Master Thesis

Developing a Category Theory Framework in Julia

Fabian Mäurer

29th April 2022

Supervisor: Prof. Dr. Ulrich Thiel

Technische Universität Kaiserslautern

Department of Mathematics

Contents

Introduction

The abstract theory of tensor categories plays a major role in modern mathematics. Many examples in the literature are presented strictly theoretically and usually not very hands on. This is not a problem whenever we understand the category sufficiently, like in the case of representation categories or graded vector spaces. In other cases like for example the centre category of graded vector spaces, there is not necessarily that much insight. To aid with those issues this thesis begins the development of a framework to work with (fusion) categories. The work will be available in form of the julia package TensorCategories.jl. Its main objective is to lay a base that will make it easy and convenient to specify categories and work with their objects and morphisms explicitly. Next to that there will be a growing set of implemented examples to quickly get started hands-on with category theory.

An important part in the progress of understanding tensor categories is the work with examples. Thus in Chapter 4 we will present some hands-on examples of tensor categories which are all functionally implemented in TensorCategories.jl. We have included the categories of (twisted) graded vector spaces which are as categorifications of group rings fantastic and easy examples of fusion categories. The next intuitive example is the category of finite dimensional representations of a finite group. These categories are also an important example since it is a first example where objects are not sets. As a further step in abstraction we provide the category of equivariant coherent sheaves on finite sets. This category is in fact equivalent to a product of representation categories as shown in [Rog21]. The equivariant coherent sheaves con also be endowed with an alternative monoidal product as described in [Lus87]. It is then known as the convolution category which is also contained in TensorCategories.jl.

In Chapter 2 we will discuss the theoretical background needed to implement a structure for finite semisimple ring categories. A semisimple multiring category $\mathcal{C}$ is as abelian category equivalent to a direct sum of vector space categories. And moreover we can construct a monoidal product from the $6j$-symbols of $\mathcal{C}$ such that the equivalence is indeed monoidal. In the next step we construct a skeletal category equivalent to $\mathcal{C}$. Thus allowing us to define a structure for abstract semisimple ring categories using only the $6j$-symbols and the multiplication table for the simple objects.

A main feature developed here is the computation of simple objects in the centre of a fusion category. The categorical centre of a fusion category can be shown to again be fusion [Müg03]. This is already a big accomplishment in the direction to describing the centre explicitly. Additionally we have an upper bound on the number and dimension of simple objects in the centre by the identity $\dim \mathcal{Z}(\mathcal{C}) = (\dim \mathcal{C})^2$. The final ingredient to provide an algorithm is given in [Müg03] where a finite condition for half-braidings on any object in $\mathcal{C}$ is presented. We combine these results in Chapter 5. Using the centre construction from TensorCategories.jl we are able to explicitly examine the actual structure of the centre of a fusion category. We will see this in action for the example Vec_G where G is either a cyclic group or S_3. Hereby the example of S_3 is quite interesting since the simple objects of $\mathcal{Z}(\mathcal{C})$ are not at all intuitive. This is a big progress in understanding the structure in comparison to abstractly constructing the simple objects via representations like in [Rog21].

1. Preliminaries

1.1. Abelian Categories

We will assume the basic notions of abelian categories to be known. A detailed introduction can either be found in the classical work [Mac71] or in the lecture notes to 'Introduction to Tensor Categories' [Thi21b] to which we will mostly comply. We will assume categories to be essentially small, i.e. the class of isomorphism classes and Hom-classes form sets.

Let k be a field.

Definition 1.1. Let $\mathcal{C}$ be a k-linear abelian category and $X \in \mathcal{C}$. A *composition series* for X is a sequence

$$0 = X_0 < X_1 < \cdots < X_n = X$$

of objects X_i such that X_i/X_{i-1} is simple.

There is a classic result called the Jordan-Hölder theorem.

Theorem 1.2. *Let*

$$0 = X_0 < X_1 < \cdots < X_n = X$$

and

$$0 = Y_0 < Y_1 < \cdots < Y_m = X$$

be two composition series for X. Then $m = n$ and there exists a permutation σ such that

$$X_i/X_{i-1} \cong Y_{\sigma(i)}/Y_{\sigma(i)-1}$$

for all i. In particular the number

$$[X : S] := |\{i \mid X_i/X_{i-1} \cong S\}$$

for a simple object S is independent of the chosen composition series. We call $[X : S]$ the multiplicity *of S in X.*

A detailed and comprehensible proof can be found in [Thi21b, Theorem 3.3.4].

Definition 1.3. Let $\mathcal{C}$ be a k-linear abelian category.

(a) We call $\mathcal{C}$ *locally finite* if all Hom-spaces are finite dimensional k-vector spaces and all objects are of finite length.

(b) An object $X \in \mathcal{C}$ is called *simple* if it has no non-trivial subobjects.

(c) An object $X \in \mathcal{C}$ is called *semisimple* if it is isomorphic to a direct sum of simple objects.

(d) We call $\mathcal{C}$ *semisimple* if all objects are semisimple.

1.2. Monoidal Categories

We want to recall the notion of a monoidal category following [Eti+16, Chapter 2].

Definition 1.4. A *monoidal category* is given by a tuple $(\mathcal{C}, \otimes, a, \mathbb{1}, \iota)$ where $\mathcal{C}$ is a category, $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is a bifunctor called the *tensor product*, $a : (- \otimes -) \otimes - \to - \otimes (- \otimes -)$ is a natural isomorphism

$$a_{X,Y,Z} : (X \otimes Y) \otimes Z \xrightarrow{\cong} X \otimes (Y \otimes Z)$$

called *associativity constraint*, $\mathbb{1} \in \mathcal{C}$ and $\iota : \mathbb{1} \otimes \mathbb{1} \to \mathbb{1}$ is an isomorphism such that

(i) for all $W, X, Y, Z \in \mathcal{C}$

$$\begin{tikzcd}
 & ((W \otimes X) \otimes Y) \otimes Z \arrow[dl, "a_{W,X,Y} \otimes \mathrm{id}_Z"] \arrow[dr, "a_{W \otimes X, Y, Z}"] & \\
(W \otimes (X \otimes Y)) \otimes Z \arrow[d, "a_{W, X \otimes Y, Z}"] & & (W \otimes X) \otimes (Y \otimes Z) \arrow[d, "a_{W,X,Y \otimes Z}"] \\
W \otimes ((X \otimes Y) \otimes Z) \arrow[rr, "\mathrm{id}_W \otimes a_{X,Y,Z}"] & & W \otimes (X \otimes (Y \otimes Z))
\end{tikzcd}$$

commutes and

(ii) the functors

$$L_1 : X \mapsto \mathbb{1} \otimes X$$
$$R_1 : X \mapsto X \otimes \mathbb{1}$$

are autoequivalences.

