Rapport de stage

Pierre-Alexandre Bazin

17 août 2022

1 Introduction : Déroulement général du stage

J'ai effectué mon stage à l'Imperial College London du 31 janvier au 17 juin 2022, encadré par le Pr Kevin Buzzard. Durant ce stage, j'ai contribué à prouver des résultats mathématiques dans *mathlib* [1], la librairie mathématique de l'assistant de preuve Lean. Ma contribution principale a été dans la formalisation de la preuve du théorème de structure des modules finiment engendrés sur un anneau principal en Lean. [5]

Le stage a débuté entièrement en distanciel, n'ayant pas pu rejoindre Londres avant mi-mars suite à des problèmes administratifs. La grande majorité de mon stage a consisté en du travail sur ordinateur dans l'assistant de preuve Lean, avec des rencontres hebdomadaires avec mon directeur de stage pour discuter de mon implémentation des résultats que j'ai prouvé en Lean et des théorèmes qui pourraient être implémentés ensuite. À cette occasion j'ai aussi pu avoir un aperçu des thèmes de recherche en théorie des nombres.

J'ai aussi assisté aux séminaires hebdomadaires "London Learning Lean" où sont présentés divers résultats mathématiques qui ont été récemment prouvés en Lean, et ai moi-même présenté mon implémentation du théorème de structure des modules finiment engendrés sur un anneau principal lors de l'une des séances. [8]

2 Présentation de Lean

2.1 Principe général

2.1.1 Énoncer des résultats

Lean est un assistant de preuve, c'est-à-dire un langage permettant d'énoncer des résultats mathématiques et de les prouver. L'ordinateur pourra alors vérifier la preuve et assurer qu'il n'y a pas d'erreur ou de cas particulier oublié. Par exemple, on peut énoncer la proposition logique suivante en Lean :

Proposition 1. *Soit A un ensemble, P une propriété portant sur les éléments de A et Q une propriété. On a alors l'équivalence :*

$$(\forall x \in A, P(x) \implies \neg Q) \iff (Q \implies \neg(\exists x \in A, P(x)).$$

```
example (A : Type) (P : A → Prop) (Q : Prop) :
  (∀ x : A, P x → ¬Q) ↔ Q → ¬(∃ x : A, P x) :=
sorry
```

FIGURE 1 – Formulation de la proposition 1 (non prouvée) en Lean.

Les ensembles mathématiques (comme A dans la proposition ci-dessus) sont représentés en Lean par des types, donc des objets de type `Type`. Les objets de type `A` correspondent alors aux éléments de A. `P` est quant à lui en Lean un objet de type `A → Prop`, c'est-à-dire une fonction qui à chaque objet de `A` associe une propriété (dans `Prop`).

Les propriétés sont elles aussi des types en Lean : en fait, étant donné une propriété `Q : Prop`, un objet de type `Q` correspond à une preuve de Q. C'est pourquoi Lean utilise le symbole `→` pour l'implication : en effet, une preuve de l'implication $P \implies Q$ correspond bien à une fonction `P → Q`, permettant d'obtenir une preuve de Q à partir d'une preuve de P.

Aussi, si `A : Type`, `P : A → Prop` et `H : ∀ x : A, P x`, `H` est une fonction prenant un objet `x : A` et renvoyant un objet `H x : P x`. On voit ici que le type de `H x` dépend de la valeur de `x` : on dit que Lean utilise la théorie des types dépendants.

2.1.2 Démontrer des résultats

Comme les propriétés en Lean sont des types dont les objets sont des preuves, il suffit pour prouver un théorème en Lean de construire un élément de type [l'énoncé du théorème]. Par exemple, prouver la proposition 1 revient à définir une fonction qui, étant donné un type `A` et des propriétés `P : A → Prop` et `Q : Prop`, renvoie un objet de type `(∀ x : A, P x → ¬Q) ↔ Q → ¬(∃ x : A, P x)`. (La fonction ainsi construite sera donc de type `∀ (A : Type) (P : A → Prop) (Q : Prop), (∀ x : A, P x → ¬Q) ↔ Q → ¬(∃ x : A, P x)`.)

Dans la figure 1, cette fonction est définie comme `sorry`, qui agit comme un "joker" pouvant être une valeur de n'importe quel type. Remplacer les preuves de résultats non prouvés par `sorry` permet d'avoir du code sans erreurs de syntaxe même lorsque la preuve d'un résultat est incomplète. En revanche, un résultat ne peut bien évidemment pas être considéré comme prouvé si sa preuve contient un `sorry` : Lean va donc renvoyer un *avertissement* à la compilation signalant que la preuve est incomplète car elle contient un `sorry`.

```
example (A : Type) (P : A → Prop) (Q : Prop) :
  (∀ x : A, P x → ¬Q) ↔ Q → ¬(∃ x : A, P x) :=
⟨λ H q E, H _ (classical.some_spec E) q,  λ H a ha q, H q ⟨a, ha⟩⟩
```

FIGURE 2 – Une preuve de la proposition 1 en Lean, qui est donc un terme du type voulu.

```
example (A : Type) (P : A → Prop) (Q : Prop) :
  (∀ x : A, P x → ¬Q) ↔ Q → ¬(∃ x : A, P x) :=
⟨λ (H : ∀ x : A, P x → ¬Q) (q : Q) (E : ∃ x : A, P x),
  (H _ (classical.some_spec E) q : false),
λ (H : Q → ¬(∃ x : A, P x)) (a : A) (ha : P a) (q : Q),
  (H q (⟨a, ha⟩ : ∃ x, P x) : false)⟩
```

FIGURE 3 – Une version plus détaillée de la preuve de la figure 2, où les types de certains termes intermédiaires sont spécifiés.

Ici, le terme construit est composé de deux éléments :
- une fonction de type `(∀ x : A, P x → ¬Q) → Q → ¬(∃ x : A, P x)`, prenant donc trois arguments `H : ∀ x : A, P x → ¬Q`, `q : Q` et `E : ∃ x : A, P x` et renvoyant un élément de type `false`.
- et une fonction de type `(Q → ¬(∃ x : A, P x)) → (∀ x : A, P x → ¬Q)`, prenant donc quatre arguments `H : Q → ¬(∃ x : A, P x)`, `a : A`, `ha : P a` et `q : Q` pour renvoyer un élément de type `false`.

Le constructeur `⟨_, _⟩` permet ensuite de regrouper ces termes (qui sont des preuves des deux sens de l'équivalence) en une preuve de l'équivalence (donc un terme de type `_ ↔ _`). Le compilateur Lean peut alors constater que le terme donné est bien du type voulu, ce qui prouve la proposition 1.

Remarque. *En Lean,* `¬Q` *est du sucre syntaxique pour* `Q → false` *: pour prouver* $\neg Q$*, il faut et il suffit en effet d'être capable d'obtenir une contradiction en ayant* Q*.* `false` *est quant à lui défini en Lean comme un type sans élément : le seul moyen d'obtenir* `false` *est donc d'appliquer une fonction de la forme* `H : ¬R` *à une hypothèse* `h : R` *pour une certaine propriété* `R`*, ce qui correspond bien mathématiquement à une contraction. Inversement, on a automatiquement la propriété* `false.elim : ∀ {A : Prop}, false → A`*, qui est simplement la fonction vide.*

Remarque. *À partir de* `a : A` *et* `ha : P a`*, on peut former* `⟨a, ha⟩ : ∃ x : A, P x`*. Inversement, à partir de* `E : ∃ x : A, P x`*, on peut obtenir* `classical.some E : A` *et* `classical.some_spec E : P (classical.some E)`*. Lorsque dans la figure 3 on crée le terme* `H _ (classical.some_spec E) q : false`*, on n'a pas besion de spécifier que le premier argument est* `classical.some E`*. En effet, comme* `H` *attend comme premier argument un* `x : A` *et comme deuxième argument un* `hx : P x`*, Lean est capable de déterminer lui-même la valeur du premier argument (ici* `classical.some E`*) à partir du type du deuxième (ici* `P (classical.some E)`*, le type de* `classical.some_spec E`*.)*

2.1.3 Les tactiques

Cependant, les preuves mathématiques ne sont pas en général vues comme la construction d'un terme. Pour écrire des preuves en Lean dans un format plus proche de celui d'une preuve mathématique, on peut utiliser des tactiques. Voici ci-dessous par exemple une preuve mathématique de la proposition 1, puis la même preuve écrite en Lean avec des tactiques :

Démonstration. On prouve l'équivalence $(\forall x \in A, P(x) \implies \neg Q) \iff (Q \implies \neg(\exists x \in A, P(x))$ par double implication.

— $\Rightarrow$: on suppose $\forall x \in A, P(x) \implies \neg Q$ (1), ainsi que Q (2) et $\exists x \in A, P(x)$ (3) et on veut aboutir à une contradiction.
Pour cela, d'après (1) il nous suffit de trouver $x \in A$ tel que $P(x)$ et de prouver Q.
L'hypothèse (3) nous donne un tel x tel que $P(x)$, et l'hypothèse (2) nous donne Q.

— $\Leftarrow$: on suppose $Q \implies \neg(\exists x \in A, P(x))$ (1). Soit $a \in A$ tel que $P(a)$ (2), on veut une contradiction en supposant Q (3).
En appliquant (1) à (3), il nous suffit pour avoir cette contradiction de trouver un $x \in A$ tel que Px.
$x = a$ convient d'après (2).

