ÉCOLE NORMALE SUPÉRIEURE DE LYON

Université Claude Bernard Lyon 1

Master de Mathématiques Avancées

Parcours Higher Algebra and Formalization

Rapport de stage

Loris PARISOT

Formalisation de la représentation de Weil pour les groupes classiques sur corps finis en L∃∀N4

encadré par Sophie MOREL

Introduction

Ce stage de M2, réalisé à l'U.M.P.A et encadré par Sophie Morel, consistait en la formalisation des résultats issus de l'article « Weil representations associated to finite fields »de Paul Gérardin.

Au début du stage, certains prérequis à la formalisation de l'article n'existait pas dans mathlib. Il a donc fallu les implémenter avant de pouvoir travailler sur l'article, celui-ci devenant alors mineur par rapport aux prérequis à implémenter.

Voici un lien vers un site qui contient : le lien vers le git, le blueprint et le dependency graph. `https://lorisparisot.github.io/Weil_representations_associated_to_finite_fields/`

1 Formalisation informatique de preuve

1.1 L'assistant de preuve LEAN

La finalité de ce stage était la formalisation informatique de résultats (ainsi que de leurs preuves) de théorie des représentations.

La formalisation informatique de preuves consiste grossièrement en traduire des énoncés mathématiques à ce qu'on appelle un *assistant de preuve* dans un langage compréhensible par un ordinateur qui sera alors en mesure de dire si les énoncés ont un sens ou non et si leurs preuves sont exactes ou non. Il existe de nombreux assistants, les plus connus étant *Rocq*, *Isabelle* et *LEAN* qui est celui qui a servi à ce projet.

Pour formaliser des résultats en *LEAN*, il a été développé une bibliothèque, *Mathlib*, regroupant tous les résultats déjà implémentés et pouvant être directement utilisés. Cette bibliothèque est collaborative : n'importe qui peut soumettre un résultat et sa preuve, qui fera alors l'objet (sur le même modèle que les publications classiques en mathématiques) d'une revue par les pairs qui déterminera si celui-ci est « bien formalisée » : respect de la façon d'écrire, temps de compilation du code, est-il assez général,... *Mathlib* possède de plus une assez grande communauté dont les membres peuvent s'inscrire sur un forum en ligne pour discuter, demander de l'aide pour formaliser et prouver des résultats, etc.

L'interface du langage *LEAN* sur le logiciel de codage *VSCode* se présente en deux parties. Une première qui permet d'écrire du code, et une seconde permettant de voir les objets mathématiques et les énoncés mis en jeu ainsi que le but de la preuve. *LEAN* utilisant la correspondance de Curry-Howard, les propositions sont représentés par des types.

1.2 Blueprint, une carte interactive pour formaliser

Blueprint est un outil développé spécialement pour les projets de formalisation en *LEAN* par Patrick Massot. Il permet de déployer automatiquement une page web hébergée par Github sur laquelle on retrouve :

- Les fichiers associés au projet de formalisation
- Une documentation complète de *Mathlib* avec les nouveaux résultats implémentés.
- Un fichier latex permettant d'écrire un article avec, pour chaque résultats, la possibilité d'ajouter un lien vers le résultat formalisé dans la documentation.
- Un graphe de dépendance des résultats implémentés, c'est-à-dire un graphe dont les sommets sont les noms des propositions et dont les arêtes indiquent les relations entre ces propositions. Le tout agrémenté d'un code couleur indiquant si le résultat est formalisé ou pas encore.

Cet outil a deux avantages majeurs. Le premier est qu'il permet de structurer le projet de formalisation et de s'organiser lors de travaux à plusieurs. Le second est qu'il a été pensé pour des mathématiciens souhaitant faire de la formalisation : il ne nécessite aucun pré requis informatique et se met en place assez facilement.

2 Groupe d'Heisenberg

2.1 Convention

L'article de P. Gérardin ne prend pas pour convention l'identification canonique de V avec $V**$, mais la suivante : pour tout $(y, x) \in V^* \times (V^*)^*$, $\langle x, y\rangle = -\langle y, x\rangle = -y(x)$.

