# Une formalisation des faisceaux et des schémas affines en théorie des types avec Coq

Laurent Chicli[1]

*1: INRIA Sophia-Antipolis, Projet Lemme,*
`lchicli@sophia.inria.fr`

**Résumé**

Nous présentons nos spécifications en théorie des types des notions mathématiques nécessaires à la définition des schémas affines, l'un des premiers éléments de la géométrie algébrique moderne. Pour cela topologie et théorie des faisceaux ont étés développées, nous décrivons ici leurs spécifications, ainsi que les problèmes rencontrés lors de ce travail.

## 1. Introduction

L'objectif du travail que nous présentons dans cet article est de commencer une formalisation de la géométrie algébrique avec la définition des schémas affines en *Coq*.

Pour présenter les schémas affines, considérons par exemple $\mathbb{C}^2$, le plan sur le corps des complexes. Puisque $\mathbb{C}$ est algébriquement clos, $\mathbb{C}^2$ est en bijection avec l'ensemble des idéaux maximaux de $\mathbb{C}[X,Y]$ (de tels idéaux s'écrivent en effet $< X-a, Y-b >$ où $a, b \in \mathbb{C}$). L'idée est d'étendre l'ensemble des points que l'on considère en ne considérant non pas seulement les idéaux maximaux de $\mathbb{C}[X,Y]$, mais tous les idéaux premiers. Les points du nouvel ensemble (dans notre exemple) correspondront soit à des points de $\mathbb{C}^2$, soit à des courbes irréductibles (excepté pour un nouveau point : l'idéal nul, sans correspondance dans $\mathbb{C}^2$).
Plus généralement on associe à un anneau $A$ l'ensemble $Spec(A)$ de ses idéaux premiers. Et parce que changer les ensembles nécessite de changer les notions de continuité et de morphisme, on construit sur $Spec(A)$ une topologie et une structure de faisceaux. Les schémas affines sont les données de tels espaces topologiques avec leurs faisceaux structuraux.

Ce travail est basé sur la formalisation de l'algèbre effectuée par Loïc Pottier [POT99], qui comprend les notions de catégorie, d'ensemble, groupe, anneau, et corps, et est effectué avec la version 6.2.3 du logiciel d'aide à la démonstration *Coq* [COQ99] (et donc avec le *Calcul des Constructions Inductives* comme théorie des types). On s'est principalement inspiré du livre de R. Hartshorne : 'Algebraic Geometry' [HAR77].

L'interface graphique *CtCoq* [CRO96] 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.

## 2. Quelques mots sur *Coq* et la représentation des ensembles

On présente ici quelques aspects de la formalisation de l'algèbre en *Coq* utilisée comme base de notre travail.

## 2.1. Sétoïdes

La structure utilisée pour représenter les ensembles est celle des setoïdes (voir par ex. [SAI95]). Un setoïde est la donnée d'un type (qui sera le type des éléments de l'ensemble) et d'une relation d'équivalence (qui servira d'égalité sur cet ensemble).

**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)*

L'image précédente correspond à ce qui apparait sous *CtCoq* avec un 'pretty-printer' approprié, 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 de *Coq*. La macro Record crée ici le type inductif `Setoid` ayant pour unique constructeur la fonction `Build_Setoid` qui demande trois arguments des types spécifiés dans les différents champs. Elle définit aussi les trois projections `Carrier, Equal, Prf_equiv` qui associeront à un setoïde son type, sa relation, et la preuve qu'il s'agit bien d'une relation d'équivalence.
Le symbole `:>` permet de déclarer la projection `Carrier` comme coercion entre le type `Setoid` et le type `Type`, de cette manière un setoïde pourra 'être vu' (à travers Carrier) comme un type. Si E est un setoïde, on pourra donc considérer un élément x 'de type' E ( ce que *Coq* instanciera automatiquement par `x : (Carrier E))`.
Les setoïdes ont l'avantage de permettre d'appréhender de manière totalement homogène les ensembles quotients : l'ensemble $E/R$ n'est rien d'autre qu'un setoïde ayant le même type Carrier que $E$ mais où la relation initiale a été changée par R.