□

```
example (A : Type) (P : A → Prop) (Q : Prop) :
  (∀ x : A, P x → ¬Q) ↔ Q → ¬(∃ x : A, P x) :=
begin
  split,  -- sépare les 2 sens du ↔
  { intros H q E, -- introduit les hypothèses
    apply H, -- applique une hypothèse
    exact classical.some_spec E,
    exact q },
  { intros H a ha q,
    exact H q ⟨a, ha⟩ }
end
```

FIGURE 4 – Preuve de la proposition 1 en Lean avec des tactiques.

À l'intérieur du `begin...end`, le code est composé de tactiques, c'est-à-dire d'instructions qui modifient l'état de la preuve. Au début (juste après le `begin`), le but est de prouver la proposition 1. L'instruction `split` va alors séparer l'équivalence en deux implications, et le but (indiqué par $\vdash$) sera alors de prouver chacune des deux implications.

```
2 goals
A : Type
P : A → Prop
Q : Prop
⊢ (∀ (x : A), P x → ¬Q) → Q → (¬∃ (x : A), P x)

A : Type
P : A → Prop
Q : Prop
⊢ (Q → (¬∃ (x : A), P x)) → ∀ (x : A), P x → ¬Q
```

FIGURE 5 – État de la preuve après l'instruction split.

Les deux morceaux de la preuve sont ensuite prouvés séparément. Pour le sens $\Rightarrow$, on commence par la tactique `intros`, qui introduit les hypothèses (1), (2) et (3) de la preuve mathématique ci-dessus (nommées `H`, `q` et `E` dans la preuve en Lean). Le but devient alors `false` et les hypothèses `H, Q, E` sont rajoutées au contexte avec `A, P, Q`. On peut ensuite appliquer `H`, qui permet d'obtenir `false` à partir de `P a` pour un `a : A` à déterminer, et `Q`. On utilise ensuite la tactique `exact` pour donner un terme de type correspondant au but actuel (ici `classical.some_spec E` dont le type est de la forme `P a` (en l'occurrence `P (classical.some E)`), puis `q` de type `Q`. L'autre sens de la preuve est similairement conclu avec `intros` et `exact`.

Lean peut alors automatiquement convertir la suite de tactiques en un terme du type désiré : par exemple ici, vu que la preuve commence par la tactique `split`, Lean va générer un terme de la forme `⟨_, _⟩` où les deux morceaux seront les termes dérivés des suites de tactiques prouvant les deux buts générés par la tactique `split`. (Ici, le terme généré sera en fait exactement celui écrit en figure 2.)

2.2 Un exemple plus avancé

2.2.1 La librairie mathématique *mathlib*

Pour prouver des résultats mathématiques plus avancés en Lean, on utilise la librairie mathématique *mathlib* [1]. Cette librairie regroupe les résultats mathématiques déjà prouvés en Lean, ce qui permet ensuite de les utiliser pour prouver des résultats plus avancés. Ma première contribution à *mathlib*, au début de mon stage, a été de définir l'infimum de deux seminormes [2], dont la définition mathématique est rappelée ci-dessous, et dont ma définition en Lean est donnée en figure 6.

Définition 1. *Étant donné deux seminormes p et q sur un espace E, on peut définir*

$$p \wedge q : x \in E \mapsto \bigwedge_{u \in E} p(u) + q(x-u).$$

Cette définition définit bien une seminorme sur E, qui est l'infimum de p et q dans l'ensemble des seminormes sur E muni de l'ordre point à point (i.e. $p \leq q$ ssi $\forall x \in E, p(x) \leq q(x)$.)

Remarque. *La figure 6 ne montre pas la définition actuelle de l'infimum de deux seminormes dans mathlib. En effet, après que j'ai ajouté cette définition à mathlib, la définition de* `seminorm` *dans mathlib a été légèrement modifiée (on commence maintenant par définir une notion plus faible de seminorme sur un groupe additif puis on passe au cas d'un espace vectoriel en ajoutant un axiome sur la multiplication par un scalaire), et la définition de l'infimum a donc dû être adaptée.*

2.2.2 Un langage orienté objets

Lean permet de définir des classes et des instances comme un langage de programmation orienté objets, ce qui est très utile pour formaliser des résultats sur des structures mathématiques. Par exemple, pour définir l'infimum de deux seminormes en Lean, j'ai défini une instance de la classe `has_inf (seminorm k E)`, où la classe `has_inf A` contient un unique champ `inf : A → A → A` pour un type `A`.

```
noncomputable instance : has_inf (seminorm 𝕜 E) :=
{ inf := λ p q,
  { to_fun := λ x, ∏ u : E, p u + q (x-u),
    triangle' := λ x y, begin
      refine le_cinfi_add_cinfi (λ u v, _),
      apply cinfi_le_of_le (bdd_below_range_add _ _ _) (v+u), dsimp only,
      convert add_le_add (p.triangle v u) (q.triangle (y-v) (x-u)) using 1,
      { rw show x + y - (v + u) = y - v + (x - u), by abel },
      { abel },
    end,
    smul' := λ a x, begin
      obtain rfl | ha := eq_or_ne a 0,
      { simp_rw [norm_zero, zero_mul, zero_smul, zero_sub, seminorm.neg],
        refine cinfi_eq_of_forall_ge_of_forall_gt_exists_lt
          (λ i, add_nonneg (p.nonneg _) (q.nonneg _))
          (λ x hx, ⟨0, by rwa [p.zero, q.zero, add_zero]⟩) },
      simp_rw [real.mul_infi_of_nonneg (norm_nonneg a), mul_add, ←p.smul, ←q.smul, smul_sub],
      refine infi_congr ((•) a⁻¹ : E → E) (λ u, ⟨a • u, inv_smul_smulₒ ha u⟩) (λ u, _),
      rw smul_inv_smulₒ ha,
    end } }
```

FIGURE 6 – Définition de l'infimum de seminormes en Lean.

En effet, lorsque Lean va évaluer `a ⊓ b` pour `a b : A`, il va chercher une instance de la classe `has_inf A` (définie plus haut dans le fichier ou dans un fichier importé par le fichier, typiquement dans *mathlib*), et s'il en trouve une comprendre `a ⊓ b` comme étant `has_inf.inf a b`. (Sinon, l'opérateur ⊓ sur `A` n'est pas défini et Lean renvoie donc une erreur). Ainsi, définir une instance de`has_inf (seminorm k E)` permet de définir l'opérateur ⊓ sur `seminorm k E`.

Pour définir cette instance, j'ai donc dû définir la fonction `seminorm.has_inf.inf`, qui prend en arguments `p q : seminorm k E` et qui renvoie une seminorme `p ⊓ q : seminorm k E`. Ici, un objet de type `seminorm k E` est défini comme étant un triplet `⟨to_fun, smul', triangle'⟩` avec `to_fun : E → ℝ` la seminorme à proprement parler, ainsi que
`smul' : ∀ (a : k) (x : E), to_fun (a • x) = ‖a‖ * to_fun x` et

`triangle' : ∀ x y : E, to_fun (x + y) ≤ to_fun x + to_fun y`
qui sont donc des preuves de l'homogénéité et de l'inégalité triangulaire.

Remarque. *Bien qu'une seminorme en Lean soit donc un triplet et pas une fonction, une instance de la classe* `has_coe_to_fun (seminorm k E) (λ _, E → ℝ)` *(définie dans mathlib juste après la définition de* `seminorm`*) permet de donner un sens à* `p u : ℝ` *pour* `p : seminorm k E` *et* `u : E`*, en l'occurrence ici* `p u := p.to_fun u`.

Cette instance de `has_inf _` définit pour l'instant simplement un opérateur binaire sur les seminormes : on a donc à ce stade prouvé que $p \wedge q$ (tel que défini dans la définition 1) est bien une seminorme (en donnant `(p ⊓ q).smul'` et `(p ⊓ q).triangle'`) mais pas que c'est l'infimum de p et q.

J'ai donc ensuite défini une instance de la classe `lattice (seminorm k E)`, qui prouve que l'ensemble des seminormes sur E est muni d'une stucture de treillis avec l'infimum défini ci-dessus et le supremum point à point (qui était déjà défini dans *mathlib*).

```
noncomputable instance : lattice (seminorm k E) :=
{ inf := (⊓),
  inf_le_left := λ p q x, begin
    apply cinfi_le_of_le (bdd_below_range_add _ _ _) x,
    simp only [sub_self, seminorm.zero, add_zero],
  end,
  inf_le_right := λ p q x, begin
    apply cinfi_le_of_le (bdd_below_range_add _ _ _) (0:E),
    simp only [sub_self, seminorm.zero, zero_add, sub_zero],
  end,
  le_inf := λ a b c hab hac x,
    le_cinfi $ λ u, le_trans (a.le_insert' _ _) (add_le_add (hab _) (hac _)),
  ..seminorm.semilattice_sup }
```

FIGURE 7 – Une instance de lattice (seminorm k E).

Ici, en plus des champs déjà trouvés dans l'instance déjà définie dans *mathlib* `seminorm.semilattice_sup : semilattice_sup (seminorm k E)` (donnant la définition de $\leq$ et $\sqcup$ dans `seminorm k E` et prouvant qu'il s'agit bien d'un ordre partiel et d'un supremum), j'ai dû définir les champs `inf` définissant l'infimum (ici, celui déjà défini dans `has_inf`) ainsi que
`inf_le_left : ∀ a b, a ⊓ b ≤ a, inf_le_right : ∀ a b, a ⊓ b ≤ b` et
`le_inf : ∀ a b c, a ≤ b → a ≤ c → a ≤ b ⊓ c` prouvant que $\sqcap$ agit bien comme un infimum.