Il suffit pour se faire de définir cette application linéaire dans *LEAN*. Toute la machinerie sur le dual d'un espace vectoriel étant d'ors et déjà dans *mathlib*, on peut s'en servir pour se faciliter une part de travail. Pour l'application réciproque, on utilise en particulier `Module.evalEquiv k V` qui est la bijection canonique entre V et V^{**}. Notre réciproque peut donc s'écrire `fun x => -((Module.evalEquiv k V).invFun (x))`, ce qui nous permet d'utiliser toute la machinerie derrière elle. L'application de V dans V^{**} est définie en utilisant la syntaxe `- LinearMap.id.flip x`.

2.2 Groupe d'Heisenberg

Considérons $\mathbb{K}$ un corps fini, V un $\mathbb{K}-$espace vectoriel de dimension fini n et V^* son dual. On définit alors une structure de groupe sur les triplets $(z, x, y) \in \mathbb{K} \times V \times V^*$.

L'ensemble $H = \{(z, x, y) \in \mathbb{K} \times V \times V^*\}$ muni de $1 := (0, 0, 0)$ et de l'opération
$\star : \left| \begin{array}{rcl} H \times H & \longrightarrow & H \\ ((z_1, x_1, y_1), (z_2, x_2, y_2)) & \longmapsto & (z_1 + z_2 + y_1(x_2), x_1 + x_2, y_1 + y_2) \end{array} \right.$ forme un groupe appelé groupe d'Heisenberg associé à V et noté $\mathcal{H}(V)$.

LEAN permet de définir directement des instances de multiplication, de neutre et d'inverse. Cela permet d'utiliser automatiquement les notation « * », « 1 »et « $^{-1}$ ». On établit également quelques lemmes « triviaux »qui permettront de réécrire plus facilement les buts et de se servir du `@[ext]` placé en préambule.

L∃∀N

```
@[ext]
structure Heisenberg where
z : k
x : V
y : Module.Dual k V

namespace Heisenberg

variable {V k}
/--Intern law over ‘Heisenberg‘ -/
instance : Mul (Heisenberg V k) where
mul H1 H2 :=  ⟨H1.z + H2.z + (H1.y H2.x), H1.x + H2.x, H1.y +
H2.y⟩

lemma mul_apply (H1 H2 : Heisenberg V k) : H1 * H2 = ⟨H1.z + H2.z
+ (H1.y H2.x), H1.x + H2.x, H1.y + H2.y⟩ := rfl

lemma add_apply (H1 H2 : Heisenberg V k) : H1 + H2 = ⟨H1.z +
H2.z, H1.x + H2.x, H1.y + H2.y⟩ := rfl

instance : One (Heisenberg V k) where
one := ⟨0, 0, 0⟩

lemma one_eq : (1 : Heisenberg V k) = ⟨0,0,0⟩ := by
rfl

lemma one_apply (H : Heisenberg V k) : H * 1 = H := by
ext u
<;> rw[one_eq,mul_apply]
<;> simp only [add_zero, map_zero]

/--Inverse of an element of ‘Heisenberg‘ by ‘mul‘ -/
instance : Inv (Heisenberg V k) where
inv H := ⟨ -H.z - (H.y (-H.x)), - H.x ,- H.y⟩
```

Notre article représente la loi de groupe par la multiplication matricielle sur des matrices 3×3, chose qui n'est pas possible en *LEAN*, les matrices étant définies par

$L\exists\forall N$

```
    def Matrix (m : Type u) (n : Type u') (α : Type v) : Type max
u u' v :=
    m → n → α
```

c'est-à-dire un tableau dont les lignes sont indexés par un type `m`, des colonnes indexées par un type `n` et dont les éléments sont tous de même type.

On montre alors aisément que la formalisation proposée définie bien un groupe. L'ajout de `[@ext]` permet d'obtenir automatiquement que si $H_1 = H_2$, alors $x_1 = x_2$, $y_1 = y_2$ et $z_1 = z_2$.

Deux théorèmes sur les groupes d'Heisenberg sont entre autres démontrés. Le premier stipule que $\mathcal{H}(V)$ est un groupe nilpotent d'ordre 2, c'est-à-dire $[\mathcal{H}(V), \mathcal{H}(V)] \neq \{1\}$ et $[\mathcal{H}(V), [\mathcal{H}(V), \mathcal{H}(V)]] = \{1\}$.
Le second énonce que $\mathcal{H}(V)$ est anti-isomorphique à $\mathcal{H}(V^*)$.