We call the object $(\mathbb{1}, \iota)$ the unit object. Usually we refer to (i) as the *pentagon axiom.*

We will in general not write the isomorphisms $\mathbb{1} \otimes X \to X$ and $X \otimes \mathbb{1} \to X$ for more readability.

The additional structure on $\mathcal{C}$ gives rise to a special category of functors preserving that structure

Definition 1.5. (a) Let $(\mathcal{C}, \otimes, \mathbb{1}, a, \iota)$ and $(\tilde{\mathcal{C}}, \tilde{\otimes}, \tilde{\mathbb{1}}, \tilde{a}, \tilde{\iota})$ be monoidal categories. A *monoidal functor* is given by a pair (F, J) where $F : \mathcal{C} \to \tilde{\mathcal{C}}$ and a natural transformation

$$J_{X,Y} : F(X) \tilde{\otimes} F(Y) \xrightarrow{\cong} F(X \otimes Y)$$

such that $F(\mathbb{1}) \cong \tilde{\mathbb{1}}$ and the diagram

$$\begin{tikzcd}
(F(X) \tilde{\otimes} F(Y)) \tilde{\otimes} \arrow[r, "\tilde{a}_{F(X),F(Y),F(Z)}"] \arrow[d, "J_{X,Y} \otimes \mathrm{id}_F(Z)"] & F(X) \tilde{\otimes} (F(Y) \tilde{\otimes} F(Z)) \arrow[d, "\mathrm{id}_F(X) \tilde{\otimes} J_{Y,Z}"] \\
F(X \otimes Y) \tilde{\otimes} F(Z) \arrow[d, "J_{X \otimes Y, Z}"] & F(X) \tilde{\otimes} (F(Y \otimes Z)) \arrow[d, "J_{X, Y \otimes Z}"] \\
F((X \otimes Y) \otimes Z) \arrow[r, "F(a_{X,Y,Z})"] & F(X \otimes (Y \otimes Z))
\end{tikzcd}$$

commutes for all $X, Y, Z \in \mathcal{C}$.

(b) A monoidal functor that is also an equivalence is called *monoidal equivalence.*

Remark 1.6. The definition of a monoidal equivalence may seem unintuitive, since it is not a priory clear that a monoidal weak inverse exists. But in fact whenever there is a monoidal functor which is an equivalence of ordinary categories, then there exists a weak inverse which which can be quipped with a monoidal structure. A detailed discussion can be found in [Thi21a].

From now on let $(\mathcal{C}, \otimes, \mathbb{1}, a, \iota)$ be a monoidal category.

Definition 1.7. Let $X \in \mathcal{C}$. An object X^* is called a *left dual* of X if there exist morphisms $\mathrm{coev}_X : \mathbb{1} \to X \otimes X^*$ and $\mathrm{ev}_X : X^* \otimes X \to \mathbb{1}$, called *coevaluation* and *evaluation*, such that

$$X \xrightarrow{\mathrm{coev}_X \otimes \mathrm{id}_X} (X \otimes X^*) \otimes X \xrightarrow{a_{X,X^*,X}} X \otimes (X^* \otimes X) \xrightarrow{\mathrm{id}_x \otimes \mathrm{ev}_X} X$$
$$X^* \xrightarrow{\mathrm{id}_{X^*} \otimes \mathrm{coev}_X} X^* \otimes (X \otimes X^*) \xrightarrow{a^{-1}_{X^*,X,X^*}} (X^* \otimes X) \otimes X^* \xrightarrow{\mathrm{ev}_X \otimes \mathrm{id}_{X^*}} X^*$$

are the identity morphisms. Analogously we call ${}^*X$ a *right dual* of X if there exist isomorphisms $\overline{\mathrm{coev}}_X : \mathbb{1} \to {}^*X \otimes X$ and $\overline{\mathrm{ev}}_X : X \otimes {}^*X \to \mathbb{1}$ such that

$$X \xrightarrow{\mathrm{id}_X \otimes \overline{\mathrm{coev}}_X} X \otimes ({}^*X \otimes X) \xrightarrow{a^{-1}_{X,{}^*X,X}} (X \otimes {}^*X) \otimes X \xrightarrow{\overline{\mathrm{ev}}_X \otimes \mathrm{id}_X} X$$
$${}^*X \xrightarrow{\overline{\mathrm{coev}}_X \otimes \mathrm{id}_X} ({}^*X \otimes X) \otimes {}^*X \xrightarrow{a_{{}^*X,X,{}^*X}} {}^*X \otimes (X \otimes {}^*X) \xrightarrow{\mathrm{id}_X \otimes \overline{\mathrm{ev}}_X} {}^*X$$

Definition 1.8. An object $X \in \mathcal{C}$ is called *rigid* if it admits a left and a right dual. The category $\mathcal{C}$ is called *rigid* if all objects are rigid.

If X, Y are objects in $\mathcal{C}$ with left duals, then any morphism $f : X \to Y$ induces a morphism $f^* : Y^* \to X^*$ via

$$\begin{aligned} f^* := Y^* &\xrightarrow{\mathrm{id}_{Y^*} \otimes \mathrm{coev}_X} Y^* \otimes (X \otimes X^*) \xrightarrow{a^{-1}_{Y^*,X,X^*}} (Y^* \otimes X) \otimes X^* \\ &\xrightarrow{(\mathrm{id}_{Y^*} \otimes f) \otimes \mathrm{id}_{X^*}} (Y^* \otimes Y) \otimes X^* \xrightarrow{\mathrm{ev}_Y \otimes \mathrm{id}_{X^*}} X^* \end{aligned}$$

There is a similar construction for right duals.

Lemma 1.9 ([Eti+16, Proposition 2.10.8])**.** *Let $\mathcal{C}$ be a monoidal category and $X \in \mathcal{C}$. If X has a left dual X^* then there are adjunction isomorphisms*

$$\mathrm{Hom}(Y \otimes X, Z) \cong \mathrm{Hom}(X, Z \otimes X^*)$$
$$\mathrm{Hom}(X^* \otimes Y, Z) \cong \mathrm{Hom}(Y, X \otimes Z)$$

Let $\mathcal{C}$ be a rigid monoidal category.

Definition 1.10. Let X be an object in $\mathcal{C}$ and $a \in \mathrm{Hom}(X, X^{**})$. Define the *(left) trace* of a

$$\mathrm{Tr}(a) : \mathbb{1} \xrightarrow{\mathrm{coev}_X} X \otimes X^* \xrightarrow{a \otimes \mathrm{id}_X} X^{**} \otimes X^* \xrightarrow{\mathrm{ev}_{X^*}} \mathbb{1}$$

Analogously define the right trace of a morphism $a \in \mathrm{Hom}(X, {}^{**}X)$.