Grâce à cette instance, on peut ensuite utiliser tous les théorèmes déjà prouvés dans *mathlib* sur les treillis (qui sont donc en fait des méthodes de la classe `lattice`) dans le cas des seminormes.

Remarque. *Dans les figures 6 et 7,* `k` *et* `E` *sont des variables globales plus haut dans le fichier, avec des instances de classes* `[normed_field k] [add_comm_group E] [module k E]` *elles aussi définies comme variables globales. Ces instances sont donc des hypothèses sur* `k` *et* `E` *qui assurent que* `k` *est un corps muni d'une valeur absolue et* `E` *un* `k`*-espace vectoriel. Les instances définies dans les figures 6 et 7 ne sont donc définies pour* `k` *et* `E` *fixés que sous réserve d'existence de ces instances sur* `k` *et* `E`.

3 Structure des modules finiment engendrés sur un anneau principal

3.1 Le résultat prouvé

J'ai consacré la majeure partie de mon stage à prouver en Lean le théorème de structure des modules finiment engendrés dans un anneau principal. Au final, j'ai prouvé la version suivante du théorème :

Théorème 2. *Soit R un anneau principal et N un R-module finiment engendré. Alors N est isomorphe (comme R-module) à un module de la forme*

$$R^n \times \bigoplus_{i \in I} R/p_i^{e_i} R,$$

où I est fini et les p_i sont des irréductibles de R.

Remarque. *Cette décomposition est en fait unique à réindexation des termes et ajout de termes triviaux R/p^0R près. Cette unicité n'a pas été prouvée en Lean.*

On a ici comme hypothèses les instances affirmant que `R` est un anneau principal et `N` est un `R`-module, et l'hypothèse `h'` affirmant que `N` est finiment engendré.

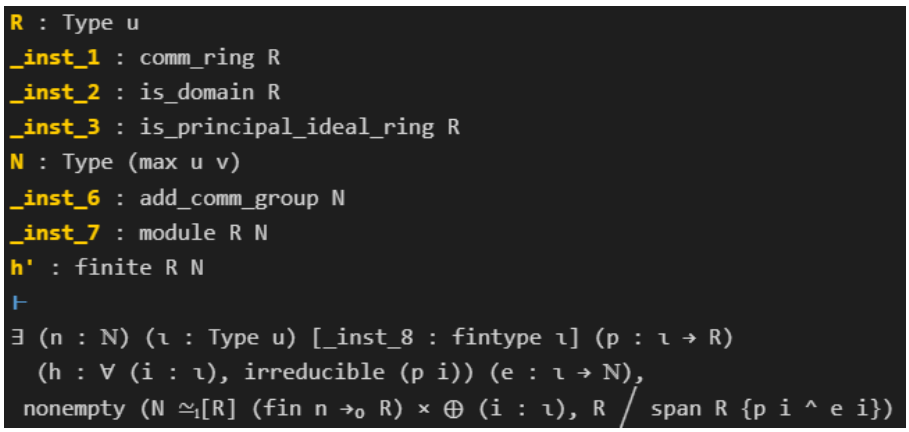

FIGURE 8 – L'énoncé du théorème 2 en Lean.

Remarque. *On a ici* `R : Type u` *et* `N : Type (max u v)`. `u` *et* `v` *sont en Lean des univers, qui existent pour des raisons techniques. En effet, le type de* `Type` *ne peut pas être* `Type` *car ceci mènerait à une contradiction (similaire au problème de l'existence d'un ensemble de tous les ensembles). Le type de* `Type` *est donc un autre type* `Type 1`*, qui vit un univers au dessus de* `Type`*. De même,* `Type 1` *est de type* `Type 2`*, un type vivant un univers au dessus de* `Type 1`*. Ici, on doit supposer pour des raisons techniques que* `N` *vit dans un univers supérieur ou égal à* `R`*. En pratique, cela ne pose en général pas de problème car on utilisera le théorème pour* `R` *et* `N` *de type* `Type` *donc vivant dans le même univers.*

La preuve du théorème 2 que j'ai formalisée dans *mathlib* est donnée ci-dessous, d'abord en Lean puis sous forme de preuve mathématique.

```
theorem equiv_free_prod_direct_sum [h' : module.finite R N] :
  ∃ (n : ℕ) (ι : Type u) [fintype ι] (p : ι → R) (h : ∀ i, irreducible $ p i) (e : ι → ℕ),
  nonempty $ N ≃ₗ[R] (fin n →₀ R) × ⨁ (i : ι), R ⧸ R ∙ (p i ^ e i) :=
begin
  haveI := is_noetherian_of_fg_of_noetherian' (module.finite_def.mp h'),
  haveI := is_noetherian_submodule' (torsion R N),
  haveI := module.finite.of_surjective _ (torsion R N).mkq_surjective,
  obtain ⟨I, fI, p, hp, e, ⟨h⟩⟩ := equiv_direct_sum_of_is_torsion (@torsion_is_torsion R N _ _ _),
  obtain ⟨n, ⟨g⟩⟩ := @module.free_of_finite_type_torsion_free' R _ _ _ (N ⧸ torsion R N) _ _ _ _,
  haveI : module.projective R (N ⧸ torsion R N) := module.projective_of_basis ⟨g⟩,
  obtain ⟨f, hf⟩ := module.projective_lifting_property _ linear_map.id (torsion R N).mkq_surjective,
  refine ⟨n, I, fI, p, hp, e,
    ⟨(lequiv_prod_of_right_split_exact (torsion R N).injective_subtype _ hf).symm.trans $
      (h.prod g).trans $ linear_equiv.prod_comm R _ _⟩⟩,
  rw [range_subtype, ker_mkq]
end
```

FIGURE 9 – La preuve du théorème 2 en Lean.

Démonstration. (du théorème 2)
Soit T le sous-module de torsion de N (contenant les $x \in N$ tels qu'il existe $a \in R$ non nul tel que $a \cdot x = 0$, noté `torsion R N` dans *mathlib*).

Comme R est noetherien et N est finiment engendré, N est noetherien donc T est finiment engendré. Par ailleurs, N/T est finiment engendré comme quotient d'un module finiment engendré.

Comme T est finiment engendré et de torsion sur R principal, par le lemme 3 ci-dessous (`equiv_direct_sum_of_is_torsion` en Lean), T est isomorphe à un $\bigoplus_{i\in I} R/p_i^{e_i}R$ pour un I fini et des p_i irréductibles de R.

Comme N/T est finiment engendré et sans torsion sur R principal, par le lemme 4 (`module.free_of_finite_type_torsion_free'` en Lean), N/T est libre donc isomorphe à un R^n.

Enfin, comme N/T est libre par ce qui précède donc projectif, on peut trouver un inverse à droite à la surjection naturelle $N \to N/T$. Ainsi, la suite exacte courte $0 \to T \to N \to N/T \to 0$ est scindée à droite.

Donc par le lemme 5, on a

$$N \simeq T \times N/T \simeq R^n \times \bigoplus_{i \in I} R/p_i^{e_i} R.$$

□

La preuve repose donc sur trois lemmes :

Lemme 3. *Si R est un anneau principal et N est un R-module de torsion finiment engendré, alors N est isomorphe à un*

$$\bigoplus_{i \in I} R/p_i^{e_i} R$$

pour un I fini et des p_i irréductibles de R.

Lemme 4. *Si R est un anneau principal, tout R-module sans torsion finiment engendré est libre.*

Lemme 5. *Si R est un anneau (quelconque) et $0 \to A \to M \to B \to 0$ est une suite exacte scindée à droite de R-modules (avec donc $j : A \to M$ injective, $g : M \to B$ surjective et $f : B \to M$ avec $\ker g = \operatorname{im} j$ et $g \circ f = \mathrm{id}_B$), alors on a un isomorphisme linéaire*

$$A \times B \simeq M.$$

Le lemme 4 était déjà prouvé dans *mathlib* ; j'ai implémenté la preuve des lemmes 3 et 5.

3.2 Preuve du lemme 5

Pour prouver le lemme 5, je me suis placé dans un contexte de théorie des catégories. J'ai uilisé la notion de catégorie abélienne (déjà définie dans *mathlib*), dont la définition est rappelée ci-dessous (définition 4) :

Définition 2. *Une catégorie C est dite préadditive si pour tous $X, Y \in C$, on peut munir $\mathrm{Hom}(X, Y)$ d'une structure de groupe abélien compatible avec la composition (c'est à dire que pour tout $f \in \mathrm{Hom}(X, Y)$ et pour tout Z, la composition à gauche (resp. à droite) par f définit un morphisme de groupes $\mathrm{Hom}(Z, X) \to \mathrm{Hom}(Z, Y)$ (resp. $\mathrm{Hom}(Y, Z) \to \mathrm{Hom}(X, Z)$).)*

Remarque. *Dans une catégorie préadditive, on peut donc parler pour tous X, Y de 0_{XY}, le neutre du groupe abélien $\mathrm{Hom}(X, Y)$. On a alors pour tout $f \in \mathrm{Hom}(X, Y)$,*

$$0_{YZ} \circ f = 0_{XZ} \text{ et } f \circ 0_{ZX} = 0_{ZY}.$$

On dit qu'une catégorie dans laquelle on peut définir de tels 0_{XY} ayant ces propriétés possède des zéros. (Les catégories préadditives possèdent donc toujours des zéros).

