Université de Nice - Sophia Antipolis
UFR SCIENCES

École doctorale de mathématique de l'UNSA

THÈSE

Présentée pour obtenir le titre de :
Docteur en SCIENCES de l'Université de Nice - Sophia Antipolis

Spécialité : MATHÉMATIQUES

par

Laurent Igal CHICLI

Équipe d'accueil : LEMME - INRIA Sophia Antipolis
Equipe de géométrie algébrique - Laboratoire J.A.DIEUDONNE

Sur la formalisation des mathématiques dans le Calcul des Constructions Inductives

Soutenue publiquement au Laboratoire J.A.DIEUDONNÉ le 26 novembre 2003 à 14h00
devant le jury composé de :

M. :	Carlos	SIMPSON	Université de Nice	Président
MM. :	Henri	LOMBARDI	Université de Franche-Comté	Rapporteurs
	Christine	PAULIN-MOHRING	LRI - Orsay	
MM. :	Benjamin	WERNER	INRIA Rocquencourt	Examinateurs
	Carlos	SIMPSON	Université de Nice	
MM. :	Loïc	POTTIER	INRIA Sophia Antipolis	Directeurs
	André	HIRSCHOWITZ	Université de Nice	

A ma maman,

Remerciements

Eh bien enfin voici venu le moment d'écrire les remerciements, et de penser aux nombreuses personnes qui m'ont aidé et accompagné. Je suis plus qu'heureux d'enfin pouvoir le faire.

Et de commencer par mes deux directeurs, Loïc et André. Merci pour votre gentillesse, pour avoir su (très ?) souvent être patient, et pressant aussi quand il le fallait. Ca a été très agréable de travailler avec deux personnes toujours disponibles, prêtes à aider, qui ont sut me proposer de nouvelles voies de recherche sans jamais rien imposer.

Je veux remercier Benjamin Werner pour chacune de nos rencontres, chaque fois très enrichissantes. Merci aussi pour la fois où tu es venu, en bas du labo de maths, me rassurer sur l'intérêt de mes résultats. Je remercie aussi Henri Lombardi d'avoir bien accepté de rapporter cette thèse, vos nombreuses remarques ont largement contribué à améliorer ce document sur bien des points.

Merci aussi à Carlos Simpson d'avoir accepté de faire partie de mon jury, mais aussi et surtout merci pour tous les échanges que nous avons pu avoir.

J'ai eu la chance de passer ces dernières années dans l'équipe Lemme de l'INRIA, composée de personnes toutes à l'écoute et toujours prêtes à aider. Yves, Laurence, Laurent, Philippe, Gilles vous êtes des gens formidables (difficile de ne pas faire pompeux et grandiloquent en écrivant des phrases pareilles, mais je vous assure que c'est sincère). Merci aussi à Antonia, mon copain Kuntal, et Nicolas. Hanane et Frédérique, partager ce bureau avec vous a été un vrai bonheur. Frédérique, je ne regretterai que deux choses : ne pas avoir goûté tes pommes dauphines "maison", et ne pas avoir (malgré de nombreuses incitations) dépassé les 163 km/h avec ta twingo.

Merci aussi à mes potes du bureau 701 à la fac : Laurent, Laurent et Fabrice, avec qui j'ai partagé ma première année de thèse. On a vécu ensemble nos premiers "pfff... j'comprends rien", nos "bon sang je dois aller voir mon chef et j'ai rien fait depuis la semaine dernière" et autres "rhaaaa.. c'est la fin du monde .. ce que je fais depuis un mois est complètement faux !". Heureusement les parties de Xblast étaient là pour nous remonter le moral, merci aussi au génial inventeur de ce jeu.

Et puis y'a aussi les potes "hors contexte" que je vais honteusement remercier globalement : Michaël, Manu, Aurélien, JD, Manu, Bernard, David, Marlène, Delphine, Nat, Sarah, Fredo... Et puis aussi Judicaël, Leatitia, Manu, Sami, ? et ?. Merci les amis.

Un merci global aussi (et toujours honteux) à toute ma petite famille que j'adore : Mon père, mes frères et soeurs, Georgie, Pierrot, Stéphane, Yohann, Déborah, mon Yann, David, Sharon. Double merci à Annie et Deborah qui vont préparer le pot.

Yaël, mon coeur, merci pour ton incroyable soutien. C'est déjà grâce à toi que j'ai eu le courage de me lever ce matin là pour me rendre à mon deuxième oral du CAPES, c'est aussi grâce à toi que j'ai passé sereinement ces dernières semaines. Mais je te dois bien plus que cela. Merci du bonheur que tu m'apportes.

Et puis il y a ma maman. Je ne vais pas la remercier, ça n'aurait aucun sens pour elle, tant elle trouve les innombrables efforts qu'elle fait pour moi absolument normaux. Maman je t'aime, je te dédie simplement ce travail.

Table des matières

Chapitre 1

Introduction

Utilisée tout d'abord par les mathématiciens comme simple moyen de réaliser des calculs numériques, l'informatique a dans un premier temps évolué pour permettre le calcul symbolique et formel par manipulation de variables. Depuis une trentaine d'années cependant, les liens entre les deux disciplines se resserrent avec l'apparition des premiers logiciels d'aide à la démonstration (un des premiers et des plus célèbres, Mizar [Try] a été créé en 1978 et réunit encore aujourd'hui une grande communauté d'utilisateurs). Il ne s'agit plus d'effectuer des opérations arithmétiques ou formelles, mais de valider des démonstrations purement théoriques. Grâce à ces *assistants de preuve* on peut développer et construire des théories mathématiques abstraites et raisonner sur les objets ainsi construits pour finalement laisser le système valider les démonstrations.

Utiliser de tels logiciels demande cependant un effort conséquent. En effet, le langage utilisé par les mathématiciens est finalement (et assez paradoxalement) très ambigu : les abus de notations y sont légions, les 'trous' dans les démonstrations pas si rares, et le sens même des phrases n'est souvent pas univoque et demande au lecteur de comprendre l'*intention* du rédacteur. Il n'est donc pas question pour un système de validation de preuve d'utiliser un tel langage, le formalisme le plus explicite est nécessaire. Passer du langage mathématique à un langage non ambigu et extrêmement détaillé où chaque étape de raisonnement est l'application d'une règle de déduction logique et qui puisse donc être analysé et validé par un ordinateur, c'est le travail de *formalisation* des mathématiques, sujet de notre travail.

La première partie de ce document présente notre formalisation dans le logiciel d'aide à la démonstration Coq des mathématiques nécessaires à la définition des *schémas affines*. Pour cela algèbre, topologie et théorie des faisceaux ont été formalisés, c'est donc l'occasion de recenser les difficultés de la formalisation des mathématiques dans cet assistant.
Dans la seconde partie on présente une étude sur les possibilités existantes de formaliser les quotients dans Coq. On s'intéresse en particulier aux types quotients dans le *Calcul des Constructions Inductives* (le formalisme à la base de Coq) et on montre qu'ils ont une expressivité moindre que celles des ensembles quotients en théorie des ensembles.
Enfin on montre qu'il est impossible de formaliser les mathématiques classiques (utilisant le tiers-exclu et l'axiome du choix propositionnels) dans le *Calcul des Constructions In-*

ductives sans entrainer l'incohérence du système.

Pourquoi formaliser ?

On peut apporter plusieurs réponses à la question des motivations de l'activité de formalisation. La plus significative est sans doute dans l'idée même d'une preuve mathématique. En effet, une preuve est déclarée valide justement lorsque l'on est *convaincu* que celle-ci peut se ramener à une preuve complètement formalisée, dont chaque pas de raisonnement est une règle de logique, et dont les points acquis sont soit des axiomes de la théorie soit des théorèmes déjà démontrés. En formalisant une théorie, on concrétise finalement l'idée (aussi précise soit-elle) que l'on avait auparavant de cette théorie.
A (très ?) long terme, si l'emploi d'un tel logiciel est généralisé, c'est aussi un moyen de standardisation et de vérification, ce qui deviendra sans doute un jour nécessaire tant les branches de la discipline se sont séparées et diversifiées, et les spécialisations devenues nécessaires.
Enfin, avoir un système d'aide à la démonstration sert aussi dans un cadre éducatif. Enseigner ce qu'est une démonstration au moyen de règles de déduction ou par la signification des symboles $\exists, \forall, \Rightarrow, \Leftrightarrow, \wedge, \ldots$ sont des choses plutôt abstraites et qui ont souvent du mal à être comprises par les étudiants. Le fait de montrer concrètement comment elles s'utilisent via un logiciel qui "élimine" correctement les quantificateurs et qui génère lui-même les énoncés à démontrer peut certainement aider à leur compréhension.

Le Calcul des Constructions Inductives et Coq

L'assistant de preuves auquel on s'intéresse dans cette thèse est Coq, il est basé sur un formalisme nommé Calcul des Constructions Inductives (*CCI*). On ne décrira pas dans ce document le *CCI* en détails, mais les références abondent maintenant : on peut citer pour les plus faciles à lire le cours de Gilles Dowek sur la théorie des types [Dow] et les premiers chapitres de la thèse de Pierre Courtieu [Cou01], et pour quelque chose de plus formel, l'article de Christine Paulin-Mohring [PM93] ou la thèse de Benjamin Werner [Wer94].
Rappelons simplement qu'il s'agit tout d'abord d'une théorie des types, c'est à dire un formalisme dans lequel les objets sont des λ-termes typés. Le Calcul des Constructions [CH88] est l'extension du λ-calcul simplement typé obtenu en remplaçant le type flèche par le type produit, plus général, pour permettre la création de types dépendants : la fonction qui à un entier naturel p associe l'ensemble de ses multiples est un type dépendant : le type de retour dépend de p. Le Calcul des Constructions Inductives est une extension du Calcul des Constructions qui permet la construction de types définis inductivement à partir de constructeurs et de fonctions définies par récurrences. Par exemple on peut y définir l'ensemble Nat des entiers naturels comme le type inductif ayant pour constructeurs :

$$O : Nat \quad et \quad S : Nat \rightarrow Nat.$$

Cet objet vient avec ses principes de récursions, permettant de démontrer des propositions quantifiées sur Nat et de construire d'autres objets ou fonctions récursivement sur Nat.

Un des aspects les plus séduisants du *CCI* est que les preuves des propositions ne sont plus des idées abstraites soumises à l'approbation des mathématiciens, mais des objets concrets du formalisme. Une preuve est un objet, une proposition un type, et une preuve d'une proposition P est simplement un terme t de type P : vérifier que t est bien une preuve exacte de P revient simplement à calculer le type de t. Le fait qu'un terme t est bien la preuve de l'énoncé donné est ainsi vérifiable par calcul et n'exige donc pas d'approbation de la part du lecteur. En plus d'être validé par Coq, ce terme pourrait très bien être vérifié par un logiciel tiers.

On dispose ainsi d'une théorie qui nous offre un double aspect : un aspect déductif avec le λ-calcul qui permet via l'isomorphisme de Curry-Howard d'énoncer et prouver des propositions, et un aspect constructif avec les types inductifs qui permettent de réellement construire des objets et raisonner avec.

Des différences de taille existent néanmoins entre le *Calcul des Constructions Inductives* et la théorie des ensembles, le formalisme logique utilisé de manière générale en mathématiques.

- La présence de l'univers propositionnel `Prop`. On a dit en effet que les preuves et les propositions sont des objets du formalisme, ils sont dans un univers qui n'a pas d'équivalent en mathématique : `Prop`. Les propositions sont ainsi de type `Prop`, et les preuves ont donc pour type un terme de type `Prop` (on dit qu'ils sont de sorte `Prop`).

- L'imprédicativité de `Set`. L'univers `Set` est pour de nombreux points la sorte jumelle de `Prop`. Le fait que ce soit la sorte des objets du discours (les entiers naturels, les rationnels, les fonctions sont de sorte `Set`), découle de la propriété dite d'élimination forte qui est autorisée dans `Set`. Cette propriété stipule que des termes obtenus par des constructeurs différents dans un type inductif sont différents (ceci permet de montrer que par exemple $0 \neq 1$ dans les entiers naturels ce que l'on ne pourrait pas faire s'ils étaient construits dans `Prop`). Jusque ici, rien de bien choquant, on pourrait dire que `Set` est l'équivalent de la classe des ensembles en théorie des ensembles. Mais voilà : `Set` est imprédicatif. L'imprédicativité de `Set` signifie qu'un terme construit en quantifiant universellement sur tous les termes de `Set` est lui aussi dans `Set`. Ceci n'a évidemment aucun équivalent en théorie des ensembles, où l'objet ainsi construit serait dans la classe supérieure.

- Le *CCI* est un cadre logique purement constructif : pas de tiers-exclu ni d'axiome du choix dans le système et il est donc impossible d'y formaliser des mathématiques classiques. On peut néanmoins heureusement les rajouter au système sous forme d'axiomes, mais *quid* alors de la cohérence du système ? Nous répondrons à cette question dans le dernier chapitre de cette thèse.

Coq [COQ] est le système d'aide à la démonstration qui met en oeuvre le *CCI*. Il permet de définir les objets du discours, d'énoncer des propriétés sur ces mêmes objets, et propose un environnement interactif pour fabriquer les preuves de ces propriétés à l'aide de tactiques. Le programme génère lui même à partir de ces tactiques le λ-terme correspondant à la preuve (ou la définition), puis vérifie bien que le λ-terme ainsi créé a pour type le but fixé.

Contenu et plan de la thèse

Cette thèse s'articule autour de deux thèmes principaux : la formalisation de mathématiques dans Coq et une étude sur les types quotients et des mathématiques classiques dans le *Calcul des Constructions Inductives* .

La première partie concerne donc nos formalisations.

• Le premier chapitre décrit notre formalisation en Coq des notions mathématiques nécessaires pour formaliser les schémas affines, l'un des premiers objets du corpus de la géométrie algébrique. Algèbre, topologie et théorie des faisceaux y sont donc formalisées avant d'être finalement réunis. Ce travail, conséquent en taille, est l'occasion de faire le point sur les facilités de développement des mathématiques dans Coq. On revient donc dans le deuxième chapitre sur cette formalisation pour en décrire les résultats, les difficultés rencontrées et certaines lacunes de Coq, ainsi que les différentes astuces que nous avons utilisées.

• Le chapitre suivant s'intéresse au lemme d'Horace : ce lemme donne un moyen itératif de répondre à la question *Quelle est la dimension de l'espace vectoriel des courbes de degré d assignées à passer par n points $P_1, \ldots, P_n$ avec les multiplicités respectives $m_1, \ldots, m_n$?* Les notions de géométrie algébrique formalisées n'étant pas suffisantes pour énoncer ce lemme, on prend le problème à rebours en en proposant une axiomatisation. On a pu utiliser cette axiomatisation pour vérifier un mémoire utilisant le lemme d'Horace, mémoire dans lequel un grand nombre d'erreurs ont ainsi put être décelées.

La deuxième partie porte sur les types quotients dans Coq. En effet l'une des grandes lacunes du *CCI* (pour le mathématicien du moins) est de ne pas proposer de notion primitive pour formaliser les ensembles quotients. Des types quotients ont néanmoins été proposés et étudiés par M.Hofmann [Hof95a, Hof95b], G.Barthe [Bar95] et S.Boutin [Bou97], et nous étudions leur expressivité et leur utilisation dans le cadre du *CCI* lorsque on les ajoute au formalisme.

• Dans un premier temps, et après avoir exposé rapidement les différentes possibilités existantes pour formaliser les ensembles quotients dans Coq, on s'intéresse dans le chapitre 4 à l'aspect pratique de leur utilisation. On a ainsi programmé une librairie de tactiques et de commandes pour à la fois simplifier et rapprocher de la pratique mathématique l'utilisation des types quotients.

• Les chapitres suivants sont plus théoriques. Le premier étudie l'expressivité des types quotients dans le *CCI*. On y démontre qu'il ne peut exister dans le *CCI* de notion catégorique de type quotient qui soit aussi expressive qu'en mathématique classique (englobant à la fois la classe des ensembles quotients et leurs propriétés) sans que cela n'entraine l'incohérence du système.

• Enfin le dernier chapitre, indépendant des types quotients, reprend une idée de la démonstration précédente pour montrer l'incohérence des mathématiques classiques dans Coq. On introduit en effet une contradiction dans le *Calcul des Constructions Inductives* en y ajoutant les axiomes propositionnels du choix et du tiers-exclu. Le *fauteur de troubles*, qui est le même pour les types quotients, est néanmoins indentifié : l'imprédicativité de la sorte `Set`.

Première partie

Formalisations

Chapitre 2

Une formalisation des faisceaux et schémas affines

2.1 Introduction

Nous présentons dans ce chapitre nos spécifications et preuves en théorie des types des notions mathématiques nécessaires à la définition des schémas affines, l'un des premiers objets de la géométrie algébrique moderne.

L'interface graphique CtCoq [PC] pour Coq a été utilisée pour simplifier la création des preuves (avec 'proof-by-pointing' [BKT94]) et pour augmenter la lisibilité des énoncés (que nous voulons aussi proche que possible du langage mathématique). Dans la mesure où cela ne nuira pas à la compréhension de nos spécifications, c'est d'ailleurs l'affichage de CtCoq que nous utiliserons pour les présenter.

Ce travail est basé sur la formalisation de l'algèbre effectuée par Loïc Pottier [Pot98], qui comprend les notions de catégorie, d'ensemble, groupe, anneau, et corps, et a été effectué avec la version 6.2.3 de Coq, puis porté à la version actuelle 7.4 avec Pcoq[1].

Ce chapitre s'articule autour de six parties. La première, fondamentale, décrit la représentation des ensembles que nous utilisons. On continue ensuite avec différentes notions de topologie comme celles d'intérieur, d'adhérence, de topologie engendrée ou de continuité. Puis avec l'algèbre : idéaux premiers, maximaux, le radical d'un idéal, les anneaux locaux et la localisation d'anneaux font partie des notions nécéssaires pour commencer à formaliser les schémas affines. La quatrième partie, la plus importante de ce travail, décrit nos spécifications de la théorie des faisceaux et se termine par la construction du faisceau associé à un préfaisceau. On s'y est fortement inspiré du livre 'Algebraic Geometry' de R. Hartshorne ([Har77]). Topologie, algèbre, et théorie des faisceaux sont ensuite réunis pour permettre la définition des schémas affines.

[1]Le portage à la version 7.4 a été réalisé par Loïc Pottier

2.2 Des mathématiques classiques

Ce developpement a été réalisé dans un cadre mathématique 'classique' : on y utilise le tiers-exclu dans `Prop` :

```
Axiom EM : (P:Prop)P\/~P.
```

et l'axiome du choix, lui aussi dans `Prop` :

```
Axiom AC :
   (A,B : Type)(R:A->B->Prop)
   ((x : A)  (exT B [y : B]  (R x y))) ->
   (exT A ->  B [f : A ->  B] (x : A)  (R x (f x))).
```

On n'utilise toutefois ce dernier axiome qu'à deux reprises : lors de la construction du faisceau associé à un préfaisceau (c.f. section 2.5.6) et dans la construction du faisceau spectral (c.f. section 2.6.2).

On montrera dans le dernier chapitre de cette thèse que ces deux axiomes, ajoutés ensembles au *CCI*, impliquent une incohérence du système. Néanmoins la cause de ce paradoxe est l'imprédicativité de la sorte `Set` que nous n'avons jamais utilisée, toutes nos constructions étant directement dans `Type`.

2.3 Représentation des ensembles

On présente ici quelques aspects de la formalisation de l'algèbre en `Coq` utilisée comme base de notre travail. Le lecteur interessé pourra se reporter à [Pot98] pour plus de détails.

2.3.1 Ensembles et Sétoïdes

La manière de représenter les ensembles conditionne tout le travail et la formalisation future, il est donc essentiel que ce choix capture à la fois tous les ensembles mathématiques et qu'il soit suffisamment flexible pour ne pas trop s'éloigner, dans son utilisation, des mathématiques standards.

La notion de type semble *a priori* convenir pour représenter les ensembles : on peut voir un type comme l'ensemble des termes qui l'ont pour type, mais il n'existe malheureusement pas encore dans `Coq` de notion de soustype (pour les parties) ni de type quotient (pour formaliser les ensembles quotients) qui soit suffisamment expressive pour englober les quotients mathématiques et leurs propriétés[2].

[2]Nous verrons dans la deuxième partie de cette thèse qu'avec la hiérarchie actuelle des univers de `Coq` et l'imprédicativité de la sorte *Set* de tels types quotients ne peuvent exister sans entraîner l'inconsistance du système.

Il faut donc trouver une structure pour combler ces lacunes du CCI, et c'est celle de sétoïde (directement inspirée des "ensembles" de Bishop ([Bis67],[BB85]), et implentés dans Coq pour la première fois par G.Huet et A.Saïbi [SH95]) que l'on utilise pour représenter les ensembles.

Un sétoïde est la donnée d'un type (dit type support), d'une relation binaire sur ce type, et d'une preuve que cette relation binaire est une relation d'équivalence. Le type support sera interprété comme le type des éléments de l'ensemble, et la relation comme l'égalité de cet ensemble. On abandonne ainsi l'égalité *interne* des termes du calcul des constructions inductives (deux termes sont égaux s'ils sont convertibles), pour la relation d'équivalence des sétoïdes.

On peut alors appréhender de manière totalement homogène les ensembles quotients : l'ensemble E/R est tout simplement le sétoïde ayant le même type support que E mais où la relation initiale a été changée par R.

L'image suivante correspond à ce qui apparait sous CtCoq avec un 'pretty-printer' (un ensemble de règles d'affichages) approprié :

Definition.
a set is a structure with the following components:
Carrier: a type *(coercion)*
Equal: a type relation on Carrier
Prf_equiv: a proof that Equal is an equivalence *(coercion)*

Elle correspond en fait au code Coq suivant :

```
Record Setoid : Type  := {
Carrier :> Type;
Equal : (relation Carrier);
Prf_equiv :> (equivalence Equal) }.
```

Un mot avec cet exemple sur la syntaxe `Record` de Coq. La macro `Record` crée ici le type inductif `Setoid` ayant un unique constructeur : la fonction `Build_Setoid` qui demande trois arguments des types spécifiés dans les différents champs. L'unique moyen de construire un sétoïde est donc de fournir un type, une relation et une preuve que cette relation est d'équivalence. La macro `Record` définit aussi les trois projections `Carrier,Equal,Prf_equiv` qui associeront à un sétoïde respectivement son type, sa relation, et la preuve qu'il s'agit bien d'une relation d'équivalence.

Le symbole >présent entre `Carrier` et `Type` permet de déclarer la projection `Carrier` comme coercion entre le type `Setoid` et le type `Type` : de cette manière un sétoïde pourra être vu, à travers `Carrier`, comme un type. Plus précisément, si E est un sétoïde, on pourra alors considérer un élément x "de type" E (alors que E n'est pas un type), parce que Coq l'aura instancié automatiquement par `x :(Carrier E)`. Cette capacité de Coq à gérer les coercions ([Saï99]) est un outil considérable dans les gros dévelopements : sans cela pour voir par exemple un anneau commutatif comme un ensemble il faudrait à chaque fois considérer plusieurs termes intermédiaires.

2.3.2 Sous-typage et parties d'un ensemble

Les parties comme prédicats

Un des moyens les plus naturels de représenter une partie A d'un ensemble E est de se donner un prédicat P sur E, c'est-à-dire dans Coq un terme de type `E->Prop` (la sorte `Prop` est le type des propositions). La partie A est simplement l'ensemble des éléments de E vérifiant P.

Puisqu'on travaille avec les sétoïdes, il faut en plus demander aux prédicats que l'on considère d'être compatibles avec l'égalité de E. Soit donc la fonction `pred_compatible` qui associe à un prédicat P la proposition $\forall x, y \in E,\ si\ (P\ x)\ et\ (y = x)\ alors\ (P\ y)$:

```
Definition pred_compatible : (E->Prop) ->Prop :=
   [P:E ->Prop] (x, y:E) (P x) -> (Equal y x) ->(P y).
```

On peut alors définir ce qu'est un prédicat dans le cadre des sétoïdes : c'est un prédicat donné avec une preuve de sa compatibilité vis-à-vis de l'égalité de E :

```
Record Predicate:=
      { Pred_fun: E ->Prop;
        Pred_compatible_prf: (pred_compatible Pred_fun)}.
```

Les parties comme ensemble

Mais pour faire de la partie de E définie par un terme P de type `Predicate` un sétoïde, il faut d'abord pouvoir donner un type à ses éléments. Comme il n'existe pas en Coq de notion primitive de sous-typage, on procède ainsi :

Etant donné un type F et une fonction i de type `F->E`, on définit le sétoïde `(subtype_set E F i)` par le type F et la relation d'équivalence xRy ssi $i(x) = i(y)$. Ce sétoïde est en bijection avec I, l'image de i, et peut donc être vu comme une partie de E.

Pour implémenter la partie de E définie par le prédicat P, il suffit alors de prendre pour F le type `subtype` suivant :

```
Record subtype :={subtype_elt : E
                  subtype_prf : (Pred_fun P subtype_elt)}
```

(les termes qui l'habitent sont les couples constitués d'un terme de type E et d'une preuve qu'il vérifie le prédicat P), et pour i la projection `subtype_elt : subtype->E`. Le terme `(subtype_set E subtype subtype_elt)` sera ainsi le sétoïde défini par le prédicat P.

On met en place les coercions nécessaires pour pouvoir voir un prédicat comme une partie et inversement. L'appartenance à une partie peut alors se définir comme la fonction `in_part` qui à un sétoïde E, un terme x de type E, et à un prédicat A sur E associe la proposition `(Pred_fun A x)` :

```
Definition in_part :=
          [E:Setoid] [x:E] [A:(Predicate E)](Pred_fun A x).
```

L'ensemble des parties

Le sétoïde définissant l'ensemble des parties de E a pour type `Predicate` et pour égalité la double inclusion :

```
Definition eq_part: (Predicate E) -> (Predicate E) ->Prop :=
  [A:(Predicate E)] [B:(Predicate E)] (x:E)
  ((in_part x A)->(in_part x B))/\((in_part x B)->(in_part x A)).
```

On montre que la double inclusion `eq_part` est bien une relation d'équivalence :

```
Lemma eq_part_equiv: (equivalence eq_part).
```

Et on peut alors définir l'ensemble des parties de E par :

```
Definition part_set: Setoid := (Build_Setoid eq_part_equiv).
```

Dans la définition de `part_set` ci-dessus on ne donne à Coq qu'un seul des trois arguments nécessaires à la fonction `Build_Setoid`. En effet, les deux arguments manquants (le type `(Predicate E)` et la relation `eq_part`) sont implicitement contenus dans le type de `eq_part_equiv`, et sont automatiquement synthétisés par Coq. Cette capacité de Coq à rechercher les arguments implicites allège considérablement le travail de formalisation, on l'utilise chaque fois que cela est possible.

Remarquons que la fonction `subtype_elt` nous permet de voir un élément d'une partie comme un élément de l'ensemble au-dessus : si `A :(part_set E)` et si `x :A` alors `(subtype_elt x) :E`. Malheureusement, comme on le verra dans le chapître 3, Coq ne permet pas de déclarer cette fonction comme coercion (elle serait paramétrée par le sétoïde E, ce qui n'est pas accepté par `Coq`). On doit ainsi à chaque fois explicitement faire appel à la fonction `subtype_elt`.

On choisit d'afficher sous `CtCoq` $x \in A$ pour le terme `(in_part x A)`, $\mathcal{P}(E)$ pour `(part_set E)`, et de cacher la fonction `subtype_elt` pour plus de lisibilité.
De même, si A et B sont deux parties de E, P un élement de A et h une preuve que `(subtype_elt P)` appartient aussi à B, on affichera $P_{[in\ B]}$ pour le terme `( !Build_subtype E B (subtype_elt P) h)`.

2.4 Topologie

2.4.1 Topologie, ensembles ouverts comme type

Une topologie sur un ensemble E est une partie de l'ensemble des parties de E, contenant E et l'ensemble vide, et vérifiant deux propriétés : stabilité par union quelconque, et par intersection finie. On commence donc par définir ce qu'est pour un ensemble de parties la stabilité par union quelconque (la propriété `union_in_top` ci-dessous) et par intersection finie (`inter_in_top`).

Soit `union_part` la fonction qui associe à une famille de parties de E l'union de tous ses éléments :

Let E be a set.
Let P be of type $\wp(\wp(E))$.
Definition.
union_part: $\wp(E)$
1: `Apply` (!Build_Predicate ? x:E , $\exists$ A:$\wp(E)$, (A $\in$ P) $\wedge$ (x $\in$ A)).

