pi_Q x ≡\pi_Q y).
```

On peut alors avoir envie d'utiliser une autre relation d'équivalence. En effet, pour l'instant on a laissé l'utilisateur construire le quotient en faisant un choix de représentants, mais il peut y avoir une façon plus pratique de dénoter l'équivalence.

Exemple 8 : équivalence associée à $\mathbb{Z}$

```
Definition equivz (x y : nat*nat) := x.1 + y.2 ≡y.1 + x.2.
Lemma equivzP : ∀x y, x ≡y mod relative = equivz x y.
```

La fonction `equivz` sera sûrement très utile, car il sera certainement plus simple de comparer `x.1 + y.2` et `y.1 + x.2` que les réduits `\pi_Q x` et `\pi_Q y`.

REMARQUE. —
Etablir l'égalité des équivalences revient en fait à prouver que la surjection canonique `\pi_Q` peut être vue comme fonction de normalisation (au sens de la section 2.3) de l'équivalence que l'on vient de définir.

4.4. *Puissance et particularités*

Grâce aux *Structures Canoniques*, on a réussi a surcharger les fonctions de passage entre un type et un de ses quotients, ainsi que des lemmes permettant de manipuler, déplacer et simplifier les fonctions génériques (principalement `\pi_Q`, `repr` et `≡mod`). Ces manipulations ne font intervenir que des réécritures basiques.

Avec les *Interfaces Quotients*, on reste dans le cadre de la théorie, car on ne pose pas d'axiome, mais une condition à remplir : le fait que le quotient soit constructible. Cette hypothèse semble assez raisonnable, car si on ne peut pas le construire, on ne peut pas calculer avec.

La contrepartie est qu'on laisse l'utilisateur construire manuellement son type quotient puis de lui faire correspondre l'interface. Cependant, il peut être mauvais de vouloir choisir un unique mécanisme de construction. En effet, l'implémentation fournie par un mécanisme générique ne prendrait pas en compte les particularités des types concrets (par exemple un inductif) sur lesquels on est amenés à travailler. Laisser ouverte la construction du quotient permet à l'utilisateur de choisir

l'implémentation qui lui convient le mieux, voire d'en avoir plusieurs et de passer de l'une à l'autre.

Au fond, il s'agit d'une théorie du transport de structure par surjection.

5. Exemples d'utilisation

À ce nouveau formalisme, je donne deux exemples d'utilisation (en plus de l'exemple jouet de $\mathbb{Z}$).

5.1. *Corps des fractions*

Grâce aux *Interface Quotient*, j'ai pu construire de manière assez simple le corps des fractions rationnelles sur un anneau intègre. Je me suis basé pour cela sur la bibliothèque de hiérarchie algébrique de SSReflect (cf [GGMR09]) nommée `ssralg`, et prouvé dans ce cadre que ma construction donnait bien lieu à un corps.

Pour effectuer cette construction, on prend comme type de base le sigma-type des paires d'éléments de l'anneau intègre tels que le deuxième élément est non nul. On choisit comme quotient le sigma-type constitué des éléments du type de base qui sont canonique.

Concrètement cela donne :

```
Variable R : idomainType. (* R est un anneau integre *)

Definition dom_rational := { x : R*R | x.2 ≠ 0 }.
Definition equivq (x y : dom) := x.1 * y.2 ≡ x.2 * y.1.
Definition canon x := choose (equivq x) x.
Definition rational := { x | canon x≡ x }.

Definition pi_rational : dom_rational →rational := (*
   ... *).
Definition repr_rational : rational →dom_rational :=
   projS1.
Lemma pi_repr_rationalK : ∀x, pi_rational (
   repr_rational x) = x.
```

```
Proof. (* ... *) Qed.