Définition 3. *Soit $f \in \mathrm{Hom}(X, Y)$ dans une catégorie possédant des zéros. Un noyau de f est un objet K muni d'un morphisme $k \in \mathrm{Hom}(K, X)$ vérifiant la propriété universelle suivante :*

$$f \circ k = 0_{KY} \text{ et } \forall L, l \in \mathrm{Hom}(L, X), f \circ l = 0_{LY} \implies \exists! u \in \mathrm{Hom}(L, K), k \circ u = l.$$

De même, un conoyau de f est un objet C muni d'un morphisme $c \in \mathrm{Hom}(Y, C)$ vérifiant la propriété universelle suivante :

$$c \circ f = 0_{XC} \text{ et } \forall L, l \in \mathrm{Hom}(Y, L), l \circ f = 0_{XL} \implies \exists! u \in \mathrm{Hom}(C, L), u \circ uc = l.$$

Exemple. *Dans la catégorie des groupes abéliens (ou des R-modules), étant donné $f : X \to Y$, l'injection $\ker f \hookrightarrow X$ est un noyau de f et la surjection $Y \twoheadrightarrow Y/\operatorname{im} f$ est un conoyau de f.*

Définition 4. *Une catégorie préadditive est dite abélienne si elle possède des produits finis, tout morphisme a un noyau et un conoyau, tout monomorphisme est un noyau et tout épimorphisme est un conoyau.*

Exemple. *La catégorie des groupes abéliens, ainsi que celle des R-modules pour un anneau R, sont abéliennes.*

Dans une catégorie abélienne, on peut alors définir une notion de suite exacte, et donc énoncer le

Lemme 6. *Si $0 \to A \to B \to C \to 0$ est une suite exacte scindée à droite (par un morphisme $C \to B$) dans une catégorie abélienne, alors on a un isomorphisme $A \boxplus C \simeq B$.*

```
noncomputable def lequiv_prod_of_right_split_exact {f : B →ₗ[R] M}
  (hj : function.injective j) (exac : j.range = g.ker) (h : g.comp f = linear_map.id) :
  (A × B) ≃ₗ[R] M :=
(({ right_split := ⟨as_hom f, h⟩,
    mono := (Module.mono_iff_injective $ as_hom j).mpr hj,
    exact := (exact_iff _ _).mpr exac } : right_split _ _).splitting.iso.trans $
  biprod_iso_prod _ _).to_linear_equiv.symm
```

FIGURE 10 – La preuve du lemme 5 dans mathlib.

Le lemme 6 n'était pas prouvé dans *mathlib*, mais le même résultat était prouvé dans le cas d'une suite exacte scindée à gauche (par un morphisme $B \to A$). J'ai donc adapté cette preuve au cas d'une suite exacte scindée à droite pour prouver le lemme 6, puis je l'ai appliqué au cas de la catégorie des modules pour prouver le lemme 5 (et un lemme similaire pour les suites exactes scindées à gauche de modules). [6]

La preuve du lemme 5 commence donc par construire une suite exacte scindée à droite dans la catégorie `Module R` des R-modules à partir des hypothèses, puis par lui appliquer la méthode `right_split.splitting.iso` (c'est-à-dire le lemme 6) pour obtenir un isomorphisme `of R M ≃ of R A ⊞ of R B` dans la catégorie des R-modules. (Ici, `of R` est l'opérateur permettant de convertir un R-module en un objet de `Module R`)

Ensuite, on compose cet isomorphisme avec l'isomorphisme naturel `biprod_iso_prod : of R A ⊞ of R B ≃ of R (A × B)` dans la catégorie `Module R`, pour pouvoir le convertir ensuite (via la méthode `to_linear_equiv`) en un isomorphisme linéaire `M ≃ₗ[R] (A × B)`.

Remarque. *Pour écrire la preuve du lemme 6 en Lean, je me suis appuyé sur la preuve déjà faite du cas des suites exactes scindées à gauche et sur l'intuition du cas des groupes abéliens. L'utilisation de l'assistant de preuve Lean a alors permis de m'assurer qu'il n'y avait pas d'erreur dans ma preuve du lemme 6.*

3.3 Les modules de torsion

Avant de prouver le lemme 3, j'ai dû définir les notions de modules et sous-modules de torsion dans *mathlib*. [3] En effet, bien que la notion de module sans torsion était définie dans *mathlib* par la classe `no_zero_smul_divisors R M` ayant un unique champ `eq_zero_or_eq_zero_of_smul_eq_zero : ∀ {c : R} {x : M}, c • x = 0 → c = 0 ∨ x = 0` (ce qui permettait d'énoncer le lemme 4), il n'y avait pas de définition pour les modules de torsion.

```
/-- The `a`-torsion submodule for `a` in `R`, containing all elements `x` of `M` such that
  `a • x = 0`. -/
@[simps] def torsion_by (a : R) : submodule R M := (distrib_mul_action.to_linear_map _ _ a).ker

/-- The submodule containing all elements `x` of `M` such that `a • x = 0` for all `a` in `s`. -/
@[simps] def torsion_by_set (s : set R) : submodule R M := Inf (torsion_by R M '' s)

/-- The `S`-torsion submodule, containing all elements `x` of `M` such that `a • x = 0` for some
`a` in `S`. -/
@[simps] def torsion' (S : Type*)
  [comm_monoid S] [distrib_mul_action S M] [smul_comm_class S R M] :
  submodule R M :=
{ carrier := { x | ∃ a : S, a • x = 0 },
  zero_mem' := ⟨1, smul_zero _⟩,
  add_mem' := λ x y ⟨a, hx⟩ ⟨b, hy⟩,
    ⟨b * a,
      by rw [smul_add, mul_smul, mul_comm, mul_smul, hx, hy, smul_zero, smul_zero, add_zero]⟩,
  smul_mem' := λ a x ⟨b, h⟩, ⟨b, by rw [smul_comm, h, smul_zero]⟩ }

/-- The torsion submodule, containing all elements `x` of `M` such that  `a • x = 0` for some
  non-zero-divisor `a` in `R`. -/
@[reducible] def torsion := torsion' R M R⁰
```

FIGURE 11 – Diverses notions de sous-modules de torsion.

Remarque. *Les définitions ci-dessus sont définies avec comme variables globales* `(R M : Type*) [comm_semiring R] [add_comm_monoid M] [module R M]`*, donc R un semianneau commu-*

tatif et M un R-semimodule. En effet, supposer l'existence d'inverses additifs dans R et M n'est pas nécessaire pour définir les sous-modules de torsion, mais la commutativité de la multiplication dans R l'est.

Ici, `torsion_by R M a` est défini comme le noyau de l'application linéaire de multiplication à gauche par `a`, ce qui permet de l'obtenir directement comme un sous-module de `M` (plutôt que comme un sous-ensemble dont il faudrait ensuite prouver que c'est un sous-module). De même, `torsion_by_set R M s` est défini comme l'infimum (donc l'intersection) des `torsion_by R M a` pour `a ∈ s`, ce qui permet de l'obtenir comme un sous-module directement (comme intersection de sous-modules).

Ensuite, `torsion' R M S` est défini pour S un monoïde commutatif agissant R-linéairement sur M, comme étant l'ensemble des $x \in M$ tels que $\exists a \in S, a \cdot x = 0$ (qui est bien un sous-module de M sous les hypothèses données). Ici, on ne peut pas l'obtenir facilement directement comme un sous-module, j'ai donc dû construire le sous-module en Lean sous la forme d'un quadruplet `⟨carrier, zero_mem', add_mem', smul_mem'⟩` où `carrier` est l'ensemble des éléments du sous-module et `zero_mem', add_mem', smul_mem'` sont des preuves axiomes de stabilité d'un sous-module.

Enfin, j'ai défini `torsion R M := torsion' R M R⁰`, où `R⁰ : submonoid R` est le sous-monoïde multiplicatif de `R` composé des éléments non diviseurs de zéro. (Dans le cas d'un anneau intègre, on obtient bien le sous-module de torsion de R). Ici, Lean est capable d'inférer (grâce à des définitions présentes ailleurs dans *mathlib*) les instances `[comm_monoid R⁰] [distrib_mul_action R⁰ M] [smul_comm_class R⁰ R M]` à partir du fait que `R⁰` soit un sous-monoïde de `R` commutatif, qui agit donc sur `M` par restriction de l'action de `R`.

Remarque. *Si R n'est pas intègre, l'ensemble des $x \in M$ tels que $\exists a \in R, a \neq 0 \wedge a \cdot x = 0$ n'est en général pas un sous-module de M : par exemple pour $R = M = \mathbb{Z}/6\mathbb{Z}$, on obtient $\{0, 2, 3, 4\}$ qui n'est pas un idéal de $R = M$.*

J'ai ensuite défini des notions de modules de torsion, qui sont donc des propriétés sur les modules (qui prennent donc un module en argument et renvoient une `Prop`, comme par exemple `is_torsion_by : ∏ (R M : Type*) [comm_semiring R] [add_comm_monoid M] [module R M], R → Prop`).

```
/-- A `a`-torsion module is a module where every element is `a`-torsion. -/
@[reducible] def is_torsion_by (a : R) := ∀ {x : M}, a • x = 0

/-- A module where every element is `a`-torsion for all `a` in `s`. -/
@[reducible] def is_torsion_by_set (s : set R) := ∀ {x : M} {a : s}, (a : R) • x = 0

/-- A `S`-torsion module is a module where every element is `a`-torsion for some `a` in `S`. -/
@[reducible] def is_torsion' (S : Type*) [has_smul S M] := ∀ {x : M}, ∃ a : S, a • x = 0

/-- A torsion module is a module where every element is `a`-torsion for some non-zero-divisor `a`.
-/
@[reducible] def is_torsion := ∀ {x : M}, ∃ a : R⁰, a • x = 0
```

FIGURE 12 – Diverses notions de modules de torsion.