Comme à chaque fois que l'on définit un prédicat, il faut montrer qu'il est compatible avec l'égalité que l'on considère. C'est évidement le cas de la définition de `union_part`, bien que l'on ne reproduise pas la preuve ici. On peut alors définir `union_in_top` :

Definition (union_in_top).
given top of type $\wp(\wp(E))$,
we define the proposition union_in_top(top) by
$\forall$ open_family:$\wp(\wp(E))$, open_family $\subset$ top $\Rightarrow$ union_part(open_family) $\in$ top

et :

Definition (inter_in_top).
given top of type $\wp(\wp(E))$,
we define the proposition inter_in_top(top) by
$\forall$ open1,open2:$\wp(E)$, open1 $\in$ top $\Rightarrow$ open2 $\in$ top $\Rightarrow$ open1 $\cap$ open2 $\in$ top

Une topologie sur E est alors :

Definition.
a Topology is a structure with the following components:
top: of type Predicate($\wp$(E)) *(coercion)*
union_in_top_prf: a proof of union_in_top(top)
inter_in_top_prf: a proof of inter_in_top(top)
total_prf: a proof of full(E) $\in$ top
empty_prf: a proof of $\varnothing$ $\in$ top

et un espace topologique est un ensemble donné avec une topologie :

Definition.
a topological space is a structure with the following components:
et_setoid: a set *(coercion)*
open: of type Topology(et_setoid)

On déclare une coercion entre le type `et` et le type `setoid`. Ainsi si `X :et` est un espace topologique alors :

- un terme `x :X` (automatiquement coercé vers `x :(Carrier (et_setoid X))`) sera un élément du sétoïde sous-jacent,

- un terme `U :(open X)` (coercé vers `(top (open X))`) sera un élément de la topologie, donc un ouvert, comme en langage mathématique courant.
  Une différence cependant : `U` ne peut être considéré comme une partie de `X`. En effet `(top (open X))` est une partie de l'ensemble des parties de `X` (de type `(Predicate (part_set X))`) et à chaque fois que l'on doit voir un ouvert comme une partie c'est donc `(subtype_elt U)` qu'il faut considérer.

2.4.2 Adhérence, intérieur, bord

Les notions de fermé, adhérence, intérieur, bord, ainsi que tous les lemmes associés s'expriment clairement. Par exemple, pour définir l'adhérence d'une partie A de E, on commence par construire l'ensemble `closed_containing` des fermés contenant A, puis on prend l'intersection de ces fermés :

Let E be a topological space.
Let A be of type $\wp$(E).
Definition.
closed_containing: $\wp(\wp$(E))
1: `Apply` (!Build_Predicate ? F: $\wp$(E) , A $\subset$ F $\wedge$ F is closed).

Definition.
let adh be intersection_part(closed_containing)
(its type is $\wp$(E)).

On montre alors les lemmes classiques ainsi que la propriété caractéristique de l'adhérence : c'est le plus petit fermé qui contienne A.

Lemma (included_adh).
A $\subset$ adh.

Lemma (adh_closed).
$\forall$ A: $\wp$(E) , adh(A) is closed.

Lemma (closed_containing_adh).
$\forall$ F,G: $\wp$(E) , F is closed $\Rightarrow$ G $\subset$ F $\Rightarrow$ adh(G) $\subset$ F.

L'intérieur d'une partie A s'exprime de façon similaire : c'est l'union de tous les ouverts inclus dans A.

Definition.
open_included: $\wp(\wp$(E))
1: `Apply` (!Build_Predicate $\wp$(E) U: $\wp$(E) , U $\subset$ A $\wedge$ U is open).
Definition.
let int be union_part(open_included)
(its type is $\wp$(E)).

Et, comme pour l'adhérence :

Lemma (int_included).
$\forall$ A,B: $\wp$(E) , A $\subset$ B $\Rightarrow$ int(A) $\subset$ int(B).

Lemma (int_open).
$\forall$ A: $\wp$(E) , int(A) is open.

Lemma (eq_set_int).
$\forall$ A: $\wp$(E) , A is open $\Rightarrow$ A = int(A).

Lemma (open_included_int).
$\forall$ A,B: $\wp$(E) , B is open $\Rightarrow$ B $\subset$ A $\Rightarrow$ B $\subset$ int(A).

On peut alors définir le bord d'une partie et la propriété de densité :

Definition.
let border be the function defined by , for all A:Predicate(E) , adh(A) $\cap$ $\neg$int(A)
(its type is $\wp$(E) $\Rightarrow$ $\wp$(E)).
Definition.
let is_dense be the function defined by , for all A,B:$\wp$(E) , B $\subset$ adh(A)
(its type is $\wp$(E) $\Rightarrow$ $\wp$(E) $\Rightarrow$ Prop).

et montrer les lemmes généraux concernant ces notions[3].

2.4.3 Outils pour construire une topologie

Il arrive que l'on construise les espaces topologiques en spécifiant non pas l'ensemble des ouverts mais celui des fermés, ou en considérant la topologie engendrée par une base d'ouvert. On décrit dans cette section les librairies qui nous permettent de telles constructions.

2.4.3.1 Topologie définie par l'ensemble des fermés

Si E est un sétoïde et F un ensemble de parties de E candidat à former l'ensemble des fermés d'une topologie sur E, il faut qu'il vérifie les quatre propriétés suivantes : le vide et E appartiennent à F, stabilité de F par intersection quelconque et union finie. On se donne donc en paramètre les preuves `F_empty`, `F_full`, `F_prop1`, et `F_prop2` de ces propositions :

Definition (infinite_inter_closed).
given close of type $\wp$($\wp$(E)) ,
we define the proposition infinite_inter_closed(close) by
$\forall$ closed_family:$\wp$($\wp$(E)) ,
closed_family $\subset$ close $\Rightarrow$ intersection_part(closed_family) $\in$ close
Definition (finite_union_closed).
given close of type $\wp$($\wp$(E)) ,
we define the proposition finite_union_closed(close) by
$\forall$ closed1,closed2:$\wp$(E) ,
closed1 $\in$ close $\Rightarrow$ closed2 $\in$ close $\Rightarrow$ (closed1 $\cup$ closed2) $\in$ close
Let F_empty be of type $\emptyset$ $\in$ F.
Let F_full be of type full(E) $\in$ F.
Let F_prop1 be of type infinite_inter_closed(F).
Let F_prop2 be of type finite_union_closed(F).

Soit `complementary` la fonction qui à un ensemble de parties associe l'ensemble des complémentaires de ces parties :

Definition.
complementary:$\wp$($\wp$(E)) $\Rightarrow$ $\wp$($\wp$(E))
1: `Intros` F. 1: `Apply` (!Build_Predicate ? C:$\wp$(E) , $\neg$C $\in$ F).

[3]Pour ne pas alourdir inutilement la description de cette formalisation, on ne donnera pas la liste exhaustive des différents lemmes et théorèmes démontrés, mais juste les premiers ou les plus expressifs d'entre eux.

On peut alors démontrer les quatres lemmes suivant :

Lemma (topology_by_closed_1).
union_in_top(complementary(F)).

Lemma (topology_by_closed_2).
inter_in_top(complementary(F)).

Lemma (topology_by_closed_3).
full(E) $\in$ complementary(F).

Lemma (topology_by_closed_4).
$\emptyset$ $\in$ complementary(F).

qui font de `(complementary F)` une topologie sur E :

Definition.
let Build_topology_by_closed be
(!Build_Topology
E complementary(F) topology_by_closed_1 topology_by_closed_2
topology_by_closed_3 topology_by_closed_4).

2.4.3.2 Topologie engendrée par une base d'ouverts

Dans notre développement, une base B d'ouverts est, comme une topologie, une partie de l'ensemble des parties de E, contenant E et l'ensemble vide, mais ne vérifiant que la stabilité par rapport aux intersections finies. C'est justement en considérant l'ensemble des parties engendrées par union quelconque d'éléments de B que l'on va obtenir une topologie dite "engendrée par B".

Definition.
a open_base is a structure with the following components:
open_base_setoid: of type Predicate($\wp$(E)) *(coercion)*
base_inter_in_top_prf: a proof of inter_in_top(open_base_setoid)
base_total_prf: a proof of full(E) $\in$ open_base_setoid
base_empty_prf: a proof of $\emptyset$ $\in$ open_base_setoid

Et on définit :

Let B be of type open_base.
Definition.
generated_part_by_union: $\wp(\wp(E))$
1: `Apply` (!Build_Predicate ? x:$\wp$(E) , $\exists$ Y:$\wp(\wp(E))$, Y $\subset$ B $\wedge$ (x = union_part(Y))).

Il ne reste plus qu'à démontrer que `generated_part_by_union` est bien une topologie :

Lemma (full_in_top_for_bases).
full(E) $\in$ generated_part_by_union.

Lemma (empty_set_in_top_for_bases).
$\varnothing$ $\in$ generated_part_by_union.

Lemma (union_in_top_for_bases).
union_in_top(generated_part_by_union).

Lemma (inter_in_top_for_bases).
inter_in_top(generated_part_by_union).

pour pouvoir définir la topologie engendrée :

Definition.
let generated_topology_o be
(!Build_Topology
E generated_part_by_union(B) union_in_top_for_bases inter_in_top_for_bases
full_in_top_for_bases emptyset_in_top_for_bases)
(its type is Topology(E)).

On obtient ainsi un terme `generated_topology_o` qui nous permettra de construire la topologie engendrée par B à condition de vérifier les propriétés nécessaires.

2.4.4 Continuité

On garde la définition la plus générale de la continuité : une fonction $f : E \to F$ est continue en un point a si quelque soit U ouvert de F contenant $f(a)$, $f^{-1}(U)$ est un ouvert de E.

Definition (cont_in_pt).
given f of type (Map E F) and a of type E ,
we define the proposition (cont_in_pt f a) by
$\forall$ W:open of F , f(a) $\in$ W $\Rightarrow$ f-1(W) is open

Et une application est continue si elle est continue en chacun des points de son domaine de définition :

Definition (cont).
given f of type (Map E F) ,
we define the proposition cont(f) by $\forall$ W:open of F , f-1(W) is open

Lemma (cont_then_conteverywhere).
$\forall$ f:(Map E F) , f is continuous $\leftrightarrow$ $\forall$ a:E , (cont_in_pt f a).

On fait bien sûr le lien avec les autres notions, équivalentes, de continuité (image inverse d'un fermé, caractérisation par l'adhérence) :

Lemma (cont_with_closed).
$\forall$ f:(Map E F) , f is continuous $\Rightarrow$ $\forall$ C:closed(F) , f^{-1}(C) is closed.

Lemma (im_of_adh).
$\forall$ f:(Map E F) , f is continuous $\Rightarrow$ $\forall$ C:$\wp$(E) , f(adh(C)) $\subset$ adh(f(C)).

Lemma (cont_with_adh_inv).
$\forall$ f:(Map E F) , ($\forall$ C:$\wp$(F) , f^{-1}(adh(C)) = adh(f^{-1}(C))) $\Rightarrow$ f is continuous.

2.5 Algèbre

La géométrie algébrique a pour objet de discuter 'algébriquement' de questions géométriques. Il est donc nécessaire de développer des librairies d'algèbre aussi complètes que possible. En particulier les anneaux, les idéaux, et certaines de leurs propriétés vont jouer un rôle primordial. On décrit dans ce chapitre les notions d'idéaux premiers, maximaux, les radicaux par rapport à une partie multiplicative quelconque, et notre formalisation de la localisation d'anneaux et d'anneaux locaux.

2.5.1 Idéaux, idéaux premiers, maximaux

On reprend la formalisation des idéaux de la librairie Algebra : un idéal est la donnée d'une partie d'un anneau contenant zéro et stable pour l'addition, vérifiant la propriété `is_ideal` des idéaux :

Let R be a ring.
Definition.
let is_ideal be
the function defined by , for all I:Predicate(R) , $\forall$ x:R , x $\in$ I $\Rightarrow$ $\forall$ a:R , a*x $\in$ I.
Definition.
a ideal is a structure with the following components:
ideal_set: of type Predicate(R) *(coercion)*
ideal_zero_prop: of type 0_R $\in$ ideal_set
ideal_sum_prop: of type $\forall$ x,y:R , x $\in$ ideal_set $\Rightarrow$ y $\in$ ideal_set $\Rightarrow$ x+y $\in$ ideal_set
ideal_prf: of type is_ideal(ideal_set)

Idéaux premiers. Un idéal premier I est un idéal propre vérifiant la propriété suivante : si $x * y$ appartient à I alors x ou y appartient à I.

Let R be a commutative ring.
Definition.
let is_prime be
the function defined by , for all I:ideal(R) , $\forall$ x,y:R , x*y $\in$ I $\Rightarrow$ (x $\in$ I) $\vee$ (y $\in$ I).
Definition.
let propre be the function defined by , for all I:$\wp$(R) , I $\neq$ full(R).

Definition.
a prime_ideal is a structure with the following components:
prime_ideal_ideal: of type ideal(R) *(coercion)*
prime_ideal_prop: a proof of prime_ideal_ideal is a prime ideal
prime_ideal_propre: a proof of prime_ideal_ideal is a proper ideal

Lemma (prime_ideal_propre2).
$\forall$ p:prime_ideal , 1_R $\notin$ p.

Lemma (is_prime_reverse).
$\forall$ p:prime_ideal , $\forall$ x,y:R , y $\notin$ p $\wedge$ x $\notin$ p $\Rightarrow$ x*y $\notin$ p.

D'autres lemmes :

Lemma (inter_included_prime).
$\forall$ p:prime ideal of R , $\forall$ I,J:ideal(R) , (I $\cap$ J) $\subset$ p $\Rightarrow$ (I $\subset$ p) $\vee$ (J $\subset$ p).

Puis les lemmes qui caractérisent les idéaux premiers en termes d'anneaux quotients (attention : ici la proposition `I is a prime ideal` correspond simplement à la proposition `(is_prime I)`).

Definition.
let idomain_proposition be
the function defined by , for all R:CRING , $\forall$ x,y:R , x $\neq$ 0_R $\Rightarrow$ y $\neq$ 0_R $\Rightarrow$ x*y $\neq$ 0_R.

Lemma (prime_integral).
I is a prime ideal $\Rightarrow$ R/I is an integral domain.

Lemma (integral_prime).
R/I is an integral domain $\Rightarrow$ I is a prime ideal.

Idéaux maximaux. Un idéal maximal est un idéal propre et maximal pour la relation d'inclusion entre idéaux :

Definition.
let is_maximal be
the function defined by , for all I:ideal(R) , $\forall$ J:ideal(R) , I $\subset$ J $\Rightarrow$ I=J $\vee$ (J = full(R)).

Definition.
a maximal_ideal is a structure with the following components:
maximal_ideal_ideal: of type ideal(R) *(coercion)*
maximal_ideal_prop: of type maximal_ideal_ideal is a maximal ideal
maximal_ideal_propre: of type maximal_ideal_ideal is a proper ideal

Une autre façon d'exprimer qu'un idéal I est maximal est de dire que l'idéal engendré par I et un élément quelconque x de R n'appartenant pas à I est l'anneau R tout entier :

Lemma (maximal_prop1).
I is a maximal ideal ⇒ ∀ x:R , x ∉ I ⇒ full(R) = I+<{x}>.

Lemma (maximal_prop1_rev).
(∀ x:R , x ∉ I ⇒ full(R) = I+<{x}>) ⇒ I is a maximal ideal

On définit ensuite la proposition `field_prop` : R est un corps si tout élément non nul est inversible :

Definition.
invertible:Predicate(R)
1: `Apply` (!Build_Predicate ? x:R , ∃ z:R , (x*z = 1_R) ∧ (z*x = 1_R)).
Definition.
let field_prop be
the function defined by , for all R:ring , ∀ x:R , x ≠ 0_R ⇒ x ∈ invertible(R).

Et à l'aide des lemmes précédents on peut démontrer la caractérisation des idéaux maximaux en termes d'anneaux quotients :

Lemma (maximal_field_prop).
I is a maximal ideal ⇒ R/I is a field

2.5.2 Opérations sur les idéaux

Considérons un anneau commutatif R, on commence par construire le sétoïde `Set_Ideal` des idéaux de R (l'égalité y est évidemment celle des parties de R) :

Let R be a commutative ring.
Definition.
Set_Ideal:set
1: `Apply` (!Build_Setoid ideal(R) !Equal(℘(R))).

Somme d'idéaux

Soit `ideal_family` une famille d'idéaux dont on veut définir la somme :

Let ideal_family be of type ℘(Set_Ideal).

On peut construire la somme des idéaux de `ideal_family` comme l'intersection des idéaux qui les contiennent tous. On construit donc d'abord l'ensemble `ideal_containing` des idéaux qui contiennent tous les idéaux de `ideal_family`, puis on en prend l'intersection. Notons qu'à ce niveau on ne peut pas utiliser la fonction `intersection_part` qui construit justement l'intersection de parties de R : une famille d'idéaux n'est pas vu comme un ensemble de parties de R.

Definition.
ideals_containing: ℘(Set_Ideal)
1: `Apply` (!Build_Predicate Set_Ideal a:ideal(R) , ∀ i:ideal_family , i ⊂ a).

Definition.
sum_ideals:ideal(R)
1: `Vinyl2` x:R , ∀ a:ideals_containing , x ∈ a.

On montre ensuite les deux propriétés caractéristiques de la somme d'idéaux. La première : tous les idéaux dont on a fait la somme sont inclus dans la somme :

Lemma (sum_ideals_prop1).
∀ a:ideal_family , a ⊂ sum_ideals.

La seconde : la somme des idéaux est bien le plus petit idéal contenant tous les idéaux dont on a fait la somme.

Lemma (sum_ideals_prop2).
∀ I:ideal(R) , (∀ J:ideal_family , J ⊂ I) ⇒ sum_ideals ⊂ I.

2.5.3 Radicaux

Élévation à la puissance dans un anneau La fonction `ring_power` ci-dessous élève l'élément r à la puissance n (on pose comme classiquement en mathématique $r^0 = 1_R$) :

Let R be a ring.
Let r be of type R.
Definition:
Let ring_power be the recursive function, with value into R,
mapping n:nat to:
Cases of n :

O=>1_R
(S n')=>ring_power(n')*r

Quelques lemmes :

Lemma (ring_power_comp).
$\forall$ R:ring , $\forall$ n1,n2:nat , $\forall$ x,y:R , x = y $\Rightarrow$ <nat> n1 = n2 $\Rightarrow$ x ^ $_{n1}$ = y ^ $_{n2}$.

Lemma (ring_one_power).
$\forall$ R:ring , $\forall$ n:nat , 1$_R$ ^ $_n$ = 1$_R$.

Lemma (ring_zero_power).
$\forall$ R:ring , $\forall$ n:nat , n >=1 $\Rightarrow$ 0$_R$ ^ $_n$ = 0$_R$.

Pour un anneau commutatif on a de plus :

Lemma (cring_power_morphism).
$\forall$ R:cring , $\forall$ n:nat , $\forall$ x,y:R , x*y ^ $_n$ = x ^ $_n$*y ^ $_n$.

Enfin un idéal $\mathfrak{p}$ premier vérifie :

Lemma (prime_power).
$\forall$ R:cring , $\forall$ x:R , $\forall$ p:prime ideal of R , $\forall$ n:nat , x ^ $_n$ $\in$ p $\Rightarrow$ x $\in$ p.

Parties multiplicatives

Une partie multiplicative d'un anneau R est une partie contenant l'unité et stable pour la multiplication :

Let R be a ring.
Definition.
a multiplicative_part is a structure with the following components:
multi_part: of type Predicate(R) *(coercion)*
multi_part_one: of type 1$_R$ $\in$ multi_part
multi_part_prop: of type $\forall$ x,y:R , x $\in$ multi_part $\Rightarrow$ y $\in$ multi_part $\Rightarrow$ x*y $\in$ multi_part

Si M est une telle partie multiplicative, on fournit les fonctions qui permettent de voir respectivement le produit de deux éléments de M et l'unité comme terme de type M.

Let M be of type multiplicative_part.
Definition.
let build_elt_mp_prod be the function defined by , for all m1,m2:M , (m1*m2$_{[\text{in } M]}$).
Definition.
let multi_part_unit be Build_subtype(multi_part_one(M)).

Et on construit `multi_part_min` la partie multiplicative minimale : celle qui ne contient que 1_R.

Definition.
multi_part_min:multiplicative_part
1: `Apply` !Build_multiplicative_part({1_R}). 1: Abstract (`Auto with` algebra).

Radical d'un idéal

On peut maintenant définir le radical d'un idéal relativement à une partie multiplicative M quelconque :

Definition.
radical: $\wp$(R)
1: `Apply` (!Build_Predicate ? x:R , ∃ m:M , ∃ n:nat , m*x ^ n ∈ I).

et démontrer les lemmes suivants :

Lemma (included_radical).
I ⊂ radical.
1: `Red`.

Lemma (radical_increasing).
I ⊂ J ⇒ (radical I M) ⊂ (radical J M).

Si R est commutatif on a aussi :

Lemma (radical_multiplicative).
∀ R:cring ,
∀ M:multiplicative part of R ,
∀ I:ideal(R) , ∀ x:(radical I M) , ∀ y:R , x*y ∈ (radical I M).

Pour le radical relatif à la partie multiplicative minimale `(radical I (multi_part_min R))` et que l'on note $\sqrt{I}$ sous `CtCoq` , on a aussi :

Lemma (radical_propre).
√I = full(R) ⇒ I = full(R).

Lemma (radical_to_natural_def).
∀ x:R , x ∈ √I ⇒ ∃ n:nat , x ^ n ∈ I.

Lemma (natural_def_to_radical).
∀ x:R , (∃ n:nat , x ^ n ∈ I) ⇒ x ∈ √I.

2.5.4 Localisation

2.5.4.1 Le localisé par rapport à une partie multiplicative

Etant donné un anneau commutatif `R` et une partie multiplicative `M` de `R`, le localisé de `R` par rapport à `M` est le quotient de `R x M` par la relation :

$$(x_1, x_2)R(y_1, y_2) \Leftrightarrow \exists m \in M \mid m(x_1 * y_2 - y_1 * x_2) = 0.$$

On commence donc par définir cette relation :

Let R be a commutative ring.
Let M be a multiplicative part of R.
Definition.
let eq_loc be
the function defined by , for all x,y:R × M , ∃ m:M , m * (x_1 * y_2 + −(y_1 * x_2)) = 0
(its type is R × M ⇒ R × M ⇒ Prop).

Puis par prouver que c'est bien une relation d'équivalence pour pouvoir considérer le quotient comme ensemble :

Lemma (eq_loc_equiv).
eq_loc is an equivalence.

Definition.
let localize_set be R × M/Build_Relation(eq_loc_comp).

On définit ensuite l'addition, la multiplication, l'opposé. Il faut à chaque fois prouver que la fonction est compatible avec l'égalité du quotient.

Lemma (localize_add_compatible).
(!fun2_compatible ? ? ? a,b:localize_set , b_2 * a_1 + a_2 * b_1/(a_2) * (b_2)).

Lemma (localize_mult_compatible).
(!fun2_compatible ? ? ? a,b:localize_set , b_1 * a_1/(a_2) * (b_2)).

Definition.
let localize_opp_fun be the function defined by , for all x:localize_set , −x_1/x_2
(its type is localize_set ⇒ localize_set).

Suivent les dix propriétés des anneaux :

Lemma (localize_ring1).
∀ x,y,z:localize_set ,
(localize_add (localize_add x y) z) = (localize_add x (localize_add y z)).

Lemma (localize_ring2).
∀ x:localize_set , (localize_add x 0/1_M) = x.

Lemma (localize_ring4).
∀ x:localize_set , (localize_add x localize_opp(x)) = 0/1M.

...

Lemma (localize_ring10).
∀ x,y,z:localize_set ,
(localize_mult (localize_add x y) z) =
(localize_add (localize_mult x z) (localize_mult y z)).

Pour finalement construire l'anneau puis l'anneau commutatif :

Definition.
localize_ring:ring

```
1: Apply (!BUILD_RING
        localize_set curry(localize_add) curry(localize_mult) 0/1M 1R/1M localize_opp).
```

Definition.
localize:cring

```
1: Apply !Build_cring(localize_ring).
```

2.5.4.2 Propriété universelle du localisé

Soit φ l'application de R vers $M^{-1}R$ qui envoie a sur $\frac{a}{1}$:

Definition.
localize_injection:a ring homomorphism : R→localize(M)

```
1: Apply (!BUILD_HOM_RING R M-1R a:R , a/1M).
```

On la note φ_M sous CtCoq, comme usuellement en mathématique.

Le localisé de R par rapport à M vérifie la propriété universelle suivante : pour tout morphisme d'anneau $f : R \to B$ tel que les éléments de $f(M)$ soient inversibles dans B, il existe un unique morphisme d'anneau $g : M^{-1}R \to B$ tel que $f = g \circ \varphi$. L'application g est en fait définie par $g(\frac{r}{m}) = f(r) * m^{-1}$.

On dispose comme souvent de deux possibilités de formalisation, selon que l'on veuille une version constructive ou non de cet énoncé.

La version non constructive serait si '$\forall m \in M,\ f(m)$ est inversible dans B' c'est-à-dire si '$\forall m \in M, \exists n \in B \mid f(m) * n = 1$' alors *il existe* g tel que $f = g \circ \varphi$. Notons que cette propriété "$\forall m \in M, \exists n \in B \mid f(m) * n = 1$" seule ne suffit pas : on ne peut pas construire la fonction g, même dans un contexte propositionel. En effet, pour passer de la proposition "$\forall m \in M, \exists n \mid \ldots$" à l'existence d'une fonction inv (nécessaire pour construire g) telle que $inv(m)$ soit pour tout m dans M l'inverse de m, il faut de plus utiliser l'axiome du choix dans Prop.

La version constructive que l'on a choisie de formaliser prend donc directement en entrée la fonction $inv : R \to B$ telle que $\forall m \in M\ f(m) * inv(m) = 1_B$, et construit la fonction g :

Definition.
univ_property_local_map:∀ B:cring ,
∀ f:a ring homomorphism : R→B ,
∀ invB:M ~→ B ,
(∀ m:M , f(m) * invB(m) = 1B) ⇒ localize(M) ~→ B
1: `Intros` B f invB H'. 1: `Apply` (!Build_Map ? ? (r/m):M−1R, f(r) * invB(m)).

Et on peut ainsi démontrer la propriété universelle :

Theorem (univ_property_localization).
∀ B:cring ,
∀ f:a ring homomorphism : R→B ,
∀ invB:M ~→ B ,
∀ H:∀ m:M , f(m) * invB(m) = 1B , (univ_property_local_map B f invB H) o φM = f.

2.5.5 Anneaux locaux

2.5.5.1 Définition et spécification

Un anneau local est un anneau qui ne contient qu'un idéal maximal, que l'on a souvent l'habitude en mathématique de considérer. Pour mimer ceci on peut définir une anneau local comme la donnée (toujours via la macro `Record`) d'un anneau R, d'un idéal maximal $\mathfrak{m}$ de R, et d'une preuve d'unicité des idéaux maximaux : si $\mathfrak{m}_1$ et $\mathfrak{m}_2$ sont deux idéaux maximaux de R, alors $\mathfrak{m}_1 = \mathfrak{m}_2$.

Definition.
let is_local be
the function defined by , for all R:cring , ∀ m1,m2:maximal ideal of R , m1=m2.

Definition.
a local_ring is a structure with the following components:
lr_ring: a commutative ring *(coercion)*
lr_maximal_ideal: a maximal ideal of R
lr_prop: of type is_local(lr_ring)

Cependant, plus tard dans notre développement (lors de la définition des espaces annelés) on a besoin d'écrire un terme qui énonce que "tel anneau R est local", et si `(is_local R)` semble correspondre, cela ne suffit cependant pas pour voir l'anneau en question comme un anneau local (puisqu'il manque la donnée de l'idéal maximal). On doit donc renoncer à inclure la donnée de l'idéal maximal dans la définition des anneaux locaux :

Definition.
a local_ring is a structure with the following components:
lr_ring: a commutative ring *(coercion)*
lr_prop: of type is_local(lr_ring)

En contrepartie, on ne pourra pas parler de l'idéal maximal d'un anneau local, et les lemmes et propriétés de l'unique idéal maximal d'un anneau local R devront en fait être quantifiés sur tous les idéaux maximaux de R. Par exemple, le théorème qui énonce que l'ensemble des inversibles d'un anneau local est le complémentaire de son idéal maximal est formalisé ainsi :

Lemma (local_ring_invertibles).
$\forall$ R:local_ring , $\forall$ M:maximal ideal of R , invertible(R) = $\neg$M.

On se donne quand même la possibilité de choisir un idéal maximal dans un anneau pour démontrer des propositions, avec l'axiome suivant :

Axiom ex_maximal_ideal:
$\forall$ R:cring , $\forall$ I:ideal(R) , I is a proper ideal $\Rightarrow$ $\exists$ M:maximal ideal of R , I $\subset$ M.

à "valeur" dans `Prop`. Ainsi, on pourra éliminer le quantificateur existentiel (et ainsi disposer de l'idéal maximal dans notre contexte) pour construire pour un terme de type `Prop`, c'est-à-dire pour démontrer une proposition, mais pas pour construire d'autres termes dans `Set` ou `Type`.

2.5.5.2 Le localisé en un idéal premier comme anneau local

On peut alors définir le localisé d'un anneau en un idéal premier $\mathfrak{p}$ comme anneau local : on commence par construire le complémentaire de $\mathfrak{p}$ comme partie multiplicative :

Let R be a commutative ring.
Let p be a prime ideal of R.
Definition.
mp_compl_prime:multiplicative part of R
1: `Apply` (!Build_multiplicative_part R $\neg$p).

L'idéal maximal du localisé est en fait l'ensemble des fractions $\frac{x_1}{x_2}$ tels que $x_1 \in \mathfrak{p}$:

Definition.
localize_prime_maximal:maximal ideal of localize(mp_compl_prime)
1: `Vinyl` x:localize(mp_compl_prime) , $x_1 \in$ p.

On montre alors qu'il est le seul idéal maximal du localisé :

Lemma (local_prime_proof1).
∀ m1:maximal ideal of localize(mp_compl_prime) , m1=localize_prime_maximal.

ce qui fait du localisé un anneau local :

Definition.
localize_prime:local_ring
1: `Apply` !Build_local_ring(localize(mp_compl_prime)).