2.3 La tactique grind

La formalisation des groupes d'Heisenberg a été l'occasion d'utiliser la tactique `grind`. Le but de cette tactique est de clore automatiquement le goal, c'est-à-dire sans lui spécifier de quelle manière procéder. Elle utilise les même techniques que les solveurs SMT, *Satisfiability Modulo Theories* qui sont des formules de premier ordre sans quantificateurs faisant intervenir des égalités. Le principe est le suivant : on construit la preuve en collectant toutes les données que l'on a faisant intervenir égalités, inégalités, booléens, puis dérive chacune de ces données (tout en gardant l'information principale), et ainsi de suite. On obtient alors une très grande quantité d'éléments, et but de la tactique est alors de clore le goal par contradiction.
Pour ce faire, `grind` utilise les principes suivants :

- *Congruence closure* : on considère la cloture réflexive, transitive et symétrique de la relation ≪ est égal à ≫à laquelle on ajoute une égalité sur les fonctions ≪ si $a = b$ et $c = d$, alors $f\ a\ c = f\ b\ d$ ≫. L'algorithme fusionne alors les classes d'équivalences jusqu'à arriver à un point fixe et essaye de trouver une contradiction.
- *Constraint propagation* : pour tout terme que `grind` découvre via *Congruence closure*, on lui applique des règles en plus : dérivation des booléens (par exemple si $A \wedge B$ vérifie `True`, alors A et B sont également `True`), projections (`<x,y>=<a,b>` se dérive en `x=a` et `y=b`).
- *Inductive Types* : si dans les classes d'équivalences de `grind` un même terme est formé par deux constructeurs différents d'un même type inductif, alors on obtient `False` (par exemple $\forall$ et $\emptyset$). Si deux termes issus d'un même constructeur se retrouvent dans la même classe d'équivalence, alors on rend leurs arguments égaux.
- Des solveurs arithmétiques et sur des anneaux commutatifs.

Contrairement à la tactique `aesop`, on ne peut voir ce que fait `grind`. Ceci a un désavantage : la tactique n'est pas performante pour ≪ générer ≫une preuve mathématiques, dans le sens où elle ne permet pas à l'utilisateur d'avoir accès à la dite preuve.

3 Représentation induite

3.1 Les représentations dans Mathlib

Les représentations ont la particularité dans Mathlib de ne pas être implementées de manière uniforme. La librairie étant collaborative, les personnes y participant proposent des résultats en lien avec leurs domaines de prédilections, donnant lieu, dans le cas des représentations, à un certain hétéroclitisme.

La première façon de penser aux représentations est celle classiquement enseigné, à savoir un morphisme $\rho : G \to GL(V)$ où V est un espace vectoriel sur un corps k. Toujours dans le souci d'avoir une librairie la plus généraliste possible, on obtient dans Mathlib la définition suivante : `Representation k G V = (G →* V →[k] V)` avec les instances `[CommSemiring k] [Monoid G] [AddCommMonoid V] [Module k V]`.

Une autre façon de voir les représentations est d'utiliser un point de vue catégorique : en effet, on peut, étant donné un monoïde G et un anneau R, introduire la catégorie des représentations $R-$linéaires de G en prenant l'action de G sur l'ensemble des modules finis sur R. On obtient la formalisation `FDRep R G = Action (FGModuleCat R) G` où `FGModuleCat R` désigne la catégorie des modules engendrés de manière finie.

Ces points de vue sont heureusement reliés entre eux via différents lemmes mais qui ont parfois tout de même des subtilités d'utilisation. Par exemple, le passage d'un élément de `FDRep R G` à la définition via les morphismes se fait via `FDRep.`ρ de manière immédiate en interprétant une représentation de f de G dans le $\mathbb{K}$-espace vectoriel V comme un morphisme d'algèbres entre $\mathbb{K}[G]$ et les endomorphismes $\mathbb{K}-$linéaires de G via $\tilde{f} : \left| \begin{array}{ccc} \mathbb{K}[G] & \longrightarrow & \mathcal{L}_k(G) \\ \sum\limits_{g \in G} a_g g & \longmapsto & \sum\limits_{g \in G} a_g f(g) \end{array} \right.$. On définit alors la multiplication externe sur V par $x \cdot v := (\tilde{f}(x))(v)$ pour tout $x \in \mathbb{K}[G]$ et tout $v \in V$.