J'ai ensuite écrit et prouvé des lemmes de base sur les modules de torsion, comme par exemple :

- `mem_torsion_by_iff : ∀ (a : R) (x : M), x ∈ torsion_by R M a ↔ a • x = 0` (vrai par définition de `torsion_by`)
- `torsion_is_torsion : is_torsion R (torsion R M)` (on a déjà dans *mathlib* une instance de `module R N` pour `N : submodule R M`, qu'on utilise ici implicitement pour donner un sens à `torsion R M` comme `R`-module)
- une instance de `no_zero_smul_divisors R (M/torsion R M)` quand `R` est supposé intègre (remarque : on a en fait utilisé cette instance dans la preuve du théorème 2 (figure 9) pour pouvoir appliquer le lemme 4 à `N/torsion R N`)
- `torsion_by_set_eq_torsion_by_span :`
  `∀ s : set R, torsion_by_set R M s = torsion_by_set R M (ideal.span s)`
  (les éléments annulés par tous les éléments de `s` sont ceux annulés par tout l'idéal engendré par `s`)

Ces lemmes de base permettent ensuite de manipuler les notions de torsion dans des preuves de résultats plus avancés, sans avoir à ressortir à chaque fois les définitions brutes (de la figure 11) de ces objets.

3.4 Preuve du lemme 3

Après avoir défini ainsi les modules et sous-modules de torsion, j'ai pu ensuite formaliser la preuve du lemme 3. Cette preuve repose sur les trois lemmes ci-dessous :

Lemme 7. *Si M est un R-module finiment engendré (R anneau commutatif quelconque) et de torsion (i.e. on a* `is_torsion R M` *avec la définition donnée en figure 12), alors il existe un idéal I de R contenant un élément non diviseur de zéro et tel que $\forall x \in M, \forall a \in I, a \cdot x = 0$ (i.e.* `is_torsion_by_set R M I`*.)*

Démonstration. Soit (u_i) une famille génératrice finie de M comme R-module et I_i l'idéal annulateur de u_i. Soit I l'intersection des I_i.

Comme M est de torsion, il existe pour tout i un a_i non diviseur de zéro tel que $a_i \cdot u_i = 0$, donc $a_i \in I_i$. On a alors $\prod_i a_i \in \bigcap_i I_i = I$, et $\prod_i a_i$ non diviseur de zéro dans R comme produit d'éléments non diviseurs de zéro.

De plus, soit $N =$ `torsion_by_set R M I`, alors pour tout i, comme $I \subset I_i$, on a $u_i \in N$. Comme les u_i engendrent M, on en déduit $N = M$, ce qui conclut. □

Remarque. *L'implémentation de cette preuve en Lean ne présente pas de difficulté et n'est pas montrée ici.*

Lemme 8. *Si R est un anneau principal, $a \in R$ non nul et M un R-module de a-torsion, avec $a = \prod_i p_i^{e_i}$ sa décomposition en facteurs premiers, alors M se décompose comme somme directe interne de ses sous-modules de $p_i^{e_i}$-torsion (notés $T_{p_i^{e_i}}$), c'est-à-dire que la fonction*

$$f \in \bigoplus_i T_{p_i^{e_i}} \mapsto \sum_i f_i \in M$$

définit un isomorphisme.

Lemme 9. *Si R est un anneau principal, p un irréductible de R et N est un R-module de p^∞-torsion (c'est-à-dire $\forall x \in N, \exists n \in \mathbb{N}, p^n \cdot x = 0$) engendré par d éléments, alors N est isomorphe à un*

$$\bigoplus_{i=1}^{d} R/p^{e_i}R$$

pour certains $e_i \in \mathbb{N}$.

On peut à partir de ces trois lemmes démontrer le lemme 3 rappelé ci-dessous :

Lemme 10. *Si R est un anneau principal et N est un R-module de torsion finiment engendré, alors N est isomorphe à un*

$$\bigoplus_{i \in I} R/p_i^{e_i}R$$

pour un I fini et des p_i irréductibles de R.

Démonstration. Par le lemme 7, N est de I-torsion pour un idéal non nul I. Comme R est principal, I est engendré par un unique élément a (non nul car I non nul) donc N est en fait de a-torsion.

Par le lemme 8, on a alors

$$N \simeq \bigoplus_i T_{p_i^{e_i}}$$

, où T_r et le sous-module de r-torsion de N et $\prod_i p_i^{e_i}$ la décompositions en facteurs premiers de a.

De plus, chaque $T_{p_i^{e_i}}$ est de p_i^∞-torsion, et finiment engendré comme sous-module de N par notherianité, donc on peut appliquer le lemme 9 pour avoir

$$T_{p_i^{e_i}} \simeq \bigoplus_{j=1}^{d_i} R/p_i^{f_{ij}}R$$

pour certains d_i, f_{ij}.

On peut alors obtenir N comme somme directe des $T_{p_i^{e_i}}$ qui sont eux-mêmes des sommes directes de modules de la forme $R/p_i^k R$, ce qui conclut. □

La preuve en Lean commence par appliquer les lemmes 8 (`submodule.is_internal_prime_power_torsion_of_pid`) et 9 (`torsion_by_prime_power_decomposition`) pour obtenir les isomorphismes $N \simeq \bigoplus_i T_{p_i^{e_i}}$ et $T_{p_i^{e_i}} \simeq \bigoplus_{j=1}^{d_i} R/p_i^{f_{ij}}R$, puis avec l'instruction `refine ⟨..., ⟨...⟩⟩` construit des valeurs de J, q_j, k_j et un isomorphisme $N \simeq \bigoplus_{j \in J} R/q_j^{k_j}R$ à partir de ces isomorphismes.

Pour construire cet isomorphisme, j'ai eu besoin d'utiliser `direct_sum.sigma_lcurry_equiv R`, qui offre un isomorphisme linéaire

$$\bigoplus_{j \in \bigsqcup_i J_i} M_j \simeq \bigoplus_{i \in I} \bigoplus_{j \in J_i} M_j$$

```
theorem equiv_direct_sum_of_is_torsion [h' : module.finite R N] (hN : module.is_torsion R N) :
  ∃ (ι : Type u) [fintype ι] (p : ι → R) (h : ∀ i, irreducible $ p i) (e : ι → ℕ),
  nonempty $ N ≃ₗ[R] ⨁ (i : ι), R ⧸ R ∙ (p i ^ e i) :=
begin
  obtain ⟨I, fI, _, p, hp, e, h⟩ := submodule.is_internal_prime_power_torsion_of_pid hN,
  haveI := fI,
  have : ∀ i, ∃ (d : ℕ) (k : fin d → ℕ),
    nonempty $ torsion_by R N (p i ^ e i) ≃ₗ[R] ⨁ j, R ⧸ R ∙ (p i ^ k j),
  { haveI := is_noetherian_of_fg_of_noetherian' (module.finite_def.mp h'),
    haveI := λ i, is_noetherian_submodule' (torsion_by R N $ p i ^ e i),
    exact λ i, torsion_by_prime_power_decomposition (hp i)
      ((is_torsion'_powers_iff $ p i).mpr $ λ x, ⟨e i, smul_torsion_by _ _⟩) },
  refine ⟨Σ i, fin (this i).some, infer_instance,
    λ ⟨i, j⟩, p i, λ ⟨i, j⟩, hp i, λ ⟨i, j⟩, (this i).some_spec.some j,
    ((linear_equiv.of_bijective (direct_sum.coe_linear_map _) h.1 h.2).symm.trans $
      (dfinsupp.map_range.linear_equiv $ λ i, (this i).some_spec.some_spec.some).trans $
      (direct_sum.sigma_lcurry_equiv R).symm.trans
      (dfinsupp.map_range.linear_equiv $ λ i, quot_equiv_of_eq _ _ _))⟩,
  cases i with i j, simp only
end
```

FIGURE 13 – La preuve du lemme 3 dans mathlib.

pour passer d'une somme directe de somme directe à une seule somme directe (en utilisant en fait l'isomorphisme inverse).

L'isomorphisme `direct_sum.sigma_lcurry_equiv R` n'était pas implémenté dans *mathlib* ; je l'ai donc implémenté moi-même [4]. La définition de `direct_sum.sigma_lcurry_equiv` est montrée ci-dessous.

```
noncomputable def sigma_lcurry_equiv : (⨁ (i : Σ i, _), δ i.1 i.2) ≃ₗ[R] ⨁ i j, δ i j :=
{ ..sigma_curry_equiv, ..sigma_lcurry R }
```

FIGURE 14 – Une somme directe de somme directes est une somme directe.