Definition 1.11. (i) A *pivotal structure* on $\mathcal{C}$ is an isomorphism of functors $a : \mathrm{id}_{\mathcal{C}} \xrightarrow{\cong} -^{**}$, i.e. a family of isomorphisms $a_X : X \to X^{**}$ such that $a_{X\otimes Y} = a_X \otimes a_Y$. A monoidal category together with a pivotal structure is called *pivotal.*

(ii) Let $\mathcal{C}$ be pivotal with pivotal structure a. Then the *categorical dimension* of an object X is

$$\dim(X) := \dim_a(X) = \mathrm{Tr}(a_X).$$

(iii) A pivotal structure such that $\dim(X) = \dim(X^*)$ is called *spherical structure.* A monoidal category with a spherical structure is called *spherical.*

(iv) Let a be a spherical structure. Define the *trace* of an endomorphism $f \in \mathrm{End}(X)$ as

$$\mathrm{Tr}(f) = \mathrm{Tr}(a_X \circ f).$$

Definition 1.12. A *braiding* on a monoidal category $\mathcal{C}$ is a natural transformation $c_{X,Y} : X \otimes Y \to Y \otimes X$ such that for all $X, Y, Z \in \mathcal{C}$ the diagrams

$$\begin{tikzcd}
 & X \otimes (Y \otimes Z) \arrow[r, "c_{X,Y\otimes Z}"] & (Y \otimes Z) \otimes X \arrow[dr, "a_{Y,Z,X}"] & \\
(X \otimes Y) \otimes Z \arrow[ur, "a_{X,Y,Z}"] \arrow[dr, "c_{X,Y}\otimes \mathrm{id}_Z"'] & & & Y \otimes (Z \otimes X) \\
 & (Y \otimes X) \otimes Z \arrow[r, "a_{Y,X,Z}"] & Y \otimes (X \otimes Z) \arrow[ur, "\mathrm{id}_Y \otimes c_{Z,X}"'] &
\end{tikzcd}$$

and

$$\begin{tikzcd}
 & (X \otimes Y) \otimes Z \arrow[r, "c_{X\otimes Y,Z}"] & Z \otimes (X \otimes Y) \arrow[dr, "a^{-1}_{Z,X,Y}"] & \\
X \otimes (Y \otimes Z) \arrow[ur, "a^{-1}_{X,Y,Z}"] \arrow[dr, "\mathrm{id}_X\otimes c_{Y,Z}"'] & & & (Z \otimes X) \otimes Y \\
 & X \otimes (Z \otimes Y) \arrow[r, "a^{-1}_{X,Z,Y}"] & (X \otimes Z) \otimes Y \arrow[ur, "c_{X,Z} \otimes \mathrm{id}_Y"'] &
\end{tikzcd}$$

commute. A monoidal category equipped with a braiding is called *braided monoidal category.*

1.3. Tensor Categories

Let k be any field. We follow definitions from [Eti+16, Chapter 4].

Definition 1.13. Let $\mathcal{C}$ be a locally finite k-linear abelian monoidal category. We call $\mathcal{C}$ a

(a) *multiring category* if $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is exact and k-bilinear on morphisms.

(b) *ring category* if it is a multiring category and $\mathrm{End}(\mathbb{1}) \cong k$.

(c) *multitensor category* if it is a rigid multiring category.

(d) *tensor category* if it is a rigid ring category.

Definition 1.14. A finite semisimple (multi)tensor category is called *(multi)fusion category.*

Let $\mathcal{C}$ be a multiring category.

Lemma 1.15. *Let $X, Y \in \mathcal{C}$ such that $X^*, Y^*, (X \oplus Y)^*$ exist. Then $(X \oplus Y)^* \cong X^* \oplus Y^*$.*

Proof. We have isomorphisms

$$(X \oplus Y) \otimes (X^* \oplus Y^*) \xrightarrow{\cong} (X \otimes X^*) \oplus (X \otimes Y^*) \oplus (Y \otimes X^*) \oplus (Y \otimes Y^*)$$

and

$$(X^* \oplus Y^*) \otimes (X \oplus Y) \xrightarrow{\cong} (X^* \otimes X) \oplus (X^* \otimes Y) \oplus (Y^* \otimes X) \oplus (Y^* \otimes Y)$$

Thus we can define evaluation and coevaluation

$$\mathrm{ev}_{X \oplus Y} = i_1 \circ \mathrm{ev}_X + i_4 \circ \mathrm{ev}_Y$$
$$\mathrm{coev}_{X \oplus Y} = \mathrm{coev}_X \circ p_1 + \mathrm{coev}_Y \circ p_4$$

where $i_1, .., i_4, p_1, ..., p_4$ are the inclusion and projection morphisms of the direct sum. It is now easy to verify that $X^* \oplus Y^*$ together with evaluation and coevaluation satisfy the dual property. □

Lemma 1.16 ([Eti+16, Proposition 4.2.10])**.** *A finite ring category with left duals is a tensor category, i.e. has right duals.*

Lemma 1.17 ([Eti+16, Lemma 4.2.11])**.** *Let $\mathcal{C}$ be a multitensor category. If X is simple, then there exists a simple Y such that $X^* \cong Y$.*

Lemma 1.18. *Let X be simple. If X has a left dual then*

$$\dim \mathrm{Hom}(\mathbb{1}, X^* \otimes X) = \dim \mathrm{Hom}(\mathbb{1}, X \otimes X^*) = 1.$$

I.e. the multiplicity of $\mathbb{1}$ in $X^* \otimes X$ is 1. Moreover $\dim \mathrm{Hom}(\mathbb{1}, X \otimes Y) = 1$ if and only if $Y = X^*$.

Proof. We have $\mathrm{Hom}(\mathbb{1}, X^* \otimes X) \cong \mathrm{Hom}(X, X) \cong \mathrm{Hom}(\mathbb{1}, X \otimes X^*)$ and $\mathrm{Hom}(X, X) \cong k$ since X is simple. The second claim follows from $\mathrm{Hom}(\mathbb{1}, X \otimes Y) \cong \mathrm{Hom}(X^*, Y)$. □

Definition 1.19. A spherical braided fusion category $\mathcal{C}$ is called *pre-modular*. Let $X_1, ..., X_n$ be representatives of the isomorphism classes of simple objects. The *S-matrix* of $\mathcal{C}$ is given by

$$S = (S_{ij})_{i,j=1,\dots,n}, \quad S_{ij} = \mathrm{Tr}(c_{X_i, X_j} \circ c_{X_j, X_i}).$$

We call $\mathcal{C}$ *modular* if S is non-degenerate.

Definition 1.20. Let $\mathcal{C}$ be an abelian category. The *Grothendddieck group* $[\mathcal{C}]$ is the free abelian group generated by the isomorphism classes $[X_i]$ of simple objects in $\mathcal{C}$.