$L\exists\forall N$

```
/-- If ‘ρ : Representation k G V‘, then ‘ρ.asModule‘ is a type
synonym for ‘V‘, which we equip with an instance ‘Module (MonoidAlgebra k
G) ρ.asModule‘.

You should use ‘asModuleEquiv : ρ.asModule ≃+ V‘ to translate terms.
-/

@[nolint unusedArguments]
def asModule (_ : Representation k G V) := V

deriving AddCommMonoid, Module k

/-- A ‘k‘-linear representation of ‘G‘ on ‘V‘ can be thought of as a
module over ‘MonoidAlgebra k G‘.
-/
noncomputable instance : Module (MonoidAlgebra k G) ρ.asModule :=
Module.compHom V (asAlgebraHom ρ).toRingHom
```

En revanche, si l'on veut passer du point de vue $\mathbb{K}[G]$ module au point de vue morphisme, la situation est plus délicate. En effet, si V est un $\mathbb{K}[G]$ module, il existe une multiplication externe $x \cdot v$ entre éléments de $\mathbb{K}[G]$ et de V telle que $(xy) \cdot v = x \cdot (y \cdot v)$. On peut alors considérer l'action de G sur V définie par $\rho(g)(v) = g \cdot v$. L'application $g \mapsto \rho(g)(-)$ définie alors une représentation de G dans V. C'est ce que fait `Representation.ofModule'`. Le soucis de cette formalisation est qu'elle repose sur l'existence d'une instance `Module k M` ainsi qu'une instance `IsScalarTower k (MonoidAlgebra k G) M` (qui est une compatibilité des multiplications) qui est quelque chose d'immédiat mathématiquement mais pas en formalisation.

La manière choisie pour palier à ça est de définir une représentation non plus sur `M`, mais sur `RestricScalars k (MonoidAlgebra k G) M`. Si l'on souhaite mettre une structure de R algèbre sur un anneau S, on a une équivalence de catégories entre la catégorie des $S-$modules et la catégorie des représentations de l'algèbre S sur R donnée par : à toute représentation f de S dans $\mathcal{L}_R(M)$, on associe le $S-$module M avec l'action $s \cdot m := f(s)(m)$ et, réciproquement, à un $S-$module M on définit une représentation $f(s)(m) := s \cdot m$. La formalisation de cette équivalence est `RestricScalars R S M`. Ce dernier est équivalent à `M`, mais se passe des instances précédemment requises. C'est donc celui-ci qui est généralement utilisé. Il est de plus l'inverse (au sens d'équivalence de catégories) de notre précédent `Representation.asModule`.

LƎ∀N

```
  /-- Build a ‘Representation k G M‘ from a ‘[Module (MonoidAlgebra k G)
M]‘.

  This version is not always what we want, as it relies on an existing
‘[Module k M]‘ instance, along with a ‘[IsScalarTower k (MonoidAlgebra k
G) M]‘ instance.

  We remedy this below in ‘ofModule‘ (with the tradeoff that the
representation is defined only on a type synonym of the original module.)
  -/
  noncomputable def ofModule’ (M : Type*) [AddCommMonoid M] [Module k M]
  [Module (MonoidAlgebra k G) M] [IsScalarTower k (MonoidAlgebra k G) M]
: Representation k G M :=
  (MonoidAlgebra.lift k G (M →[k] M)).symm (Algebra.lsmul k k M)

  /-- Build a ‘Representation‘ from a ‘[Module (MonoidAlgebra k G) M]‘.

  Note that the representation is built on ‘restrictScalars k
(MonoidAlgebra k G) M‘, rather than on ‘M‘ itself.
  -/
  noncomputable def ofModule : Representation k G (RestrictScalars k
(MonoidAlgebra k G) M) :=
  (MonoidAlgebra.lift k G (RestrictScalars k (MonoidAlgebra k G) M →[k]
RestrictScalars k (MonoidAlgebra k G) M)).symm (RestrictScalars.lsmul k
(MonoidAlgebra k G) M)
```

On peut noter que ces points de vue multiples sur la librairie des représentations peuvent mener à des résultats parfois curieux.
Par exemple, le caractère d'une représentation est implémentée dans *Mathlib* via `FDRep`, c'est-à-dire d'un point de vue catégorique. Mais si l'on se penche sur sa définition, on voit qu'elle n'utilise aucunement des résultats de théorie des catégories car est simplement définie comme étant la trace de l'application linéaire donnée par `FDRep.`ρ. L'une des raisons pourrait être qu'il n'y dans la librairie aucune lemme permettant de passer de à .