Ici, on a déjà défini la fonction naturelle `⨁ (i : Σ i, α i), δ i.1 i.2 → ⨁ i (j : α i), δ i j` et prouvé qu'il s'agissait d'un isomorphisme de groupes (`sigma_curry_equiv`) et une application `R`-linéaire (`sigma_lcurry R`). On peut donc directement en déduire qu'on a un isomorphisme linéaire `sigma_lcurry_equiv`. La définition de la fonction `sigma_curry` elle-même (avant ajout d'une structure de morphisme) est donnée ci-dessous (page suivante) :

On manipule ici des éléments de somme directe, qui sont donc des suites à support fini. J'ai donc dû définir `sigma_curry f` comme une suite à support fini (dont les valeurs sont elles-mêmes des suites à support fini, puisqu'on a ici une double somme directe). Ainsi, j'ai construit `sigma_curry f` à partir de la fonction `mk`, qui étant donné un ensemble fini E (ici défini à partir du support de f) et une fonction g définie sur E (ici définie à partir de f), construit la suite à support fini qui vaut g sur E et 0 ailleurs.

J'ai ensuite prouvé que l'on a toujours `sigma_curry f i j = f ⟨i, j⟩` (lemme `sigma_curry_apply` dans la figure 15), en distinguant selon que `⟨i, j⟩` est dans le support de `f` (auquel cas `sigma_curry f i j` est défini comme égal à `f ⟨i, j⟩`) ou non (auquel cas `sigma_curry f i j` est défini comme égal à 0, mais on a aussi `f ⟨i, j⟩ = 0`.) Ceci permet de prouver que `sigma_curry` fonctionne bien comme attendu et donc ensuite qu'il préserve les structures de groupe ou de R-module. J'ai ensuite défini de même la fonction réciproque `sigma_uncurry` pour prouver la bijectivité de `sigma_curry` et obtenir un isomorphisme.

Remarque. *Dans l'utilisation de* `direct_sum.sigma_lcurry_equiv` *faite dans la figure 13, on ne manipule que des sommes directes finies, ce qui aurait pu être utilisé pour éviter ces considérations sur les suites à support fini. Cependant, définir* `sigma_curry` *dans mathlib pour des sommes directes finies ou infinies permettra de réutiliser cet isomorphisme dans d'autres preuves faisant intervenir des sommes directes infinies.*

J'ai aussi créé des isomorphismes similaires pour la réindexation de sommes directes ou l'ajout d'un terme dans une somme directe, qui seront utiles dans la preuve du lemme 9. [4]

```
/--The natural map between `Π₀ (i : Σ i, α i), δ i.1 i.2` and `Π₀ i (j : α i), δ i j`.  -/
noncomputable def sigma_curry [Π i j, has_zero (δ i j)] (f : Π₀ (i : Σ i, _), δ i.1 i.2) :
  Π₀ i j, δ i j :=
by { classical,
  exact mk (f.support.image $ λ i, i.1)
    (λ i, mk (f.support.preimage (sigma.mk i) $ sigma_mk_injective.inj_on _) $ λ j, f ⟨i, j⟩) }

@[simp] lemma sigma_curry_apply [Π i j, has_zero (δ i j)] (f : Π₀ (i : Σ i, _), δ i.1 i.2)
  (i : ι) (j : α i) :
  sigma_curry f i j = f ⟨i, j⟩ :=
begin
  dunfold sigma_curry, by_cases h : f ⟨i, j⟩ = 0,
  { rw [h, mk_apply], split_ifs, { rw mk_apply, split_ifs, { exact h }, { refl } }, { refl } },
  { rw [mk_of_mem, mk_of_mem], { refl },
    { rw [mem_preimage, mem_support_to_fun], exact h },
    { rw mem_image, refine ⟨⟨i, j⟩, _, rfl⟩, rw mem_support_to_fun, exact h } }
end
```

FIGURE 15 – La fonction sigma_curry que j'ai définie dans mathlib.

3.5 Preuve du lemme 8

J'ai prouvé le lemme 8 dans le contexte général d'un anneau de Dedekind R, dont la définition est rappelée ci-dessous :

Définition 5. *Un anneau intègre R est dit de Dedekind si tout idéal non nul peut être décomposé comme produit d'idéaux maximaux. (Cette décomposition est alors nécessairement unique.)*

Dans un anneau de Dedekind, on a alors la généralisation du lemme 8 :

Lemme 11. *Si R est un anneau de Dedekind, I idéal de R non nul et M un R-module de I-torsion (c'est-à-dire* `is_torsion_by_set R M I`*), avec $I = \prod_i P_i^{e_i}$ sa décomposition en facteurs premiers, alors M se décompose comme somme directe interne de ses sous-modules de $P_i^{e_i}$-torsion.*

Remarque. *En Lean, j'ai en fait commencé par prouver le lemme 11, puis j'ai appliqué le lemme 7 pour passer au cas d'un module de torsion finiment engendré sur un anneau de Dedekind, puis j'ai appliqué au cas d'un anneau principal pour obtenir le lemme* `is_internal_prime_power_torsion_of_pid` *utilisé plus haut.*

En fait, le lemme 11 peut encore être généralisé, avec la version ci-dessous.

Lemme 12. *Soit R un anneau commutatif (quelconque), (P_i) une famille finie d'idéaux de R deux à deux premiers entre eux (c'est-à-dire que pour tous i et j, P_i et P_j engendrent R). Soit M un R-module de $\bigcap_i P_i$-torsion. Alors M se décompose comme somme directe interne de ses sous-modules de P_i-torsion.*

Remarque. *Pour des idéaux P_i deux à deux premiers entre eux, on a $\bigcap_i P_i = \prod_i P_i$. En effet, étant donné deux idéaux I, J premiers entre eux, on a $i \in I$ et $j \in J$ tels que $i + j = 1$. Ainsi, pour $k \in I \cap J$, on a $k = ik + kj \in IJ$ (et inversement on a toujours $IJ \subset I \cap J$). Le résultat se généralise à un nombre fini d'idéaux par récurrence.*

Pour passer du lemme 12 au lemme 11, il suffit alors de remarquer que pour $P_i \neq P_j$ des idéaux maximaux distincts, on a toujours $P_i^{e_i}$ et $P_j^{e_j}$ premiers entre eux deux à deux. En effet, on a alors $P_i \subsetneq P_i + P_j$, donc comme P_i maximal on a $P_i + P_j = R$. Soit alors $p \in P_i, q \in P_j$ avec $p + q = 1$, on a alors $1 = (p+q)^{e_i+e_j} \in P_i^{e_i} + P_j^{e_j}$ en développant le binôme, donc ces idéaux sont premiers entre eux.

Pour prouver le lemme 12 en Lean, on utilise une caractérisation de la décomposition en somme directe interne (qui était déjà prouvée dans *mathlib*) :

Théorème 13. *Soit R un anneau et M un R-module, et (A_i) une famille de sous-modules de M. Alors M se décompose en somme directe interne des A_i si et seulement si on a les deux conditions suivantes :*

1. *les A_i sont indépendants, c'est-à-dire que pour tout i, $A_i \cap \bigvee_{j \neq i} A_j = \{0\}$, où $\bigvee A_j$ est le sous-module engendré par les A_j.*
2. *les A_i engendrent M, i.e. $\bigvee_i A_i = M$.*

La preuve du lemme 12 peut alors être conclue avec les trois lemmes ci-dessous, dont j'ai implémenté la preuve dans *mathlib* :

Lemme 14. *Soit (I_i) une famille finie non vide d'idéaux d'un anneau commutatif R. Alors les I_i sont premiers entre eux deux à deux si et seulement si les $\bigcap_{j\neq i} I_j$ engendrent R.*

Démonstration. .

— $\Rightarrow$: si les I_i sont premiers entre eux deux à deux, on peut trouver pour tous $i \neq j$, des $a_{ij} \in I_i, a_{ji} \in I_j$ tels que $a_{ij} + a_{ji} = 1$. En développant le produit

$$1 = \prod_{i\neq j}(a_{ij} + a_{ji})$$

, on obtient une somme de termes. De plus, pour tous $i \neq j$, chacun de ces termes est un multiple de a_{ij} ou de a_{ji}, donc dans I_i ou dans I_j.Ainsi, chacun de ces termes est dans tous les I_i sauf au plus un, donc est dans l'un des $\bigcap_{j\neq i} I_j$ (on a ici utilisé que la famille des (I_i) était non vide). On a donc exprimé 1 comme somme d'éléments des $\bigcap_{j\neq i} I_j$, ce qui prouve qu'ils engendrent R.

— $\Leftarrow$: Si les $\bigcap_{j\neq i} I_j$ engendrent R, alors on, peut écrire

$$1 = \sum_i a_i,$$

avec $a_i \in \bigcap_{j\neq i} I_j$ pour tout i. On a alors pour $i \neq j$, $a_i \in \bigcap_{j\neq i} I_j \subset I_j$ et $\sum_{k\neq i} a_k \in I_i$, donc

$$1 = \sum_{k\neq i} a_k + a_i \in I_i + I_j$$

et I_i et I_j sont premiers entre eux.

□

Remarque. *Le développement du produit avec l'argument "tous les termes de la somme obtenue ne peuvent pas ne pas être dans deux idéaux distincts" n'est pas facilement formalisable en Lean. J'ai donc à la place prouvé le résultat en Lean par récurrence sur la taille de la famille (I_i).*

Lemme 15. *Soit R un anneau commutatif, M un R-module et (P_i) une famille finie d'idéaux de R premiers entre eux deux à deux. Notons T_I le sous-module de I-torsion de M pour un idéal I. On a alors*

$$\bigvee_i T_{P_i} = T_{\bigcap_i P_i}.$$

Démonstration. Si (P_i) est la famille vide, on a bien $\bigvee_i T_{P_i} = T_{\bigcap_i P_i}$.