If $\mathcal{C}$ is a multiring category the monoidal product induces a multiplication on $\mathcal{Gr}(\mathcal{C})$ given by

$$[X] \otimes [Y] := \sum_{X_i} [X \otimes Y : X_i] X_i$$

where $[X \otimes Y : X_i]$ denotes the multiplicity of X_i in a composition series for $X \otimes Y$ from Theorem 1.2. This makes $[\mathcal{C}]$ into a ring called *Grothendieck ring.*

Remark 1.21. That the above actually defines a ring is not a priori clear. A proof for associativity can be found in [Eti+16, Lemma 4.5.1]. That $[\mathbb{1}]$ is a unit is clear.

Remark 1.22. A more natural way to define the Grothendieck group, respectively ring, is to consider the set of equivalence classes of objects with the addition and multiplication given by the direct sum and monoidal product of representatives. This is well defined and the resulting group isomorphic to the one defined above. For more details follow [Thi21b, Section 3.5].

Now let $\mathcal{C}$ be a semisimple multiring category over k. Let Vec be the category of finite dimensional k-vector spaces.

Definition 1.23. Let V be a finite dimensional k-vector space. Then an object $X \in \mathcal{C}$ representing the functor $\mathrm{Hom}_{\mathrm{Vec}}(V, \mathrm{Hom}_{\mathcal{C}}(X, -)) \colon \mathcal{C} \to \mathrm{Set}$ is denoted by $V \otimes X$.

I.e. if the functor is representable, then

$$\mathrm{Hom}_{\mathcal{C}}(V \otimes X, Y) \cong \mathrm{Hom}_{\mathrm{Vec}}(V, \mathrm{Hom}_{\mathcal{C}}(X, Y))$$

for all $Y \in C$.

Lemma 1.24. *Let V be an n-dimensional vector space and $\mathcal{C}$ an abelian k-linear category. Then $V \otimes X$ exists for all simple $X \in \mathcal{C}$ and is isomorphic to nX.*

Proof. Consider the functor $\mathrm{Hom}_{\mathcal{C}}(X, -)$ and let $n = \dim V$. By the Yoneda Lemma $\mathrm{Hom}_{\mathcal{C}}(X, -)$ is an embedding. Hence

$$\mathrm{Hom}_{\mathcal{C}}(nX, Y) \cong \mathrm{Hom}_{\mathrm{Vec}}(\mathrm{Hom}_{\mathcal{C}}(X, nX), \mathrm{Hom}_{\mathcal{C}}(X, Y)) \cong \mathrm{Hom}_{\mathrm{Vec}}(K^n, \mathrm{Hom}_{\mathcal{C}}(X, Y))$$

□

Corollary 1.25. *Let $\{X_i\}$ be a family of representatives for the isomorphism classes of simple objects in $\mathcal{C}$ and define $H_{ij}^l := \mathrm{Hom}_{\mathcal{C}}(X_l, X_i \otimes X_j)$. Then*

$$X_i \otimes X_j \cong \bigoplus_l H_{i,j}^l \otimes X_l$$

Proof. Since $\mathcal{C}$ is semisimple there is a decomposition $X_i \otimes X_j \cong \bigoplus\limits_l k_l X_l$. Now every morphism from X_k into $X_i \otimes X_j$ is given by the component morphisms. As $\dim \mathrm{Hom}_{\mathcal{C}}(X_l, X_k) = \delta_{lk}$ we have $\dim \mathrm{Hom}_{\mathcal{C}}(X_l, X_i \otimes X_j) = k_l$. □

There are natural isomorphisms

$$\begin{aligned} (X_{i_1} \otimes X_{i_2}) \otimes X_{i_3} &\cong \bigoplus_k \bigoplus_j H_{i_1 i_2}^j \otimes H_{j i_3}^k \otimes X_k \\ X_{i_1} \otimes (X_{i_2} \otimes X_{i_3}) &\cong \bigoplus_k \bigoplus_l H_{i_1 l}^k \otimes H_{i_2 i_3}^l \otimes X_k \end{aligned} \tag{1.1}$$

Thus the associativity constraints reduce to linear isomorphisms

$$\Phi_{i_1 i_2 i_3}^k \colon \bigoplus_j H_{j i_3}^k \otimes H_{i_1 i_2}^j \cong \bigoplus_l H_{i_1 l}^k \otimes H_{i_2 i_3}^l \tag{1.2}$$

Definition 1.26. The component maps

$$\left(\Phi^{k}_{i_1 i_2 i_3}\right)_{jl} : H^{j}_{i_1 i_2} \otimes H^{k}_{j i_3} \to H^{k}_{i_1 l} \otimes H^{l}_{i_2 i_3}$$

are called *6j-symbols*.

2. Reconstructing Semisimple Multiring Categories from 6j-Symbols

2.1. An Equivalent Category

In this chapter we want to discuss in which sense a locally finite semisimple multiring category $\mathcal{C}$ is given by its grothendieck ring, i.e. the fusion rule, and associativity contraints. It quickly becomes clear that such reconstruction can only be done up to equivalence. Hence we will establish an equivalence between the category C and a direct sum[1] $\bigoplus_{X_i}$ Vec of finite dimensional vector spaces where the X_i form a set of representatives for the simple objects in $\mathcal{C}$. We follow the construction from [TY98] where the equivalence is established if the category is finite and the unit object is simple. We use a different notation and will consider the more general setting.

Let $\mathcal{C}$ be a semisimple multiring category with unit $\mathbb{1}$. Denote by $\{X_i\}_{i\in\mathcal{I}}$ the set of simple objects and by $\mathcal{I}_0 \subset \mathcal{I}$ the set such that $\mathbb{1} \cong \bigoplus_{i\in\mathcal{I}_0} X_i$.

Lemma 2.1. *Every object $X \in \mathcal{C}$ is uniquely determined by the family $(\mathrm{Hom}(X_i, X))_{i\in\mathcal{I}}$ of vector spaces.*

Proof. This is clear, since as sets $\mathrm{Hom}(X_i, X) = \mathrm{Hom}(X_i, Y)$ if and only if $X = Y$. □

Intuitively the collection of the Hom-sets represents the decomposition of X into its direct summands. This allows us to define a functor.

Lemma 2.2. *The map*

$$F_0 : \mathrm{Obj}(\mathcal{C}) \to \mathrm{Obj}\left(\bigoplus_{i\in\mathcal{I}} \mathrm{Vec}\right), \quad X \mapsto (\mathrm{Hom}(X_i, X))_{i\in\mathcal{I}}$$

together with the maps

$$F_{X,Y} : \mathrm{Hom}(X,Y) \to \mathrm{Hom}(F_0(X), F_0(Y)), \quad f \mapsto \begin{pmatrix} \mathrm{Hom}(X_i, X) \to \mathrm{Hom}(X_i, Y) \\ g \mapsto f \circ g \end{pmatrix}_{i\in\mathcal{I}}$$

for all $X, Y \in \mathcal{C}$ defines a functor $F : \mathcal{C} \to \bigoplus_{i\in\mathcal{I}} \mathrm{Vec}$.