Canonical Structure rational_qT := Eval hnf in QuotType
    pi_repr_rationalK.
```

On définit ici la canonicité grâce à la fonction de choix `choose`. En effet, on suppose qu'il existe sur un anneau intègre une fonction de choix c qui prend une propriété P à valeurs booléennes ainsi qu'un élément par défault x et retourne x si $\neg P(x)$ et sinon donne un élément $c(P, x)$ tel que $\forall Q, (Q \Leftrightarrow P) \Rightarrow c(P, x) = c(Q, x)$. Cf [GGMR09] pour plus de précisions.

Cette fonction de choix s'étend alors aux couples d'éléments de cet anneau et permet donc de faire un choix dans la classe d'équivalence (il suffit de donner à cette fonction de choix la propriété "être équivalent à x").
REMARQUE. —
Dans cetains cas d'anneaux bien concrets, la fonction de choix est reconstruite automatiquement grâce aux mécanismes de SSREFLECT. Par exemple, tout type contenant un ensemble dénombrable d'éléments est muni automatiquement d'une fonction de choix ("être dénombrable" est en effet une condition suffisante à l'existence d'un tel algorithme).

Ainsi, en appliquant cette construction à `relative`, on obtient le corps des fractions $\mathbb{Q}$. Les calculs n'auront pas forcement une complexité raisonnable, mais ce n'est pas un problème car le but de cette construction est avant tout de pouvoir faire des preuves.

On peut alors définir les opérations sur le type de base, prouver qu'elles sont compatibles avec le passage au quotient et les transformer en opérations sur le quotient.

5.2. *Polynômes multivariés*

Pour simplifier l'utilisation des polynômes multivariés, on peut les voir comme un quotient de l'algèbre libre engendrée par les indéterminées et les scalaires. Cette vision des choses a l'avantage d'être complètement symétrique en les indéterminées (il n'y en a pas une qui soit prépondérante) et de coller le mieux à la

représentation qu'un humain s'en fait. On appellera le type correspondant `multi_term`.

```
Variable R : idomainType.

Inductive multi_term :=
  | Coef : R →multi_term
  | Var : nat →multi_term
  | Sum : multi_term →multi_term →multi_term
  | Prod : multi_term →multi_term →multi_term.
```

On définit alors le type `multinom` comme quotient de `multi_term` par la relation d'équivalence qui égalise deux habitants de `multi_term` représentant le même polynôme.

Pour ce faire, on commence par construire un type `multi n` des polynômes à n indéterminées, par itération de `poly`, la construction des polynômes univariés, ainsi qu'une fonction d'interpretation de `multi_term` dans `multi n`.

```
Fixpoint multi n := if n is S n' then poly (multi n')
   else R.

Fixpoint interp n (m : multi_term) : multi n :=
  match m with
    | Coef r ⇒ multiC n r
    | Var i ⇒
      (if i < n as b return (((i < n) = b →multi n))
        then fun iltn ⇒ cast_multi (subnK iltn) 'X_i
        else fun _ ⇒ 0) (refl_equal (i < n))
    | Sum p q ⇒ interp n p + interp n q
    | Prod p q ⇒ interp n p * interp n q
  end.
```

REMARQUE. —
On ne détaille pas ici les objets suivants :

– la fonction `multiC : ∀n : nat, R →multi n` qui permet de plonger `R` dans `multi n`,

– la fonction

```
> cast_multi : ∀(k j n : nat), k + j = n →
                multi j →multi n
```

qui permet de plonger `multi j` dans `multi n` (et donc, dans notre cas particulier, `multi (S i)` dans `multi n`),

– la notation `X_i` qui permet de construire le monôme unitaire de degré 1 de `poly (multi i)= multi (S i)`

– et enfin le lemme

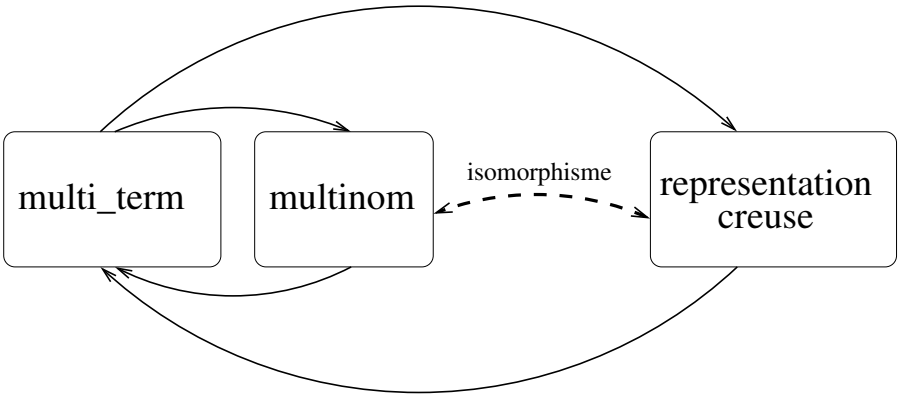
```
> subnK : ∀(i n : nat), i < n →
          (n - (S i)) + (S i) = n
```

Une fois ceci établi, on décrit la relation d'équivalence `equivm` de la manière suivante