3.2 Représentation induite

L'article utilise la notion de représentation induite et celle de son caractères. Ces notions étaient étrangères à *Mathlib* lorsque nos avons eu besoin de nous en servir, il a donc fallu les définir. Plusieurs options s'offrent alors à nous.

La première est de la définir ad hoc. Le problème de cette manière de procéder est qu'elle peut vite se révéler fastidieuse. Il est en effet bien plus aisé, lorsque l'on fait de la formalisation, de définir des objets puis de prouver des propriétés dessus plutôt que de les construire comme on le ferait traditionnellement en mathématiques, c'est-à-dire.

La deuxième option serait d'utiliser des notions de théorie des représentations déjà implémentés dans la librairie et plus particulièrement celle de coinduite. La coinduite est bla bla bla. Dans le cas des groupes finis, celle-ci est isomorphe à l'induite. Notre article ne traitant que de groupes finis, cela permettrait de s'économiser de la formalisation. En revanche, plus que de l'induite c'est de ses propriétés que nous avons besoin. Et la coinduite ne permet pas d'espérer formaliser facilement la réciprocité de Frobenius dont nous aurons besoin.

Vient ensuite la solution choisie, l'utilisation de l'algèbre de groupe : on se donne G un groupe, H un sous-groupe de G et θ une représentation de H dans un espace vectoriel W sur un corps $\mathbb{K}$. Soit W' le $\mathbb{K}[H]$ module associé à θ. On construit ρ la représentation de G donnée par le $\mathbb{K}[G]$ module tensoriel $V := \mathbb{K}[G] \otimes_{\mathbb{K}[H]} W$.

Cette façon de traiter la représentation induite possède néanmoins un désavantage : elle utilise le produit tensoriel. En effet, $M \otimes N$ n'est défini que pour des modules M et N définis sur un même anneau commutatif. L'une des raisons de cette absence est la façon avec laquelle $LEAN$ fonctionne. Or, $\mathbb{K}[H]$ n'est a priori pas un anneau commutatif. En revanche, il est clair que si H est un sous-groupe commutatif de G, alors $\mathbb{K}[H]$ est commutatif. Or, l'article de P. Gérardin se concentre sur des induites depuis le centre, donc depuis un sous-groupe commutatif.

Il s'agit ensuite de vérifier que cette représentation vérifie deux propriétés suivantes :

- $W \equiv \mathbb{K}[H] \otimes_{\mathbb{K}[H]} W$ est bien un sous $\mathbb{K}[H]$ module de V.
- La réciprocité de Frobenius : si E est un $\mathbb{K}[G]$ module, on a un isomorphisme entre $\mathrm{Hom}_{\mathbb{K}[G]}(V, E)$ et $\mathrm{Hom}_{\mathbb{K}[H]}(W, E)$.

La preuve du premier point est une preuve calculatoire ne présentant pas spécialement de difficulté. La preuve du second point utilise le lemme suivant : si A est un un anneau commutatif, B une $A-$algèbre, M et N deux $A-$modules, alors on a un isomorphisme $A-$linéaire $\mathrm{Hom}_B(B \otimes_A M, N) \simeq \mathrm{Hom}_A(M, N)$. Pour le montrer, on fourni une application $A-$linéaire et vérifie sa bijectivité, à savoir l'application $\Psi : \left| \begin{array}{ccc} \mathrm{Hom}_B(B \otimes_A M) & \longrightarrow & \mathrm{Hom}_A(M, N) \\ \varphi & \longmapsto & \Psi_\varphi : \left| \begin{array}{ccc} M & \longrightarrow & N \\ x & \longmapsto & \varphi(1 \otimes_A x) \end{array} \right. \end{array} \right.$. Cette application est bien injective : si $\Psi_\varphi = \Psi_\phi$, alors pour tout $m \in M$, $\varphi(1 \otimes_A m) = \phi(1 \otimes_A m)$ et donc $\varphi = \phi$. Pour la surjectivité, on fabrique un antécédent explicite de l'application B linéaire $\varphi : M \to N$ ainsi : on pose ψ application B linéaire de $B \otimes_A M$ dans N définie par $\psi(b \otimes_A m) = b\varphi(m)$. Ainsi, on a bien $\Psi_\psi(x) = \psi(1 \otimes_A x) = \varphi(x)$ pour tout x de M.