## 2.2. Sous-typage et parties d'un ensemble

Le moyen le plus naturel de représenter une partie A d'un ensemble E est de se donner un prédicat P sur E (de type `E -> Prop`) compatible avec l'égalité de E :

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

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

Pour faire de la partie de $E$ définie par $P$ un setoïde, il faut 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 $F \to E$, on peut définir le setoïde `(subtype_set E F i)` par le type F et la relation d'équivalence $x\ R\ y$ ssi $i(x) = i(y)$. Ce setoïde 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 de type subtype seront donc des couples constitué 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 setoïde définit 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 alors peut se définir ainsi :

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

(i.e. la fonction in_part qui à un setoïde E, un terme x de type E, et à un prédicat A sur E associe la proposition `(Pred_fun A x))`, et l'ensemble des parties d'un setoïde :

```
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)).

Lemma eq_part_equiv: (equivalence eq_part).

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

Dans la définition ci-dessus on ne donne à *Coq* qu'un seul des trois arguments nécessaires à la fonction `Build_Setoid`. En effet, les deux arguments manquant (le type `(Predicate E)` et la relation `eq_part`) sont implicitement contenu dans le type de `eq_part_equiv`,et sont automatiquement synthétisés par *Coq*.

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 *Coq* ne permet pas de déclarer cette fonction comme coercion (elle serait paramétrée par le setoïde E, ce qui n'est pas accepté par *Coq*).

On choisit d'afficher sous *CtCoq* $x \in A$ pour le terme `(in_part x A)`, et de cacher la fonction `subtype_elt` pour plus de lisibilité.

# 3. Topologie

## 3.1. 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.

Let E be a set.
**Definition.**
  union_part: $\wp(\wp(\mathrm{E})) \Rightarrow \wp(\mathrm{E})$
 1: `Intros` P. 1: `Apply` (!Build_Predicate ? x:E , $\exists$ A: $\wp$(E) , (A $\in$ P) $\wedge$ (x $\in$ A)).

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

**Definition** (inter_in_top).
  given top of type $\wp(\wp(\mathrm{E}))$ ,
  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
**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 setoïde sous-jacent et un terme `U :(open X)` un élément de la topologie, donc un ouvert, comme en langage mathématique courant.

Les notions de fermés, adhérences, intérieurs, bords, ainsi que tout 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 prends 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 propriétés caractéristiques de l'adhérence : c'est le plus petit fermé contenant 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.

## 3.2. Topologie engendrée par une base d'ouverts

Dans notre développement, une base d'ouverts B 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 la 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 $\varnothing$ $\in$ open_base_setoid

On définit l'ensemble des parties engendrées par union d'éléments d'une base B par :

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).
$\emptyset$ $\in$ 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)).

## 3.3. Continuité

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

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

**Lemma** (im_of_adh).
∀ f:(Map E F), f is continuous ⇒ ∀ C:℘(E), f(adh(C)) ⊂ adh(f(C)).

**Lemma** (cont_with_adh_inv).
∀ f:(Map E F), (∀ C:℘(F), $f^{-1}$(adh(C)) = adh($f^{-1}$(C))) ⇒ f is continuous.

Après avoir formalisé ces différentes notions de topologie, on a assez de matière pour passer à la théorie des faisceaux.

# 4. Théorie des faisceaux

## 4.1. Catégorie associée à un espace topologique

### 4.1.1. 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.

### 4.1.2. Une première idée

Formaliser l'ensemble des objets ne pose pas de problèmes, il s'agit de la topologie elle-même. Mais pour l'ensemble des morphismes, la discussion sur l'inclusion éventuelle de U dans V mérite attention. La première idée est de se donner en paramètre une fonction de décision sur l'inclusion :

```
Parameter incl_dec :(U,V:(open X)){included U V}+{~(included 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 exclus) 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 désavantages. Le premier est que cette fonction de décision doit être dans la plupart des cas considérée comme un axiome car l'inclusion dans un ensemble non dénombrable n'est justement pas décidable. Le second est qu'elle oblige régulièrement à travailler avec des cas 'vides' (dans les 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 a pas l'habitude de considérer 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.