2.6 Théorie des faisceaux

2.6.1 Catégorie associée à un espace topologique

2.6.1.1 Définition

Soit X un espace topologique. Pour définir la notion de préfaisceau sur X, on doit d'abord définir la catégorie `Top_cat` dont les objets sont les ouverts de X, et dont l'ensemble des morphismes $Hom(U, V)$ entre deux ouverts est soit le singleton contenant l'injection canonique $U \hookrightarrow V$ si $U \subset V$, soit l'ensemble vide.

2.6.1.2 Une première idée

Formaliser l'ensemble des objets ne pose pas de problème, il s'agit de la topologie elle-même. Mais pour l'ensemble des morphismes, la discussion sur l'inclusion éventuelle de U dans V est problématique.

La première idée serait de se donner en paramètre une fonction de décision sur l'inclusion :

Parameter (incl_dec). ∀ U,V:open of X , { U ⊂ V }+{ ¬U ⊂ V }

Dans le *Calcul des Constructions Inductives* l'existence d'une telle fonction signifie que l'on peut décider constructivement (c'est-à-dire sans utiliser le tiers-exclu ou l'axiome du choix) de l'inclusion de deux ouverts et permet de construire d'autres objets en raisonnant par cas. On peut donc s'en servir pour définir $Hom(U, V)$.

Cette spécification a deux inconvénients. Le premier est que cette fonction de décision doit être dans la plupart des cas considérée comme un axiome car l'inclusion entre deux parties d'un ensemble n'est pas décidable en général. Le second est qu'elle oblige régulièrement à travailler avec des cas vides (dans le cas où il n'y a pas inclusion $Hom(U, V) = \emptyset$), cas résolubles aisément avec `Coq` , mais pénibles à la longue et que l'on ne considère jamais en mathématiques. Pour définir la composition de morphismes $Hom(U, V) \times Hom(V, W) \rightarrow$

$Hom(U,W)$ on doit par exemple distinguer quatre cas selon les différentes possibilités d'inclusion de U dans V et V dans W.

2.6.1.3 Spécifications

Notre solution[4] consiste à considérer $Hom(U,V)$ comme le sétoïde dont les éléments sont les preuves, toutes égales, que $U \subset V$. Ceci est tout à fait possible dans le *Calcul des Constructions Inductives* où les preuves sont des objets comme les autres. Ainsi que U soit ou non inclus dans V la spécification de $Hom(U,V)$ correspondra bien à sa définition mathématique : dans le premier cas ce sera un singleton, dans l'autre l'ensemble vide.

Let U, V be open of X .
Definition.
a prf_included is a structure with the following components:
prf_included_prf: of type U ⊂ V *(coercion)*

Definition.
let eq_triv be the function defined by , for all t1,t2:prf_included , True
(its type is prf_included ⇒ prf_included ⇒ Prop).
Lemma (eq_triv_equiv).
eq_triv is an equivalence.

Definition.
let HomTop be the set Build_Setoid(eq_triv_equiv).

Cette fois-ci, aucun axiome n'est nécessaire, et seuls les cas significatifs mathématiquement, ceux où U est bien inclus dans V, doivent être étudiés. Par exemple, pour définir la composition de deux morphismes on doit construire un élément de `(HomTop U W)` à partir d'un élément de `(HomTop U V)` et d'un élément de `(HomTop V W)`, c'est à dire une preuve de $U \subset V$ sous les hypothèses $U \subset V$ et $V \subset W$, ce qui est immédiat.

2.6.2 Foncteurs contravariants et préfaisceaux

On a choisi de formaliser les préfaisceaux et faisceaux à valeurs dans la catégorie CRING des anneaux commutatifs. On ne peut à priori pas généraliser dans `Coq` la notion de préfaisceaux à toute catégorie C parce qu'on y utilise l'égalité des sétoïdes. Il suffit cependant de remplacer dans notre développement le mot clé CRING par celui de n'importe quelle autre catégorie dont les objets peuvent être coercés vers les sétoïdes pour obtenir la notion de préfaisceau correspondante. La seule restriction concerne les spécifications des structures d'anneaux induites, qui devront évidemment être adapté.

Un préfaisceau d'anneaux commutatifs sur un espace topologique X est un foncteur contravariant de la catégorie `(Top_cat X)` dans la catégorie CRING des anneaux commu-

[4] souflée par André Hirschowitz

tatifs.

Les foncteurs contravariants entre deux catégories C_1 et C_2 sont définit comme dans [SH95], en se donnant une application entre les objets et une autre entre les morphismes, satisfaisant les deux propriétés : $F(id_{C_1}) = id_{C_2}$, et $F^*(goh) = F^*(h)oF^*(g)$.

Definition.
a functor is a structure with the following components:
fctr_ob: a function of type Ob(C1) $\Rightarrow$ Ob(C2) *(coercion)*
fctr_morph: of type $\forall$ a,b:Ob(C1) , (a $\longrightarrow$ b) $\leadsto$ (fctr_ob(a) $\longrightarrow$ fctr_ob(b))
im_of_id_prf: of type $\forall$ a:Ob(C1) , (fctr_morph a a)(Ida) = Idfctr_ob(a)
distrib_prf: of type $\forall$ a,b,c:C1 ,
$\forall$ fa:a $\longrightarrow$ b ,
$\forall$ fb:b $\longrightarrow$ c ,
(fctr_morph a c)(fb o fa) =
((fctr_morph b c)(fb) o (fctr_morph a b)(fa))

Et on peut finalement écrire :

Definition.
let CRpresheaf be (Cfunctor Top_cat(X) CRING).

Un préfaisceau F associe donc à chaque ouvert U de X un anneau $F(U)$, et à chaque inclusion $U \subset V$ un morphisme d'anneaux $F^*(U \subset V) : F(V) \to F(U)$, dit morphisme de restriction, que l'on notera ρ_{VU}. Si $s \in F(V)$, on note aussi $s_{|U}$ l'élément $F^*(U \subset V)(s)$ de $F(U)$. Toutes ces notations sont possibles avec CtCoq , les énoncés restent ainsi lisibles, malgré la complexité grandissante du code. Par exemple, le terme qui apparait sous CtCoq comme :

s|W = s|V|W.
correspond en fait au code :

```
(Equal
     (((Cfctr_morph F W U)
          (Build_prf_included (included_trans pv pu))) s)
     (((Cfctr_morph F W V) pv) (((Cfctr_morph F V U) pu) s))).
```

2.6.3 Faisceaux

2.6.3.1 Définition

Un faisceau est un préfaisceau vérifiant les deux propriétés suivantes :

- *propriété d'unicité* : si $s, s' \in F(U)$ et si $\{V_i\}_{i \in I}$ est un recouvrement ouvert de U tel que $\forall i\ s_{|V_i} = s'_{|V_i}$ alors $s = s'$ dans $F(U)$,

- *propriété de recollement* : si on a une collection d'éléments $s_i \in F(V_i)$, $\{V_i\}_{i \in I}$ étant toujours un recouvrement ouvert de U, tels que $s_{i|V_i \cap V_j} = s_{j|V_i \cap V_j}$ alors il existe $s \in F(U)$ tel que $\forall i \ s_{|V_i} = s_i$.

2.6.3.2 Spécifications de la propriété d'unicité

On commence par définir les recouvrements ouverts d'une partie A de E, c'est une partie de l'ensemble des ouverts qui doit vérifier la propriété suivante :

Definition.
let covering_prop be
the function defined by , for all P:℘(open of E) ,
A:℘(E) ,
∀ x:E , x ∈ A ⇒ ∃ Pi:open of E , (Pi ∈ P) ∧ (x ∈ Pi).

Record Open_covering_of [A:℘(E)]: Type **:=** {
Open_covering_setoid: of type ℘(open of E) *(coercion)*;
Open_covering_prf: a proof of (covering_prop Open_covering_setoid A) }.

Let U be a open of X.
Let P be of type Open_covering_of(U).
Let s, s' be of type F(U) .

Definition.
let equal_over_a_covering be ∀ V:Open_covering_setoid(P) , s|U ∩ V = s'|U ∩ V.

La propriété ci-dessus est un peu différente de celle donnée dans la définition des faisceaux. En effet on restreint s et s' à $U \cap V$ et pas à V simplement parce que même si V fait partie du recouvrement de U il n'est pas forcément inclus dans U (un recouvrement peut 'déborder'). Cette propriété définie, on peut formaliser la première propriété des faisceaux ainsi :

Definition.
let unicity_presheaf be
the function defined by , for all U:open of X ,
P:Open_covering_of(U) ,
s,s':F(U) , (!equal_over_a_covering U P s s') ⇒ s = s'.

2.6.3.3 Spécifications de la propriété de recollement

Le problème pour la seconde propriété est de se donner la collection d'éléments s_i. On voudrait considérer le type suivant :

Definition.
a stick_elt is a structure with the following components:
stick_elt_set: a open of X
stick_elt_elt: of type F(stick_elt_set)

en faire un sétoïde, puis dire que se donner une telle collection c'est se donner une partie de ce sétoïde. On a donc besoin pour cela d'une égalité sur ce type. La notion d'égalité la plus naturelle serait $x = x'$ si `(part x) = (part x')` et `(elt x) = (elt x')`, mais elle ne peut être codée telle quelle parce que les types de `(elt x)`et de `(elt x')`, respectivement `(F(part x))`et `(F(part x'))`, sont différents (`(elt x)` et `(elt x')` n'appartiennent donc pas au même sétoïde).

Remarquons par contre que si `(part x) = (part x')`, on a en particulier l'inclusion `(part x)`$\subset$`(part x')` et `(elt x')` peut donc être vu comme un élément de type `(F (part x))` via les morphismes de restrictions. On dira donc que $x = x'$ si `(part x)=(part x')` et $(elt\ x) = (elt\ x')_{|(partx)}$:

Definition.
let equ_stick_elt be
the function defined by , for all st1,st2:stick_elt ,
$\exists$ proof_equal_sets:st1$_{\text{SET}}$ = st2$_{\text{SET}}$,
st1$_{\text{ELT}}$ = st2$_{\text{ELT}|\text{st1_SET}}$.

Lemma (equ_stick_equiv).
equ_stick_elt is an equivalence.
Proof:

Definition.
let Stick_elt be the set Build_Setoid(equ_stick_equiv).

Si `s :Stick_elt` on note sous `CtCoq` s$_{\text{ELT}}$ le terme `(stick_elt_elt s)`et s$_{\text{SET}}$ le terme `(stick_elt_set s)`.

Une fois l'ensemble `Stick_elt` des éléments de la forme $(U, s \in\ F(U))$ définit, on peut écrire la propriété de recollement. Une donnée de recollement (`sticking_data_on` en `Coq`) sera la donnée d'une partie `st_part` de `Stick_elt`, d'une preuve que l'ensemble des ouverts associés forme bien un recouvrement de U, et d'une preuve que $\forall (U, s_U), (V, s_V) \in st\_part,\ s_{U|U\cap V} = s_{V|U\cap V}$.

Let U be a open of X.
Let P be of type Open_covering_of(U).
Let s, s' be of type F(U) .

Definition.
let equal_over_intersections be
the function defined by , for all Vi,Vj:open of X ,
si:F(Vi) , sj:F(Vj) , si|Vi ∩ Vj = sj|Vi ∩ Vj.

Record sticking_data_on [U:open of X]: Type **:=** {
st_part: of type Predicate(Stick_elt(F)) *(coercion)*;
st_covering_prf: of type ∀ x:U , ∃ st:st_part , x ∈ st_SET;
st_prf: of type ∀ U,V:st_part , (equal_over_intersections U_ELT V_ELT) }.

La propriété de recollement sera alors :

Definition.
let sticking be
∀ U:open of X ,
∀ st:sticking_data_on(U) , ∃ s:F(U) , ∀ Si:st , (equal_over_intersections s Si_ELT).

Et on peut enfin définir les faisceaux d'anneaux commutatifs :

Record CRsheaf [X:topological space]: Type **:=** {
pre: a presheaf of commutative rings over X *(coercion)*;
unicity_prf: of type ∀ U:open of X ,
∀ P:Open_covering_of(U) , ∀ s,s':U , (unicity_presheaf(P) s s');
sticking_prf: of type sticking(pre) }.

2.6.4 Germes, fibres et morphismes

2.6.4.1 Germes et fibres

La fibre F_p d'un préfaisceau F en un point p de X est l'ensemble des couples formés d'un ouvert U contenant p et d'un élément s de $F(U)$, quotienté par la relation $(U, s \in F(U)) \sim (V, s' \in F(V))$ ssi $(\exists W|\ p \in W \wedge (W \subset U \cap V) \wedge s_{|W} = s'_{|W})$. La souplesse des sétoïdes permet de directement définir les fibres `(Stalk F p)` :

Definition.
a germ is a structure with the following components:
germ_set: a open of X
germ_prf: of type P ∈ germ_set
germ_elt: of type F(germ_set)

Definition.
let equ_germ be
the function defined by , for all g1,g2:germ ,
$\exists$ W:open of X ,
(P $\in$ W) $\wedge$
$\exists$ p1:W $\subset$ $g1_{\_set}$, $\exists$ p2:W $\subset$ $g2_{\_set}$, $g1_{\_elt|W}$ = $g2_{\_elt|W}$.

Lemma (equ_germ_equiv).
equ_germ is an equivalence.
Proof:

Definition.
let stalk be the set Build_Setoid(equ_germ_equiv).

Les anneaux $F(U)$ induisent une structure d'anneau sur les fibres : par exemple on peut définir la somme de deux germes en p $(U,s)+(V,s')$ par $(U \cap V, s_{U\cap V} + s'_{U\cap V})$, il suffit de montrer que cette fonction est compatible avec l'égalité des fibres :

Definition.
stalk_plus:stalk $\Rightarrow$ stalk $\Rightarrow$ stalk
1: `Intros` g1 g2.
1: `Apply` (!Build_germ
$g1_{\_set}$ $\cap$ $g2_{\_set}$ (in_part_intero germ_prf(g1) germ_prf(g2))
$g1_{\_elt|g1\_set \cap g2\_set}$+$g2_{\_elt|g1\_set \cap g2\_set}$).

Lemma (ringstalk1).
$\forall$ x,x',y,y':stalk , x = x' $\Rightarrow$ y = y' $\Rightarrow$ (stalk_plus x y) = (stalk_plus x' y').
Proof:

On définit de même la multiplication, le zéro $(X, 0 \in F(X))$, l'unité $(X, 1 \in F(X))$, puis on montre les propriétés nécessaires pour enfin faire des fibres des anneaux.

2.6.4.2 Morphismes

Comme pour toutes les structures mathématiques il existe une notion de morphisme pour les préfaisceaux. Un morphisme ϕ entre deux préfaisceaux F et G sur X est la donnée, pour tout ouvert U de X, de morphismes d'anneaux $\phi(U) : F(U) \to G(U)$, compatibles avec les morphismes de restrictions.

$$\begin{array}{cccc} \phi(V): & F(V) & \longrightarrow & G(V) \\ & \downarrow \rho^F_{VU} & \circlearrowleft & \downarrow \rho^G_{VU} \\ \phi(U): & F(U) & \longrightarrow & G(U) \end{array}$$

Definition.
a presheaf_morphism is a structure with the following components:
morphism_of: of type ∀ U:open of X , F(U) ⟶ G(U) *(coercion)*
commut_prf: of type ∀ U,V:open of X , ∀ t:(prf_included V U) , (ρUV o U) = (V o ρUV)

Un tel morphisme ϕ sera dit injectif (resp. surjectif) si, pour tout ouvert U, $\phi(U)$ est injectif (resp. surjectif). Par définition un morphisme de faisceau est un morphisme entre les préfaisceaux sous jacents : la coercion entre faisceau et préfaisceau permet de ne pas avoir à donner de nouvelle définition.

Enfin, un morphisme de faisceau induit naturellement sur toutes les fibres un morphisme d'anneau $\phi_p : F_p \to G_p$, en envoyant un germe représenté par un couple $(U, s \in (F\ U))$ vers $(U, \phi(U)(s) \in (G\ U))$.

Let X be a topological space.
Let F, G be presheaf of commutative rings over X .
Let P be of type X.
Let f be of type (presheaf_morphism F G).
Definition.
let fun_induced_in_stalks be
the function defined by , for all germ_f:(Stalk F P) ,
(!Build_germ
X G P germ_f_set germ_prf(germ_f)
f(germ_f_set)(germ_f_elt))
(its type is (Stalk F P) ⇒ (Stalk G P)).

On prouve que cette fonction est bien compatible avec l'égalité sur les germes pour en faire une application entre les fibres, puis toutes les propriétés nécessaires pour en faire un morphisme d'anneau :

Lemma (fun_comp_in_stalks).
fun_compatible(fun_induced_in_stalks).

Lemma (fun_induced_in_stalks_plus).
∀ x,y:(Stalk F P) ,
fun_induced_in_stalksx+y = fun_induced_in_stalks(x)+fun_induced_in_stalks(y).

Lemma (fun_induced_in_stalks_zero).
fun_induced_in_stalks(0(Stalk F P)) = 0(Stalk G P).

Lemma (fun_induced_in_stalks_mult).
∀ x,y:(Stalk F P) ,
fun_induced_in_stalksx*y = fun_induced_in_stalks(x)*fun_induced_in_stalks(y).

Lemma (fun_induced_in_stalks_one).
fun_induced_in_stalks(1(Stalk F P)) = 1(Stalk G P).

Definition.
let map_induced_in_stalks be
(!BUILD_HOM_RING
(Stalk F P) (Stalk G P) fun_induced_in_stalks fun_comp_in_stalks
fun_induced_in_stalks_plus fun_induced_in_stalks_zero fun_induced_in_stalks_mult
fun_induced_in_stalks_one)
(its type is (!Hom RING (Stalk F P) (Stalk G P))).

Ces morphismes induits `(map_induced_in_stalks f p)` sont ainsi formalisés, on les affiche f_p sous `CtCoq` .

2.6.5 Nature locale des faisceaux

On a montré le théorème suivant : un morphisme de faisceau est un isomorphisme (au sens de morphisme injectif et surjectif) si tous les morphismes induits dans les fibres sont des isomorphismes (la réciproque est vraie). Ce théorème n'est vrai que pour les faisceaux, il utilise explicitement les deux propriétés qui les définissent. On va détailler une partie de la preuve : le morphisme $\phi : F \to G$ est injectif si et seulement si tous les morphismes induits par ϕ dans les fibres le sont (attention c'est faux pour la surjectivité).

Soit donc U un ouvert et $s \in F(U)$ tel que $\phi(U)(s) = 0$. Si en tout point les morphismes induits sont injectifs alors on va avoir autour de chacun de ces points p un ouvert W_p tel que $s_{|W_p} = 0$:

Lemma (localy_zero).
Phi(U)(s) = 0 ⇒
(∀ P:X , Phi_P is injective) ⇒
∀ P:X ,
∀ t:P ∈ U , ∃ Wp:open of X , ∃ pw:(prf_included Wp U) , (P ∈ Wp) ∧ (s|Wp = 0).

Si on considère l'ensemble $\mathcal{W}$ de ces ouverts W_p, le lemme ci-dessus nous permet de démontrer qu'il forme un recouvrement ouvert de U. Cependant former l'ensemble exacte de ces ouverts ne peut se faire sans utiliser l'axiome du choix (on passe de 'il existe W_p' à 'l'ensemble des W_p'). Pour éviter cela, on construit un ensemble plus gros : celui de tous les ouverts W tels que $s_{|W} = 0$.

Definition.
a_good_setoid: $\wp$(open of X)
1: Apply (!Build_Predicate ? W:open of X , ∃ pu:(prf_included W U) , s|W = 0).

Puisque c'est un sur-ensemble de $\mathcal{W}$, il forme lui aussi un recouvrement de U :

Definition.
a_good_covering:(morphism_of Phi U)(s) = 0 ⇒
(∀ P:X , Phi_P is injective) ⇒ Open_covering_of(U)
1: Intros H' H'0. 1: Apply (!Build_Open_covering_of ? U)(a_good_setoid).

Par l'application de la première propriété des faisceaux on aura donc $s = 0$, et il vient :

Lemma (injective_phi).
∀ Phi:(presheaf_morphism F G) ,
(∀ P:X , Phi_P is injective) ⇒ ∀ U:open of X , Phi(U) is injective.

2.6.6 Faisceau associé

On a formalisé de deux façons le faisceau associé à un préfaisceau. La première en suivant la définition de R. Hartshorne [Har77], la seconde à l'aide de la notion de crible. On décrit ici la version de R. Hartshorne.

2.6.6.1 La définition de R. Hartshorne

Considérons un préfaisceau F. On définit le faisceau F^* associé à F par les applications suivantes :

– Entre les objets :

  A un ouvert U on fait correspondre l'anneau des applications $s : U \rightarrow \bigsqcup_{p \in U} F_p$ qui vérifient les deux conditions suivantes :

  i) $\forall p \in U,\ s(p) \in F_p$
  ii) $\forall p \in U,\ \exists V$ voisinage de p, $\exists t \in F(V)$ tel que $\forall q \in V\ \ s(q) = (V, t) \in F_q$

– Entre les morphismes :

  Si $U \subset V$ on définit la restriction $F(V) \rightarrow F(U)$ par la restriction des applications à U.

R. Hartshorne laisse au lecteur le soin de démontrer que le préfaisceau ainsi défini est en fait un faisceau, ce que l'on s'est proposé de faire en Coq .

2.6.6.2 Des applications dépendantes

L'ensemble des fonctions $s : U \to \bigsqcup_{p \in U} F_p$ qui vérifient $\forall p \in U,\ s(p) \in F_p$ peuvent être vues en Coq par les termes qui ont pour type le type dépendant `(p :(subtype_elt U))(Stalk F (subtype_elt p))`. L'idée est donc de faire de ce type dépendant un sétoïde. Soit donc f un terme de ce type. Puisque f doit représenter une fonction, on doit avoir `x=y -> (f x)=(f y)`, mais cette dernière égalité n'a pas de sens puisque le premier terme `(f x)` est de type `(Stalk F x)` et le second `(f y)` de type `(Stalk F y)`. On ne peut donc pas considérer la proposition `x=y -> (f x)=(f y)`.

De manière générale, on peut procéder de la manière suivante. Soit `E :Setoid` et `F :E ->Setoid`, on demande en paramètre une fonction G qui à `x,y` de type `E` et une preuve `p` que `x=y` associe une fonction `(G x y p) :(F x)->(F y)` qui, en quelque sorte, reconstruit un élément de type `(F x)` en un élément de type `(F y)`. On pourra alors écrire non pas `x=y -> (f x)=(f y)` mais `(p :x=y)(f y)=((G x y p)(f x))`.

On formalise la fonction `G` de la manière suivante, on veut évidemment que cette fonction soit indépendante de la preuve de `x=y` donnée en paramètre :

Let E be a set.
Let F be a function of type E $\Rightarrow$ set.
Definition.
a map_for_equality is a structure with the following components:
equality_map: of type $\forall$ x,y:E , $\forall$ p:x = y , F(x) $\leadsto$ F(y) *(coercion)*
indep_proof_prf: of type $\forall$ x,y:E ,
$\forall$ pxy,p'xy:x = y ,
(equality_map x y pxy) = (equality_map x y p'xy)

on peut alors définir les applications dont le type d'arrivée dépend ainsi de l'argument par :

Let G be of type map_for_equality(F).
Definition.
a mapd is a structure with the following components:
mapd_fun: of type $\forall$ x:E , F(x) *(coercion)*
mapd_prf: of type $\forall$ x,y:E , $\forall$ prf_equ:x = y , y = (G x y prf_equ)(x)

On met sur ce type l'égalité extensionelle classique :

Definition.
let equ_mapd be the function defined by , for all f1,f2:mapd , ∀ x:E , f1(x) = f2(x)
(its type is mapd ⇒ mapd ⇒ Prop).
Lemma (equ_mapd_equiv).
equ_mapd is an equivalence.

Definition.
let MAPd be the set (!Build_Setoid ? ? equ_mapd_equiv).

2.6.6.3 Remarque sur l'égalité des sigma-types.

Cette même idée nous permet aussi d'avoir une égalité significative sur les sigma-types. On peut en effet définir l'égalité de `(a :A,b :(F a))` et `(a' :A',b' :(F a'))`, qui pose le même problème, par ∃ `p :a=a' | b'=(G a a' p b)` .

Pour que cette définition soit valide, c'est-à-dire pour que la relation ci-dessus soit bien une relation d'équivalence, il faut de plus demander à `G` de vérifier deux propriétés :

- $\forall x : A, \forall p : x = x, (G\ x\ x\ p) = Id_{(Fa)}$
- $\forall x, y, z : A, \forall p_{xy} : x = y, \forall p_{yz} : y = z, \forall p_{xz} : x = z,$
  $(G\ y\ z\ p_{yz}) o (G\ x\ y\ p_{xy}) = (G\ x\ z\ p_{xz})$

2.6.6.4 Spécifications et preuves

Dans notre cas, on prend pour E le sétoïde `(subtype_elt U)`, et pour F la fonction `point_to_stalk` qui à `p :(subtype_elt U)` associe `(Stalk F (subtype_elt p))`. L'application G sera la fonction `fun_stalk_to_same` qui à partir de `p1,p2 :(subtype_elt U)` et d'une preuve p que `p1=p2` reconstruira tout germe de F_{p1} en un germe de F_{p2} :

Definition.
let point_to_stalk be the function defined by , for all P:U , (Stalk F P).
Lemma (in_part_comp_l).
∀ A:℘(E) , ∀ x,y:E , x ∈ A ⇒ y = x ⇒ y ∈ A.
Proof:

Definition.
let fun_stalk_to_same be
the function defined by , for all p1,p2:U ,
prf:p1 = p2 ,
st1:(Stalk F p1) ,
(Build_germ (in_part_comp_l germ_prf(st1) Sym(prf))
st1$_{_\mathrm{elt}}$)
(its type is ∀ p1,p2:U , ∀ prf:p1 = p2 , point_to_stalk(p1) ⇒ point_to_stalk(p2)).

Pour plus de commodité par la suite on fait de `fun_stalk_to_same` une application qui renvoie non pas une simple application de F_{p1} vers F_{p2}, mais un morphisme d'anneau. Après avoir vérifié toute les propriétés nécessaires, on peut enfin définir l'ensemble des fonctions qui nous intéressent par :

Definition.
as_map_equality:(!map_for_equality U point_to_stalk)
1: `Apply` (!Build_map_for_equality U point_to_stalk hom_stalk_to_same).

Definition.
let local_data_map be MAPd(as_map_equality).

Les fonctions qui définissent le faisceau associé doivent en outre vérifier la propriété (ii), on construit donc l'ensemble de ces fonctions comme la partie de `local_data_map` vérifiant :

Definition.
associated_sheaf_setoid:℘(local_data_map)

1: `Apply` (!Build_Predicate ?
s:local_data_map ,
∀ p:U ,
∃ V:open of X ,
∃ proof_neigh:p ∈ V ,
∃ proof_included:V ⊂ U ,
∃ t:F(V) , ∀ q:V , s(q$_{[\mathrm{in}\ U]}$) = (in_stalk F subtype_prf(q))(t)).

On fait alors de ce sétoïde un anneau, puis on définit les morphismes de restrictions. On prouve enfin que le préfaisceau ainsi contruit est bien un faisceau :

Lemma (as_first_prop).
∀ U:part(top(X)) ,
∀ P:Open_covering_of(U) , ∀ s,s':(Cfctr_ob as_presheaf(F) U) , (unicity_presheaf P s s').

Lemma (as_second_prop).
sticking(as_presheaf(F)).

Definition.
asheaf:sheaf of commutative rings over X
1: `Apply` (!Build_CRsheaf X as_presheaf(F) as_first_prop as_second_prop). **Defined**.

Les deux preuves précédentes sont les plus compliquées de la construction du faisceau associé. On n'en donne pas le détail ici parce qu'elles sont très proches de celles que nous verrons plus tard pour la définition du faisceau spectral (c.f. section 4.72).

2.7 Schémas Affines

2.7.1 Le spectre d'un anneau comme espace topologique

2.7.1.1 Le spectre d'un anneau comme ensemble

Soit R un anneau commutatif, comme ensemble, son spectre est l'ensemble de ses idéaux premiers (avec bien sûr pour égalité celle des parties de R) :

```
Let R be a commutative ring.
Definition.
   spec:set
 1: Apply (!Build_Setoid prime ideal of R !Equal(℘(R))).
```

2.7.1.2 La topologie de Zariski sur (spec R)

Si $\mathfrak{I}$ est un idéal de R, on définit $V(\mathfrak{I})$ (noté de cette manière avec `CtCoq` mais correspondant au terme `Coq (primes_containing I)`) comme l'ensemble des idéaux premiers de R qui contiennent $\mathfrak{I}$.

```
Definition.
   primes_containing:∀ I:ideal(R) , ℘(spec)
 1: Intros I. 1: Apply (!Build_Predicate spec x:spec , I ⊂ x).
```

Les ensembles de la forme $V(\mathfrak{I})$ forment en fait les fermés d'une topologie sur `(spec R)`, que l'on va donc construire en utilisant la fonction `Build_topology_by_closed` vue en 3.1. On note `(Spec R)` cet espace topologique.

On commence donc par construire l'ensemble des $V(\mathfrak{I})$:

```
Definition.
   set_of_closed_for_zarisky:℘(℘(spec))
 1: Apply (!Build_Predicate ? p:℘(spec) , ∃ a:Set_Ideal(R) , p = V(a)).
```

On montre facilement que les propriétés `F_empty` `F_full` de la section 3.1 : le vide s'écrit `(primes_containing (ideal_full R))` (où `(ideal_full R)` est l'anneau R vu comme ideal) et l'ensemble des idéaux premiers de R s'écrit `(primes_containing (ideal_nul R))` :

Lemma (spec_top_prop1).
$\varnothing \in$ set_of_closed_for_zarisky.