C'est ici que l'on voit tout l'intéret d'avoir choisi le point de vue de $\mathbb{K}[G]$: la réciprocité de Frobenius devient une application de ce résultat deux fois : une première fois pour avoir l'isomorphisme $\mathrm{Hom}_{\mathbb{K}[G]}(\mathbb{K}[G] \otimes_{\mathbb{K}[\mathcal{Z}_G]} W, E) \simeq \mathrm{Hom}_{\mathbb{K}[\mathcal{Z}_G]}(W, E)$ et une seconde fois en prenant l'inverse de $\mathrm{Hom}_{\mathbb{K}[\mathcal{Z}_G]}(\mathbb{K}[\mathcal{Z}_G] \otimes_{\mathbb{K}[\mathcal{Z}_G]} W) \simeq \mathrm{Hom}_{\mathbb{K}[\mathcal{Z}_G]}(W, E)$

On obtient alors le résultat formalisé en seulement une ligne :

$L\exists\forall N$

```
    noncomputable def iso_induced_as_tensor (E : Type*)
[AddCommMonoid E] [Module (MonoidAlgebra k G) E] [Module
(MonoidAlgebra k (Subgroup.center G)) E] [inst6 : IsScalarTower
(MonoidAlgebra k (Subgroup.center G)) (MonoidAlgebra k G) E]:

    ((tensor k G W θ) →[MonoidAlgebra k G] E) ≃l[MonoidAlgebra k
(Subgroup.center G)] ((module_sub_rep k G W θ) →[MonoidAlgebra k
(Subgroup.center G)] E) := by

    exact ((iso_hom_tens (MonoidAlgebra k (Subgroup.center G))
(θ.asModule) E (MonoidAlgebra k G)).trans (iso_hom_tens
(MonoidAlgebra k (Subgroup.center G)) θ.asModule E
    (MonoidAlgebra k (Subgroup.center G))).symm)
```

3.3 Caractère de l'induite

Le but de cette section est de formaliser le fait que « le caractère de l'induit est l'induit du caractère »(on se concentre sur le cas où l'on induit depuis le centre de G).

Etant donnée f une fonction centrale sur un sous-groupe H d'un groupe G, on définit l'induit de f par la fonction F_f définie par $F_f(s) = \frac{1}{Card(H)} \sum\limits_{t \in G \,\wedge\, t^{-1}st \in H} f(t^{-1}st)$ pour tout s dans G. La fonction F_f est alors une fonction centrale sur G.

La notion de fonction centrale étant absente de Mathlib, on la formalise ainsi :

$L\exists\forall N$

```
class conj_class_fun where
Fun :  G → W
conj_property : ∀ (x : G), ∀ (g : G), Fun (g⁻¹ * x * g) = Fun x
```

Dans le but de formaliser le résultat escompté, il faut introduire la décomposition $\mathbb{K}[G] \equiv \bigoplus\limits_{i=1}^{r} g_i \cdot \mathbb{K}[\mathcal{Z}_G]$ où $(g_i)_{i \in [\![1,r]\!]}$ désigne un système de représentant de $G/\mathcal{Z}_G$.
L'algèbre d'un groupe fini est la donnée de trois éléments : un groupe fini d'ordre n, un espace vectoriel V de dimension n et d'une base $(\mathrm{e}_g)_{g \in G}$ de V indexée par G. Dans *Mathlib*, on définit cet objet via `MonoidAlgebra k G := G →₀ k`, c'est-à-dire par les applications à support fini de G dans k.
Pour montrer cette décomposition, on introduit certains résultats sur les groupes et leur quotient

par leur centre. On définit un système de représentant de G par l'image de l'application $g \mapsto \bar{g}$ et on lui associe deux autres application : une qui envoie un élément sur son représentant, et une autre $g \mapsto h$ où $g = \bar{g}h$. :