Proof. Clearly $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ for all $X \in \mathcal{C}$. Let $f : X \to Y,\ g : Y \to Z$ be morphisms. Then

$$F(g) \circ F(f) = (h \mapsto g \circ h)_{i\in\mathcal{I}} \circ (h \mapsto f \circ h)_{i\in\mathcal{I}} = (h \mapsto g \circ f \circ h)_{i\in\mathcal{I}} = F(g \circ h)$$

□

[1] Here the direct sum means the direct sum in the category of abelian categories. This is the subcategory of the cartesian product in which only finitely many entries are non-zero.

The important result now is the following.

Theorem 2.3. *F is an equivalence of categories.*

Proof. We need to show that F is fully faithful and essentially surjective.

Fully faithful: Let $X, Y \in \mathcal{C}$ and $f \in \mathrm{Hom}(X, Y)$. If $f \neq 0$ there exists at least one inclusion $i_k : X_k \to X$ such that $f \circ i_k \neq 0$. Hence $F(f) = 0 \iff f = 0$ and $F_{X,Y}$ is injective. In reverse let $(h_i)_{i\in\mathcal{J}}$ be a family of morphisms $h_i : \mathrm{Hom}(X_i, X) \to \mathrm{Hom}(X_i, Y)$. We define

$$f = \sum_{i\in\mathcal{J}} h_i \circ p_i$$

where p_i the projections onto the X_i. Then clearly $F(f) = (h_i)_{i\in\mathcal{J}}$. Thus F is fully faithful.

Essentially surjective: Let $(V_i)_{i\in\mathcal{J}}$ be a family of vector spaces such that $\dim V_i = 0$ for almost all i. Let $d_i := \dim V_i$. Consider

$$X = \bigoplus_{d_i\neq 0} d_i X_i$$

We have

$$\dim(\mathrm{Hom}(X_i, X)) = d_i$$

and thus $F(X) \cong (V_i)_{i\in\mathcal{J}}$. □

Remark 2.4. Note that in theorem 2.3 it is important that we consider the direct sum of vector space categories and not the direct product, since otherwise the functor is not essentially surjective.

We need the following result for which a proof can be found at [nLaa].

Lemma 2.5. *If a functor $F : \mathcal{C} \to \mathcal{D}$ between categories $\mathcal{C}$ and $\mathcal{D}$ has a left/right adjoint then F preserves (co)limits.*

Thus the equivalence from Theorem 2.3 is an equivalence of abelian categories, since equivalences have left and right adjoints.

In the next step we define a monoidal structure on $\bigoplus_{i\in\mathcal{J}} \mathrm{Vec}$. Recall from corollary 1.25 that the multiplicity of a simple object X_l in a tensor product $X_i \otimes X_j$ is given by the dimension of the space $H^l_{ij} := \mathrm{Hom}(X_l, X_i \otimes X_j)$. This notation allows us to define a monoidal product on the objects of $\bigoplus_{i\in\mathcal{J}} \mathrm{Vec}$.

$$(V_i)_{i\in\mathcal{J}} \otimes (W_j)_{j\in\mathcal{J}} = \left(\bigoplus_{i,j\in\mathcal{J}} H^k_{ij} \otimes V_i \otimes W_j \right)_{k\in\mathcal{J}} \tag{2.1}$$

On morphisms we define the monoidal product similarly as

$$(f_i)_{i\in\mathcal{J}} \otimes (g_j)_{j\in\mathcal{J}} = \left(\bigoplus_{i,j\in\mathcal{J}} \mathrm{id}_{H^k_{ij}} \otimes f_i \otimes g_j \right)_{k\in\mathcal{J}} \tag{2.2}$$

Remark 2.6. Equations (2.1) and (2.2) depend on the order of the summands of the direct sum.

It remains to define the associativity constraints and the unit object. The $6j$-symbols give us isomorphisms (1.2) defining $a_{X_{i_1},X_{i_2},X_{i_3}}$ after applying the direct sum decomposition (1.1):

$$\Phi^k_{i_1 i_2 i_3} : \bigoplus_j H^j_{i_1 i_2} \otimes H^k_{j i_3} \cong \bigoplus_l H^k_{i_1 l} \otimes H^l_{i_2 i_3}.$$

Let $(U_{i_1}),(V_{i_2}),(W_{i_3})$ be families of vector spaces. Then we have natural isomorphisms

$$\begin{aligned} ((U_{i_1}) \otimes (V_{i_2})) \otimes (W_{i_3}) &\cong \left(\bigoplus_{i_1,i_2,i_3 \in \mathcal{I}} \bigoplus_{j \in \mathcal{I}} H^j_{i_1 i_2} \otimes H^k_{j i_3} \otimes U_{i_1} \otimes V_{i_2} \otimes W_{i_3} \right)_{k \in \mathcal{I}} \\ (U_{i_1}) \otimes ((V_{i_2}) \otimes (W_{i_3})) &\cong \left(\bigoplus_{i_1,i_2,i_3 \in \mathcal{I}} \bigoplus_{l \in \mathcal{I}} H^k_{i_1 l} \otimes H^l_{i_2 i_3} \otimes U_{i_1} \otimes V_{i_2} \otimes W_{i_3} \right)_{k \in \mathcal{I}} . \end{aligned} \tag{2.3}$$

Now the associativity is defined in the obvious way by combining the isomorphisms from the $6j$-symbols. And therefore the category $\bigoplus \mathrm{Vec}$ with the given tensor product admits the same $6j$-symbols as $\mathcal{C}$.

The unit object is defined to be the image $F(\mathbb{1})$ of the unit object in $\mathcal{C}$:

$$F(\mathbb{1}) = (\mathrm{Hom}(X_i, \mathbb{1}))_{i \in \mathcal{I}}$$

By construction the functor F is monoidal and hence an equivalence of multiring categories.

2.2. A Skeleton

For the computational aspect of this thesis it would be much easier if in all cases where objects are isomorphic they are actually identical. To achieve this we have the notion of a skeletal category.

Definition 2.7. A *skeletal* category is a category such that for objects X, Y we have $X \cong Y \iff X = Y$.