Sinon, on a déjà pour tout $i, T_{P_i} \subset T_{\bigcap_i P_i}$ donc $\bigvee_i T_{P_i} \subseteq T_{\bigcap_i P_i}$. Réciproquement, soit $x \in T_{\bigcap_i P_i}$, on cherche alors à montrer $x \in \bigvee_i T_{P_i}$.

Pour cela, utilisons le lemme 14 pour obtenir une décomposition $1 = \sum_i a_i$ avec $a_i \in \bigcap_{j\neq i} P_j$ pour tout i. On a alors

$$x = \sum_i a_i \cdot x,$$

où $a_i \cdot x \in T_{P_i}$ pour tout i, ce qui conclut. En effet, pour tout $b \in P_i$, on a $b \cdot (a_i \cdot x) = (ba_i) \cdot x = 0$, car $ba_i \in P_i \cap \bigcap_{j\neq i} P_j = \bigcap_j P_j$ et $x \in T_{\bigcap_j P_j}$. □

Lemme 16. *Sous les hypothèses du lemme 15, les T_{P_i} sont indépendants.*

Démonstration. Pour tout i, on a

$$\begin{aligned} T_{P_i} \cap \bigvee_{j\neq i} T_{P_j} &= T_{P_i} \cap T_{\bigcap_{j\neq i} P_j} \text{ par le lemme 15} \\ &= T_{P_i + \bigcap_{j\neq i} P_j} \text{ car on a toujours } T_I \cap T_J = T_{I+J} : \text{un élément est annulé par } I \text{ et } J \text{ ssi il est annulé par } I + J \\ &= T_R \text{ car } P_i \text{ est premier avec tous les autres } P_j, \text{ donc avec leur intersection} \\ &= \{0\} \text{ car si } x \in T_R, \text{ on a } x = 1 \cdot x = 0. \end{aligned}$$

donc les T_i sont indépendants. □

Pour prouver le lemme 12, il suffit alors d'appliquer le théorème 13 (`direct_sum.is_internal_submodule_of_independent_of_supr_eq_top`), et les deux conditions recherchées sur les $A_i = T_{P_i}$ nous sont alors données par les lemmes 16 (`sup_indep_torsion_by_ideal`) et 15 (`supr_torsion_by_ideal_eq_torsion_by_infi` ; ici, on a $T_{\bigcap_i P_i} = M$ pour M de $\bigcap_i P_i$-torsion).

```
lemma torsion_by_set_is_internal {p : ι → ideal R}
  (hp : (S : set ι).pairwise $ λ i j, p i ⊔ p j = ⊤)
  (hM : module.is_torsion_by_set R M (∏ i ∈ S, p i : ideal R)) :
  direct_sum.is_internal (λ i : S, torsion_by_set R M $ p i) :=
direct_sum.is_internal_submodule_of_independent_of_supr_eq_top
  (complete_lattice.independent_iff_sup_indep.mpr $ sup_indep_torsion_by_ideal hp)
  ((supr_subtype'' ↑S $ λ i, torsion_by_set R M $ p i).trans $
    (supr_torsion_by_ideal_eq_torsion_by_infi hp).trans $
    (module.is_torsion_by_set_iff_torsion_by_set_eq_top _).mp hM)
```

FIGURE 16 – La preuve du lemme 12 repose sur le théorème 13.

3.6 Preuve du lemme 9

On rappelle l'énoncé du lemme 9 que l'on va maintenant chercher à prouver.

Lemme 17. *Si R est un anneau principal, p un irréductible de R et N est un R-module de p^∞-torsion (c'est-à-dire $\forall x \in N, \exists n \in \mathbb{N}, p^n \cdot x = 0$) engendré par d éléments, alors N est isomorphe à un*

$$\bigoplus_{i=1}^{d} R/p^{e_i}R$$

pour certains $e_i \in \mathbb{N}$.

Avant de prouver ce lemme, j'ai commencé par définir en Lean une notion de p-ordre et de prouver plusieurs propriétés dessus, données ci-dessous.

Définition 6. *Soit R un anneau, $p \in R$ et M un R-module de p^∞-torsion. On définit alors le p-ordre de $x \in M$ comme*

$$\omega_p(x) := min\{n \in \mathbb{N}, p^n \cdot x = 0\}.$$

Remarque. *Il suffit en fait d'avoir une structure de monoïde multiplicatif R muni d'une action sur M pour définir cette notion. Lorsque j'ai écrit cette définition en Lean, le compilateur a alors pu m'indiquer que j'avais donné des hypothèses non nécessaires (non utilisées) sur R et j'ai ensuite pu affaiblir les hypothèses.*

Lemme 18. *Soit R un anneau commutatif, $p \in R$ et M un R-module de p^∞-torsion, engendré par une famille finie (s_j). Alors il existe j tel que s_j soit de p-ordre maximal dans M. M est alors de $p^{\omega_p(s_j)}$-torsion.*

Démonstration. Soit j maximisant $\omega_p(s_j)$. On a alors pour tout $i, \omega_p(s_i) \leq \omega_p(s_j)$ donc $p^{\omega_p(s_j)} \cdot s_i = 0$. Comme les s_i engendrent M, on en déduit que M est de $p^{\omega_p(s_j)}$-torsion, puis que $\omega_p(x) \leq \omega_p(s_j)$ pour tout $x \in M$ (puisque $p^{\omega_p(s_j)} \cdot x = 0$.) □

Lemme 19. *Soit R un anneau principal, p irréductible de R et M un R-module de p^∞-torsion. Alors pour $x \in M$, l'idéal de torsion de x (c'est-à-dire $I_x := \{a \in R, a \cdot x = 0\}$) est engendré par $p^{\omega_p(x)}$.*

Démonstration. On a $p^{\omega_p(x)} \cdot x = 0$, donc $p^{\omega_p(x)} \in I_x$. Ainsi, comme R est principal, I_x est engendré par un diviseur de $p^{\omega_p(x)}$, qui est donc de la forme p^k car p irréductible. Par définition de $\omega_p(x)$, on a alors $\omega_p(x) = k$ comme voulu. □

Remarque. *Pour pouvoir prouver ce lemme, j'ai dû préalablement prouver que les diviseurs de puissances de p sont des puissances de p quand p irréductible dans un anneau factoriel.*

Lemme 20. *Soit R un anneau principal, p irréductible de R et M un R-module de p^∞-torsion. Soit y de p-ordre maximal $n = p^{\omega_p(y)}$ dans M (autrement dit, M est de $p^{\omega_p(y)}$-torsion.) Soit $k \in \mathbb{N}$ et $x \in M$ tel que $p^k \cdot x \in R \cdot y$. Alors $p^k \cdot x$ est aussi un multiple de p^k dans $R \cdot y$, i.e. $\exists a \in R, p^k \cdot x = p^k \cdot a \cdot y$.*

Démonstration. Si $k > n$, on a $p^k \cdot x = 0 = p^k \cdot 0 \cdot y$ car M est de p^n-torsion.

Sinon, on a $p^n \cdot x = 0$ (car M de p^n-torsion) donc $p^k \cdot x$ est de p^{n-k}-torsion. Or, les éléments de p^{n-k}-torsion dans $R \cdot y \simeq R/p^n R$ sont exactement les multiples de p^k dans $R \cdot y$, d'où la conclusion. (On a bien $R \cdot y \simeq R/p^n R$ car $p^n R$ est l'idéal de torsion de y par le lemme 19.) □

Lemme 21. *Soit R un anneau principal, p irréductible de R et M un R-module de p^∞-torsion. Soit z de p-ordre maximal $n = p^{\omega_p(z)}$ dans M, $k \in \mathbb{N}$ et $f : R/p^k R \to M/R \cdot z$ R-linéaire. Alors on peut trouver $x \in M$ de p^k-torsion tel que $x \equiv f(1)$ dans $M/R \cdot z$. 5autrement dit, un élément de p^k-torsion de $M/R \cdot z$ (ici $f(1)$) est nécessairement l'image d'un élément de p^k-torsion de M par la surjection canonique.)*

Démonstration. On a dans $M/R \cdot z, p^k \cdot f(1) = f(p^k = 0 \in R/p^k R) = 0$.

Soit $x' \in M$ tel que $x' \equiv f(1)$ dans $M/R \cdot z$. On a alors $p^k \cdot x' \equiv p^k \cdot f(1) = 0$ dans $M/R \cdot z$, autrement dit $p^k \cdot x' \in R \cdot z$. Par le lemme 20, on peut alors trouver un $az \in R \cdot z$ tel que $p^k \cdot az = p^k \cdot x'$. On a alors $p^k \cdot (x' - az) = 0$ et $x' - az \equiv x' \equiv f(1)$ dans $R/R \cdot z$ (car $az \in R \cdot z$), comme voulu avec $x = x' - az$. □

On peut maintenant prouver le lemme 9.

Démonstration. On prouve le résultat par récurrence sur d. Si $d = 0$, alors $M = \{0\}$ est bien une somme directe vide.

Soit maintenant N engendré par $d + 1$ éléments $s_0, ..., s_d$. Par le lemme 18, on peut trouver s_j de p-ordre maximal $n = \omega_p(s_j)$ dans M. On a alors par le lemme 19 que $p^n R$ est l'idéal de torsion de s_j donc $R \cdot s_j \simeq R/p^n R$.