Lemma (spec_top_prop2).
full(spec) $\in$ set_of_closed_for_zarisky.

Les propriétés `finite_union_closed` (l'union de deux fermés est un fermé) et `infinite_inter_closed` (une intersection quelconque de fermés est un fermé) reposent essentiellement sur le fait que $i)$ $V(\mathfrak{a}) \cup V(\mathfrak{b}) = V(\mathfrak{a} \cap \mathfrak{b})$ et $ii)$ $\cap V(\mathfrak{a}_i) = V(\sum \mathfrak{a}_i)$, ce qu'il nous faut donc démontrer.

La première égalité $i)$ est une conséquence du lemme `inter_included_prime` vu dans la section `4.1` : si un idéal premier contient l'intersection de deux idéaux, alors il contient l'un ou l'autre de ces idéaux.

Lemma (union_primes_containing).
$\forall$ I,J:ideal(R) , V(I) $\cup$ V(J)=V(I$\cap$J).

Lemma (spec_top_prop4).
finite_union_closed(set_of_closed_for_zarisky).

Pour exprimer la propriété $ii)$, on définit d'abord la fonction qui associe à une famille d'idéaux $\{\mathfrak{a}_i\}$ l'ensemble des $V(\mathfrak{a}_i)$:

Definition.
family_of_primes_containing:Predicate(Set_Ideal(R)) $\Rightarrow$ Predicate($\wp$(spec))
1: `Intros` Ai. 1: `Apply` (!Build_Predicate $\wp$(spec) p:Predicate(spec) , $\exists$ ai:Ai , p = V(ai)).

On peut alors démontrer, en utilisant les lemmes qui caractérisent la somme d'idéaux ($\sum \mathfrak{a}_i$ est le plus petit idéal qui contient tous les $\mathfrak{a}_i$) :

Lemma (primes_containing_sum_ideals).
$\forall$ ideal_collection: $\wp$(Set_Ideal(R)) ,
V(sum_ideals(ideal_collection)) =
intersection_part(family_of_primes_containing(ideal_collection)).

On a donc bien :

Lemma (spec_top_prop3).
infinite_inter_closed(set_of_closed_for_zarisky).

Et on peut définir le spectre d'un anneau :

Definition.

```
let spec_top be
 (!Build_topology_by_closed
    spec set_of_closed_for_zarisky spec_top_prop1 spec_top_prop2 spec_top_prop3
    spec_top_prop4).
```

Definition.

```
let Spec be the topological space Build_et(spec_top).
```

2.7.2 Le faisceau spectral

Le faisceau spectral $\mathcal{O}_{(Spec\ R)}$ sur (Spec R) est défini de la manière suivante. Il associe à tout ouvert U de (Spec R) (U est donc aussi un ensemble d'idéaux premiers de R) l'anneau des applications $s : U \to \bigsqcup_{p \in U} R_p$ telles que $\forall p \in U, s(p) \in R_p$ qui sont localement des fonctions constantes, c'est-à-dire telles que quelque soit p dans U il existe un voisinage V de p inclus dans U et des éléments a, f de R tels que $\forall q \in V, f \notin q$, et $s(q) = \frac{a}{f}$ dans R_p. Si U est inclus dans V, le morphisme de restriction associe simplement à s sa restriction à V.

Pour formaliser ce faisceau, on doit donc :

- définir l'ensemble des fonctions s ci-dessus
- y poser la structure d'anneau naturelle
- définir les morphismes de restrictions
- construire le préfaisceau
- enfin montrer qu'il s'agit bien d'un faisceau en démontrant qu'il vérifie la propriété d'unicité et de recollement

Définir $\mathcal{O}_{(Spec\ R)}(U)$ d'abord comme ensemble demande plusieurs étapes. Comme pour la construction du faisceau associé, les fonctions de $\mathcal{O}_{(Spec\ R)}(U)$ sont des fonctions dépendantes ($s(p)$ appartient à l'anneau local R_p qui dépend de p), on doit donc avant tout créer la fonction qui à une preuve t de $p_1 = p_2$ associe une fonction de R_{p_1} vers R_{p_2} qui reconstruit un élement de R_{p_1} en élément de R_{p_2} :

Definition.

```
f1:∀ p,q:U , ∀ H:p = q , A(p) ⇒ A(q)
1: Intros p q H x. 1: Elim x. 1: Intros num den. 1: Elim den. 1: Intros a Ha.
1: Cut a ∈ (mp_compl_prime 1!A q). 1: Intros Ha'.
1: Exact (!Build_cart_type A (mp_compl_prime 1!A q) num (a[in (mp_compl_prime 1!A q)])).
1: Simpl in Ha. 1: Simpl. 1: EAuto with algebra. Defined.
```

Puis en faire une application :

Definition.

```
f2:∀ p,q:U , ∀ H:p = q , A(p) ~⟶ A(q)
1: Intros. 1: Apply (!Build_Map ? ? f1(H)).
```

Et ensuite prouver que cette fonction est indépendante de la preuve t de $p_1 = p_2$:

Definition.

```
    f3:(!map_for_equality  U  p:U , localize_prime(p))
Apply  (!Build_map_for_equality  U  p:U ,  localize_prime(p)  f2).
```

On peut ainsi construire le sétoïde `SP_setoid_carrier : (MAPd f3 ? ? )` des fonctions $U \to \bigsqcup_{p \in U} R_p$ telles que $\forall p \in U,\ s(p) \in R_p$. L'ensemble $\mathcal{O}_{(Spec\ R)}(U)$ est la partie de ce sétoïde formés des applications localement constantes : on définit donc le prédicat suivant :

Definition.

```
    let  scheme_presheaf_pred  be
     the function defined by , for all   s:scheme_presheaf_setoid_carrier ,
                                              ∀ p:U ,
                                              ∃ V:open of  Spec(A) ,
                                              ∀ HVU:V ⊂ U ,
                                              (p ∈ V) ∧
                                               ∃ a:A ,
                                               ∃ f:A ,
                                               ∀ q:V ,
                                               ∃ Hf:f ∉ q ,
                                               s(q[in U]) = (!Build_cart_type  A  ¬q  a  (f[in ¬q])).
```

puis on montre qu'il est compatible avec l'égalité des éléments de $\mathcal{O}_{(SpecR)}(U)$ c'est-à-dire l'égalité des fonctions dépendantes. On peut finalement construire le sétoïde $\mathcal{O}_{(SpecR)}(U)$:

Definition.

```
    scheme_presheaf_setoid: ℘(scheme_presheaf_setoid_carrier)
Apply  (!Build_Predicate  scheme_presheaf_setoid_carrier    scheme_presheaf_pred).
```

La structure d'anneau doit maintenant être posée sur cet ensemble. Par exemple, pour définir la somme on doit d'abord définir la fonction `SP_plus_sub` de type $SP\_setoid \to SP\_setoid \to (p : U)R_p$ qui associe à deux fonctions leur somme comme fonction dépendante, puis on en fait un élément de `SP_setoid_carrier` en montrant qu'elle vérifie la propriété des applications dépendantes. Il ne reste plus qu'à montrer que la somme ainsi définie est bien localement constante et on peut enfin définir la somme `SP_plus`.

Evidement le même travail doit être réalisé pour définir la multiplication, l'unité et le zéro, pour ensuite pouvoir démontrer les douze propriétés des anneaux et finalement construire $\mathcal{O}_{(SpecR)}(U)$ comme anneau commutatif.

A cause des multiples structures empilées ces définitions et ces preuves sont très fastidieuses. Pour les preuves il est nécessaire de s'arrêter au bon moment dans le dépliage des structures et donc d'utiliser judicieusement les fonction `Opaque` et `Transparent` qui permettent de spécifier à Coq les termes qu'il ne doit pas déplier lors des simplifications.

L'anneau $\mathcal{O}_{(Spec\ R)}(U)$ construit, on peut définir les morphismes de restriction. D'abord la fonction :

```
Let A be of type CRING.
Let U, V be open of Spec(A) .
Definition.
   SP_restr_fun:∀ prf:(prf_included V U) , SP_cring(U) ⇒ SP_cring(V)
 1: Intros prf f.
1: Pose z = (p:V , f(Build_subtype(prf_included_prf(prf) p subtype_prf(p)))).
1: Cut ∀ x,y:V , ∀ H:x = y , z(y)=(f3(V) x y H)(z(x)). 1: Intros H; Try Assumption.
1: Simpl.
1: Apply (!Build_subtype
          scheme_presheaf_setoid_carrier(V) scheme_presheaf_setoid(V)
          (!Build_mapd ? ? f3(V) z H)).
```

Puis l'application, en montrant que les restrictions de deux fonctions dépendantes égales sont égales :

```
Lemma (SP_restr_fun_comp).
   ∀ prf_inc:(prf_included V U) ,
   ∀ f1,f2:SP_cring(U) , f1 = f2 ⇒ (SP_restr_fun prf_inc f1) = (SP_restr_fun prf_inc f2).
```

On va plus loin et on fait de la restriction un morphisme d'anneau, pour avoir plus de facilités dans la suite. Enfin, on construit le préfaisceau $\mathcal{O}_{(Spec\ R)}$ en montrant les deux propriétés de la définition des foncteurs :

```
Lemma (SP_restr_hom_idprop).
   ∀ U:top(Spec(A)) ,
   (SP_restr_hom_map U U)(!Hom_id Top_cat(Spec(A)) U) = IdSP_cring(U).

Lemma (SP_restr_hom_comp_prop).
   ∀ a,b,c:Top_cat(Spec(A)) ,
   ∀ fa:a ⟶ b ,
   ∀ fb:b ⟶ c ,
   (SP_restr_hom_map a c)((Hom_comp a b c)((fb,fa))) =
    ((SP_restr_hom_map a b)(fa) o (SP_restr_hom_map b c)(fb)).
```

Definition.
SP:presheaf of commutative rings over Spec(A)

```
1: Apply (!Build_Cfunctor
        Top_cat(Spec(A)) CRING !SP_cring(A) SP_restr_hom_map
        SP_restr_hom_idprop SP_restr_hom_comp_prop).
```

Le plus gros du travail réside néanmoins dans les deux propriétés qu'il faut démontrer pour faire de ce préfaisceau un faisceau : la propriété d'unicité et celle de recollement. Détaillons la preuve de la seconde.

2.7.3 Les schémas affines

2.7.3.1 Les espaces annelés

Un espace annelé $(X, \mathcal{O}_X)$ est la donnée d'un espace topologique X et d'un faisceau d'anneau $\mathcal{O}_X$ sur cet espace. On dit qu'il est localement annelé si toutes les fibres $\mathcal{O}_{X,p}$ du faisceau sont des anneaux locaux.

On formalise les espaces annelés et localement annelés de la manière la plus naturelle :

Definition.
a ringed space is a structure with the following components:
rs_et: a topological space *(coercion)*
rs_sheaf: a sheaf of commutative rings over rs_et *(coercion)*

Record rs_hom [X:ringed space; Y:ringed space]: Type **:=** {
rs_hom_map: of type X~–>Y (continuous) *(coercion)*;
rs_hom_morphism: of type a morphism of presheaves : Y→rs_hom_map*X *(coercion)* }.

Definition.
let is_lrs be
the function defined by , for all X:ringed space , ∀ P:X , is_local((Stalk rs_sheaf(X) P)).
Definition.
a locally ringed space is a structure with the following components:
lrs_rs: a ringed space *(coercion)*
lrs_prop: of type is_lrs(lrs_rs)

2.7.3.2 Les schémas affines

Un espace affine est la donnée du couple $((Spec\ R), \mathcal{O}_{Spec\ R})$, où $\mathcal{O}_{Spec\ R}$ est le faisceau spectral de la section précédente, vu comme un espace localement annelé. Il nous faut donc prouver en Coq que toutes les fibres du faisceau spectral sont des anneaux locaux :

```
Let A be of type CRING.
Definition.
    let affine_scheme_rs be (!Build_rs Spec(A) Affine_sheaf(A)).
Lemma (affine_scheme_local_prop).
    is_lrs(affine_scheme_rs).
```

Et on peut enfin définir un schéma affine :

```
Definition.
    let affine_scheme_lrs be the locally ringed space
      (!Build_lrs affine_scheme_rs affine_scheme_local_prop).
```

2.8 Conclusion

Le développement présenté ici permet donc de formaliser les schémas affines, une des toutes premières notions de la géométrie algébrique. Ils ne forment évidemment qu'une infime partie du corpus de la géométrie algébrique, mais la définition de la notion générale de schéma et de leurs premières propriétés géométriques ne parait pas si lointaine. La difficulté résidera sans doute dans les notions de recollement et d'isomorphisme qui devront être formalisé de manière assez souple pour être réellement utilisables.

Avec l'avancée de ce travail cependant, les difficultés se sont faites de plus en plus importantes, et la vitesse de progression forcément de plus en plus lente. Dans le chapitre suivant on fait donc l'inventaire de ces problèmes et des différents remèdes que l'on a tenté pour y remédier.

Chapitre 3

Résultats, analyses et critiques

3.1 Sur la forme

3.1.1 Des documents lisibles par des non-experts

L'un des objectifs de ce travail a été de produire des documents lisibles par des mathématiciens non experts de Coq. Pour cela, les possibilités de CtCoq puis Pcoq en matière de notations étendues ont été utilisées. Ainsi on espère que tout mathématicien reconnaitra le lemme suivant :

Let F be a presheaf of commutative rings over X.
Lemma (restr_restr_subproof).
$\forall$ W,V,U:open of X , $\forall$ s:F(U) , $\forall$ pv:W $\subset$ V , $\forall$ pu:V $\subset$ U , $s|_W = s|_V|_W$.

Alors que le code Coq sous jacent est moins identifiable :

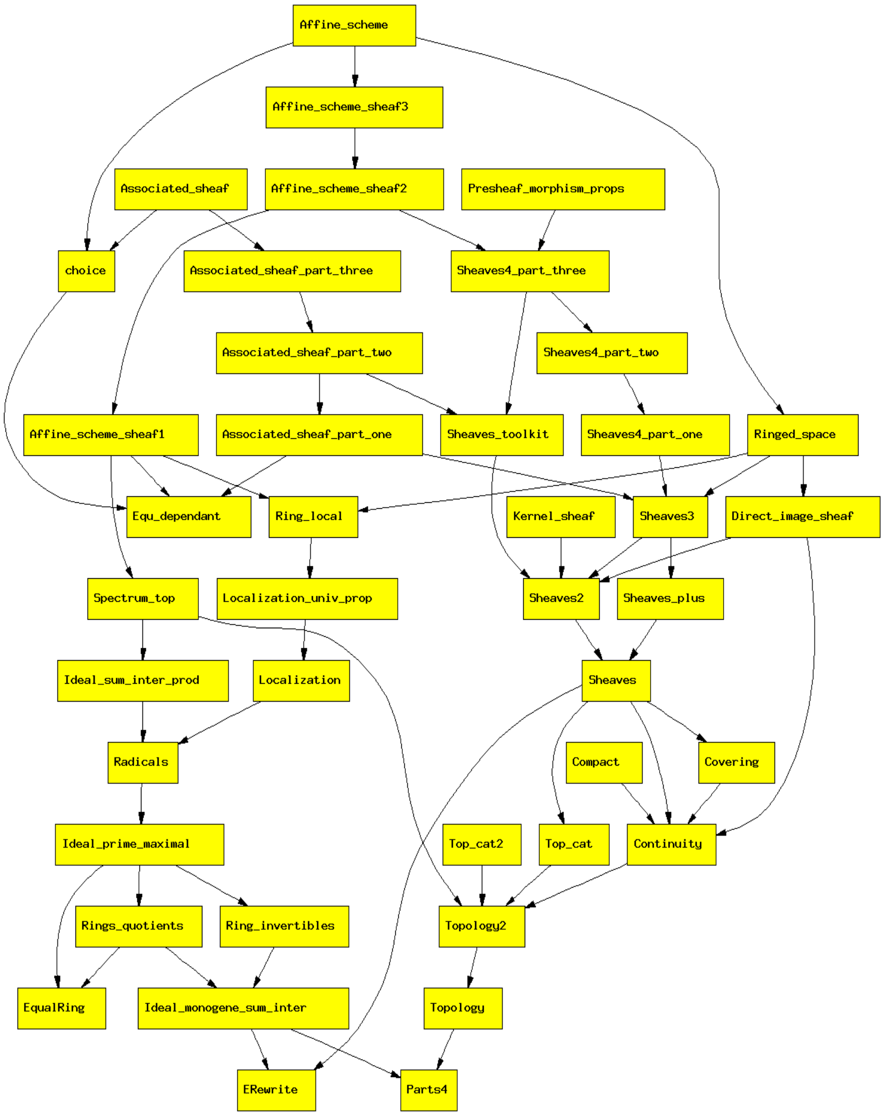
```
Variables F :  (CRpresheaf X).
Lemma restr_restr_subproof:
 (W, V, U : (open X))
 (s : (F U))
 (pv : (prf_included W V))
 (pu : (prf_included V U))
  (Equal
   (((Cfctr_morph F W U) (HomTop_compos pu pv)) s)
   (((Cfctr_morph F W V) pv) (((Cfctr_morph F V U) pu) s))).
```

Mais plus que simplement un artifice visuel permettant de lire des documents Coq déjà réalisés, c'est surtout pendant l'activité de preuve que de telles notations sont utiles. En effet les termes que l'on considère sont de plus en plus 'gros' avec l'avancée de notre travail, d'une part parce que l'empilement des théories implique souvent un empilement de termes, et d'autre part parce que les objets dépendent souvent aussi de propriétés, c'est à dire dans Coq de termes de preuves, souvent énormes et qui apparaissent donc dans les objets que l'on considère. Ainsi dans notre formalisation de la théorie des faisceaux les buts successifs et les hypothèses font souvent plus d'une dizaine de lignes et seraient illisibles sans un affichage évolué.

L'apport de CtCoq (et de son successeur Pcoq) ne tient pas seulement dans l'affichage. La technique de *preuve-en-cliquant* (*proof-by-pointing* [BKT94]) s'est révélé être un puissant outil : il est très agréable de pouvoir générer en très peu de clics une succession complexe de tactiques.

3.1.2 Vue d'ensemble du developpement

Voici le graphe de dépendance du developpement réalisé :



Les notions formalisées couvrent :

- en algèbre : les idéaux (somme, produit, intersection, premiers, maximaux), les radicaux par rapport à une partie multiplicative quelconque, les propriétés sur les inversibles des anneaux et enfin les anneaux locaux et les localisés par rapport aux idéaux premiers,

- en topologie : la définition des topologies par les ouverts, les fermés, par une base d'ouverts ; les notions d'adhérence, intérieur, bord, densité ; la continuité, les topologies séparées et les compacts,

- en théorie des faisceaux : la définition des préfaisceaux, faisceaux, morphismes de faisceaux, leurs fibres et les propriétés qui lient les morphismes de faisceaux et les morphismes entre les fibre. Le faisceau noyau et le faisceau image direct d'un faisceau par un morphisme de faisceau et enfin la construction du faisceau associé.

Toutes ces notions sont finalement réunies pour aborder la géométrie algébrique avec la définition au final des espaces affines.

3.2 Sur le fond

3.2.1 Les problèmes rencontrés

Le développement que nous avons présenté a subi deux remaniements complets, à chaque fois dictés par la nécessité d'alléger un travail toujours de plus en plus lourd.

Deux principaux problèmes ont considérablement ralenti l'évolution de ce dévelopement. Plus que simplement ralentir le travail de formalisation, ils imposent une certaine différence avec la "manière de faire" en mathématique.

3.2.1.1 Limitations des coercions et le problème du sous-typage

Le premier de ces problèmes concerne les limitations des coercions autorisées par Coq. En effet les coercions paramétrées ne sont pas gérées par le système et cela empêche l'utilisateur de déclarer la fonction `subtype_elt` comme coercion.

On est donc obligé d'écrire explicitement cette fonction pour voir un élément d'une partie comme un élément de l'ensemble au-dessus.

Ce qui peut sembler anecdotique devient vite un véritable puzzle avec l'empilement des différentes structures et théories dans Coq. Si on considère par exemple le terme servant à définir une des propriétés du faisceau associé, affiché ainsi par CtCoq :

```
1: Apply (!Build_Predicate ?
          asso_fun:local_data_map ,
          ∀ P:U ,
          ∃ V:open of  X ,
          ∃ proof_neigh:P ∈ V ,
          ∃ proof_included:V ⊂ U ,
          ∃ t:F(V) , ∀ Q:V , asso_fun(Q[in ?]) = (in_stalk F subtype_prf(Q))(t)).
```

il correspond en fait au code *Coq* suivant :

```
Apply (!Build_Predicate ?
   [asso_fun:local_data_map] (P:(subtype_elt U))
   (exT   (open X) [V:(open X)]
    (exT  (in_part (subtype_elt P) (subtype_elt V))
    [proof_neigh:(in_part (subtype_elt P) (subtype_elt V))]
     (exT (included (subtype_elt V) (subtype_elt U))
     [proof_included:(included (subtype_elt V) (subtype_elt U))]
       (exT (F V) [t:(F V)]
        (Q:(subtype_elt V))
        (Equal  (asso_fun
         (!Build_subtype ? ? (subtype_elt Q)
         (proof_included (subtype_elt Q) (subtype_prf Q))))
         ((in_stalk F (subtype_prf Q)) t))))))).
```

et contient pas moins de douze fois la fonction `subtype_elt` ! Remarquons encore une fois que des notations évoluées sont essentielles pour comprendre les termes que l'on manipule.

Le plus gênant est que cette fonction `subtype_elt` n'a aucune correspondance dans la pratique mathématique : pour le mathématicien un entier pair EST un entier, et un élément d'une topologie sur X EST une partie de X, ce qui n'est évidement pas le cas dans Coq. Cette fonction est si peu naturelle que souvent l'écriture d'un terme dans notre formalisation des faisceaux (là où les termes sont les plus gros) s'est déroulée selon la méthode :

- Ecriture du terme
- Coq : Erreur de type à cet endroit
- Ahh oui j'ai du oublié un `subtype_elt` ici
- Coq : Autre erreur de type ici
- Mince, il doit manquer un autre `subtype_elt` ..
- Coq :
- Ok j'ai compris, ça doit être un `subtype_elt`
- etc...

3.2.1.2 Les sétoïdes

Une autre difficulté vient elle du choix même de la structure utilisée pour représenter les ensembles : les sétoïdes. S'ils sont pour le moment nécessaires pour permettre le travail avec des ensembles quotients, ils obligent à travailler avec une égalité au-dessus de celle généralement utilisée dans Coq (l'égalité de Leibniz). Les étapes de réécriture dans les preuves, très courantes, se font donc par application de lemmes de compatibilité successifs, ce qui est très coûteux en terme de mémoire et d'efficacité.

Ce problème, ajouté à l'empilement des différentes structures mathématiques et des différentes coercions, ralentit considérablement Coq. Il n'a par exemple pas été rare d'avoir à attendre de cinq à quinze minutes l'exécution d'une tactique (étape élémentaire de preuve) dans les démonstrations relatives au faisceau associé.

De plus, utiliser les sétoïdes oblige à démontrer les lemmes de compatibilité pour toute nouvelle application, même (et surtout, parce que le cas est bien plus courant) lorsque les sétoïdes ne représentent pas de structure quotient. Là encore, ce n'est pas que prouver ces lemmes soit difficile, mais que leur nombre (chaque proposition, chaque fonction doit être compatible) et leur inutilité (lorsque l'ensemble que l'on considère n'est pas un ensemble quotient, il ne devrait pas être nécessaire de les démontrer) rend le travail de formalisation extrêmement pénible et fastidieux.

3.2.1.3 Coq n'est pas ré-entrant

Le fait que Coq ne soit pas ré-entrant, c'est-à-dire qu'il n'arrive pas toujours à typer des expressions qu'il a pourtant lui même généré, pose lui aussi problème.

En effet, lors de la construction des preuves, les buts successifs sont souvent le résultat (entre autres) de plusieurs étapes de beta-iota-delta réduction. Lorsque l'on doit ensuite appliquer un théorème en donnant pour argument un des sous-termes du but courant et que celui-ci n'est pas accepté par Coq(parce qu'il n'arrive pas à le re-typer), c'est à l'utilisateur de reconstituer un terme valide. Le pire étant lorsque c'est un terme de preuve qu'il faut reconstituer, ce qui arrive régulièrement puisque ceux-ci font partie de la majorité des objets que l'on a considérés et construits.

3.2.2 Une solution au problème d'effectivité.

On l'a dit, avec l'avancée de la formalisation, le nombre croissant de structures empilées les unes sur les autres, de coercions à gérer par le système et d'arguments implicites que Coq a de plus en plus de mal à rendre explicite ont considérablement ralenti la vitesse d'exécution de Coq. Lors du portage à la version 7.4 de Coq l'ordinateur à même souvent échoué à terminer la compilation de certains fichiers (la taille mémoire du processus coq augmentait jusqu'à dépasser les limites du PC).

3.2.2.1 Tout mettre à plat

Pour y remédier, nous avons tout d'abord tenté de limiter au maximum l'utilisation des coercions en mettant toutes les structures *à plat*, c'est-à-dire en n'empilant plus les structures. Un anneau n'est ainsi plus vu comme un groupe auquel on rajoute une loi de multiplication interne ayant certaines propriétés mais comme un ensemble avec deux lois, un zéro, une unité et la série bien connues de propriétés :

Definition.
a ring is a structure with the following components:
ring_set: a set *(coercion)*
ring_add: a law of composition on ring_set
ring_zero: of type ring_set
ring_opp: of type ring_set $\sim\!\rightarrow$ ring_set
ring_mult: a law of composition on ring_set
ring_unit: of type ring_set
ring_assoc: a proof that ring_add is associative
ring_com: of type commutative(ring_add)
ring_zero_r: of type ring_zero is a unit on right for ring_add
ring_zero_l: of type ring_zero is a unit on left for ring_add
ring_opp_l: of type (inverse_l ring_add ring_zero ring_opp)
ring_opp_r: of type (inverse_r ring_add ring_zero ring_opp)
ring_mult_assoc: a proof that ring_mult is associative
ring_unit_r: of type ring_unit is a unit on right for ring_mult
ring_unit_l: of type ring_unit is a unit on left for ring_mult
ring_dist_r: of type (dist_r ring_mult ring_add)
ring_dist_l: of type (dist_l ring_mult ring_add)

On perd certes énormément en élégance mais cette méthode permettait quand nous l'utilisions de gagner en vitesse avec un rapport allant du simple au double.

3.2.2.2 Renommer les sous termes propositionnels

L'autre source de ralentissement est la présence de termes de preuves dans une grande proportion des termes que l'on considère. Ces termes de preuves peuvent être énormes et sont re-typé par Coq à chaque fois qu'il les rencontre.

Une première solution est d'utiliser la tactique `Abstract` de manière systématique. `Abstract` prends une tactique en argument (celle-ci doit résoudre complètement le sous-but en cours) et sauvegarde un lemme auxiliaire correspondant au sous-but en cours et sa preuve. L'objet construit globalement ne contiendra donc pas la preuve mais uniquement le nom du lemme auxiliaire. On gagne ainsi en place mémoire et en temps puisque Coq ne re-type rien.
Le problème c'est qu'il faut justement donner la preuve entière en argument à la tactique `Abstract`, et donc passer d'un script de preuve linéaire comme :

```
Simpl.
```

```
Exists (single (full E)).
Split.
Unfold included.
Intros.
Cut (Equal x (full E)).
Intros.
Apply in_part_comp_l with (full E).
Apply base_total_prf.
Auto with algebra.
Auto with algebra.
Apply included_antisym.
Intros.
Auto with algebra.
```

à :

```
Abstract (
Simpl;  (Exists (single (full E));
        (Split;
        [Unfold included;
          (Intros;
          (Cut (Equal x (full E));
            [Intros;
            (Apply in_part_comp_l with (full E);
             [Apply base_total_prf | Auto with algebra]) |
               Auto with algebra])) |
          Apply included_antisym; (Intros; Auto with algebra)]))
 ).
```

Pour faire cela, il faut dans un premier temps construire la preuve, puis la rejouer afin de reconstituer les différents embranchements de celle-ci quand un but se subdivise en plusieurs sous-but et finalement la condenser en une unique succession de tactiques. Cette activité se révèle au final très fastidieuse.

La solution consiste évidemment à automatiser tout cela par une commande. La commande `DefinedAbstract`[1] que l'on a dévellopé s'utilise à la place de la commande `Defined`. Elle fonctionne ainsi : avant de sauvegarder le terme en cours, elle le parcours récursivement afin d'y trouver d'éventuels sous-termes de sorte `Prop` (des termes de preuves). Lorsque qu'elle trouve un tel terme, elle le sauvegarde sous un autre nom et le remplace dans le terme initial par ce nom.

L'utilisation systématique de `DefinedAbstract` a enfin permis à la version 7.4 de Coq de compiler notre dévelopement, et accélère de manière assez spectaculaire la vitesse d'execution.

[1]Cette commande à été écrite en commun avec Loïc Pottier dans la version de Coq 6.3.1 (pour les quotients, cf section 5.2.1.2) puis portée par Loïc Pottier à la version Coq7.4 .

3.3 Conclusion

Le développement présenté dans ce chapitre représente 19000 lignes de code, et permet donc de formaliser une des toutes premières notions de la géométrie algébrique, les schémas affines.

Le mécanisme de coercions, ainsi que la synthèse automatique des arguments implicites se sont avérés être des atouts primordiaux tout au long de ce travail, allégeant à la fois la lecture des énoncés, l'écriture des termes, et la compréhension de ceux-ci par l'utilisateur.