Our goal now is to show that a multiring category $\mathcal{C}$ is equivalent to skeletal multiring category. Considering arbitrary categories the result is well known and straight forward [nLab]. We construct a skeletal category $\bar{\mathcal{C}}$. Objects of $\bar{\mathcal{C}}$ are representatives $\bar{X}$ for the isomorphism classes of objects in $\mathcal{C}$. The Hom-spaces are the Hom-spaces between representatives. Now to obtain a functor $^- : \mathcal{C} \to \bar{\mathcal{C}}$ we need to choose one isomorphism $i_X : X \xrightarrow{\cong} \bar{X}$ for all X in the isomorphism class of $\bar{X}$. This allows us to define a 'conjugation'

$$(^-) : \mathrm{Hom}(X,Y) \to \mathrm{Hom}(\bar{X},\bar{Y}), \quad (f : X \to Y) \mapsto (\bar{f} : \bar{X} \xrightarrow{i_X^{-1}} X \xrightarrow{f} Y \xrightarrow{i_Y} \bar{Y})$$

which clearly is functorial. Also the functor is by construction surjective and fully faithful, hence an equivalence. Also again by Lemma 2.5 we get that an equivalence of abelian categories.

Now it is well known that Vec is as an abelian category equivalent to the category $\overline{\mathrm{Vec}}$ with objects are natural numbers and $\mathrm{Hom}(m,n) = \mathrm{Mat}_{m \times n}$ [Thi21b, Section 2.4]. Therefore clearly as abelian categories there are equivalences

$$\mathcal{C} \cong \bigoplus_{i \in \mathcal{I}} \mathrm{Vec} \cong \bigoplus_{i \in \mathcal{I}} \overline{\mathrm{Vec}}$$

Let $(n_i), (m_j) \in \bigoplus \overline{\mathrm{Vec}}$. Then we can define the obvious monoidal product

$$(n_i) \otimes (m_j) = \left(\sum_{i,j \in \mathcal{J}} \dim H_{ij}^k n_i m_j \right)_{k \in \mathcal{J}}$$

on objects with associativity constraints given by $\overline{a_{X,Y,Z}}$. The monoidal product on morphisms is defined analogously.

Remark 2.8. The associativity constraints in $\overline{\mathrm{Vec}}$ are not unique and depend on the choices of representatives. But they are unique up to natural isomorphism.

Remark 2.9. We also could have stated that every monoidal category is equivalent to a strict one. This is widely known as Mac Lanes strictness theorem [Eti+16, Theorem 2.8.5]. But the equivalent strict category might not be handleable and also there is not always a strict and skeletal category equivalent to $\mathcal{C}$. [Eti+16, Remark 2.8.7]

Remark 2.10. It would be tempting to assume that the direct sum in a skeletal category is unique. This is in general false since a direct sum is more than just an object. It is equipped with injection and projection morphisms. Thus even though $X \oplus Y = Y \oplus X$ as objects there might be different injection and projection maps such that the direct sums are not equal as (co)cones.

3. TensorCategories.jl and the Framework

The main product of this thesis is the julia package TensorCategories.jl. The package is hosted on Github and can be installed following the instructions from the documentation[1]. julia is the perfect choice to develop such a package since the high-level, high-performance language provides multiple dispatch and just-in-time computation. This enables us to implement very generic methods and keep the interface intuitive. Also we do not need to implement algebraic structures. These are provided by the package Oscar.jl[2]. Oscar is a project combining multiple computer algebra systems like GAP, Singular, Polymake with julia packages like AbstractaAlgebra.jl and Nemo.jl. Especially the GAP.jl package will become handy since it handles all group and representation related tasks.

3.1. The Idea

The core aim of TensorCategories.jl is to provide a framework that makes it easy and convenient to define categories, objects, morphisms and operations between them. At the current developing status the focus will be concentrated on (multi)ring categories and especially fusion categories.

Many things in category theory are defined generically using special morphisms and objects like direct sums, (co)kernels or duals including their characteristic morphisms. The philosophy of our project is to implement as many generic methods as possible to allow a workflow that does not require to implement everything from scratch when a new category is defined.

A second goal is to implement common and well known examples of (tensor) categories such that their can be hands-on examination of the objects and morphisms. In Chapter 5 we will see this in action for objects in the categorical center of a fusion category where in the literature examples are often only computed via equivalences and not explicitly.

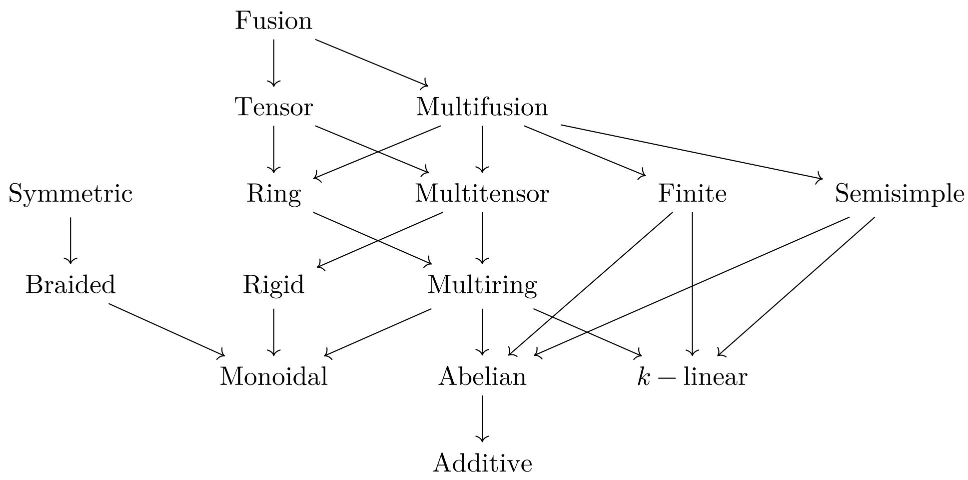
3.2. The Framework

In TensorCategories.jl there are the following abstract types defined:

```julia
abstract type Category end

abstract type Object end

abstract type Morphism end

abstract type VectorSpaceObject <: Object end

abstract type HomSpace <: VectorSpaceObject end
```

[1] https://github.com/FabianMaeurer/TensorCategories.jl.jl
[2] https://github.com/oscar-system/Oscar.jl

Figure 3.1.: Implication graph.




```
abstract type VectorSpaceMorphism <: Morphism end

abstract type HomSet end

abstract type HomSpace <: VectorSpaceObject end
```

That there are abstract types for vector spaces and vector space morphisms is due to the fact that there are many objects which have this structure, e.g. Hom-spaces. Also note that there are no abstract types for ablian, monoidal, e.t.c. categories. There is no practical use for these abstract types other than checking for the structure and this can be done by methods just fine.

In figure Figure 3.1 we can see how properties and structures are implied by each other. For every node in the graph there is a function `is___(C::Category)`::Bool. The graph is implemented in such a way that if a call returns true on any of the nodes, then calling a function below will also return true. This makes implementation of categories more convenient since only the highest level has to be set via `is___(C::MyCategory) = true`.