```
Definition equivm p q := ∀n, interp n p = interp n q.
```

Alors `multinom` sera le quotient de `multi_term` par la relation `equivm`, et un grand nombre de ses propriétés seront relevées de `multi n`. Ceci permet de montrer sans trop d'efforts que le type obtenu représente un anneau intègre (pourvu que `R` le soit).

En déclarant d'autres types comme quotient de l'algèbre libre (par exemple des représentations creuses), on hérite de la structure d'anneau intègre, et de toutes les procédures liées à cette structure que l'on aurait déjà écrites sur les polynômes univariés.



Cette multiplicité des représentations qui confèrerait immédiatement un avantage calculatoire (en passant par la représentation creuse par

exemple) permettrait potentiellement de simplifier certaines preuves, puisque la réprésentation sous forme d'algèbre libre possède une symétrie dont la construction itérative ne dispose pas.
REMARQUE. —
Ce travail est actuellement en cours, d'où l'emploi du conditonnel.

6. Conclusion

Les *Interfaces Quotients* apportent un point de vue nouveau sur la question puisqu'on ne cherche pas à les construire et qu'ils s'intègrent apparemment mieux aux formalisations. La notion ne demande pas de modification théorique, ne fait pas perdre de calcul, ne demande pas d'extension de *Coq*. Il reste donc de la bureaucratie à gérer, ce qui est effectivement fait et bien caché par les *Structures Canoniques*.

Par nature, cette interface est réutilisable au dessus de n'importe quel quotient du moment qu'il a été construit et qu'on dispose d'une procédure de décision de l'égalité sur le type de base. Et il n'y a rien de plus à démontrer que les faits qui doivent de toute façon être montrés, quelque soit le formalisme.

Tous les codes se trouvent sur ma page web :
http://perso.crans.org/cohen/jfla2010/codes

Pour faire tourner les *Interfaces Quotients*, il faut se procurer *Coq* (http://coq.inria.fr/) et SSREFLECT (http://www.msr-inria.inria.fr/Projects/math-components)

Références

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now ! In *PLPV '07 : Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, New York, NY, USA, 2007. ACM.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art : the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[BCP03] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *J. Funct. Program.*, 13(2) :261–293, 2003.

[BGOP08] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Paşa. Canonical big operators. In *TPHOLs 2008*, volume 5170 of *LNCS*, pages 86–101, 2008.

[Chi03] Laurent Chicli. *Sur la formalisation des mathématiques dans le Calcul des Constructions Inductives*. PhD thesis, Université de Nice-Sophia Antipolis, Nice, 2003.

[Cou01] Pierre Courtieu. Normalized types. In *Proceedings of CSL2001*, volume 2142 of *Lecture Notes in Computer Science*, 2001.

[CPS03] Laurent Chicli, Loïc Pottier, and Carlos Simpson. Mathematical quotients and quotient types in Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, number 2646 in LNCS, pages 95–107. Springer, 2003.

[FT63] Walter Feit and John G. Thompson. Solvability of groups of odd order. *Pacific Journal of Mathematics*, 13(3) :775–1029, 1963.

[GGMR09] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009.

[GM08] Georges Gonthier and Assia Mahboubi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2008.

[GMR+07] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A modular formalisation of finite group theory. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2007.

[Hof95a] Martin Hofmann. *Extensional concepts in intensional type theory*. Phd thesis, University of Edinburgh, 1995.

[Hof95b] Martin Hofmann. A simple model for quotient types. In *Proceedings of TLCA'95, volume 902 of Lecture Notes in Computer Science*, pages 216–234. Springer, 1995.

[Saï98] A. Saïbi. *Algèbre Constructive en Théorie des Types, Outils génériques pour la modélisation et la démonstration, Application à la théorie des Catégories*. PhD thesis, Université Paris VI, 1998.

[SO08] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In *TPHOLs 2008*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, August 2008.