### 4.1.3. Spécifications

Notre solution (due à André Hirschowitz) est de considérer $Hom(U, V)$ comme le setoïde 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.

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 $\Rightarrow$ prf_included $\Rightarrow$ 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, définir la composition de deux morphismes revient à prouver que $U \subset W$ sous les hypothèses $U \subset V$ et $V \subset W$, ce qui est immédiat.

## 4.2. Foncteurs contravariants et préfaisceaux

Un préfaisceau d'anneaux commutatifs sur un espace topologique $X$ est un foncteur contravariant de la catégorie $(Top\_catX)$ dans la catégorie $CRING$ des anneaux commutatifs.

On a choisi de formaliser les préfaisceaux et faisceaux d'anneaux commutatifs. Il suffit cependant de remplacer dans notre développement le mot clé CRING par celui ne n'importe quelle autre catégorie dont les objets peuvent être coercé vers les setoïdes pour obtenir la notion de préfaisceau correspondante. La seule restriction concerne les spécifications des structures d'anneaux induites, qui devront évidemment être adaptés.
Les foncteurs contravariants entre deux catégories $C_1$ et $C_2$ sont définit comme dans [SAI95], en se donnant une application $F$ entre les objets et une autre $F^*$ entre les morphismes, satisfaisant 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 CRPresheaf := [X:et](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 $\rho_{VU}$. Si $s \in F(V)$, on note aussi $s_{|U}$ l'élément $F^*(U \subset V)(s)$ de $F(U)$.
Toutes ces notations sont définissables 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))).
```

## 4.3. Faisceaux

Un faisceau est un préfaisceau vérifiant deux propriétés :
- Si $s \in F(U)$ et si $\{V_i\}_{i\in I}$ est un recouvrement ouvert de U tel que $\forall i\ s_{|V_i} = 0$ alors $s = 0$ dans $F(U)$,
- Si on a une collection d'éléments $s_i \in F(V_i)$ tels que $s_{i|V_i\cap V_j} = s_{j|V_i\cap V_j}$ alors il existe $s$ dans $F(U)$ tel que $\forall i\ s_{|V_i} = s_i$.

Le problème est ici, pour le second point, de se donner la collection d'éléments $s_i$. On voudrait prendre 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 setoïde, puis dire que se donner une telle collection c'est se donner une partie de ce setoïde. On a donc besoin pour cela d'une égalité sur ce type. La notion la plus naturelle d'égalité 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.
Remarquons par contre que si `(part x)=(part x')`, on a en particulier `(part x)`$\subset$`(part x')`et donc `(elt x')` peut 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 $(eltx) = (elt\ x')_{|(part\ x)}$.

**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.

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

## 4.4. Germes, fibres, morphismes

### 4.4.1. 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 $(U, s \in F(U))\ R\ (V, s' \in F(V))$ ssi $(\exists W|\ p \in W,\ W \subset U \cap V,\ s_{|W} = s'_{|W})$.
La souplesse des setoïdes 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 $\in$ 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 $\subset$ $g2_{\_set}$ , $g1_{\_elt|W}$ = $g2_{\_elt|W}$.

Les anneaux $F(U)$ induisent une structure d'anneau sur les fibres : par exemple on peut définir la somme de deux germes $(U, s) + (V, s')$ 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))
$g1_{\_elt|g1\_set \cap g2\_set}$+$g2_{\_elt|g1\_set \cap g2\_set}$). **Defined**.
**Lemma** (ringstalk1).
$\forall$ x,x',y,y':stalk , x = x' $\Rightarrow$ y = y' $\Rightarrow$ (stalk_plus x y) = (stalk_plus x' y').