La taille et la complexité des termes considérés grandissent néanmoins de manière non négligeable tout au long de l'avancée de notre travail (plus d'une quinzaine de lignes de code, arguments implicites et coercions cachées, pour certains termes). En cela les spécificités de CtCoq (possibilités de notations étendues, d'édition structurée du code et *proof-by-pointing*), ont été nécessaires pour mener à bien notre projet.

Le manque de sous-types (qui permettraient de ne plus utiliser la fonction `subtype_elt`) et de types quotients (qui nous permettraient de ne plus utiliser les setoïdes) sont les principaux problèmes qui ont ralenti notre travail.

Chapitre 4

Horace

Le lemme d'Horace géométrique ([Hir85],[Mig97]) dont il est question ici donne un moyen de répondre à la question suivante : étant donné un degré d et n points $P_1, \ldots, P_n$ en position générique dans le plan, affectés de multiplicités $m_1, \ldots, m_n$: quelle est la dimension de l'espace vectoriel des courbes de degré d passant par les n points P_i avec la multiplicité au moins m_i ? Plus précisément, il donne un moyen (à condition de faire un choix judicieux de différents paramètres) de réduire ce problème à un problème de degré plus faible. En itérant l'application du lemme, on arrive ainsi et ainsi à un problème de degré très faible dont on connait les caractéristiques.

L'objectif initial de cette thèse était d'arriver à formaliser ce lemme d'Horace mais malheureusement les problèmes soulevés au chapitre précédent sont tels que cet espoir est bien vite devenu irréalisable en un temps raisonnable. On a donc tout de même formalisé une version axiomatique et partielle de ce théorème, toutefois suffisante pour être appliquée à la vérification numérique des utilisations qui en sont faites.

La vérification avec notre developpement d'un mémoire ([dS]) portant sur l'étude des courbes de genre 5 a ainsi pu mettre en avant un grand nombre d'erreurs.

4.1 Le lemme d'Horace

Le lemme d'Horace, dont on propose dans ce chapitre une formalisation partielle, a pour sujet l'étude des systèmes linéaires de courbes planes de degré fixé et à multiplicités imposées en r points du plan projectif.

Si on se fixe un degré d et r multiplicités $m_1, \ldots, m_r$, et si on note $\underline{d}$ la *classe* des courbes planes de degré d ayant la multiplicité au moins m_i en chaque point P_i, les questions auxquelles on s'intéresse sont les suivantes :

- Quelle est la dimension de l'espace vectoriel défini par $\underline{d}$?

– La courbe générique de $\underline{d}$ est-elle géométriquement irréductible ?

– Cette même courbe est-elle lisse en dehors des points P_i ?

Les spécifications réalisées au chapitre 4 ne sont pas suffisantes pour parler des propriétés géométriques des courbes (irréductibilité, lissité), on se donne donc ces dernières sous forme de paramètre, seules les conditions numériques et celles qui peuvent s'y réduire seront vérifiées. En particulier on ne s'intéressera qu'à la première question : la dimension de $\underline{d}$.

4.1.1 Cadre, définitions et notations

On se place dans le plan projectif complexe $\mathbb{P}^2_{\mathbb{C}}$.Considérons un degré d et r multiplicités $m_1, \ldots, m_r$. Si $(P_1, \ldots, P_r)$ désigne le r-uplet générique de $\mathbb{P}^2_{\mathbb{C}}$, nous noterons $\underline{d} = (d; m_1 \ldots, m_r)$ la *classe* des courbes planes de degré d ayant la multiplicité au moins m_i en chaque point P_i.

Par exemple, $\underline{d} = (5; 2, 1, 1)$, que l'on notera aussi de manière abrégée $(5; 2, 1^2)$, désigne la classe des quintiques assignées à passer par trois points simples P_1, P_2, P_3 et ayant un point double en P_1.

Les points P_i peuvent être en position générique dans le plan, mais aussi dans des positions plus *spécialisées* (ce sera le cas dans l'utilisation du lemme d'Horace). On note $\mathcal{L}_x(\underline{d})$ le système linéaire des courbes de degrés d assignées à passer par les points P_i avec la multiplicité m_i quand les points P_i sont en position x.

Enfin, $\mathcal{D}_x(\underline{d})$ désignera la courbe générique de $\mathcal{L}_x(\underline{d})$.

L'espace vectoriel des courbes planes de degré d est de dimension $C^2_{d+2} = \frac{(d+1)(d+2)}{2}$, et imposer à une courbe de passer par un point avec une multiplicité m revient à imposer à toutes les dérivées partielles de cette courbe à s'annuler en ce point jusqu'à l'ordre $m-1$, ceci impose donc $\frac{m(m+1)}{2}$ conditions.

Définition 4.1.1 *La dimension attendue du système $\underline{d}$ lorsque les points P_i sont en position générique est*

$$Max \left\{ \chi(\underline{d}) = \frac{(d+1)(d+2)}{2} - \sum_{i=1}^{r} \frac{m_i(m_i+1)}{2}, 0 \right\}.$$

On dit que le système $\underline{d}$ est régulier si sa dimension réelle est égale à sa dimension attendue. On dit encore qu'il est effectif si sa dimension réelle est strictement positive.

Tous les systèmes ne sont pas réguliers : celui des coniques assignées à passer par deux points doubles en position générique est de dimension attendue nulle, pourtant la droite 'double' passant par ces deux points satisfait au problème.

Définition 4.1.2 *Le genre arithmétique de $\underline{d}$ est :*

$$g(\underline{d}) = \frac{(d-1)(d-2)}{2} - \sum_{i=1}^{r} \frac{m_i(m_i-1)}{2}.$$

Définition 4.1.3 *Le nombre d'intersection de deux classes $\underline{c}$ et $\underline{d}$ est :*

$$\underline{c}.\underline{d} = cd - \sum_{i=1}^{r} m_i n_i$$

Il correspond, si $\underline{c}$ et $\underline{d}$ sont sans composantes communes, au nombre de points d'intersections de $D(\underline{d})$ et $D(\underline{c})$ dans le plan éclaté aux points P_i. On peut le voir aussi comme le nombre de points d'intersection qu'il reste dans $\mathbb{P}^2_{\mathbb{C}}$ quand on a compté les points P_i avec leurs multiplicités.

4.1.2 Principe de la méthode d'Horace

Avant de citer le lemme d'Horace géométrique, donnons-nous un exemple (très) simple pour en dégager le principe.

Intéressons nous à la classe des cubiques assignées à passer par un point double et 6 points simples. Soit D la courbe générique de ce système.

$$\underline{d} = (3; 2, 1^6), \;\; \chi(d) = 10 - 3 - 6 = 1.$$

Considérons la droite C_1 passant par P_2 et P_3 sur laquelle on *spécialise* P_4 et P_5 (on leur impose d'appartenir à C_1, ils ne sont donc plus en position générique dans le plan).

D'après le théorème de Bezout, si D et C_1 n'ont pas de composante commune le nombre de points d'intersection de D et C_1 (multiplicitées comptées) est $3.1 = 3$. Or quatre points P_2, P_3, P_4, P_5 appartiennent à $D \cap C_1$, la droite C_1 est donc une composantede D. L'équation de D se factorise donc par celle de la droite C.

On *factorise* alors le problème, et la question se réduit donc (dans notre position spécialisée) à connaitre la dimension du système des coniques passant par un point double P_1 et deux points simples P_6 et P_7 de classe $d_1 = (2; 2, 0^4, 1)$.

Considérons alors simplement la droite C_2 passant par P_1 et P_6, le nombre de points d'intersection (multiplicités comptées) avec la conique D_1 est trois, un de trop, et la droite

C_2 est donc une composante de D_1. Encore une fois le système se réduit à celui d'une droite passant par deux points simple P_1 et P_7 : une unique droite.

Dans notre spécialisation il y a donc une unique cubique ayant un point double en P_1 et passant par les 6 points simples $P_2, ...P_7$. Avoir une dimension donnée pour un système est une condition ouverte par semi-continuité sur la position des points$P_1, \ldots, P_7$, aussi trouver une spécialisation des points P_i pour laquelle notre système a bien la dimension attendue nous assure que c'est bien le cas sur tout un ouvert contenant cette position.

Le lemme d'Horace géométrique généralise ce type de démonstration à la spécialisation sur une courbe irréductible du plan quelconque (et non pas simplement une droite). L'idée est la suivante : si $|\underline{d}|$ est le système dont on veut étudier la dimension et la courbe générique, on choisit une courbe irréductible C du plan, puis on spécialise certains des points sur C, jusqu'à ce que cette courbe devienne une composante fixe du système. On peut alors 'soustraire' cette courbe au système précédent, pour obtenir un nouveau système, dit résiduel. Sous certaines conditions numériques et géométriques, l'étude du système initial se réduit alors à celle du système résiduel. La dimension de ce nouveau système étant inférieure, les applications successives du lemme d'Horace géométrique permet souvent de se ramener à des systèmes de degrés très bas dont on connaît les caractéristiques.

4.1.3 Le lemme d'Horace géométrique

Lemme 4.1.4 *(Lemme d'Horace) Soient $\underline{d} = (d; m_1, \ldots, m_r) \in \mathbb{Z}^{r+1}$ et x, y deux points de X, l'ensemble des r-uplets de points distincts de $(\mathbb{P}^2)^r$, tels que x se spécialise en $\underset{\smile}{y}$. Soit C une courbe de $\mathbb{P}^2_y$ et $\widetilde{C}_y$ sa transformée dans le plan éclaté S_y. On suppose que $\widetilde{C}_y$ est géométriquement irréductible et lisse.*

Dimension et spécialisation.

Supposons que

1. *$\underline{d}.\underline{c} + 1 - g(\underline{c}) = -a$ où $a \geq 0$,*
2. *en position y, il y a $g(\underline{c})$ points libres sur C, affectés de multiplicité supérieure ou égale à 1,*
3. *si $a \geq 1$, il y a $a+1$ points libres sur C, affectés de multiplicité supérieures ou égales à 1 tels que :*
   en position x, ces $a+1$ points sont libres dans le plan $\mathbb{P}^2$
   en position y, ils sont libres sur la courbe C,
4. *si $a = 0$, $\mathcal{L}_y(\underline{d} - \underline{c})$ a la dimension attendue $\chi(\underline{d})$*
   si $a \geq 1$, $\mathcal{L}_y(\underline{d} - \underline{c} - P^C_{i_1} - \ldots - P^C_{i_a+1})$ a la dimension attendue $\chi((\underline{d} - \underline{c}) - a - 1 = \chi((\underline{d}) - 1$
   alors $\mathcal{L}_y(\underline{d})$ a la dimension attendue.

Irréductibilité.

Si de plus $\chi(\underline{d}) \geq 1$ *et*

5. *Si* $a = 0$*, la classe* $\underline{c}$ *n'est pas effective en position* x*,*
6. $\mathcal{D}_y(\underline{d} - \underline{c})$ *est géométriquement irréductible*

alors $\mathcal{D}_y(\underline{d})$ *est géométriquement irréductible et réduite .*

Lissité.

Si enfin

7. *lorsque* $a = 0$*,* y *est lisse dans l'adhérence de* x*,*
8. *si* $\mathcal{D}_y(\underline{d} - \underline{c})$ *rencontre* $\widetilde{C}_y$ *transversalement,*
9. *si* $\mathcal{D}_y(\underline{d} - \underline{c}) \cap \widetilde{C}_y$ *est irréductible (pas géométriquement),*
10. *et si* $\mathcal{D}_y(\underline{d} - \underline{c})$ *est lisse,*

alors $\mathcal{D}_y(\underline{d})$ *est lisse.*

4.1.4 Un exemple d'application de la méthode

L'exemple suivant, que nous donnons à titre d'exemple d'application du lemme et de la méthode d'Horace, est tiré du mémoire de Benoîte de Saporta ([dS]) qui porte sur l'étude des systèmes linéaires des courbes de genre 5.

Citons Benoîte de Saporta :

Considérons le système des courbes de degré 16 assigné à passer par 10 points de multiplicité 5. On a $\underline{d} = (16; 5^{10})$ *et la dimension attendue est* $\chi = 3$*. On peut toujours rajouter à un système des points simples sans en changer la régularité : ici on se ramène donc au système* $\underline{d} = (16; 5^{10}, 1^2)$ *qui à 1 pour dimension attendue.*

En position initiale x_0 *les points* $P_1, ...P_{12}$ *sont en position générique dans le plan. Appliquons le lemme d'Horace avec la cubique* C *ayant un noeud en* P_1 *et passant par* $P_2, ..., P_7$*. On spécialise* P_8 *et* P_9 *sur* C*. On a ainsi* $\underline{c} = (3; 2, 1^8, 0^3)$ *et* $g(\underline{c}) = 0$*. En position actuelle* x_1*,* P_8 *et* P_9 *ne sont plus en position générique mais spécialisés sur* C*. Vérifions les hypothèses du lemme d'Horace :*

– 1. $\underline{\text{d}}.\underline{\text{c}}$ +1 = -1. On a $a = 1$.

– 2. $g(\underline{c}) = 0$, il n'y a donc rien à vérifier.

– 3. il y a bien 2 points libres sur C (c'est à dire spécialisés sur C uniquement) : P_8 et P_9, qui étaient génériques dans $\mathbb{P}^2_{\mathbb{C}}$ en position x_0.

Le point n°4 est plus délicat. La classe résiduelle $\underline{d}_1$ *est égale à* $(13; 3, 4^8, 5, 1^2)$*. On a* $\underline{d}_1.\underline{c} = 1$*. Par ailleurs* $D(\underline{d}_1 - P_8^C - P_9^C)$ *et* C *se coupent au moins deux fois dans le plan éclaté aux points* P_i*, et* C *est irréductible : donc* C *est une composante fixe du système* $\mathcal{L}_y(\underline{d}_1 - P_8^C - P_9^C)$*. On peut donc soustraire* C *sans changer la dimension réelle.*

Considérons donc la nouvelle résiduelle $\underline{d}_2 = (10; 1, 3^8, 5, 1^2)$*, on réapplique le lemme d'Horace plusieurs fois pour calculer sa dimension.*

Soit C_2 *la cubique passant par* $P_2, \ldots, P_{10}$*, on y spécialise* P_{11}*. On a* $\underline{c}_2 = (3, 0, 1^1 0, 0)$*,* $g(\underline{c}_2) = 1$ *et :*

– 1. $\underline{d}_2.\underline{c}_2 + 1 - 1 = 0$. On a $a = 0$.
– 2. $g(\underline{c}_2) = 1$, P_{11} est bien libre sur C_2.
– 3. $a = 0$ il n'y a rien à vérifier.

Le point 4 nous ramène à la résiduelle $\underline{d}_3 = (7; 1, 2^8, 4, 0, 1)$ *dont il faut encore démontrer qu'elle a bien la dimension attendue.*

Soit C_3 *la droite* P_8, P_9*, où on spécialise* P_{10}*. On a* $\underline{c}_3 = (1, 0^7, 1^3, 0^2)$*,* $g(\underline{c}_3) = 0$ *et :*

– 1. $\underline{d}_3.\underline{c}_3 + 1 = 0$. On a $a = 0$.
– 2. $g(\underline{c}_3) = 0$, il n'y a donc rien à vérifier.
– 3. $a = 0$ il n'y a rien à vérifier non plus

On est donc ramené à $\underline{d}_4 = (6; 1, 2^6, 1^2, 3, 0, 1)$*. On exploite cette fois-ci la droite* $C_4 = (P_6, P_{10})$ *en y spécialisant* P_7*. On a* $\underline{c}_4 = (1, 0^5, 1^2, 0^2, 1, 0^2)$*,* $g(\underline{c}_4) = 0$ *et :*

– 1. $\underline{d}_4.\underline{c}_4 + 1 = 0$.
– 2. et 3. $g(\underline{c}_4) = 0$ et $a = 0$, il n'y a donc rien à vérifier.

On a $\underline{d}_5 = (5; 1, 2^4, 1^4, 2, 0, 1)$*. On spécialise simplement* P_5 *sur* C_3 *. On a maintenant* $\underline{c}_3 = (1, 0^4, 1, 0^2, 1^3, 0^2)$ *et toujours* $g(\underline{c}_3) = 0$ *:*

– 1. $\underline{d}_5.\underline{c}_3 + 1 - 0 = 0$.
– 2. et 3. $g(\underline{c}_3) = 0$ et $a = 0$, il n'y a donc rien à vérifier.

$\underline{d}_6 = (4; 1, 2^3, 1^3, 0^2, 1, 0, 1)$*, on spécialise* P_4 *sur* C_4*. Il vient*
$\underline{c}_4 = (1, 0^3, 1, 0, 1^2, 0^2, 1, 0^2)$*,* $g(\underline{c}_4) = 0$ *et :*

- 1. $\underline{d}_6.\underline{c}_4 + 1 - 0 = 0$.
- 2. et 3. $g(\underline{c}_4) = 0$ et $a = 0$, il n'y a donc rien à vérifier.

Finalement on est ramené à la résiduelle $\underline{d}_7 = (3; 1, 2^2, 1^2, 0^6, 1)$, où tous les points qui sont de multiplicités non nulle sont en position générique. La question est ainsi : quelle est la dimension réelle du système linéaire des cubiques assignées à passer par deux points doubles et quatre points simples ? La réponse est connue : 0, qui est bien la dimension attendue. En remontant pas à pas, notre système initial est donc bien régulier.

Bien que numériquement tout à fait sans erreur, cette démonstration est malheureusement fausse, l'une des utilisation faite du lemme d'Horace n'est pas valide. On s'en rendra compte en essayant de la valider sous Coq avec notre developpement.

4.2 Notre formalisation

4.2.1 Classes et courbes

Formaliser une classe $\underline{d}$ est facile : c'est simplement une liste d'entiers relatifs de taille $n + 1$. Plutôt que de considérer des listes dépendantes (incluant donc la taille de la liste dans sa construction), on choisit simplement de considérer une liste de taille libre et de contrôler la taille lors de l'utilisation du lemme d'Horace[1].

Definition.
let Pic be list(Z).

Il faut aussi formaliser la courbe C dont il est question dans le lemme d'Horace et sur laquelle on va spécialiser des points encore libres. Afin de ne pas introduire de notion de courbe, on la verra aussi comme une liste d'entiers naturels c : comme pour les classes le premier élément c_1 est interprété comme le degré de la courbe et les suivants comme de conditions pour la courbe à passer par les points P_i avec la multiplicité c_{i+1}, avec l'idée que ces conditions suffisent à définir la courbe. Evidement on pourrait fixer n'importe quelle condition, on demandera donc dans le lemme d'Horace de vérifier que la classe ainsi définie est effective.

Definition.
Let curve be the inductive type (of type Set) with constructors:
curvedef_points: list(Z) $\Rightarrow$ curve

[1] On a aussi formalisé une version avec des listes dépendantes, plus élégante mais plus difficile à utiliser et surtout plus lente lors des calculs du lemme d'Horace. `http ://www-sop.inria.fr/lemme/personnel/Laurent.Chicli/`.

4.2.2 Dimensions, genre, nombre d'intersection.

Les calculs de la dimension attendue, du genre, et du nombre d'intersection ne posent pas de problèmes particulier en Coq. Par exemple on définit la dimension attendue ainsi :

```
Definition khi :=
   [cl:Pic]
      Cases cl of
          nil => ZERO
         | (cons d m) =>
             (Zminus
                (Zdiv2 (Zmult (Zplus d (1)) (Zplus d (2))))
                (sum_elt_nat_fun (length m) m nbcond))
      end.

Definition dim_attendue := [c:Pic]
          (max_Z (Zminus (khi c) (S O)) (Zinv (S O))).
```

Dans CtCoq, on ommet l'affichage des cas inutiles, et cela donne :

Definition.
let χ be
the function defined by , for all (d,m):Pic ,
$[(d + (1)) * (d + (2))]/2 - \sum_{i=1..length(m)}$ nbcond(m_i).
Definition.
let dim_attendue be the function defined by , for all c:Pic , Max (χ(c) $-$ 1,$-$1).

De même on définit le genre d'une classe :

Definition.
let genre_fun be the function defined by , for all m:Z , $[m * (m - 1)]/2$
(its type is Z $\Rightarrow$ Z).
Definition.
let genre be
the function defined by , for all (d,m):Pic ,
$[(d - (1)) * (d - (2))]/2 - \sum_{i=1..length(m)}$ genre_fun(m_i).

et le nombre d'intersection :

Definition.
let intersection_number be
the function defined by , for all (d1,m1),(d2,m2):Pic d1 $*$ d2 $- \sum_{i=1..length(m1)}$ (m1*m2)$_i$.

4.2.3 Positions et Spécialisations

Dans le lemme d'Horace on part d'une position x que l'on spécialise (on assigne des points sur des courbes) pour arriver à la position y. Un point peut donc au cours des application successives du lemme d'Horace, soit être libre dans le plan, soit libre sur une courbe (il a été spécialisé sur cette courbe), soit à l'intersection de deux courbes (parce qu'il à été de nouveau spécialisé sur une autre courbe, il n'est dans ce cas plus libre sur aucune des deux courbes).

On définit donc le type inductif `position` suivant :

Definition.
Let position be the inductive type (of type Set) with constructors:
free: position
specialised: curve $\Rightarrow$ position
specialised_at_inter: curve $\Rightarrow$ curve $\Rightarrow$ position

Et une position x sera simplement la liste des informations de position des points $P_1, ..., P_n$:

Definition.
let Positions be list(position).

Pour indiquer les points que l'on souhaite spécialiser sur la courbe C, on créé simplement un type inductif `yesno` avec deux constructeurs `yes` et `no` :

Definition.
Let yesno be the inductive type (of type Set) with constructors:
yes: yesno
no: yesno

Definition.
let to_specialise be list(yesno).

La liste [no,no,yes,yes,no,no] de type `to_specialize` signifiera donc que l'on souhaite spécialiser sur la courbe C les points P_3 et P_4.

4.2.4 Le lemme d'Horace

4.2.4.1 Formalisation de l'en-tête du lemme

Reprenons le début de l'intitulé du lemme d'Horace :

Soient $\underline{d} = (d; m_1, \ldots, m_r) \in \mathbb{Z}^{r+1}$ *et* x, y *deux points de* X*, l'ensemble des* r*-uplets de points distincts de* $(\mathbb{P}^2)^r$*, tels que* x *se spécialise en* y*.*

On part donc d'une position x (la liste des positions des points P_i) et plutot que de se donner une position y et de vérifier que x se spécialise en y on se donne x et une liste de points (sous forme de liste de termes de type `yesno`) à spécialiser sur la courbe C : on calcule ensuite y.

La fonction `new_position` qui réalise ce calcul fonctionne de la manière suivante. On parcours les listes x de type `Positions` et l de type `to_specialize` récursivement : au rang i si l_i est `yes` c'est à dire si on veut spécialiser le point P_i alors :

- si x_i est `free` on l'envoie sur `(specialised c)` où c est la liste définissant la courbe C,
- si P_i est simplement spécialisé sur une courbe D (x_i est `(specialised d)`), donc encore libre sur D, on envoie x_i sur l'intersection `(specialised_at_inter d c)`,
- si P_i est déja spécialisé à une intersection, on ne peut le spécialiser ailleurs : on l'envoie donc sur n'importe quoi (`(specialised c)` dans notre formalisation) et une autre propriété (`well_to_specialize`, chargée justement de vérifier la validité de la spécialisation) n'autorisera pas l'utilisation du lemme d'Horace.

En `Coq`, on obtient le code suivant :

Definition:
Let new_position be the recursive function, with value into curve $\Rightarrow$ list(position),
mapping x:list(position),l:to_specialise, to:
c:curve , Cases of x :

```
nil=>nil(position)
(cons a b)=>Cases of l :

        nil=>x
        (cons lhead ltail)=>Cases of lhead :

                yes=>Cases of a :

                    free=>specialised(c),(new_position b ltail c)
                    (specialised d)=>
                    (specialised_at_inter d c),
                     (new_position b ltail c)
                    (specialised_at_inter e1 e2)=>
                    specialised(c),(new_position b ltail c)

                no=>a,(new_position b ltail c)
```

La fonction `well_to_specialize` est donc chargée de vérifier si on essaie pas de spécialiser un point qui est déjà à une intersection de deux courbes, mais aussi de vérifier qu'on essaie pas de spécialiser sur une courbe un point qui sert justement à définir cette courbe. Elle renvoie donc `false` si

– l_i est `yes` et x_i est `(specialised_at_inter nimporte quoi)`

– l_i est `yes` et c_i est différent de 0.

En `Coq` :

Definition:
Let well_to_specialize be the recursive function, with value into bool,
mapping x:list(position),c:list(Z),l:to_specialise, to:
Cases of l :

```
nil=>true
(cons lhead ltail)=>Cases of lhead :

        yes=>Cases of x :

            nil=>true
            (cons xhead xtail)=>Cases of xhead :

                    (specialised_at_inter nimporte quoi)=>false
                    other=>Cases of c :

                        nil=>true
                        (cons chead ctail)=>
                        Cases of chead = (0) :

                        true=>
                        (well_to_specialize xtail ctail ltail)
                        false=>false

        no=>Cases of x :

            nil=>true
            (cons xhead xtail)=>Cases of c :

                    nil=>true
                    (cons chead ctail)=>
                    (well_to_specialize xtail ctail ltail)
```

Il reste une partie de ce premier bout d'énoncé que l'on n'a pas vérifié : x, y *deux points de* X*, l'ensemble des* r*-uplets de points distincts de* $(\mathbb{P}^2)^r$. On doit donc vérifier qu'en partant d'une liste x de points distincts, la nouvelle position `(new_position x l c)` est bien aussi consituée de points distincts. Cela peut arriver lors de spécialisations *abusives* : si trois points sont déjà à l'intersection d'une droite et une cubique, on ne peut y spécialiser un quatrième point sans que celui ci soit confondu avec l'un des trois points (toujours par Bezout).

Pour éviter cela, on calcule le nombre d'intersection de la classe de c dans la nouvelle position `(new_position x l c)` et de la classe d dans la position x pour toutes les courbes

d où sont déja spécialisés des points (on a dit que ce nombre est le nombre de points d'intersection *qu'il reste* si on a comptés les points P_i) : si ce nombre est positif ou nul, la spécialisation est valide, sinon c'est qu'il y a trop de points que l'on essaye de spécialiser à l'intersection de c et d.

On doit donc :

- A) écrire la fonction qui calcule la classe d'une courbe C en position y,
- B) avoir la liste des courbes utilisées pour décrire la position x,
- C) vérifier enfin tous les nombres d'intersection.

A) On l'a dit, dans notre formalisation une courbe C est donnée par un degré et la liste des points par lesquels elle est assignée à passer et qui sont ainsi censés la définir (ce qui est vérifié par une autre propriété dans notre lemme d'Horace). Par exemple la droite (P_1, P_2) est formalisée par (1 ;1,1).

La classe $\underline{c}$ de la courbe C en position x est l'élément de $\mathbb{Z}^{n+1}$ défini par $\underline{c}_1 = c_1$ et, pour $i > 1$:

- $\underline{c}_i = c_i$ si P_i est générique ou spécialisé sur une ou deux courbes autres que c

- $\underline{c}_i = 1$ si P_i est spécialisé sur c ou à l'intersection de deux courbes dont l'une est c.

On a vu qu'on vérifie (avec `well_to_specialize`) qu'on ne cherche pas à spécialiser un point P_i sur la courbe C s'il sert à définir cette courbe : cela assure le non chevauchement des cas ci-dessus.

En Coq, pour calculer la classe de la courbe C en position x, on construit d'abord la fonction `construct_class` suivante, qui renvoie la liste des multiplicités de $\underline{c}$. Comme on doit toujours vérifier l'égalité des courbes sur lesquels sont spécialisés les points P_i avec c, on prend deux fois la courbe c en paramètre : la première sera destructurée, la seconde conservée pour les tests d'égalité.

Definition:
Let construct_class be the recursive function, with value into curve ⇒ list(Z),
mapping c:list(Z),x:list(position), to:
catester:curve , Cases of c :

```
            nil=>nil
            (cons  chead  ctail)=>Cases of x :

                              nil=>nil(Z)
                              (cons  xhead  xtail)=>Cases of xhead :

                                                free=>
                                                chead,
                                                 (construct_class
                                                    ctail  xtail  catester)
                                                (specialised  classe1)=>
                                                Cases of classe1 =  catester :

                                                true=>
                                                (1),
                                                 (construct_class
                                                    ctail  xtail  catester)
                                                false=>
                                                chead,
                                                 (construct_class
                                                    ctail  xtail  catester)

                                                (specialised_at_inter
                                                   classe1  classe2)=>
                                                Cases of classe1 =  catester  ∨b
                                                         classe2 = catester :

                                                true=>
                                                (1),
                                                 (construct_class
                                                    ctail  xtail  catester)
                                                false=>
                                                chead,
                                                 (construct_class
                                                    ctail  xtail  catester)
```

La classe de c en position x, que l'on note $c_{[x]}$ dans CtCoq, se définit alors simplement :

Definition.

```
let  classe  be
  the function defined by , for all  c:curve ,
                                   x:Positions ,  d°(c),(construct_class  mults(c)  x  c)
 (its type is  curve  ⇒  Positions  ⇒  Pic).
```

B) On obtient la liste des courbes sur lesquelles les points P_i sont spécialisés simplement par filtrage sur la position x :

Definition:

```
Let  list_all_curves  be the recursive function,  with value into  list(curve),
mapping  x:list(position)  to:
Cases of x :

nil=>nil(curve)
(cons  xh  xt)=>Cases of xh :

            free=>list_all_curves(xt)
            (specialised  C)=>C,list_all_curves(xt)
            (specialised_at_inter  C  D)=>C,D,list_all_curves(xt)
```

Et pour alléger les calculs on élimine toutes les redondances de cette liste :

Fixpoint

```
  elim_redondances  l:list(curve):  list(curve)  :=
     (list_without_repeat_cons   l  nil(curve))
  with
  list_without_repeat_cons  l:list(curve):  list(curve)  ⇒  list(curve)  :=
     intermed:list(curve) ,  Cases of l :

                      nil=>intermed
                      (cons  lh  lt)=>Cases of (in_bool_list   intermed   lh) :

                                 true=>(list_without_repeat_cons    lt  intermed)
                                 false=>(list_without_repeat_cons    lt  lh,intermed)

                      .
```

C) On peut enfin vérifier tous les nombres d'intersection, c'est la fonction `verif_specialise_list` suivante qui en est chargée, appliquée à la liste de courbe calculée précédemment.