L∃∀N

```
abbrev set: Set G := by
exact Set.range (@Quotient.out G (QuotientGroup.con (Subgroup.center
G)).toSetoid )

noncomputable def G_to_syst: G → ↑(system_of_repr_center.set G) :=
by
intro g
unfold system_of_repr_center.set
refine ⟨ Quotient.out (Quotient.mk ((QuotientGroup.con
(Subgroup.center G)).toSetoid) g), by simp⟩

noncomputable def G_to_center : G → Subgroup.center G := by
intro u
exact ((QuotientGroup.mk_out_eq_mul (Subgroup.center G) u).choose)⁻¹
```

Le principal avantage de mettre en place cette machinerie est de pouvoir implémenter différents lemmes `simp` reliant ces applications et s'en servir pour simplifier des expressions. Cela permet également d'avoir des notations plus compactes et lisibles.

La preuve de la décomposition s'effectue en deux temps. On montre d'abord qu'on a un isomorphisme entre $\mathbb{K}[\mathcal{Z}_G]$ et $g \cdot \mathbb{K}[\mathcal{Z}_G]$ avec g un élément du système de représentant de $G/\mathcal{Z}_G$, ce qui est obtenu via l'application $x \mapsto gx$.
On montre ensuite que les éléments du système de représentants vus comme élément de $\mathbb{K}[G]$ forment une base de $\mathbb{K}[G]$ en tant que $\mathbb{K}[\mathcal{Z}_G]$ algèbre. La preuve n'a rien de bien compliqué sur le papier : une interversion de somme finie. Cependant, l'implémentation en *LEAN* est un peu plus fastidieuse à cause de la gymnastique entre éléments de $\mathbb{K}[\mathcal{Z}_G]$, $\mathbb{K}[G]$ et G. On donne ensuite explicitement les applications liant $\mathbb{K}[G]$ et $\bigoplus_{i=1}^{r} g_i \cdot \mathbb{K}[\mathcal{Z}_G]$, à savoir $\Psi : \left| \begin{array}{ccc} \bigoplus_{i=1}^{r} g_i \cdot \mathbb{K}[\mathcal{Z}_G] & \longrightarrow & \mathbb{K}[G] \\ (x_z)_{z \in [\![1,r]\!]} & \longmapsto & \sum_{i=1}^{r} gi \cdot x_i \end{array} \right.$.
Pour implémenter la première, on utilise un lemme sur les sommes directes :

L∃∀N

```
theorem DirectSum_eq_sum_direct (ι : Type*) [hi : Fintype ι] (β : ι → Type
w) [(i : ι) → AddCommMonoid (β i)] [DecidableEq ι] (x : (i : ι) → β i)
    (j : ι) : (Σ (i : ι), (DirectSum.of β i) (x i)) j = x j  := by
    have := Finset.sum_apply (a := j) (g := fun i ↦ (DirectSum.of β i) (x i))
(s := Finset.univ)
    rw [DFinsupp.finset_sum_apply, Finset.sum_eq_single j]
    · simp only [DirectSum.of_eq_same]
    · intro a _ ha
    exact DirectSum.of_eq_of_ne _ _ _ ha
    · simp only [Finset.mem_univ, not_true_eq_false, DirectSum.of_eq_same,
IsEmpty.forall_iff]
```

Ce lemme est intéressant, dans le sens où sa preuve ne se comporte pas comme l'on aimerait qu'elle se comporte. L'énoncé est assez simple, la preuve est assez longue. Mais si l'on rajoute un `simp[DirectSum]`, alors on peut clore le but directement via `exact Fintype.sum_dite_eq j fun j_1 h ↦ Eq.symm h ▷ x j_1`.