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.

### 4.4.2. Morphismes

Comme pour toutes les structures mathématiques il existe une notion de morphisme pour les préfaisceaux. Un morphisme $\phi$ 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.

**Definition.**
a presheaf_morphism is a structure with the following components:
*morphism_of*: of type $\forall$ U:open of X , F(U) $\longrightarrow$ G(U) *(coercion)*
*commut_prf*: of type $\forall$ U,V:open of X , $\forall$ t:(prf_included V U) , ($\rho_{UV}$ o U) = (V o $\rho_{UV}$)

Un tel morphisme 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-jascents : 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 $(U, s)$ sur $(U, \phi(U)(s))$. Ces morphismes induit sont formalisés, on les affiche $\phi\_p$ sous *CtCoq*.

## 4.5. 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 contraposée 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 *ssi* tous les morphismes induits par $\phi$ dans les fibres le sont.
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 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 :

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

qui formera bien sur 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 $s = 0$, et donc :

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

## 4.6. 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.

### 4.6.1. 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'ensemble des applications $s : U \to \bigsqcup_{p\in U} F_p$ 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 $F(V) \to F(U)$ 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*.

### 4.6.2. 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 de type `(p :(subtype_elt U))(Stalk F (subtype_elt p))`. On va faire de l'ensemble de ces termes un setoïde.

Soit $f$ un terme de type `(p :(subtype_elt U))(Stalk F (subtype_elt p))`. Puisque f doit représenter une fonction, on voudrait que si x=y alors (f x)=(f y), mais là encore cette dernière égalité n'a pas de sens puisque le premier terme est de type (Stalk F x) et le second de type (Stalk F y).

De manière générale, on peut procéder ainsi :
Soit `E :Setoid` et `F :E ->Setoid`, on demande en paramètre une fonction G qui à x,y :E et une preuve p que x=y associe une fonction `(F x)->(F y)` (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 les termes f de type `(x :E)(F x)` qui vérifient, pour tout `x` et `y` de type `E` et étant donné une preuve p que `x=y`, `(f y)=((G x y p) (f x))`.

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)

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

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

**Remarque :** Cette même idée nous permet aussi d'avoir une égalité significative sur les sigma-types : l'égalité de `(a :A,b :(f a))` et de `(a' :A',b' :(f a'))` pourra être définie par $\exists$ `p :a=a' | b'=(G a a' p b)`. Pour que cette définition soit valide, c'est-à-dire pour que la relation ainsi définie soit une relation d'équivalence, il faut de plus demander à G de vérifier deux propriétés :
- $\forall x : A, \forall p : x = x, (G\ x\ x\ p) = Id_{(fa)}$
- $\forall x, y, z : A, \forall p_{xy} : x = y, \forall p_{yz} : y = z, \forall p_{xz} : x = z,\ (G\ y\ z\ p_{yz}) o (G\ x\ y\ p_{xy}) = (G\ x\ z\ p_{xz})$

### 4.6.3. Spécifications et preuves

Dans notre cas, on prends pour E le setoïde `(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).
$\forall$ A:$\wp$(E) , $\forall$ x,y:E , x $\in$ A $\Rightarrow$ y = x $\Rightarrow$ y $\in$ A.