This does not mean that whenever those functions return false there does not exist such structure or property. Indeed there is no use for a positive return value if the associated functionality is not implemented.

3.2.1. Categories

In TensorCategories.jl a category is a struct extending the abstract type `Category`. For an arbitrary category there are no requirements so far. Objects in this category extend the abstract type `Object` and morphisms extend `Morphism`. The objects and morphisms demand at least some datum of the parent category, domain and codomain respectively. So the minimal structure for a category is the following.

```
struct MyCategory <: Category end
```

```julia
struct MyObject <: Object
    parent::MyCategory
end

struct MyMorphism <: Morphism
    domain::MyObject
    codomain::MyObject
end
```

The axioms of a category require us to define at least methods for the identity and composition of morphisms. Moreover it is always handy to be able to check whether two objects are isomorphic.

```julia
id(X::MyObject)::MyMorphism

compose(f::MyMorphims, g::MyMorphism)::MyMorphim

isisomorphic(X::Object, Y::Object)::Tuple{Bool, MyMorphism}
```

Composition is assumed to be in order, i.e. `compose(f,g)` computes $g \circ f$.

3.2.2. Categories with Additional Structure or Property

Most categories of interest will have additional structures or properties. These have to be implemented and the corresponding indicator method `is___(C::MyCategory)` has to be overwritten. A full set of methods attributed to the specific structures are listed in Table 3.1.

For direct sums and tensor products it is only required to implement methods for two objects. For more input objects or iterators TensorCategories.jl performs the extension.

Hom-Spaces

In any k-linear category the Hom-spaces are vector spaces. Therefore any type of Hom-space has to implement `VectorSpaceObject`.

```julia
struct MyHomSpace <: HomSpace
    X::MyObject
    Y::MyObject
    basis::Vector{MyMorphism}
    parent::VectorSpaces
end
```

Then all functionality that is applicable to morphisms also applies to Hom-spaces. The integration in Hom-functors requires the constructor to be the following.

```julia
Hom(X::MyObject, Y::MyObject)::MyHomSpace
```

3.3. Simple Generic Functionality

There is some generic functionality already available. The most interesting example is the centre of a fusion category which will be discussed in Chapter 5.

3.3.1. Images, Traces and Dimension

If `kernel` and `cokernel` are defined for morphism f we can generically compute the image object.

```
function image(f::Morphism)
    C,c = cokernel(f)
    return kernel(c)
end
```

Having methods `dual, ev, coev` we are able to define a generic method `left_trace` computing the left trace by just writing out the basic definition:

```
function left_trace(f::Morphism)
    V = domain(f)
    W = codomain(f)
    C = parent(V)
    if V == zero(C) || W == zero(C) return zero_morphism(one(C),one(C)) end

    if V == W
    return ev(left_dual(V)) ∘ ((spherical(V)∘f) ⊗ id(left_dual(V))) ∘ coev(V)
    end
    return ev(left_dual(V)) ∘ (f ⊗ id(left_dual(V))) ∘ coev(V)
end
```

Similarly the method `right_trace` is defined. Since we are usually interested in the left trace (which in many examples coincides with the right trace) we set `tr(f::Morphism) = left_trace(f)`. The dimension is immediate by computing the trace of the identity

```
dim(X::Object) = base_ring(X)(id(X))
```

where we assume a conversion function converting a morphism in $\mathrm{Hom}(X, X) \cong k$ into a scalar.

3.3.2. Decompose Objects in a Finite Semisimple Multiring Category

Knowing that $\mathcal{C}$ is a finite semisimple multiring category we can decompose an object X into its simple components. The only requirement is a function

```
simples(C::Category)::Vector{<:Object}
```

returning a full set of non-isomorphic simple objects. Then

```
function decompose(X::Object)
    C = parent(X)
    S = simples(C)
    dimensions = [dim(Hom(X,s)) for s ∈ S]
    return [(s,d) for (s,d) ∈ zip(S,dimensions) if d > 0]
end
```

computes a decomposition. We can also obtain an isomorphism into the direct sum. Let $X_1, ..., X_r$ be representatives for the simple objects. Let $f_{i1}, ..., f_{id_i}$ be a basis of $\mathrm{Hom}(X, X_i)$ and $g_{i1}, ..., g_{id_i}$ a basis of $\mathrm{Hom}(X_i, D)$ where $D = \bigoplus_{i=1}^{r} \dim \mathrm{Hom}(X_i, X) X_i$. Then $\sum_s \sum_{i=1}^{d_s} f_{si} \circ g_{si}$ is an isomorphism. This is combined in the method `decompose_morphism(X::Object)::Morphism.`

3.3.3. The Grothendieck Ring of a Finite Semisimple Multiring Category

Computing the Grothendieck ring in case of a semisimple (multi)Ring category is rather simple. The only additional functionality required is given by

```
isisomorphic(X::MyObject, Y::MyObject)::Tuple{Bool, MyMorphism}
```

which checks for isomorphism. Then the function

```
grothendieck_ring(C::Category)
```

computes the ring $[\mathcal{C}]$. The generic implementation returns an associative algebra of type `Hecke.AlgAss`. This is not ideal since the functionality around this type is not (yet) very reliable.

3.3.4. Opposite Category

TensorCategories.jl provides a wrapper structure for the opposite category. This makes it easier to work with special functors like the contravariant Hom-functor. It implements `Category` and inherits all categorical operations on morphisms and objects from its underlying category.

```
struct OppositeCategory <: Category
C::Category
end

struct OppositeMorphism <: Morphism
domain::OppositeObject
codomain::OppositeObject
m::Morphism
end

struct OppositeObject <: Object
parent::OppositeCategory
X::Object
end
```

3.3.5. Product Category

Similarly the product category is an important tool when we want to define functors. Thus we implement a type where category, objects and morphisms extend the basic structures and provide the necessary operations component-wise.

```
struct ProductCategory{N} <: Category
factors::Tuple
end

struct ProductObject{N} <: Object
parent::ProductCategory{N}
```

```
factors::Tuple
end

struct ProductMorphism{N} <: Morphism
domain::ProductObject{N}
codomain::ProductObject{N}
factors::Tuple
end
```

3.4. Functors

A functor $F : \mathcal{C} \to \mathcal{D}$ between categories is defined by a map on objects and a map on morphisms such that $F(g \circ f) = F(g) \circ F(f)$. Thus in TensorCategories.jl whenever a functor type extending `abstract type Functor` is defined it has to admit at least domain and codomain, as well as an object map and a morphism map.

```
struct MyFunctor <: Functor
    domain::Category
    codomain::Category
    obj_map
    mor_map
end
```

3.4.1. Examples of Functors

The most prevalent examples of functors are the Hom-functors. Let $\mathcal{C}$ be a k-linear category and $X \in \mathcal{C}$. We have three functors:

$$\begin{aligned}
\mathrm{Hom}(X, -) &: \mathcal{C} \to \mathrm{Vec} \\
\mathrm{Hom}(-, X) &: \mathcal{C}^{op} \to \mathrm{Vec} \\
\mathrm{Hom}(-, -) &: \mathcal{C}^{op} \times \mathcal{C} \to \mathrm{Vec}
\end{aligned}$$

These can be constructed naturally with the syntax `Hom(X,:)`, `Hom(:,X)` and `Hom(C)`. Note that for the latter two the constructions from Section 3.3 are necessary.