Definition:

```
Let  verif_specialise_list  be the recursive function,  with value into  curve  ⇒
                                                                 to_specialise  ⇒
                                                                 Positions  ⇒  bool,
mapping  listofcurve:list(curve)  to:
c:curve ,  l:to_specialise ,  x:Positions ,  Cases of listofcurve :

                              nil=>true
                              (cons  c1  listofcl2)=>Cases of c1 =  c :

                                          true=>
                                          (verif_specialise_list  listofcl2  c  l  x)
                                          false=>
                                          Cases of c1[x].c[(new_position  x  l  c)] :

                                          (POS  pos)=>
                                          (verif_specialise_list  listofcl2  c  l  x)
                                          0=>
                                          (verif_specialise_list  listofcl2  c  l  x)
                                          (NEG  neg)=>false
```

Definition.

```
   let   verif_specialise  be
    the function defined by , for all   c:curve ,
                              x:Positions ,
                              l:to_specialise ,
                              (verif_specialise_list
                                  elim_redondances(list_all_curves(x))  c  l  x).
```

A ce point de nos formalisations, on est donc capable de formaliser l'en-tête du lemme d'Horace :

- La position de départ x est une liste d'élements de type `Position` représentant des points qui sont soit libres dans le plan, soit spécialisé et libre sur une courbe, soit à l'intersection de deux courbes.
- La courbe C est donnée sous forme de liste d'entiers (on la définit donc comme la courbe assignée à passer par tels et tels points avec telle multiplicité).
- La position y est calculée à partir de x et de la liste des points que l'on souhaite spécialiser sur C.

Cette formalisation nécessite plusieurs vérifications :

- `well_to_specialise` vérifie à la fois que l'on essaye pas de spécialiser un point qui est déjà à l'intersection de deux courbes mais aussi que l'on essaye pas de spécialiser sur une courbe un point qui sert justement à la définir.
- `verif_specialise` vérifie elle que la spécialisation est valide, c'est à dire que les points P_i restent bien distincts, en vérifiant les nombres d'intersection de la classe de la nouvelle courbe avec les classes de toutes les courbes déjà considérées.
- la courbe C étant définie par des restrictions à passer par des points dans une position

donnée, on doit vérifier que la classe ainsi définie est bien effective.

4.2.4.2 Horace

On peut maintenant formaliser le lemme d'Horace, on en donne deux versions utilisables dans notre développement.

Cas où l'ajustement est nul. La première est le cas le plus simple d'utilisation du lemme d'Horace, et aussi le cas le plus fréquemment utilisé, lorsque l'on spécialise exactement le nombre de points nécessaires sur C pour en faire une composante fixe du système, c'est-à-dire lorsque $-a = \underline{d}.\underline{c}+1-g(\underline{c}) = 0$. Dans ce cas le lemme d'Horace peut se réécrire :

Lemme 4.2.1 *(Lemme d'Horace simple) Soient $\underline{d} = (d; m_1, \ldots, m_r) \in \mathbb{Z}^{r+1}$ et x, y deux points de X, l'ensemble des r-uplets de points distincts de $(\mathbb{P}^2)^r$, tels que x se spécialise en y. Soit C une courbe de $\mathbb{P}^2_y$ et $\widetilde{C}_y$ sa transformée dans le plan éclaté S_y. On suppose que $\widetilde{C}_y$ est géométriquement irréductible et lisse.*

Supposons que

1. $\underline{d}.\underline{c} + 1 - g(\underline{c}) = 0$,
2. *en position y, il y a $g(\underline{c})$ points libres sur C, affectés de multiplicité supérieure ou égale à 1,*
3. *la condition 3 devient triviale,*
4. $\mathcal{L}_y(\underline{d} - \underline{c})$ *a la dimension attendue* $\chi(\underline{d})$

alors $\mathcal{L}_y(\underline{d})$ a la dimension attendue.

La première condition s'énonce facilement : on définit tout d'abord l'ajustement :

Definition.
let ajustement be the function defined by , for all d,c:Pic , Zopp((d.c + 1) − g(c))
(its type is Pic ⇒ Pic ⇒ Z).

puis l'hypothèse s'écrit simplement `(ajustement d (class c (new_position x l c)))=0`.

La seconde condition demande de compter le nombre de points libres sur C (ni libres dans le plan, ni spécialisé à l'intersection de C avec une autre courbe) affectés d'une multiplicité supérieure ou égale à 1. On le fait simplement par filtrage sur la nouvelle position `(new_position x l c)` et sur les multiplicités du système en cours. C'est la fonction `nb_specialised_curve` suivante que l'on utilise (`d` sera la liste des multiplicités du sytème) :

Definition:
Let nb_specialised_curve be the recursive function, with value into curve $\Rightarrow$ nat,
mapping d:list(Z),x:list(position), to:
c:curve , Cases of x :

```
    nil=>O
    (cons  a  b)=>Cases of d :

          nil=>O
          (cons  dh  dtail)=>Cases of dh >  O :

                true=>(nb_specialised_curve  dtail  b  c)
                false=>Cases of a :

                      free=>(nb_specialised_curve  dtail  b  c)
                      (specialised  acurve)=>Cases of c =   acurve :

                            true=>
                            (nb_specialised_curve
                                dtail  b  c)+1
                            false=>
                            (nb_specialised_curve
                                dtail  b  c)

                      (specialised_at_inter  c1  c2)=>
                      (nb_specialised_curve  dtail  b  c)
```

et la condition 2 devient, si `d` est le système en cours, `c` la liste correspondant à la courbe C, `l` la liste des points à spécialiser sur C et `x` la position des points P_i avant spécialisation : `(nb_specialised_curve (mults d) (new_position x l c) c) >= (genre (class c (new_position x l c)) ))`.

En passant Pour compter le nombre de points ainsi libres sur C, on a eu besoin d'un critère d'égalité de courbes. On choisit dans notre formalisation de dire que deux courbes sont égales si elles sont égales comme listes. Pourtant, la droite (P_1,P_2) est égale à la droite (P_1,P_3) si P_3 a été spécialisé sur (P_1,P_2), sans que les listes qui les définissent (resp. $(1,1,1,0)$ et $(1,1,0,1)$) ne soient égales. C'est la condition `verif_specialise` que l'on a décrit auparavant qui va empêcher un tel cas : on ne pourra pas considérer la droite $(1,1,0,1)$ dans cette position si on a déja considéré $(1,1,1,0)$: le nombre d'intersection des deux classes est -1.

En se donnant en paramètre la notion d'irréductibilité et de lissité, on peut alors formaliser ce lemme d'Horace simple :

Definition.

let effective be the function defined by , for all c:Pic , x:Positions , (real_dim c x) >= 0.

Parameter (irreductible_dans_l_eclate). curve $\Rightarrow$ Positions $\Rightarrow$ Prop

Parameter (lisse_dans_l_eclate). curve $\Rightarrow$ Positions $\Rightarrow$ Prop

Axiom Horace_simple:
$\forall$ x:Positions ,
$\forall$ d:Pic ,
$\forall$ c:curve ,
$\forall$ l:to_specialise ,
length(d) = length(x)+1 $\Rightarrow$
Le systeme d−c[(new_position x 1 c)] est regulier en position (new_position x 1 c) $\Rightarrow$
(well_to_specialize x mults(c) l) = true $\Rightarrow$
(verif_specialise c x l) = true $\Rightarrow$
(ajustement d c[(new_position x 1 c)]) = 0 $\Rightarrow$
(nb_specialised_curve mults(d) (new_position x 1 c) c) >= g(c[(new_position x 1 c)]) $\Rightarrow$
c[x] est effective en position x $\Rightarrow$
La courbe c est irreductible dans l'eclate en position x $\Rightarrow$
La courbe c est lisse dans l'eclate en position x $\Rightarrow$
Le systeme d est regulier en position x.

Cas où le nombre d'intersection du système résiduel avec la classe de la courbe vaut 1. Lorsque l'ajustement a est strictement positif, il est impossible d'utiliser le lemme d'Horace de manière récursive, puisque l'on bute sur la quatrième hypothèse, que l'on ne peut vérifier en réappliquant le lemme d'Horace :

4) si $a \geq 1$, $\mathcal{L}_y(\underline{d}-\underline{c}-P_{i_1}^C-\ldots-P_{i_a+1}^C)$ a la dimension attendue $\chi((\underline{d}-\underline{c})-a-1 = \chi((\underline{d})-1$

Lorsque le nombre d'intersection du système résiduel et de la classe de la courbe est égal à 1 ($(\underline{d}-\underline{c}).\underline{c} = 1$), cela est toutefois possible. En effet $\mathcal{D}_y(\underline{d}-\underline{c}-P_{i_1}^C-\ldots-P_{i_a+1}^C)$ et C se coupent au moins deux fois dans le plan éclaté au points P_i : une de trop puisque $\underline{d}.\underline{c} = 1$: c'est donc que C est une composante fixe du système $\mathcal{L}_y(\underline{d}-\underline{c}-P_{i_1}^C-\ldots-P_{i_a+1}^C)$. On peut alors soustraire C sans changer la dimension réelle. Ainsi la quatrième hypothèse devient dans notre cas :

4) $\mathcal{L}_y(\underline{d}-\underline{c}-\underline{c})$ a la dimension attendue.

Le lemme d'Horace devient alors :

Lemme 4.2.2 *(Lemme d'Horace moins simple) Soient $\underline{d} = (d; m_1, \ldots, m_r) \in \mathbb{Z}^{r+1}$ et x, y deux points de X, l'ensemble des r-uplets de points distincts de $(\mathbb{P}^2)^r$, tels que x se spécialise en y. Soit C une courbe de $\mathbb{P}_y^2$ et $\widetilde{C}_y$ sa transformée dans le plan éclaté S_y. On suppose que $\widetilde{C}_y$ est géométriquement irréductible et lisse.*

Supposons que

1. $\underline{d}.\underline{c} + 1 - g(\underline{c}) = -a$ *où* $a > 0$,
2. *en position* y, *il y a* $g(\underline{c})$ *points libres sur* C, *affectés de multiplicité supérieure ou égale à 1,*
3. *il y a* $a+1$ *points libres sur* C, *affectés de multiplicité supérieures ou égales à 1 tels que :*
   en position x, *ces* $a+1$ *points sont libres dans le plan* $\mathbb{P}^2$
   en position y, *ils sont libres sur la courbe* C,
4. $(\underline{d} - \underline{c}).\underline{c} = 1$ *et* $\mathcal{L}_y(\underline{d} - \underline{c} - \underline{c})$ *a la dimension attendue.*

   alors $\mathcal{L}_y(\underline{d})$ *a la dimension attendue.*

La condition 3 demande que dans la nouvelle position il y ait au moins $a+1$ points de multiplicité supérieure ou égale à 1 qui soient spécialisés sur C et qui aient été libres auparavant. Pour compter de tels points, il suffit de faire la différence entre `(nb_specialised_curve (mults d) (new_position x l c) c)` et `(nb_specialised_curve (mults d) x c)`.

En Coq, cela donne :

Axiom Horace_moins_simple:
∀ x:Positions ,
∀ d:Pic ,
∀ c:curve ,
∀ l:to_specialise ,
length(d) = length(x)+1 ⇒
Le systeme $d - c_{[(\text{new\_position x l c})]} - c_{[(\text{new\_position x l c})]}$ est regulier en position (new_position x l c) ⇒
(well_to_specialize x mults(c) l) = true ⇒
(verif_specialise c x l) = true ⇒
(ajustement d $c_{[(\text{new\_position x l c})]}$) > 0 ⇒
(nb_specialised_curve mults(d) (new_position x l c) c) >= g($c_{[(\text{new\_position x l c})]}$) ⇒
(nb_specialised_curve mults(d) (new_position x l c) c) −
(nb_specialised_curve mults(d) x c) >= (ajustement d $c_{[(\text{new\_position x l c})]}$) + (1) ⇒
$d - c_{[(\text{new\_position x l c})]}.c_{[(\text{new\_position x l c})]}$ = (1) ⇒
$c_{[x]}$ est effective en position x ⇒
La courbe c est irreductible dans l'eclate en position x ⇒
La courbe c est lisse dans l'eclate en position x ⇒
Le systeme d est regulier en position x.

4.3 Tests et résultats

4.3.1 Exemple de session

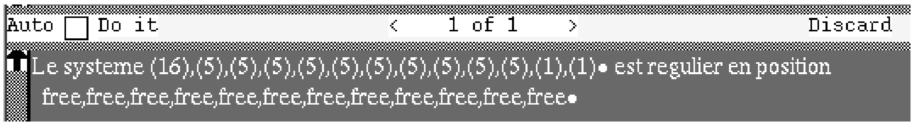
Nous allons reprendre la démonstration donnée en section 4.1.5 afin d'essayer de la valider sous Coq. Il s'agit donc d'étudier le système des courbes de degré 16 assignées à passer par 10 points génériques de multiplicité 5. Ceci s'ennonce ainsi sous Coq :

Lemma (Courbe30).
Le systeme (16),(5),(5),(5),(5),(5),(5),(5),(5),(5),(5)• est regulier en position free_list((10)).

On commence donc par ajouter deux points simple génériques (cela ne change rien à la régularité du système) avec la tactique `Add_free_point` :

1: `Simpl.` 1: `Add_free_point.` 1: `Add_free_point.`

Le but en cours devient donc (capture d'écran de CtCoq) :


Auto Do it < 1 of 1 > Discard
Le systeme (16),(5),(5),(5),(5),(5),(5),(5),(5),(5),(5),(1),(1)• est regulier en position
free,free,free,free,free,free,free,free,free,free,free,free•

On va maintenant successivement appliquer les lemmes `Horace_simple` et `Horace_moins_simple` en suivant la démonstration et donc en utilisant les différentes courbes et spécialisations, dans l'ordre. Voici le script Coq que nous allons lancer pas-à-pas :

1: `Horace1` [(3),(2),(1),(1),(1),(1),(1),(1)•] no,no,no,no,no,no,no,yes,yes,no,no,no•.
1: `Horace0` [(3),(0),(1),(1),(1),(1),(1),(1),(1),(1),(1)•] no,no,no,no,no,no,no,no,no,no,yes,no•.
1: `Horace0` [(1),(0),(0),(0),(0),(0),(0),(0),(1),(1)•] no,no,no,no,no,no,no,no,no,yes,no,no•.
1: `Horace0` [(1),(0),(0),(0),(0),(0),(1),(0),(0),(0),(1),(0),(0)•]
no,no,no,no,no,no,yes,no,no,no,no,no,no•.
1: `Horace0` [(1),(0),(0),(0),(0),(0),(0),(0),(1),(1),(0),(0),(0)•]
no,no,no,no,yes,no,no,no,no,no,no,no,no•.

Les tactiques `Horace0` et `Horace1` correspondent respectivement à l'utilisation des théorèmes `Horace_simple` et `Horace_moins_simple`, dont elles simplifient et résolvent les hypothèses qui se réduisent à du calcul. Commençons donc par l'utilisation du lemme d'Horace avec la cubique C ayant un noeud en P_1 et passant par $P_2, ..., P_7$, en y spécialisant P_8 et P_9. Puisque l'ajustement a vaut 1, on utilise `Horace1`. Après l'application de la tactique

```
Horace1 [(3),(2),(1),(1),(1),(1),(1),(1),(1)]
        no,no,no,no,no,no,no,yes,yes,no,no,no .
```

Coq génère 4 sous-buts. Le premier correspond au système résiduel :

```
Auto [ ] Do it                 <    1 of 4    >                    Discard
Le systeme (10),1,(3),(3),(3),(3),(3),(3),(3),(3),(5),1,1• est regulier en position
 free,
  free,
   free,
    free,
     free,
      free,
       free,
        specialised([(3),(2),(1),(1),(1),(1),(1),(1)•]),
         specialised([(3),(2),(1),(1),(1),(1),(1),(1)•]),free,free,free•
```

Le suivant demande de prouver que la cubique définie est bien effective dans cette position :

```
Auto [ ] Do it                 <    2 of 4    >                    Discard
(3),(2),(1),(1),(1),(1),(1),(1),O,O,O,O,O• est effective en position
 free,free,free,free,free,free,free,free,free,free,free,free•
```

Le troisième et le quatrième sous-but correspondent à la lissité et l'irréductibilité de la cubique :

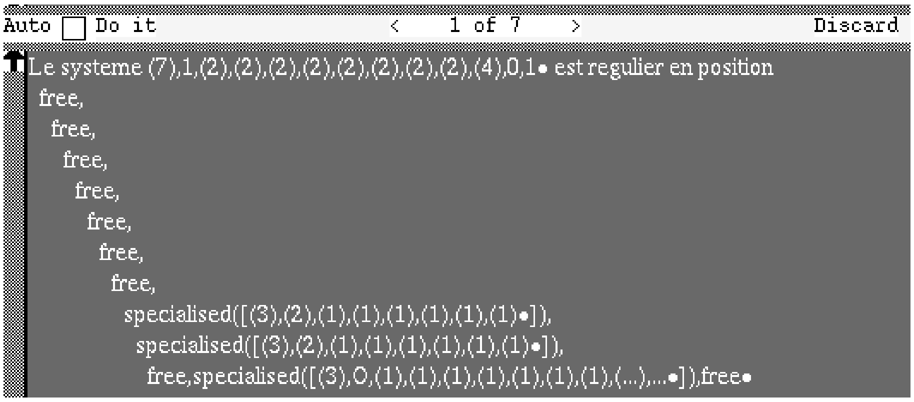
```
Auto [ ] Do it                 <    3 of 4    >                    Discard
La courbe [(3),(2),(1),(1),(1),(1),(1),(1)•] est irreductible dans l'eclate en  position
 free,free,free,free,free,free,free,free,free,free,free,free•
```

```
Auto [ ] Do it                 <    4 of 4    >                    Discard
La courbe [(3),(2),(1),(1),(1),(1),(1),(1)•] est lisse dans l'eclate en  position
 free,free,free,free,free,free,free,free,free,free,free,free•
```

Les autres hypothèses, conditions numériques ou conditions de validité des spécialisations, ont toutes étés résolues. On peut donc continuer l'étude de notre système, cette fois ci en considérant la cubique C_2 passant par $P_2, \ldots, P_{10}$ et en y spécialisant P_{11} (cette fois-ci l'ajustement est nul et on utilise `Horace0`).

```
Horace0 [(3),(0),(1),(1),(1),(1),(1),(1),(1),(1),(1),(1)]
        no,no,no,no,no,no,no,no,no,no,yes,no.
```

On a maintenant 7 sous-buts (régularité du nouveau système résiduel plus les conditions d'effectivité, irréductibilité et lissité de la nouvelle courbe), les autres hypothèses ont encore été résolues :



Continuons avec la droite (P_8, P_9), où on spécialise P_{10} :

```
Horace0 [(1),(0),(0),(0),(0),(0),(0),(0),(1),(1)]
        no,no,no,no,no,no,no,no,no,yes,no,no.
```

On a 10 sous-buts, tout va bien :

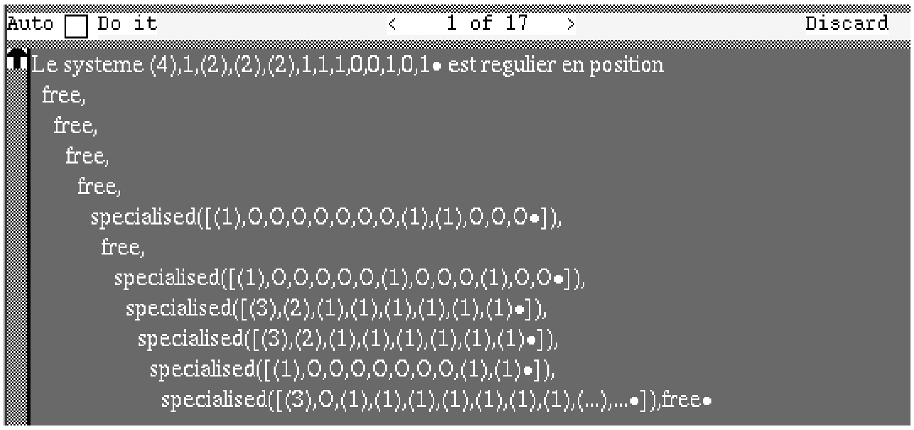
Et après avoir exploité la droite (P_6, P_{10}) en y spécialisant P_7 on obtient :

13 sous-buts, tout va encore pour le mieux.

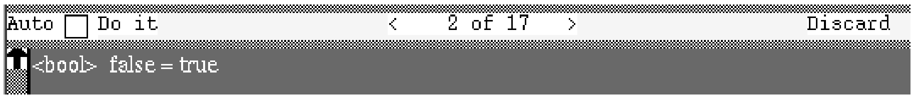
Dans l'application suivante du lemme d'Horace, on spécialise simplement P_5 sur la droite (P_8, P_9) :

```
Horace0 [(1),(0),(0),(0),(0),(0),(0),(0),(1),(1)]
        no,no,no,no,yes,no,no,no,no,no,no,no.
```

Et on obtient :



17 sous-buts : l'une des hypothèses habituellement résolue ne l'est pas cette-fois ci. Il s'agit de la condition qui vérifie la validité de la spécialisation, qui se réduit sur :



et qui indique donc que l'on a tenté d'appliquer le lemme d'Horace avec une spécialisation non valide. En effet, en vérifiant le problème *à la main* on se rend compte de l'erreur :

la droite (P_8, P_9) aurait 4 points d'intersection avec la conique C_2 : P_8, P_9, P_{10} et P_5, un de trop.

4.3.2 Résultats

On s'est servi de notre dévelopement pour vérifier les utilisations du lemme d'Horace faites dans [dS], en vérifiant pas-à-pas les courbes et spécialisations utilisées. Pour ce faire, on a du rajouter différentes notions que l'on n'a pas documentées dans ce document (ajout de points libres à un système, transformations de Cremona ...) mais qui sont disponibles dans l'archive Coq[2]. Les résultats, assez étonnants, montrent l'utilité d'une telle formalisation. En effet, 35% des systèmes étudiés contenaient une erreur, souvent due au positionement non valide des points. Nous n'avons pas pu vérifier tous les systèmes non plus : à peu près 10% des systèmes utilisaient par contre des notions mathématiques que nous n'avions pas formalisées.

4.3.3 Coq était-il bien nécessaire ?

On peut s'interroger sur la nécessité d'un logiciel de preuve pour le developpement que nous venons de décrire. En effet, une fois le travail de spécification effectué, ne pouvait-on pas réaliser tout ceci sous Maple ou directement en caml ?

La réponse est qu'en grande partie, oui, on pourrait très bien simuler le comportement général de cette librairie Coq en la programmant en maple ou caml. On peut cependant relever plusieurs points qui rendent son interet à sa spécification dans un logiciel de preuves :

a) L'espoir reste ouvert quand à la possibilité d'instancier les axiomes de la librairie.

b) Toujours le même aspect (mais avouons que c'est chipoter) : "Jusqu'où fait-on confiance à Maple pour effectuer des calculs et des manipulations de listes ?"

c) Tout n'est pas décrit dans ce chapître et l'aspect preuve n'est pas totalement absent de la librairie. En effet, un certain nombre de théorèmes ont été démontrés qui "controlent" les fonctions qui réalisent des calculs dans le lemme d'Horace (par exemple les fonctions doivent bien rendre des listes de la bonne taille etc..). Mais surtout le lemme d'Horace tel qu'implenté laisse des obligations de preuves qui ne se réduisent pas au calcul (par exemple "telle courbe est irréductible", "lisse" etc). Ces obligations peuvent se démontrer en utilisant d'autres théorèmes généraux (toujours donnés sous forme d'axiomes), et on peut toujours penser à une extension de la librairie dans ce sens.

Un autre aspect interessant est le fait que la construction interactive de preuves dans

[2]Disponible ici : `http ://www-sop.inria.fr/lemme/personnel/Laurent.Chicli/`

Coq permet de se rapprocher de l'utilisation mathématique du lemme d'Horace : on procède exactement de la meme manière (et dans le meme ordre) pas à pas en construisant une figure géométrique (sur le dessin en mathématiques et dans le contexte en Coq) et en réduisant ainsi le système étudié, pour enfin arriver à un cas déjà connu.

4.4 Conclusion

On a donc essayé dans ce chapitre de donner une spécification du lemme d'Horace. Au final, on dispose d'une librairie qui permet d'utiliser ce théorème de manière interactive et qui pourrait, pourquoi pas, servir à l'enseignement de celui-ci : en effet on peut facilement faire des erreurs en l'appliquant, et celles-ci sont difficilement décelables "à mains nues". En vérifiant un mémoire portant sur l'étude des courbes de genre 5, on a pu déceler grâce à ce developement un grand nombre d'erreurs.

Deuxième partie

Les quotients dans Coq

Chapitre 5

Formaliser les ensembles quotients dans Coq

On s'intéresse ici aux différentes manières d'aborder les ensembles quotients dans Coq. Plusieurs approches existent : les sétoïdes tout d'abord, les types normalisés, les types congruences intentionnels et enfin les types quotient. On s'attardera sur ces derniers parce qu'ils sont les seuls à la fois à permettre des substitutions lors des raisonnements (l'égalité considérée est celle de Leibniz) et à englober tous les quotients mathématiques.

L'utilisation de ces types quotients dans Coq, lorsque l'on les rajoute comme axiomes dans le système, n'est cependant pas toujours aisée ni très proche de la pratique mathématique. On décrit donc une série de commandes et de tactiques que l'on a implanté dans Coq afin d'en rendre l'usage plus évident et intuitif.

Malheureusement ces mêmes types quotients échouent à rendre compte des propriétés d'un quotient fonctionnel particulier que nous décrirons à la section 7.3 et qui sera le sujet d'étude du chapitre 8.

5.1 Différentes approches

5.1.1 Les sétoïdes

La première approche, sans doute la plus élégante conceptuellement parce qu'elle englobe tous les ensembles et traite uniformément les ensembles 'normaux' et les ensembles quotients, est celle des sétoïdes. Un sétoïde est la donnée d'un type support (le type des éléments de l'ensemble) et d'une relation d'équivalence qui est considérée comme l'égalité de l'ensemble. Un quotient n'est donc qu'un sétoïde particulier.

Malheureusement les avantages des sétoïdes sont aussi leurs incovénients. Le fait que ensembles et ensembles quotients soient traités uniformément implique d'avoir à démontrer pour un ensemble qui n'est pas un ensemble quotient, que chaque fonction, chaque prédicat définis dessus soient compatibles avec l'égalité. Le fait aussi que l'égalité considérée ne soit pas celle de Leibniz interdit la substitution lors des raisonnements et la remplace par l'utilisation successive de lemmes de compatibilité. Rien de bien difficile certes, mais par contre cela engendre un travail très fastidieux et ce qui était élégant conceptuellemnt ne l'est plus du tout dans la pratique.

5.1.2 Les types normalisés

Une autre approche est celle de P.Courtieu qui définit dans sa thèse ([Cou01]) la notion de quotient calculatoire. Un quotient E/R est dit calculatoire s'il existe une fonction nf (dite " de normalisation de E/R ") de E dans E telle que

$$\forall x, y \in E,\ x\ R\ y \Leftrightarrow nf(x) = nf(y).$$

Une telle fonction nf calcule donc un représentant particulier parmi tous les représentant d'une classe d'équivalence et établit une bijection entre E/R et $nf(E)$. P.Courtieu étend le CCI avec des types qu'il nomme *types normalisés* afin de représenter ces quotients et montre que les propriétés importante du CCI (normalisation forte, décidabilité du typage, comportement calculatoire) sont conservées.

Malheureusement cette extension du CCI est limitée en ce qu'elle ne considère qu'une classe restreinte des quotients : les quotients calculatoires, et ne permet pas de considérer la notion mathématique de fibre de faisceaux que nous avons formalisé (cf. section 4.5.4.1) et qui donne justement un exemple de quotient non calculatoire.

Rappellons brièvement la définition d'une fibre de faisceau (un préfaisceau F envoie en particulier un ouvert U sur un anneau $F(U)$) : la fibre F_p de F en un point p est l'ensemble des couples formés d'un ouvert U contenant p et d'un élément s de $F(U)$, quotienté par la relation

$$(U, s \in F(U)) \sim (V, s' \in F(V)) \Leftrightarrow (\exists W |\ p \in W \wedge (W \subset U \cap V) \wedge s_{|W} = s'_{|W}).$$

Si l'on veut définir une fonction de normalisation de ce quotient, il faut trouver un représentant canonique de la classe de $(U, s \in F(U))$, ce qui est impossible : une idée serait de considérer le plus petit ouvert W contenant p et définir ce représentant canonique comme $(W, s_{|W})$ (bien en relation avec $(U, s \in F(U))$), mais un tel ouvert n'existe pas.

Ainsi, s'ils apportent une solution puissante pour la formation des quotients, les types normalisés ne sont pas toujours adaptés à la formalisation des mathématiques.