De plus, $N/R \cdot s_j$ est engendré par d éléments, qui sont les images des $(s_i)_{i \neq j}$ par la surjection canonique (car s_j est envoyé sur 0). Et on a toujours $N/R \cdot s_j$ de p^∞-torsion, donc par hypothèse de récurrence on a un isomorphisme

$$f : N/R \cdot s_j \simeq \bigoplus_{i=1}^{d} R/p^{e_i} R$$

pour certains e_i. On a donc la suite exacte courte :

$$0 \to R/p^n R (\simeq R \cdot s_j) \to N \to \bigoplus_{i=1}^{d} R/p^{e_i} R (\simeq N/R \cdot s_j) \to 0.$$

Pour obtenir un isomorphisme $N \simeq R/p^n R \times \bigoplus_{i=1}^{d} R/p^{e_i} R$ (ce qui conclurait), il suffit par le lemme 5 de trouver un scindage à droite de cette suite exacte, c'est-à-dire une fonction $h : \bigoplus_{i=1}^{d} R/p^{e_i} R \to N$ telle que $\forall x, h(x) \equiv f^{-1}(x)$ dans $N/R \cdot s_j$.

Pour construire une telle fonction, il suffit de construire les $h(1_i)$ pour les $1_i \in R/p^{e_i} R$ engendrant $\bigoplus_{i=1}^{d} R/p^{e_i} R$, avec pour seule contrainte (en plus de celles déjà posées) d'avoir $h(1_i)$ de p^{e_i}-torsion dans M. Le lemme 21 nous permet justement d'obtenir un tel $h(1_i) \equiv f^{-1}(1_i)$ dans $N/R \cdot s_j$ et de p^{e_i}-torsion dans M. □

Pour formaliser cette preuve du lemme 9 en Lean (figure 17), j'ai commencé par faire une `induction d with d IH generalizing N`, ce qui permet de prouver le résultat par récurrence sur `d` en incluant le quantificateur sur `N` dans l'hypohèse de récurrence. Ainsi, j'ai pu ensuite ans l'hérédité appliquer l'hypothèse de récurrence à un autre module N que le module initial (en l'occurrence ici $N/R \cdot s_j$ au lieu de N). Ensuite, j'ai prouvé l'initialisation (cas $d = 0$), puis l'hérédité. Pour l'hérédité, j'ai commencé par appliquer le lemme 18 `exists_is_torsion_by` pour récupérer s_j, puis j'ai appliqué l'hypothèse de récurrence `IH` sur $N/R \cdot s_j$. J'ai appliqué ensuite le lemme 21 `exists_smul_eq_zero_and_mk_eq` pour récupérer les $h(1_i)$, puis j'ai pu définir l'isomorphisme $N \simeq \bigoplus_{i=1}^{d+1} R/p^{e_i} R$ dans la grande instruction `refine`. Définir cet isomorphisme nécessite ici de construire h à partir des $h(1_i)$, puis de composer l'isomorphisme obtenu par le lemme 5 avec entre autres les isomorphismes $R \cdot s_j \simeq R/p^n R$ et $\_ \times \bigoplus_{i=1}^{d} \_ \simeq \bigoplus_{i=1}^{d+1} \_$. La suite de la preuve consiste à vérifier les hypothèses qui ont été laissées en suspens plus haut dans la preuve, comme par exemple le fait que h scinde la suite exacte ou que $N/R \cdot s_j$ a une famille génératrice à d éléments.

4 Conclusion

J'ai donc prouvé durant ce stage le théorème 2 de classification des modules finiment engendrés sur un anneau principal (sans l'unicité de la décomposition) dans l'assistant de preuve Lean. [5] La formalisation de cette preuve m'a amené à implémenter différents résultats intermédiaires qui peuvent être utiles ailleurs, par exemple sur les suites exactes et les sommes directes. J'ai aussi pu explorer des généralisations possibles de certaines parties du théorème, notamment sur les anneaux de Dedekind.

Ce résultat est dorénavant disponible dans *mathlib*, ce qui permet de l'utiliser dans la preuve d'autres résultats en Lean. J'ai par exemple pu l'appliquer au cas $R = \mathbb{Z}$ pour obtenir la classification des groupes abéliens finis [7] (ici encore, sans l'unicité de la décomposition).

```
theorem torsion_by_prime_power_decomposition (hN : module.is_torsion' N (submonoid.powers p))
  [h' : module.finite R N] :
  ∃ (d : ℕ) (k : fin d → ℕ), nonempty $ N ≃ₗ[R] ⨁ (i : fin d), R ⧸ R ∙ (p ^ (k i : ℕ)) :=
begin
  obtain ⟨d, s, hs⟩ := @module.finite.exists_fin _ _ _ _ _ h', use d, clear h',
  unfreezingI { induction d with d IH generalizing N },
  { use λ i, fin_zero_elim i,
    rw [set.range_eq_empty, submodule.span_empty] at hs,
    haveI : unique N := ⟨⟨0⟩, λ x, by { rw [← mem_bot _, hs], trivial }⟩,
    exact ⟨0⟩ },
  { haveI : Π x : N, decidable (x = 0), classical, apply_instance,
    obtain ⟨j, hj⟩ := exists_is_torsion_by hN d.succ d.succ_ne_zero s hs,
    let s' : fin d → N ⧸ R ∙ s j := submodule.quotient.mk ∘ s ∘ j.succ_above,
    obtain ⟨k, ⟨f⟩⟩ := IH _ s' _; clear IH,
    { have : ∀ i : fin d, ∃ x : N,
        p ^ k i • x = 0 ∧ f (submodule.quotient.mk x) = direct_sum.lof R _ _ i 1,
      { intro i,
        let fi := f.symm.to_linear_map.comp (direct_sum.lof _ _ _ i),
        obtain ⟨x, h0, h1⟩ := exists_smul_eq_zero_and_mk_eq hp hN hj fi, refine ⟨x, h0, _⟩, rw h1,
        simp only [linear_map.coe_comp, f.symm.coe_to_linear_map, f.apply_symm_apply] },
      refine ⟨_, ⟨⟨⟨⟨
        @lequiv_prod_of_right_split_exact _ _ _ _ _ _ _ _ _ _ _ _ _
          ((f.trans ulift.module_equiv.{u u v}.symm).to_linear_map.comp $ mkq _)
          ((direct_sum.to_module _ _ _ $ λ i, (liftq_span_singleton.{u u} (p ^ k i)
            (linear_map.to_span_singleton _ _ _) (this i).some_spec.left : R ⧸ _ →ₗ[R] _)).comp
            ulift.module_equiv.to_linear_map)
          (R ∙ s j).injective_subtype _ _).symm.trans $
        ((quot_torsion_of_equiv_span_singleton _ _ _).symm.trans $
          quot_equiv_of_eq _ _ $ ideal.torsion_of_eq_span_pow_p_order hp hN _).prod $
          ulift.module_equiv).trans $
        (@direct_sum.lequiv_prod_direct_sum R _ _ _
          (λ i, R ⧸ R ∙ p ^ @option.rec _ (λ _, ℕ) (p_order hN $ s j) k i) _ _).symm).trans $
        direct_sum.lequiv_congr_left R (fin_succ_equiv d).symm⟩⟩,
      { rw [range_subtype, linear_equiv.to_linear_map_eq_coe, linear_equiv.ker_comp, ker_mkq] },
      { rw [linear_equiv.to_linear_map_eq_coe, ← f.comp_coe, linear_map.comp_assoc,
          linear_map.comp_assoc, ← linear_equiv.to_linear_map_eq_coe,
          linear_equiv.to_linear_map_symm_comp_eq, linear_map.comp_id,
          ← linear_map.comp_assoc, ← linear_map.comp_assoc],
        suffices : (f.to_linear_map.comp (R ∙ s j).mkq).comp _ = linear_map.id,
        { rw [← f.to_linear_map_eq_coe, this, linear_map.id_comp] },
        ext i : 3,
        simp only [linear_map.coe_comp, function.comp_app, mkq_apply],
        rw [linear_equiv.coe_to_linear_map, linear_map.id_apply, direct_sum.to_module_lof,
          liftq_span_singleton_apply, linear_map.to_span_singleton_one,
          ideal.quotient.mk_eq_mk, map_one, (this i).some_spec.right] } },
    { exact (mk_surjective _).forall.mpr
      (λ x, ⟨(@hN x).some, by rw [← quotient.mk_smul, (@hN x).some_spec, quotient.mk_zero]⟩) },
    { have hs' := congr_arg (submodule.map $ mkq $ R ∙ s j) hs,
      rw [submodule.map_span, submodule.map_top, range_mkq] at hs', simp only [mkq_apply] at hs',
      simp only [s'], rw [set.range_comp (_ ∘ s), fin.range_succ_above],
      rw [← set.range_comp, ← set.insert_image_compl_eq_range _ j, function.comp_apply,
        (quotient.mk_eq_zero _).mpr (mem_span_singleton_self _), span_insert_zero] at hs',
      exact hs' } }
end
```

FIGURE 17 – Preuve du lemme 9 en Lean.

5 Liens

1. la librairie mathématique *mathlib* de Lean
2. définition 1 de l'infimum de seminormes
3. définition des sous-modules de torsion
4. isomorphismes pour manipuler les sommes directes
5. preuve du théorème 2 de classification des modules finiment engendrés sur un anneau principal
6. preuve du lemme 5 sur les suites exactes courtes scindées à droite
7. classification des groupes abéliens finis
8. séminaire "London Learning Lean" du 19 mai où je présente ma preuve du théorème 2 en Lean