**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$_{\text{elt}}$)
(its type is $\forall$ p1,p2:U , $\forall$ prf:p1 = p2 , point_to_stalk(p1) $\Rightarrow$ point_to_stalk(p2)).

Pour plus de commodité par la suite on fait de `fun_stalk_to_same` une application qui ne renvoie non pas une simple application de $F_{p1}$ vers $F_{p2}$, mais un morphisme d'anneau. Après avoir vérifié toutes les propriétés nécessaires , on peut enfin définir l'ensemble des fonctions qui nous interessent 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 faisceaux associés doivent en outre vérifier la propriété (ii) :

**Definition.**
associated_sheaf_setoid: $\wp$(local_data_map)

1: `Apply` (!Build_Predicate ?
s:local_data_map ,
$\forall$ p:U ,
$\exists$ V:open of X ,
$\exists$ proof_neigh:p $\in$ V ,
$\exists$ proof_included:V $\subset$ U ,
$\exists$ t:F(V) , $\forall$ q:V , s(q$_{[\text{in U}]}$) = (in_stalk F subtype_prf(q))(t)).

On définit à partir de là les morphismes de restrictions, puis on prouve que le préfaisceau ainsi contruit est bien un faisceau (cf [CHI00] pour plus de détails).

# 5. Spectre et Schémas Affines

## 5.1. Le spectre d'un anneau comme espace topologique

Soit A un anneau commutatif, comme ensemble, son spectre est l'ensemble de ses idéaux premiers.

Let R be a commutative ring.
**Definition.**
spec:set
1: Apply (!Build_Setoid prime ideal of R !Equal($\wp$(R))).

Si $\mathfrak{I}$ est un ideal de A, on définit $V(\mathfrak{I})$ (en *Coq* (`primes_containing I`)) comme l'ensemble des idéaux premiers de A qui contiennent $\mathfrak{I}$. L'ensemble de ces $V(\mathfrak{I})$ forment en fait les fermés d'une topologie sur spec(A). On note $Spec(A)$ cet espace topologique.

**Definition.**
primes_containing:$\forall$ I:ideal(R) , $\wp$(spec)
1: Intros I. 1: Apply (!Build_Predicate spec x:spec , I $\subset$ x).

**Definition.**
spec_top:Topology(spec)
1: Apply !Build_Topology(O:$\wp$(spec) , $\exists$ p:ideal(R) , O = $\neg$V(p)).

**Definition.**
let Spec be the topological space (!Build_et spec spec_top).

## 5.2. Un préfaisceau particulier pour définir les schémas affines

Le schéma affine défini par $A$ est la donnée de l'espace topologique $(Spec\ A)$ muni d'un faisceau $\mathcal{O}_{(SpecA)}$ qui sera le faisceau associé au préfaisceau $\mathfrak{F}$ que l'on décrit ici :
Pour tout ouvert $U$ de $(SpecA)$ on définit $\mathfrak{F}(U)$ comme l'anneau constitué des couples $(g, h)$ de $A^2$ tels que $\forall p \in U, h \notin p$. L'égalité, la somme et le produit sont définit comme pour les fractions, par exemple $(g_1, h_1) + (g_2, h_2) = (g_1 * h_2 + g_2 * h_1, h_1 * h_2)$, le zéro est $(0_A, 1_A)$, et l'unité $(1_A, 1_A)$.

Let U be a open of Spec(A).
**Definition.**
a spec_pre_elt is a structure with the following components:
*sp_num*: of type A
*sp_den*: of type A
*sp_prf*: of type $\forall$ p:prime ideal of A , p $\in$ U $\Rightarrow$ sp_den $\notin$ p

**Definition.**
let equ_spec_pre be
the function defined by , for all s,s':spec_pre_elt , $s_{num}*s'_{den}$ = $s'_{num}*s_{den}$
(its type is spec_pre_elt $\Rightarrow$ spec_pre_elt $\Rightarrow$ Prop).
**Definition.**
spec_pre_set:set
1: Apply (!Build_Setoid spec_pre_elt equ_spec_pre).
**Definition.**
spec_pre_plus:spec_pre_elt $\Rightarrow$ spec_pre_elt $\Rightarrow$ spec_pre_elt
1: Intros s s'. 1: Apply (!Build_spec_pre_elt $s_{num}*s'_{den}+s'_{num}*s_{den}$ $s_{den}*s'_{den}$).
**Definition.**
spec_pre_zero:spec_pre_elt
1: Apply (!Build_spec_pre_elt 0 1).