5.1.3 Types congruences intentionnels

Une autre solution est apportée par S.Boutin dans sa thèse [Bou97] avec la notion de *types congruences intentionnels*. Implantés dans Coq, leur utilisation est des plus élégantes et ressemble fortement à la pratique mathématique de définition d'algèbres libres quotientés par des équations. Par exemple, pour définir $\mathbb{Z}$ dans le prototype de Coq (dévellopé lui aussi par Samuel Boutin) intégrant les types congruences intentionnels on écrit simplement :

```
Congruence Z : Set :=
  inZ : nat -> nat -> Z
With Equations
  Zcongruence:
  (a1,a2,t:nat) (inZ a1 a2) = (inZ (plus a1 t) (plus a2 t)).
```

Pour définir une fonction dont la source est un quotient, là aussi la pratique mathématique est de mise : on définit d'abord la fonction, puis le système demande de valider la définition en montrant que la fonction est compatible avec les égalités de congruence du quotient.

Malheureusement là aussi la classe des quotients définissables est limitée, en particulier la relation d'équivalence ne doit être donnée que sous forme d'équations quantifiées universellement. Pour reprendre notre exemple des fibres de faisceaux, la relation

$$(\exists W |\ p \in W \wedge (W \subset U \cap V) \wedge s_{|W} = s'_{|W})$$

ne rentre évidemment pas dans ce cas.

5.1.4 Types quotients dans Coq

A part les sétoïdes qui permettent d'appréhender les quotients dans leur globalité (mais qui sont à la fois trop pénibles à utiliser et finalement peu élégants), les autres notions de quotients (les types congruence intentionnels et les types normalisés) ne permettent donc pas de formaliser tous les quotients mathématiques. On bute en particulier sur l'un des objets que nous avons formalisé au chapître 4 : les fibres de faisceaux.

On s'intéresse donc à une notion beaucoup plus catégorique : les types quotients, qui nous permettront de formaliser cet exemple problèmatique.

5.1.4.1 Un système de type quotient minimal

Les types quotients que nous considérons ici ont été étudiés par M.Hofmann dans [Hof95a, Hof95b], G.Barthe dans [Bar95] et S.Boutin dans [Bou97]. Ils sont une formalisation quasi-directe de la pratique mathématique des quotients, à laquelle on a cependant retranché la notion de choix de représentant (qui serait comme on le verra contradictoire). Ils consistent en la donnée, pour un type E et une relation d'équivalence R :

- d'un type quo représentant le type quotient,
- d'une fonction class : E -> quo pour le passage au quotient,
- de deux propositions caractérisant l'égalité des classes,
- d'un moyen constructif de descendre dans le quotient les fonctions compatibles avec R.
- Enfin, puisqu'on n'a pas de fonction de choix de représentant d'une classe, on se donne une propriété énonçant l'existence (dans Prop) de tels représentants.

Plus précisément, dans Coq, on formalise la donnée de ces types quotients de la manière suivante :

```
Record type_quotient [E : Type; R : (Relation E);
                      p : (Equivalence 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 R f))
     (x:E)((comp (quo_lift F f H) class) x)== (f x);

   quo_surj: (c:quo)(ExT x:E | c==(class x))}.
```

Et ajouter ces types quotients à Coq revient à se donner l'axiome quotient donnant un type quotient pour tout type E et relation d'équivalence R :

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

5.2 Utilisation des types quotients dans Coq.

L'utilisation de ces types quotients dans Coq est cependant finalement assez peu conviviale, elle a plusieurs inconvénients :

- la définition du quotient et des fonctions dont le type source est un quotient prend plusieurs étapes qu'on aimerait voir simplifiées,
- l'utilisation des propriétés des quotients nécessite le rappel d'arguments non implicites,
- enfin, les termes de preuves apparaissent dans le corps des objets construits (`quo_lift` et `quo_lift_prop` prennent des preuves en arguments) ce qui les rend souvent illisibles.

On a donc créé une librairie constitués de théories, commandes et tactiques pour permettre à la fois un travail plus facile avec les quotients et plus proche de la pratique mathématique[1]. Pratiquement, l'utilisation de cette librairie ressemble beaucoup à l'utilisation des types congruences intentionnels de S. Boutin.

5.2.1 Description de la librairie

5.2.1.1 QDefine

La première de ces commandes, `QDefine`, permet de définir le type quotient à partir d'une liste d'équations et ainsi simuler la création de types inductifs avec des équations liant les constructeurs. Le but de cette commande est de simplifier la création du quotient et de nommer les propriétés génériques des quotients, spécialisées au quotient créée.

La commande demande les arguments suivants :

QDefine `type_de_base quotient eq [liste d'équations]` .

où :
- `type_de_base` est le type que l'on veut passer au quotient (ce peut bien sur être un type inductif),
- `quotient` est le nom du quotient à créer,
- `eq` le nom de l'égalité utilisée dans la liste suivante,
- `[liste d'équations]` les équations qui vont définir l'égalité dans le quotient.

Le principe de fonctionnement de `QDefine` est le suivant. Elle commence par définir la clôture reflexive symétrique transitive de la relation binaire engendrée par les équations fournies, en définissant le type inductif suivant :

```
Inductive eq:type_de_base->type_de_base->Prop:=
  eq_refl:(x:type_de_base)(eq x x)
```

[1]Travail réalisé en commun avec Loïc Pottier

```
| eq_sym:(x,y:type_de_base)(eq x y) -> (eq y x)
| eq_trans:(x,y,z:type_de_base)(eq x y)->(eq y z) -> (eq x z)
| eq_1:eq1
| eq_2:eq2
| ...
| eq_n:eqn.
```

où eq1, ...,eqn sont les équations données dans `[liste d'equations]`.

Elle prouve ensuite le lemme `(eq_equivalence)` énonçant que la relation `eq` ainsi définie est une relation d'équivalence (c'est immédiat puisqu'on a considéré une clôture reflexive symétrique transitive), puis contruit le quotient par :

```
Definition quotient := (quotient eq_equivalence).
```

Enfin, `QDefine` génère et démontre les lemmes qu'elle nomme `quotient_eqi` dont les énoncés correspondent aux équations données, mais "plongées" dans le quotient.

Exemple de session Coq : la définition de Z/pZ. Après avoir lancé Coq et chargé la librairie Qtools :

```
Welcome to Coq 7.3 (May 2002)

Coq < Require Qtools.
[Loading ML file qtools.cmo ...done]
```

On peut définir le quotient $\mathbb{Z}/p\mathbb{Z}$ par la relation classique

$$z_1 R z_2 \Leftrightarrow \exists k \in \mathbb{Z} | z_1 = z_2 + k * p$$

avec la commande :

```
Coq < QDefinition Zp :=quotient Z with eqZp verifying
        (z1,z2:Z)(EXT k:Z | z1 =‘ z2 + k*p‘)->(eqZp z1 z2).

The quotient Zp will be defined.
eqZp is defined
eqZp_ind is defined
Warning: the hint: EApply eqZp_trans will only be used by EAuto
eqZp_equivalence is defined
Zp is defined
```

```
class_Zp is defined
Zp_comp is defined
Zpeq1 :
(z1,z2:Z)(EXT k:Z | `z1 = z2+k*p`)->(class_Zp z1)==(class_Zp z2)
Zpeq1 is defined
```

L'égalité `eqZp`, puis `Zp` sont définis :

```
Coq < Print eqZp.

Inductive eqZp  : Z->Z->Prop :=
      eqZp_refl : (x:Z)(eqZp x x)
    | eqZp_sym : (x,y:Z)(eqZp x y)->(eqZp y x)
    | eqZp_trans : (x,y,z:Z)(eqZp x y)->(eqZp y z)->(eqZp x z)
    | eqZp_1 : (z1,z2:Z)(EXT k:Z | `z1 = z2+k*p`)->(eqZp z1 z2)

Coq < Print Zp.

Zp = (quotient Z eqZp eqZp_equivalence)
     : (type_quotient Z eqZp eqZp_equivalence)
```

Puis `class_Zp` (la fonction `class` spécialisée à notre quotient) et `Zpeq1` (l'équation donné, plongée dans le quotient) :

```
Coq < Print class_Zp.

class_Zp = (class Z eqZp eqZp_equivalence Zp)
           : Z->Zp

Coq < Check Zpeq1.

Zpeq1:
(z1,z2:Z)(EXT k:Z | `z1 = z2+k*p`)->(class_Zp z1)==(class_Zp z2)
```

5.2.1.2 QLift et QDefinedLift

La commande `Qlift` permet à l'utilisateur de définir une fonction d'arité 1 ou 2 dont les espaces source et arrivé peuvent être des quotients, à partir d'une fonction définie sur les représentants des classes. Sur les espaces sources la fonction donnée doit évidemment être compatible avec la relation d'équivalence : `Qlift` tente de le démontrer et si elle échoue laisse le soin à l'utilisateur de le faire.

La commande demande les arguments suivants :

QLift nom : type := fonction.

où :

- `nom` est le nom de la fonction que l'on veut définir.
- `type` est le type de la fonction que l'on veut définir. Ce peut être une fonction d'arité 1 ou 2, les ensembles sources et but peuvent être des quotients.
- `fonction` est la fonction que l'on souhaite passer au quotient.

Exemple de session : la multiplication par 3 dans Zp

```
Coq < QDefinition triple_Zp : Zp -> Zp := [z:Z](Zmult `3` z).

You must prove the following goals to validate your definition.
Don't forget to save it with the 'QDefinedLift' command.

1 subgoal

  x : Z
  x' : Z
  H : (eqZp x x')
  ============================
   (z1,z2:Z)(EXT k:Z | `z1 = z2+k*p`)->(eqZp `3*z1` `3*z2`)
```

Pour valider notre définition, Coq nous demande de prouver que la fonction donnée est bien compatible avec la relation d'équivalence qui définit Zp.

```
triple_Zp < Intros; Apply eqZp_1.

1 subgoal

  x : Z
  x' : Z
  H : (eqZp x x')
  z1 : Z
  z2 : Z
  H0 : (EXT k:Z | `z1 = z2+k*p`)
```

```
  ============================
   (EXT k:Z | `3*z1 = 3*z2+k*p`)

triple_Zp < Elim H0;Intros k H1;Exists `3*k`;Ring;Ring;Omega.

Subtree proved!
```

Si on sauvegarde notre définition par un `Defined` habituel, le terme `triple_Zp` construit fera plusieurs dizaines de lignes de code (et l'utilisation dans notre exemple des tactiques `Ring` et `Omega`, très verbeuses, n'arrangent rien) :

```
 triple_Zp1 =
(quo_lift2 eqZp_equivalence
  [x,x':Z; H:(eqZp x x')]
   (eqZp_ind [z,z0:Z](eqZp `3*z` `3*z0`) [x0:Z](eqZp_refl `3*x0`)
     [x0,y:Z; _:(eqZp x0 y); H1:(eqZp `3*x0` `3*y`)]
      (eqZp_sym `3*x0` `3*y` H1)
     [x0,y,z:Z;
      _:(eqZp x0 y);

... 3 pages de terme de preuve ...

     (Pmult Z (Pconst Z `3`) (Pvar Z (Right_idx End_idx)))
                  (Pmult Z
                    (Pmult Z (Pconst Z `3`)
                      (Pvar Z (Left_idx End_idx))) (Pvar Z End_idx)))))
          H0)) x x' H))
     : Zp->Zp
```

En effet `triple_Zp` est défini par `Qlift` avec la fonction `quo_lift` des axiomes quotients, qui demande en argument la preuve de compatibilité de la fonction donnée. Ceci génère des termes énormes qui ont l'inconvénient à la fois d'être difficiles à gérer par `Coq` (il doit retyper ces termes de preuves à chaque fois qu'il les rencontre), mais surtout d'être illisibles.

Pour remédier à cela la commande `QDefinedLift` fait le même travail que `Defined` en sauvegardant le terme ainsi construit dans le contexte, mais avant cela, elle le simplifie de la manière suivante. Elle parcourt récursivement tous les sous termes de l'objet à sauvegarder et, dès qu'elle rencontre une preuve (donc un sous-terme de sorte `Prop`), elle la sauvegarde sous un autre nom dans le contexte pour la remplacer dans l'objet par son nom.

`QDefinedLift` prouve aussi le lemme de réécriture de la fonction nouvellement définie (i.e. le lemme énonçant $(f(class\ x)) == (f_1\ x)$ si f est la fonction définie à partir de f_1).

Si on reprend notre exemple précédent :

```
triple_Zp < Elim H0;Intros k H1;Exists `3*k`;Ring;Ring;Omega.

Subtree proved!

triple_Zp < QDefinedLift.

triple_Zp is defined.
triple_Zp_rewrite_rule is defined
```

Le terme de preuve contenu auparavant dans `triple_Zp` à été sauvegardé sous le nom `triple_Zp_subproof` et cette fois-ci `triple_Zp` est bien plus lisible :

```
Coq < Print triple_Zp.
triple_Zp =
(quo_lift2 eqZp_equivalence triple_Zp_subproof)
     : Zp->Zp
```

Le lemme de compatibilité a lui bien été généré :

```
Coq < Check triple_Zp_rewrite_rule.

triple_Zp_rewrite_rule
     : (x:Z)(triple_Zp (class_Zp x))==(class_Zp ([z:Z]`3*z` x))
```

5.2.1.3 QSimpl et QElim

QSimpl La tactique `QSimpl` utilise les lemmes de reécriture que génère QDefinedLift et simplifie toutes les occurrences de $(f(class\ x))$ où f est une fonction définie par `Qlift`.

QElim L'axiome `quo_surj` des types quotients nous assure qu'un prédicat sur un type quotient est vrai pour tout élément de ce type quotient s'il est vrai pour tout élément de la forme `(class x)`. La tactique `QElim` sert à prouver une propriété dépendant d'un élément `x` d'un type quotient en le remplaçant par un élément `(class x0)`.

```
Coq < Lemma injective : (x,y:Zp)(triple_Zp x)==(triple_Zp y) -> (x == y).

injective < Intros x y; QElim x; QElim y.

1 subgoal

  x : (quotient Z eqZp eqZp_equivalence)
  y : (quotient Z eqZp eqZp_equivalence)
  x0 : Z
  x1 : Z
  ============================
   (triple_Zp (class_Zp x0))==(triple_Zp (class_Zp x1))
   ->(class_Zp x0)==(class_Zp x1)
```

`QElim x` puis `QElim y` ont bien remplacé `x` et `y` par respectivement `(class_Zp x0)` et `(class_Zp x1)`. On a maintenant dans notre but des termes `(triple_Zp (class_Zp y))` et puisque `triple_Zp` à été obtenue en 'descendant' dans le quotient la fonction `[x :Z](Zmult ‘3‘ z)`, `QSimpl` va pouvoir les simplifier :

```
injective < QSimpl.

 ============================
   (class_Zp ‘3*x0‘)==(class_Zp ‘3*x1‘)->(class_Zp x0)==(class_Zp x1)
```

La preuve se termine ensuite simplement en passant de l'égalité des classes d'équivalences à des éléments en relation.

Même si ces commandes sont assez minimalistes (on pourrait imaginer des tactiques plus spécialisées, des moyens différents de définir les quotients, pas forcément par clotûre réflexive symétrique transitive ...), elles permettent de montrer qu'il peut être aisé de travailler avec les types quotients dans Coq.

5.3 Echec dans la formalisation d'un quotient fonctionnel simple

Comme le remarque M.Hofmann dans ([Hof95a], p.254-255), ces types quotients ne permettent cependant pas de rendre compte des propriétés de tous les quotients mathématiques. Il le prouve en donnant l'exemple suivant.

Soient deux ensembles E et F, R une relation d'équivalence sur F et π_R la fonction de passage au quotient de F vers $\frac{F}{R}$. On peut définir sur l'espace fonctionnel $E \to F$ la relation d'équivalence S suivante :

$$fSg \Leftrightarrow \forall x \in E \ f(x)Rg(x).$$

Par définition de S, l'application qui compose toute fonction $f : E \to F$ avec le passage au quotient π_R est compatible avec S, et il existe donc un morphisme naturel θ faisant commuter le diagramme suivant :

$$\begin{tikzcd} E \to F \arrow[r, "\pi_R \ \circ \ ."] \arrow[d, "\pi_S"] & E \to \dfrac{F}{R} \\ \dfrac{E \to F}{S} \arrow[ur, dashed, "\theta"] & \end{tikzcd}$$

En théorie des ensembles (où l'axiome du choix et l'extensionalité des fonctions sont donnés), ce morphisme est clairement injectif (si $\pi_R \ o \ f = \pi_R \ o \ g$ i.e. $\forall x \in E,\ [f(x)]_R = [g(x)]_R$ on a bien fSg) et surjectif (si $f : E \to \frac{F}{R}$, elle est l'image par θ de la fonction $\pi_S(g)$ où g envoie x sur un représentant de $f(x)$).

Dans le Calcul des Constructions Inductives, étendu par les types quotients, l'application θ peut être définie (à condition toutefois de se donner l'axiome d'extensionalité des fonctions) et montrée injective, mais pas surjective.

En effet, pour définir θ, il suffit en appliquant `quo_lift` de prouver la compatibilité de $\pi_R \ \circ \ .$ pour la relation S :

$$\begin{array}{lll} & f \ S \ g & \\ \forall x \in E & f(x) \ R \ g(x) & \text{par définition de S,} \\ \forall x \in E & \pi_R(f(x)) = \pi_R(g(x)) & \text{par application de quo\_comp,} \\ & \pi_R \circ f = \pi_R \circ g & \text{par l'extensionalité des fonctions.} \end{array}$$

L'injectivité est elle obtenue en remontant le raisonnement précédent :

$$\begin{array}{lll} & \pi_R \circ f = \pi_R \circ g & \\ \forall x \in E & \pi_R(f(x)) = \pi_R(g(x)) & \text{par propriété des fonctions} \\ \forall x \in E & f(x) \ R \ g(x), & \text{par application de quo\_comp\_rev,} \\ & f \ S \ g, & \text{par définition de S,} \end{array}$$

Mais la surjectivité pose problème. En effet, l'unique antécédent (en théorie des ensemble) d'une fonction $f : E \to \frac{F}{R}$ est la fonction $\pi_S(g)$ où g envoie x sur un représentant de $f(x)$. Pour prouver l'existence d'une telle fonction, il faudrait en donner un témoin, et donc la construire (dans un contexte propositionel), ce qui est impossible puisqu'on ne dispose pas de fonction de choix de représentant.

Il est impossible aussi de se servir de la proposition `quo_surj` assurant l'existence de représentants : passer de la proposition

$$\forall x : E,\ \exists c : F \mid (class\ c) = (f\ x)$$

à l'existence de la fonction g :

$$\exists g : E \to F \mid (class\ (g\ x)) = (f\ x)$$

revient à utiliser l' axiome du choix à valeur dans `Prop` :

```
Axiom AC :
   (A,B : Type)(R:A->B->Prop)
   ((x : A)  (exT B [y : B]  (R x y))) ->
   (exT A ->  B [f : A ->  B] (x : A)  (R x (f x))).
```

que nous n'avons pas dans notre contexte.

Pour résoudre ce problème, M.Hofmann ([Hof95a], p.255) suggère (dans le cadre du Calcul des Constructions) d'étendre le système de type quotient en rajoutant comme propriété la surjectivité de l'application θ.

5.4 Conclusion

On dispose d'une notion de quotient dans Coq à la fois très catégorique et d'une relative facilité d'utilisation grâce aux différentes commandes et tactiques que nous avons développées. Elles permettent de formaliser le quotient sur lequel on butait jusqu'à présent : les fibres des faisceaux.

Leur expressivité cependant ne semble que partielle aussi : si on peut formaliser tous les quotients mathématiques, ils n'ont pas forcément les mêmes propriétés : on en a vu un exemple, déja cité par Martin Hofmann dans sa thèse [Hof95a].

La question se pose donc naturellement : est-il possible d'étendre ces types quotients dans Coq (soit par l'axiome du choix vu plus haut soit par une quelquonque propriété) afin de rendre compte de l'exemple problématique ? C'est ce que propose M.Hofmann dans le cadre du Calcul des Constructions. C'est le sujet d'étude du chapitre suivant.

Chapitre 6

Limite des types quotients dans Coq

6.1 Introduction

On montre[1] dans ce chapitre qu'il ne peut y avoir de notion catégorique de type quotient dans le *Calcul des Constructions Inductives* qui soit aussi expressive (en englobant la classe des quotients et leurs propriétés) que les ensembles quotients en mathématique classique.

Pour cela, nous allons reprendre le système de type quotient minimal que nous avons vu au chapitre précédent, et montrer qu'il est impossible de l'étendre de manière à pouvoir démontrer la surjectivité du morphisme θ vu dans la section 5.3, surjectivité pourtant effective en mathématique classique.

Rapellons brièvement l'exemple qui pose problème. Soient E et F deux ensembles, R une relation d'équivalence sur F et π_R la fonction de passage au quotient de F vers $\frac{F}{R}$. On peut définir sur l'espace fonctionnel $E \to F$ la relation d'équivalence S suivante :

$$fSg \Leftrightarrow \forall x \in E \ f(x)Rg(x).$$

Par définition même de la relation S, l'application qui compose toute fonction $f : E \to F$ avec le passage au quotient π_R est compatible avec S, et il existe donc un morphisme naturel θ faisant commuter le diagramme suivant :

$$\begin{tikzcd}
E \to F \arrow[rr, "\pi_R \ o \ ."] \arrow[d, "\pi_S"] & & E \to \dfrac{F}{R} \\
\dfrac{E \to F}{S} \arrow[urr, dashed, "\theta"] & &
\end{tikzcd}$$

[1]Les résultats exposés dans ce chapitre proviennent d'un travail commun avec Carlos Simpson et Loïc Pottier.

En théorie des ensembles, où l'axiome du choix et l'extensionalité des fonctions sont donnés, ce morphisme est un isomorphisme. Dans le Calcul des Constructions Inductives implanté dans *Coq* et étendu par les types quotients de M.Hofmann [Hof95a] l'application θ peut être définie et montrée injective, mais pas surjective.

Le but de ce chapitre est donc de montrer que l'on ne peut pas étendre ces types quotients de manière à rendre compte de cette surjectivité : les deux types `E->F/R` et `(E->F)/S` ne sont pas et ne peuvent être rendus isomorphes dans le CCI de Coq, où l'imprédicativité et l'élimination forte coexistent pour la sorte `Set`.

Le plan adopté dans ce chapitre est le suivant. Le problème est formalisé dans Coq dans la deuxième partie, puis on explique la différence dans Coq entre la proposition d'*existence* d'un objet et la donnée *constructive* de celui-ci. Dans la quatrième partie on trouve un équivalent catégorique à notre problème : l' *existence* en général d'une section au passage au quotient (π_R).

Dans la partie suivante, on rappelle un résultat de [Pot01] : la donnée *constructive* d'une section au passage au quotient est contradictoire. Ce sera l'occasion de décrire quelques résultats connus sur les quotients dans le CCI : l'adaptation du résultat de Diaconescu [Dia75] par Werner/Lacas [LW99] et les résultats obtenus par Maria Emilia Maietti [Mai99].

Mais notre problème est équivalent à *l'existence* des sections au passage au quotient et nous montrons l'incohérence de cette hypothèse dans la sixième partie. La première étape consiste à montrer l'existence d'une fonction `Proptobool` de `Prop` vers `bool` envoyant les propositions vraies sur `true` et les propositions fausses sur `false`. La deuxième étape se déroule ainsi : dans un contexte où `False` (de type `Prop`) est le but à prouver, nous pouvons utiliser le principe d'élimination sur l'existence de `Proptobool`, et ainsi nous en servir pour construire un tiers-exclu dans Set, qui lui est contradictoire ([Pot01], [Geu01]).

Les preuves données dans ce chapître ont toutes été formalisées et validées dans Coq, on donne le script correspondant aux définitions et aux énoncés des lemmes démontrés au fur et à mesure de leur description[2].

6.2 Formalisation du problème dans Coq

Reprenons donc notre formalisation des types quotients :

```
Record type_quotient [E : Type; R : (Relation E);
                      p : (Equivalence R)] : Type := {
```

[2]On trouvera le script complet incluant les preuves à l'adresse suivante : `http ://www-sop.inria.fr/lemme/personnel/Laurent.Chicli/types2002.zip`.

```
   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)((comp (quo_lift F f H) class) x)== (f x);
   quo_surj: (c:quo)(ExT x:E | c==(class x))}.

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

Muni des types quotients, on peut définir la relation d'équivalence S et le morphisme θ dans Coq (l'extentionalité des fonctions est nécessaire pour cela mais est consistante avec les types quotients [Hof95a]) :

```
Variable E,F:Type.
Variable R:(Relation E); p:(Equivalence R).

Definition S:=[f,g:F->E](x:F)(R (f x) (g x)).

Lemma Sequiv: (Equivalence S).

Axiom Extensionality :
   (F,G:Type)(f,g:F->G)((x:F)(f x) == (g x))->f == g.
```

On prouve la compatibilité de $\pi_R \circ .$ avec S, puis on définit θ :

```
Lemma ps:(compatible S [f:F->E](comp (class (quotient p)) f)).

Definition theta:=(!quo_lift ? ? ? (quotient Sequiv) ? ? ps).
```

Comme on l'a vu dans la section 5.3, l'injectivité de θ ne pose pas problème :

```
Lemma theta_injective:
      (f,g:(quotient Sequiv))(theta f)==(theta g)->f==g.
```

6.3 Prop et Set.

A la base de la hiérarchie d'univers de Coq , deux sortes particulières : Prop et Set. La première, Prop est destinée à représenter le type des propositions logiques : une proposition dans Coq est un type P de type `Prop`. La seconde, `Set`, est elle destinée (de part ces propriétés) aux objets du discours : l'ensemble des entiers naturels, les listes, les programmes sont généralement construit dans `Set`.

`Set` étant destiné à représenter des données, et puisqu'on veut celles-ci non dégénérées (i.e. on veut pouvoir démontrer que $0 \neq 1$), on doit, d'un point de vue technique, autoriser l'élimination de `Set` vers `Type`. L'élimination de `Set` vers `Set` et de `Set` vers `Prop` sont alors déduites par *cumulativité des univers.*

D'autre part l'univers `Prop` est lui fait pour être le type des propositions, sur lesquelles on veut pouvoir raisonner de manière intuitionniste ou classique. Or le mathématicien "classique" veut pouvoir ne pas distinguer les preuves d'une même proposition, d'une part parce que certains axiomes classiques implique l'égalité des preuves de proposition, d'autre par parce qu' un élément d'une partie d'un ensemble ne doit pas dépendre de la preuve qu'il appartient à cette partie. Pour qu'il ne soit pas possible de démontrer que deux preuves sont différentes, on doit cette fois-ci interdire l'élimination de `Prop` vers `Set` ou `Type`. En effet sans cela on pourrait démontrer que les deux preuves de `True` $\vee$ `True` sont distinctes en éliminant `True` $\vee$ `True` vers les booléens.

L'élimination (ou le raisonnement par cas) d'un terme de sorte `Prop` donc est autorisée dans Coq pour construire un terme de sorte `Prop`, mais pas pour construire un terme de sorte `Set` ou `Type`.

Ainsi, l'existence d'un objet x vérifiant une propriété P étant ainsi définie dans `Prop` :

```
Inductive exT [A : Type; P : A->Prop]  : Prop :=
      exT_intro : (x:A)(P x)->(ExT P)
```

on pourra faire une élimination sur l'existence de x (et donc disposer de x dans notre contexte) pour construire un terme de sorte `Prop`(la preuve d'une proposition quelquonque) mais pas pour construire un terme de sorte `Set`(et donc un objet proprement dit). En ce sens affirmer l'existence d'un objet est donc plus faible que de se le donner dans `Set` : dans le premier cas on ne pourra s'en servir que pour démontrer des propositions, dans le second cas on pourra aller plus loin et construire de nouveaux objets avec.

Dans la suite de ce chapitre la différence entre les deux notions est fondamentale et chaque référence à l'existence d'un objet désignera bien uniquement la proposition affirmant l'existence de celui-ci.

6.4 L'existence d'une section au passage au quotient

Revenons à notre problème initial. En fait, pour F et R donnés, la surjectivité des morphismes θ quelque soit E est équivalente à l'existence d'une section au passage au quotient π_R, c'est-à-dire une fonction s telle que

$$\exists s : \frac{F}{R} \to F \mid \pi_R o s = Id_{\frac{F}{R}}.$$

En effet, si θ est surjective dans le cas où $E = \frac{F}{R}$, la proposition ci-dessus se déduit de

$$\exists s : \frac{F}{R} \to F \mid \pi_R o s = \epsilon(g) \text{ ou } g : \frac{\frac{F}{R} \to F}{S}.$$

En appliquant ensuite le principe d'induction `quo_ind` on est ramené au but

$$\forall h : \frac{F}{R} \to F,\ \exists s : \frac{F}{R} \to F \mid \pi_R o s = \theta(\pi_S(h))\ ,$$

ce qui trivialement vrai puisque $\theta(\pi_S(h)) = \pi_R o h$ selon la propriété `quo_lift_prop`.

Inversement, s'il existe une fonction $s : \frac{F}{R} \to F$ telle que $\pi_R o s = Id_{\frac{F}{R}}$, alors quelque soit E l'application θ correspondante est surjective : pour tout $f : E \to \frac{F}{R}$ on a :

$$\theta(\pi_S(s o f)) = \pi_R o(s o f) = (\pi_R o s) o f = f.$$

Dans la suite, seul le sens direct nous interesse, et dans Coq donc, en se donnant pour hypothèse la surjectivité de θ :