L∃∀N

```
theorem DirectSum_eq_sum_direct (ι : Type*) [hi : Fintype ι] (β : ι → Type w)
[(i : ι) → AddCommMonoid (β i)] [DecidableEq ι] (x : (i : ι) → β i) (j : ι) :
(Σ (i : ι), (DirectSum.of β i) (x i)) j = x j  := by
simp[DirectSum]
exact Fintype.sum_dite_eq j fun j_1 h ↦ Eq.symm h ▷ x j_1
```

Or, `DirectSum` est implémenté dans *Lean* comme étant Π_0 `i, β i`, c'est-à-dire une fonction dépendante à support fini. Il n'y a donc a priori aucune raison que `simp` n'arrive pas à ouvrir la définition. La raison de ce blocage est le manque de lemme à propos des sommes directes.

On arrive finalement au lemme suivant :

L∃∀N

```
  /--Given a representative system ‘system_of_repr_center.set G‘ of ‘G/
(Subgroup.center G)‘, we have a ‘MonoidAlgebra k (Subgroup.center G)‘ linear
bijection between ‘MonoidAlgebra k G‘ and the direct sum of ‘g · (MonoidAlgebra
k (Subgroup.center G))‘ for ‘g : system_of_repr_center.set G‘.-/

  noncomputable def MonoidAlgebra_direct_sum_system_of_repr_center_set :
MonoidAlgebra k G ≃[MonoidAlgebra k (Subgroup.center G)] DirectSum
(system_of_repr_center.set G) (fun g => gkH_set k G g) := by

  have := DirectSum_equiv_linearmap (MonoidAlgebra k (Subgroup.center G))
(system_of_repr_center.set G) (fun g => gkH_set k G g) (fun g => MonoidAlgebra
k (Subgroup.center G))  (fun g => (gkH_set_iso_kH_module k G g))
  exact LinearEquiv.trans (MonoidAlgebra_direct_sum_1 k G) this.symm
```

L'idée de la preuve de la formule du caractère de l'induite, est d'utiliser cette décomposition pour en obtenir une autre : on note (V, ρ_V) une représentation du centre de G et (W, ρ_W) son induite. Avec la décomposition précédente et le point de vue $\mathbb{K}[G]$-module, on a $W = \mathbb{K}[G] \otimes_{\mathbb{K}[\mathcal{Z}_G]} V = \bigoplus_{i=1}^{r} W_i$ avec $W_i := g_i \cdot \mathbb{K}[\mathcal{Z}_G] \otimes_{\mathbb{K}[\mathcal{Z}_G]} V$.

On introduit ce résultat de théorie des groupes : « pour tout $g \in G$, il existe $\sigma \in \mathfrak{S}_r$ et $h_1, \ldots, hr$ des éléments du centre de G tel que $gg_i = g_{\sigma(i)} h_i$ où les g_i sont les éléments du système de représentants ». Le σ en question étant la fonction qui a `gbar` associe `G_to_syst G (g⁻¹*gbar)`. On définit donc cette permutation et on obtient le lemme suivant :

L∃∀N

```
      theorem equiv_perm_exists (g : G) : ∃ (σ : Equiv.Perm
(system_of_repr_center.set G)), ∀ (gbar : system_of_repr_center.set G),
g*gbar= (σ gbar) * G_to_center G (g*gbar) := by
      use equiv_perm G g
      intro gbar
      exact equiv_perm_fun_apply G g gbar
```

On en déduit que pour tout $i \in [\![1, r]\!]$, $\rho_W(g)(W_i) = W_{\sigma(i)}$, et ainsi W_i est stabilisé si et seulement si $\sigma(i) = i$, c'est-à-dire si $gg_i = g_i h_i$, donc si $g_i^{-1} g g_i \in \mathcal{Z}_G$.
On se donne ensuite une base $(e_1, \ldots, e_n)$ de V. On va ensuite montrer que la famille $(g_i \otimes e_k)_{k \in [\![1,n]\!]}$

est une base de W_i. On a alors $\rho_W(g)(g_i \otimes e_j) = g_{\sigma(i)} \otimes \rho_V(h_i)e_s$. Le caractère de l'induite est alors :

$$\begin{aligned}
\mathrm{Tr}(\rho_W(g)) &= \sum_{\sigma(i)=i} \mathrm{Tr}(\rho_V(h_i)) \\
&= \sum_{i,\ g_i^{-1}gg_i \in H} \mathrm{Tr}(\rho_V(g_i^{-1}gg_i))
\end{aligned}$$

qui est la formule voulue.