Les restrictions sont définies de manière naturelle : si $U \subset V$ on envoi un couple (g,h) de $\mathfrak{F}(V)$ sur ce même couple vu comme élément de $\mathfrak{F}(U)$, la condition $\forall p \in U, h \notin p$ étant naturellement satisfaite puisque $U \subset V$.

Une fois le préfaisceau $\mathfrak{F}$ ( `(spec_presheaf A)` en *Coq*) formalisé, on peut enfin définir le faisceau $\mathcal{O}_{(SpecA)}$ par `(associated_sheaf (spec_presheaf A))`.

# 6. Problèmes rencontrés

Deux principaux problèmes ont été rencontrés lors de ce développement.

## 6.1. Limitations des coercions

Le premier concerne les limitations des coercions autorisées par *Coq*. En effet les coercions paramétrées ne sont pas gérées par le système et cela oblige l'utilisateur à explicitement faire appel à la fonction `subtype_elt` pour voir un élément d'une partie comme 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` ! Notons de plus que les possibilités étendues de notations de *Ctcoq* sont ici essentielles pour comprendre les termes que l'on manipule.

## 6.2. Les setoïdes

Le second problème vient lui du choix même de la structure utilisée pour représenter les ensembles : les setoïdes. 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 de *Coq*, et donc souvent à procéder par réécritures successives dans les preuves, 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 pas été rare d'avoir à attendre de cinq à quinze minutes l'exécution d'une tactique (étape élémentaire de preuve).

# 7. Conclusion

Le développement présenté dans cet article (représentant 19000 lignes de codes) 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 dans un terme 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 de manière non négligeable tout au long de l'avancée de notre travail (jusqu'à plus d'une dizaine de lignes de code pour certains termes). En cela les spécificités de *CtCoq* (possibilités de notations étendues mais surtout d'édition structurée du code), ont été nécessaires pour mener à bien notre projet.

Le manque de soustypes (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 ralentit notre travail.

Les schémas affines ne forment évidemment qu'une infime partie du corpus de la géométrie algébrique, mais la définition de la notion générale de schémas et de leurs premières propriétés géométriques ne parait pas inaccessible. La difficulté résidera sans doute dans les notions de recollement et d'isomorphisme qui devront être formalisés de manière assez souple pour être réellement utilisables.

# Références

[BKT94] Yves Bertot, Gilles Kahn, and Laurent Théry, "Proof by Pointing", Symposium on Theoretical Aspects Computer Science (STACS), Sendai (Japan), LNCS 789, Avril 1994.

[CHI00] Laurent Chicli. Une formalisation des schémas affines avec *Coq*. Rapport de Recherche INRIA (à paraître).

[COQ99] L'équipe *Coq*. The Coq Proof Assistant Reference Manual v6.3. `http ://pauillac.inria.fr/coq/doc-fra.html`

[CRO96] Projet CROAP, INRIA Sophia-Antipolis. The Ctcoq system . `http ://www-sop.inria.fr/croap/ctcoq/ctcoq-fra.html`

[GOD73] Roger Godement, Topologie Algébrique et théorie des faisceaux , Editions Hermann, 1973.

[HAR77] R. Hartshorne, Algebraic Geometry, Springer Verlag, 1977.

[LAN84] Serge Lang, Algebra (second edition), Addison-Wesley Publishing Company, 1984.

[POT99] Loïc Pottier. Basic notions of algebra, Disponible à l'adresse : `http ://pauillac.inria.fr/coq/contribs/algebra.html`

[SAI95] A. Saibi and G. Huet, "Constructive Category Theory" In Proceedings of the joint CLICS-TYPES Workshop on Categories and Type Theory, Goteborg (Sweden), January 95.