```
Hypothesis theta_surj : (f : F ->  (quotient p))
  (!exT ? [g : (quotient Sequiv)](theta g) == f).
```

on montre :

```
Lemma step1: (!exT ? [s:(quo (quotient p))->E]
(comp (class (quotient p)) s) == (Id (quotient p))).
```

La surjectivité du morphisme θ est donc ramenée à la question de la possibilité ou non d'étendre les types quotients par l'existence d'une section au passage au quotient. Dans la prochaine partie, nous allons brièvement rapeller certains résultats de [Pot01] où une telle section est donnée constructivement et montrer l'incohérence de cette hypothèse. Ce n'est que dans la section 9.6 que nous traiterons du cas où seule l'*existence* de la section est donnée en hypothèse.

6.5 Une fonction de choix de représentant donnée constructivement est contradictoire.

On se place donc ici dans un cadre où en plus de nos types quotients une fonction de choix de représentant `s : quo -> E` est donnée, vérifiant :

$$E \underset{s}{\overset{c}{\rightleftarrows}} quo \qquad \forall x : quo,\ (c(s\ x)) == x.$$

(où c est la fonction `(class quo)`).

6.5.1 L'astuce de Diaconescu

B.Werner et S.Lacas ont montré dans [LW99] (en adaptant l'astuce de Diaconescu [Dia75]) que dans ce cas, étant donné une proposition P quelconque, on peut utiliser les quotients pour construire un terme de type `{P}+{~P}` décidant si P est vrai ou non. Ils procèdent de la manière suivante.

Soit $E = bool$ et la relation R définie par :

$$x\ R\ y \Leftrightarrow x = y \vee P,$$

il s'agit bien d'une relation d'équivalence, et si on considère le quotient E/R associé (il est important de noter qu'il dépend de P), on a :

$$\begin{array}{rcl} (s(c\ true)) =_{bool} (s(c\ false)) & \Leftrightarrow & (c\ true) == (c\ false) \\ & \Leftrightarrow & P \end{array}$$

Et l'égalité dans $bool$ étant décidable, on obtient ainsi la décidabilité de P.

6.5.2 Incohérence du tiers-exclu dans `Set`

En généralisant la construction précédente pour toute proposition P, on construit un terme de type `(P :Prop){P}+{~P}`, c'est-à-dire le tiers-exclu fort, dans la sorte `Set`.

Pour conclure sur l'incohérence, on utilise justement le fait que le tiers-exclu dans la sorte `Set` est contradictoire dans `Coq`.

Le lecteur interressé pourra se reporter à [Pot01][3] pour une démonstration de cette incohérence. Celle-ci s'inspire du résultat démontré par Barbanera-Berrardi dans [BB96] (l'axiome du choix[4] avec le tiers-exclu implique proof-irrelevance) et l'adapte à `Set` (ce qui est possible puisque `Set` est imprédicatif) dans `Coq` pour montrer que le tiers-exclu fort

[3] Cette formalisation supposait, en plus du tiers-exclu fort, l'axiome eqT_rec qui permet faire de la réécriture sur des objets de sorte `Set` si on dispose de deux termes égaux pour l'égalité de Leibniz dans Type. Bien que non contradictoire avec le CCI, cet axiome n'était pas inclus dans `Coq` pour préserver le mécanisme d'extraction. Ce n'est toutefois plus le cas depuis la version 7.3 de `Coq` où eqT_rec a été définitivement intégré au système.

[4] Cet axiome du choix, formalisé dans `Coq` pour démontrer ce paradoxe, est défini dans `Prop` pour des propositions, et est prouvable dans `Coq` puisque l'élimination est autorisé de `Prop` vers `Prop`.

implique l'égalité de tous les termes de même type et de sorte `Set`, ce qui est faux (par exemple on peut montrer que $0 \neq 1$ dans le type `Nat` des entiers naturels de Coq).

Un démonstration plus compliquée, de Hermann Geuvers, peut se trouver dans [Geu01]. Elle est basée sur une adaptation de Thierry Coquand ([Coq89]) du paradoxe de Hurkens ([Hur95]). Cette preuve utilise elle aussi de manière forte l'imprédicativité de `Set`.

6.5.3 Travaux reliés

On peut citer les travaux de Maria Emilia Maietti. Dans [Mai99], elle montre qu'étendre la théorie des types de Martin-Löf avec des types quotients sans section mais avec la règle d'unicité des preuves d'égalité implique le tiers-exclu.

Si l'on essaye d'adapter ces résultats dans le Calcul des Constructions Inductives, on peut prouver dans `Coq`[5](avec nos types quotients) que l'axiome du choix avec l'unicité des preuves d'égalités implique le tiers-exclu dans `Prop`.

Si on essaye d'adapter cette preuve pour obtenir le tiers exclus fort (dans `Set`), on doit utiliser un axiome du choix fort (qui rend non pas l'existence dans `Prop` d'une fonction mais la fonction dans `Set`), qui lui implique de toute façon directement notre première hypothèse : la section s donnée constructivement.

6.6 Même l'existence d'une section au passage au quotient est contradictoire.

Revenons à notre problème initial : la surjectivité de θ est, on l'a vu, équivalente à l'*existence* de la section. La section `s : quo -> E` n'est donc plus donnée, mais on dispose de la proposition nous assurant son existence :

$$E \underset{s}{\overset{c}{\rightleftarrows}} quo \qquad \exists s : quo \to E,\ \forall x : quo,\ (c\ (s\ x)) == x.$$

On introduit ici une nouvelle astuce, l'ancienne astuce de Diaconescu ne servant plus dans ce cas (on verra en section 9.6.3 pourquoi).

6.6.1 De Prop vers bool

Le but de cette section est de démontrer l'existence d'une fonction `Proptobool : Prop -> bool` envoyant les propositions vraies sur `true` et fausses sur `false`, c'est-à-dire telle

[5]Le script coq peut être trouvé à cette adresse :
`http ://www-sop.inria.fr/lemme/personnel/Laurent.Chicli/t02_rel_work.zip`

que :

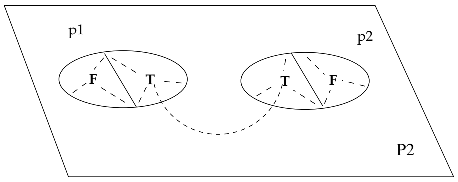
$$\forall P : Prop,\ P \leftrightarrow (Proptobool\ P) == true. \qquad (i)$$

6.6.1.1 Un quotient particulier

On va utiliser l'existence de la section pour le quotient particulier suivant. Considérons donc le type inductif $P2$ constitué de deux copies de `Prop` :

```
Inductive P2 : Type :=
   p1: Prop ->  P2
 | p2: Prop ->  P2 .
```

On définit la relation suivante sur les éléments de $P2$: toutes les propositions équivalentes d'une même copie de Prop sont mises en relation, ainsi que les propositions vraies de p_1 et p_2.



Dans `Coq`, cela donne :

```
Inductive R : P2 -> P2 ->  Prop :=
   R_12: (x, y : Prop) x -> y ->  (R (p1 x) (p2 y))
 | R_21: (x, y : Prop) x -> y ->  (R (p2 x) (p1 y))
 | R_11: (x, y : Prop) (iff x y) -> (R (p1 x) (p1 y))
 | R_22: (x, y : Prop) (iff x y) -> (R (p2 x) (p2 y)) .

Lemma Requiv: (Equivalence R).
```

et on définit ainsi le type quotient `QP2` :

```
Definition QP2 := (quotient Requiv).
```

6.6.1.2 Quelques définitions et propriétés

Puisque l'on cherche à définir un terme de sorte `Prop` (on veut prouver *l'existence* de la fonction `Proptobool`), on peut éliminer le quantificateur existentiel sur s et disposer de s dans notre contexte pour construire `Proptobool`.

Considérons alors la fonction $f = s \circ c$, de P2 dans lui-même. Elle a les deux propriétés suivantes :

– $\forall x : P2,\ xRf(x)$: (1)

$$\begin{array}{rcl} \forall x : P2,\ xR(fx) & \Leftrightarrow & (c\ x) == (c\ (f\ x)) \\ & \Leftrightarrow & (c\ x) == (c\ (s\ (c\ x))) \\ & \Leftrightarrow & (c\ x) == (c\ x) \end{array}$$

– $\forall x, y : P2,\ xRy \Rightarrow f(x) == f(y)$: (2)

$$\begin{array}{rcl} \forall x, y : P2,\ xRy & \Leftrightarrow & (c\ x) == (c\ y) \\ & \Leftrightarrow & (s(c\ x)) == (s(c\ y)) \end{array}$$

Définissons encore les fonctions `in_p1` et `in_p2`, d'appartenance à `p1` et `p2` :

```
Definition in_p1: P2 ->  bool :=
   [x:P2]Cases x of (p1 p)=>true  | (p2 p)=> false end.

Definition in_p2 : P2 ->  bool :=
   [x:P2]Cases x of (p1 p)=>false | (p2 p)=> true end.
```

Un élément de $P2$ de la forme $p_1(P)$ ne peut être en relation avec un élément de la forme $p_2(Q)$ que si la proposition P est vraie :

$$\forall P, Q : \mathrm{P}rop,\ p_1(P)\ R\ p_2(Q) \rightarrow P. \qquad (3)$$

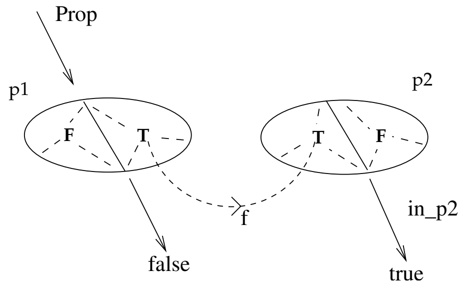
La preuve est évidente et se construit aisément dans `Coq` , simplement en faisant une inversion sur l'hypothèse.

6.6.1.3 L'existence de la fonction `Proptobool`

On dispose maintenant de toutes les briques nécessaires pour montrer l'existence de la fonction `Proptobool`. On procède en distinguant par cas suivant la valeur de `(in_p2 (f (p1 True)))`, pour définir le témoin de l'existence de `Proptobool` :

```
Exists Cases (in_p2 (f (p1 True))) of
          true  => [p : Prop] (in_p2 (f (p1 p)))
        | false => [p : Prop] (in_p1 (f (p2 p)))
       end.
```

La fonction ainsi définie satisfait bien la propriété $i)$: dans le premier cas, `(in_p2 (f (p1 True)))= true`, la fonction `Proptobool` est égale à $in\_p2 \circ f \circ p1$:



On veut montrer que

$$\forall P : Prop,\ P \leftrightarrow (Proptobool\ P) == true.$$

Soit donc P une proposition, si P est vraie, on a :

$$P \leftrightarrow True$$

et donc

$$p1(P)\ R\ p1(True).$$

Par (2), il s'en suit que

$$f(p1(P)) == f(p1(True))$$

et finalement :

$$Proptobool(P) == in\_p2(f(p1(P))) == in\_p2(f(p1(True))) == true.$$

Inversement maintenant, si $Proptobool(P) == in\_p2(f(p1(P))) == true$, on obtient en distinguant par cas sur $f(p_1(P))$:

$$f(p_1(P)) == p_2(Q),$$

l'autre cas $f(p_1(P)) == p_1(Q)$ étant absurde par hypothèse. On a aussi, par (1) :

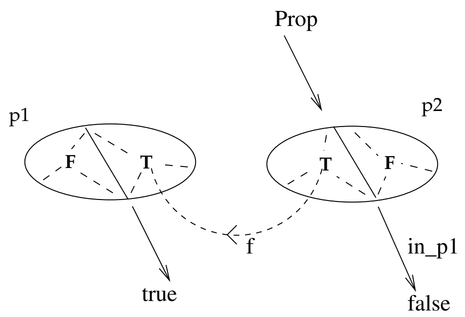
$$p_1(P)\ R\ f(p_1(P)),$$

et donc :

$$p_1(P)\ R\ p_2(Q).$$

On obtient ainsi par la propriété (3) une preuve de P.

Dans le second cas, si `(in_p2 (f (p1 True)))= false` on envoie cette fois-ci `Prop` dans `bool` à travers $in\_p1 \circ f \circ p2$, dont on démontre de manière analogue qu'elle vérifie bien la propriété i).



On montre ainsi dans Coq , sans utiliser aucune forme de tiers-exclu :

```
Lemma existence_decision:
(!exT ? [Proptobool:Prop->bool]
         (P:Prop)( P <-> (Proptobool P) = true)).
```

6.6.2 Incohérence

L'idée pour montrer l'incohérence dans ce cas est la même qu'auparavant : utiliser `Proptobool` pour démontrer le tiers-exclu dans `Set`, qui lui est contradictoire. Cependant, on ne peut pas ici montrer `(P :Prop){P}+{~P}` à l'aide de `Proptobool` puisqu'on ne dispose a priori pas de la fonction elle même mais uniquement de son existence, et que l'élimination d'un objet de `Prop` (le quantificateur existentiel) n'est pas possible lorsque la sorte d'arrivée est `Set` ce qui est le cas ici.

On contourne la difficulté en remarquant que ce que l'on cherche à démontrer est une proposition : `False : Prop`. On peut donc, pour construire un terme de type `False`, faire une élimination sur l'existence de `Proptobool` et ainsi en disposer dans notre contexte. On ne fait qu'ensuite une coupure avec `(P :Prop)({P}+{~P})`, que l'on démontre aisément pour tout `P` par analyse par cas sur `(Proptobool P)`.

```
Lemma incoherence: False.
Generalize existence_decision.
Intros yH.
Elim yH; Intros Proptobool H; Clear yH.
Cut Strong_EM.
Exact paradoxe.
Intros P.
Generalize (refl_equal ? (Proptobool P)).
Pattern -1 (Proptobool P); Case (Proptobool P).
Intros H0.
Case (H P).
Intros H1 H2.
Left; Exact (H2 H0).
Intros hyp; Right.
Unfold not;Intros p.
Case (H P).
Intros H0 H1.
Rewrite (H0 p) in hyp; Inversion hyp.
Qed.
```

6.6.3 Retour sur l'astuce de Diaconescu

On peut se demander si l'astuce consistant à éliminer les quantificateurs existentiels alors que le but à prouver est `False` rend inutile toute distinction entre la donnée constructive d'un objet et la simple donnée de son existence lorsqu'il s'agit de prouver une proposition.

Est-ce que dans notre cas on pourrait se servir des constructions de Werner/Lacas/Diaconescu pour démontrer l'incohérence de l'existence de la section au passage au quotient ? Il semble que la réponse soit négative.

En effet la construction du quotient de la section 3.1 depend d'une proposition donnée `P`, on peut donc en fait construire un terme de type

$$H_0 : \quad \forall P : Prop,\ \forall s : E/R_P \to E,\ \forall x\ c(s(x)) == x \Rightarrow \{P\} + \{\tilde{}P\}.$$

Par ailleurs, on a dans dans notre contexte une proposition nous assurant pour tout P l'existence d'une section au passage au quotient :

$$H_1 : \forall P : Prop,\ \exists s : E/R_P \to E \ \mid\ \forall x\ c(s(x)) == x.$$

Et nous voulons utiliser l'incohérence du tiers-exclu dans `Set` :

$$H_2 : ((P : Prop)\{P\} + \{\tilde{}P\}) \to False.$$

Le but courant étant `False` sous les hypothèses H_0, H_1, H_2, il y a deux possibilités pour continuer la preuve :

– soit on fait une coupure avec `(P :Prop){P}+{~P}`, avant d'introduire `P` ; mais le nouveau but courant (`{P}+{~P}`) est maintenant dans `Set` et on ne peut plus faire d'élimination sur le quantificateur universel de `(H1 P)`.

– soit on essaie tout d'abord de faire cette élimination sur le quantificateur existentiel de `H1`, mais pour prouver plus tard `(P :Prop){P}+{~P}` on devrait faire ceci pour toute proposition `P`, ce qui est bien sûr impossible.

Ainsi les constructions de la section 9.6.1 permettent d'aller plus loin que l'astuce de Diaconescu (et donc de prouver l'incohérence) en traitant toutes les propositions avec un seul type quotient.

6.7 Conclusion

On a montré en utilisant un unique quotient, que même la proposition d'existence d'une section au passage au quotient est contradictoire dans Coq . Le morphisme θ n'est donc pas surjectif en général dans CCI+types quotients. En ce sens l'expressivité des types quotients dans Coq semble définitivement moindre que celle des quotients mathématiques.

C'est la sorte `Set` qui pose problème, et plus précisément son imprédicativité. En effet, celle-ci est essentielle dans la preuve de l'inconsistance du tiers-exclus fort et sans elle, le modèle ensembliste classique booléen permet de montrer que la surjectivité de θ n'est plus contradictoire. Une des solutions pour permettre à Coq de disposer de types quotients ayant les mêmes propriétés sur les espaces fonctionnels que les quotients mathématiques serait donc d'éteindre l'imprédicativité de Set.

Chapitre 7

Incohérence des mathématiques classiques dans Coq

7.1 Introduction

Le Calcul des Constructions Inductives implanté dans `Coq` est, on l'a vu, un cadre logique constructif. Il est impossible d'y démontrer des théorèmes de mathématique classique utilisant le tiers-exclu ou encore l'axiome du choix, à moins de les y rajouter sous forme d'axiomes dans le système. Se pose alors la question de la cohérence de tels ajouts.

La séparation dans `Coq` des univers `Set` et `Type` de l'univers propositionnel `Prop` au niveau des principes d'élimination conduit à penser que l'on peut rajouter dans `Prop` ces axiomes (tiers-exclu et axiome du choix) sans d'une part empêcher le caractère constructif des constructions dans `Set` et `Type`, et sans d'autre part engendrer d'inconsistances dans le système . On va voir dans ce chapitre qu'il n'en est rien, et que l'on peut facilement introduire des incohérences dans le système en ajoutant des axiomes dans `Prop` dont l'équivalent en théorie des ensembles classique est pourtant vrai.

En effet, l'une des idées utilisées dans le chapitre précédent pour démontrer l'inconsistance de types quotients+surjectivité de la fonction θ dans `Coq` peut être utilisée pour montrer que l'axiome du choix unique à valeurs dans `Prop` est incohérent avec le tiers-exclu faible (dans `Prop` lui aussi).

7.2 Preuve

L'axiome du choix que l'on considère ici est le suivant :

```
Axiom AC :
   (A,B : Type)(R:A->B->Prop)
   ((x : A)  (exT B [y : B]  (R x y))) ->
```

```
((x : A) (y1, y2 : B) (R x y1) -> (R x y2) ->  y1 == y2) ->
(exT A ->  B [f : A ->  B] (x : A)  (R x (f x))).
```

Il affirme pour deux types A et B, et une relation R entre éléments de A et de B, que si pour tout x de A il existe un unique y tel que $(R\ x\ y)$ alors il existe (il ne s'agit donc que d'une proposition) une fonction $f : A \to B$ telle que pour tout x on ait $(R\, x\, f(x))$.

Le tiers-exclu faible dont il s'agit s'écrit quant à lui :

```
Axiom EM : (P : Prop)P\/~P.
```

Muni de ces deux axiomes, on peut montrer l'existence d'une fonction `Proptobool` vérifiant pour toute proposition $P : (ProptoboolP) == true \leftrightarrow P$.
Pour ce faire, on applique l'axiome du choix à la relation :

$$\begin{array}{rcl} R & : & Prop \to bool \to Prop \\ & := & [P : Prop][b : bool]P \leftrightarrow b = true \end{array}$$

Et il nous faut juste prouver :

$$\forall P : Prop, \exists! b : bool,\ P \leftrightarrow b = true$$

ce qui se fait facilement avec le tiers-exclu : il suffit de distinguer selon P ou ~P et de choisir `true` ou `false` en conséquence.

```
Lemma existence_decision:
 (!exT ? [f : Prop ->  bool] (P : Prop)  (iff P (f P) == true)).
Apply (!AC Prop bool [P : Prop] [b : bool]  (iff P b == true)).
Intros P; Elim (EM P).
Intros p; Exists true; Split; Intuition.
Intros p; Exists false; Split; Intuition.
Inversion H.
Intros x y1 y2.
Elim (EM x).
Intros xx H0 H;Elim H0;Elim H;Intros;Transitivity true;Intuition.
Case y1; Case y2; Intuition.
Elim (H (H3 (refl_eqT bool true))).
Qed.
```

On construit ensuite un terme de type `False` à partir de l'existence de `Proptobool` en reprenant la preuve de la section 6.6.2 :

Dans un contexte de but `False`, on fait une élimination sur l'existence de `Proptobool`. On dispose ainsi de cette fonction dans le contexte. On fait ensuite une coupure sur le tiers-exclu dans `Set` que nous savons contradictoire, et on prouve celui-ci à l'aide de `Proptobool` :

```
Lemma incoherence: False.
Generalize existence_decision.
Intros ed;Elim ed; Intros Proptobool H; Clear yH.
Cut Strong_EM.
Exact EM_paradoxe.
Intros P.
Generalize (refl_equal ? (Proptobool P)).
Pattern -1 (Proptobool P); Case (Proptobool P).
Intros H0.
Case (H P).
Intros H1 H2.
Left; Exact (H2 H0).
Intros hyp; Right.
Unfold not;Intros p.
Case (H P).
Intros H0 H1.
Rewrite (H0 p) in hyp; Inversion hyp.
Qed.
```

7.3 Implications

Encore une fois, l'incohérence provient du fait que l'on peut prouver le tiers-exclu fort dans un contexte prouvant `False`, et que celui-ci est contradictoire dans `Coq`, du fait de l'imprédicativité de `Set`.

En l'état actuel de Coq, et tant qu'il n'y aura pas moyen de désactiver `Set` ou simplement son imprédicativité, on ne peut donc pas rajouter à la fois le tiers-exclu et l'axiome du choix dans `Prop`. Ainsi toute tentative de formalisation de mathématiques classiques sera forcément limitée et partielle.

Qu'en est-il de la formalisation de la théorie des faisceaux décrite dans le chapitre 4 ? Elle utilise à la fois l'axiome du choix dans `Prop` et le tiers-exclu. Seul le passage à une version de `Coq` sans un `Set` imprédicatif pourra valider ce développement.

Chapitre 8

Conclusion

Nous avons présenté dans cette thèse deux principaux travaux, le premier autour de la formalisation dans Coq des schémas affines, le second une étude sur l'expressivité des types quotients dans Coq.

Dans le premier, on a formalisé l'ensemble des mathématiques nécessaires à la définition des schémas affines, premier pas vers une formalisation de la géométrie algébrique. Algèbre, topologie, et faisceaux ont été formalisés avant d'être finalement réunis, tout ceci mettant en avant plusieurs difficultés : lacunes de Coq ou choix de formalisation sur lesquels il a fallu revenir.
L'objectif premier était d'aller plus loin et d'énoncer le lemme d'Horace ; on en a tout de même donné une formalisation axiomatique vérifiant les conditions numériques de son application. Cette axiomatisation a pu être utilisée pour vérifier les utilisations du lemme d'Horace dans un mémoire ([dS]) et a mis en avant un grand nombre d'erreurs dans celui-ci.
Les problèmes (à la fois d'efficacité et d'élégance) rencontrés au cours de nos formalisations étant pour beaucoup dus à l'utilisation des sétoïdes comme structure pour représenter les ensembles et parmi eux les ensembles quotients, on s'est intéressé aux extensions existantes du *Calcul des Constructions Inductives* permettant de formaliser les quotients.

Le deuxième thème de cette thèse a donc été l'étude des types quotients dans Coq. On s'est d'abord intéressé à leur utilisation en implantant une librairie de commandes et tactiques Coq pour pouvoir les utiliser de manière facile et élégante.
On s'est ensuite intéressé à l'expressivité de ces types quotients dans Coq , en s'interrogeant sur une lacune de ces derniers relevée par M.Hofmann ([Hof95a], p.254-255). Alors que celui-ci propose de rajouter la propriété manquante aux types quotients dans le cadre du Calcul des Constructions, on a montré qu'elle était contradictoire dans le CCI de Coq, montrant ainsi que les types quotients dans Coq avaient une expressivité définitivement moindre que les quotients en mathématiques classiques. L'origine de la contradiction est la présence simultanée de l'imprédicativité et de l'élimination forte dans la sorte `Set`.

Enfin, les idées utilisées dans la démonstration précédente ont permis de démontrer qu'il était impossible d'ajouter à la fois le tiers exclu et l'axiome du choix (dans `Prop`) dans Coq, réduisant ainsi les possibilités d'y formaliser les mathématiques classiques. Le coupable est là encore désigné : l'imprédicativité de `Set`, que l'on espère donc un jour voir devenir facultative ...

Bibliographie

[Bar95] G. Barthe. Extensions of pure type systems. In Springer-Verlag, editor, *Proceedings of TLCA'95, volume 902 des LNCS, pages 16-31*, 1995.

[BB85] Errett Bishop and Douglas Bridges. *Constructive Analysis.* Springer-Verlag, 1985.

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

[Bis67] Errett Bishop. *Foundations of Constructive Analysis.* McGraw Hill, New York, 1967.

[BKT94] Yves Bertot, Gilles Kahn, and Laurent Théry. "proof by pointing". In *Symposium on Theoretical Aspects Computer Science (STACS),Sendai (Japan), LNCS 789*, 1994.

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

[CH88] Thierry Coquand and Gerard Huet. *The Calculus of Constructions.* Information and Computation, 76 , pages 95-120, 1988.

[COQ] INRIA Roquencourt COQ, Projet LOGICAL. The coq proof assistant. http ://coq.inria.fr.

[Coq89] Thierry Coquand. An introduction to type theory, 1989. notes de l'école d'été FPCL.

[Cou01] P. Courtieu. Représentation d'algèbres non libres en théorie des types. Thèse de doctorat, Université Paris-Sud, 2001.

[Dia75] R. Diaconescu. Axiom of choice and complementation. In *Proceedings of A.M.S., vol. 51, pages 176-178)*, 1975.

[Dow] Gilles Dowek. Théorie des types, cours de dea programmation. http ://www.lix.polytechnique.fr/~dowek/Cours/theories_des_types.ps.gz.

[dS] Benoîte de Saporta. Systèmes linéaires de courbes de genre 5. http ://enslyon.free.fr/rapports/maths/Benoite_de_Saporta_2.ps.gz.

[Geu01] H. Geuvers. Inconsistency of classical logic in type theory, 2001. http ://www.cs.kun.nl/ herman/note.ps.gz.

[Gir86] Jean-Yves Girard. *The system F of variables types : Fifteen years later*, volume 45 : 159 192 of *Theoretical Computer Science*. 1986.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[God73] Roger Godement. *Topologie Algébrique et théorie des faisceaux.* Hermann, 1973.

[Har77] Robin Hartshorne. *Algebraic Geometry.* Springer Verlag, 1977.

[Hir85] André Hirschowitz. La méthode d'horace pour l'interpolation à plusieurs variables, 1985. Manus. Math. 50 (1985) 337-388.

[Hof95a] M. Hofmann. *Extensional concepts in intensional type theory.* Thèse de doctorat, LFCS Edinburgh, 1995.

[Hof95b] M. Hofmann. A simple model for quotient types. In *Proceedings of TLCA'95, vol. 902 of LNCS, Springer-Verlag*, 1995.

[Hur95] A. Hurkens. A simplification of girard's paradox. In Springer-Verlag, editor, *Proceedings of TLCA'95, volume 902 des LNCS, pages 266-278*, 1995.

[Lan84] Serge Lang. *Algebra (second edition).* Addison-Wesley Publishing Company, 1984.

[Log] Projet Logical. *The Coq Proof Assistant Reference Manual, http ://pauillac.inria.fr/coq/doc-fra.html.*

[LW99] Stéphane Lacas and Benjamin Werner. Which choices imply the excluded middle ? http ://pauillac.inria.fr/~werner/, 1999.

[Mai99] Maria Emilia Maietti. About effective quotients in constructive type theory. In *TYPES'98, LNCS 1657,pp 164-178*, 1999.

[Mig97] T. Mignon. *Systèmes linéaires de courbes planes.* Thèse de doctorat, Université de Nice Sophia Antipolis, 1997.

[Miq] A. Miquel. Sémantiques dénotationnelles du calcul des constructions. http ://pauillac.inria.fr/~miquel/publi/semcc.ps.

[Miq00] Alexandre Miquel. A model for impredicative type systems with universes, intersection types and subtyping. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science (LICS'00)*, 2000.

[PC] INRIA Sophia Antipolis Projet CROAP. The ctcoq system. http ://www-sop.inria.fr/croap/ctcoq/ctcoq-fra.html.

[PM93] Christine Paulin-Mohring. *Inductive definitions in the system COQ.* LNCS 664, Typed Lambda Calculi and applications, pages 329-354. Springer Verlag, 1993.

[Pot98] Loïc Pottier. Basic notions of algebra. In *http ://pauillac.inria.fr/coq/contribs/algebra.html*, 1998.

[Pot01] Loïc Pottier. Quotients dans le CCI Rapport de recherche INRIA RR-4053. , INRIA, 2001.

[Saï99] Amokrane Saïbi. *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories.* Thèse de doctorat, INRIA Rocquencourt, 1999.

[SH95] Amokrane Saïbi and Gerard Huet. Constructive category theory. In *joint CLICS-TYPES Workshop on Categories and Type Theory, Goteborg (Sweden)*, 1995.

[Try] A. Trybulec. The mizar project. http ://mizar.uwb.edu.pl/.

[Wer94] Benjamin Werner. *Méta-théorie du Calcul des Constructions Inductives.* Thèse de doctorat, Université de Paris 7, 1994.