In Chapter 4 we will see some more implementations of Functors.

Table 3.1.: Attributes and corresponding methods.

Attribute	Implied	Additional Methods

Additive		`dsum(X::MyObject, Y::MyObject)::MyObject`
		`zero(C::MyCategory)::MyObject`
		`dsum(f::MyMorphism, g::MyMorphism)::MyMorphism`
		`+(f::MyMorphism, g::MyMorphism)::MyMorphism`
		`zero_morphism(X::MyObject, Y::MyObject)::MyMorphism`

Abelian	Additive	`ker(f::MyMorphism)::Tuple{MyObject,MyMorphism}` `coker(f::MyMorphism)::Tuple{MyObject, MyMorphism}`
Linear		`+(f::MyMorphism, g::MyMorphism)::MyMorphism` `*(x::FieldElem, f::MyMorphism)::MyMorphism`

Monoidal		`tensor_product(X::MyObject, Y::MyObject)::MyObject`
		`one(C::MyCategory)::MyObject`
		`tensor_product(f::MyMorphism, g::MyMorphism)::MyMorphism`

Rigid	Monoidal	`dual(X::MyObject)::MyObject`
		`ev(X::MyObject)::MyMorphism`
		`coev(X::MyObject)::MyMorphism`

Finite	Abelian Linear	`simples(C::MyCategory)::Vector{MyObject}`
Semisimple	Abelian Linear	
Multiring	Abelian Linear Monoidal	
Multitensor	Multiring Rigid	
Multifusion	Multitensor Finite Semisimple	
Ring	Multiring	`(::Field)(f::MyMorphism)::FieldElem`
Tensor	Ring Rigid	
Fusion	Tensor Finite Semisimple	

4. Examples

4.1. Finite sets

As a very basic example we take a look at the category Set of finite sets.

Definition 4.1. The category Set has as objects the class of finite sets. Morphisms are maps between sets.

As implemented the category of finite sets has not that much structure, but at least product and coproduct are available. Let X, Y be finite sets. Then product and coproduct are given by

$$X \prod Y = X \times Y$$
$$X \coprod Y = X \dot{\cup} Y$$

With TensorCategories.jl we can define objects and morphisms within Set as well as create products and coproducts.

julia

For sets there are the types `SetObject <: Object`, `SetMorphism <: Morphism` and `Sets <: Category`. We can define a `SetObject` from every collection.

```
julia> set = Sets()
Category of finte sets

julia> X = SetObject([1,2,3,4]); Y = SetObject([:v1,:v2,:v3])
Set([:v3, :v2, :v1])

julia> parent(X)
Category of finte sets

julia> coproduct(X,Y)
Set(Any[4, 2, :v3, :v2, 3, 1, :v1])

julia> X × Y
Set([(2, :v2), (3, :v2), (1, :v3), (4, :v3), (1, :v1), (2, :v3), (4, :v1), (2,
 ↪  :v1), (3, :v3), (1, :v2), (4, :v2),
(3, :v1)])
```

4.2. Finite Dimensional Vector Spaces

The category of finite dimensional vector spaces over an arbitrary field k is a prevalent example for a category.

Definition 4.2. The category Vec of finite dimensional k-vector spaces has all finite dimensional vector spaces as objects and k-linear maps as morphisms.

We will not be able to implement the category Vec precisely. Since vector spaces are usually infinite sets we must break them down to a discrete level. There are two generic ways to obtain that. The first being to implement the equivalent subcategory of the vector spaces k^n and the second is to consider pairs (V, b) where b is an ordered basis for V. We choose the second mainly to emphasise on the non-strictness of Vec. These categories are canonically equivalent. In both settings morphisms can be expressed unambiguously by matrices.

The category of finite dimensional (based) vector spaces is a simple example for a tensor category. The operations for direct sums and tensor products are performed on the basis. Recall that for finite dimensional vector spaces V, W with bases $v_1, ..., v_r$ and $w_1, ..., w_s$ the direct sum is given by the vector space $V \oplus W$ with ordered basis $v_i, ..., v_r, w_1, ..., w_s$. Thus for direct sums the bases of the two spaces. Having fixed this order the direct sum of two morphisms $f : V_1 \to W_1$, $g : V_2 \to W_2$ given by matrices M_f, M_g is defined as

$$f \oplus g : V_1 \oplus V_2 \to W_1 \oplus W_2, \quad x \to \begin{pmatrix} M_f & 0 \\ 0 & M_g \end{pmatrix} x.$$

The kernel of a morphism $f : V \to W$ given by a matrix M_f is defined by the right null space of M_f, i.e. the solution set of $M_f x = 0$. Let $b_1, ..., b_l$ be a basis for this space. Then the categorical kernel is given by the tuple

$$\ker f = (k^l, \phi), \quad \phi : k^l \to V, \ x \mapsto (b_1 \cdots b_l)x.$$

Similarly the cokernel is defined by the left null space, i.e. the solution set of $xM_f = 0$. Let $b_1, ..., b_l$ be a basis for this space. Then the categorical cokernel is given by the tuple

$$\operatorname{coker} f = (k^l, \psi), \quad \psi : W \to k^l, \ x \mapsto (b_1 \dots b_l)^t x.$$

The monoidal product of two spaces V, W with bases $v_1, ..., v_r$ and $w_1, ..., w_s$ is defined by the well known tensor product $V \otimes W$ with basis $(v_i \otimes w_j)_{i=1,...,r, j=1,...,s}$ ordered in such a way that the indices are prioritised from left to right. This implies that the tensor product of two morphism $f : V_1 \to W_1$ and $g : V_2 \to W_2$ defined by matrices M, N is given by the Kronecker product.

$$f \otimes g : V_1 \otimes V_2 \to W_1 \otimes W_2, \quad x \mapsto \begin{pmatrix} M_{11}N & \cdots & M_{1s}N \\ \vdots & \ddots & \vdots \\ M_{r1}N & \cdots & M_{rs}N \end{pmatrix} x$$

The associativity constraints are given by the identity matrices interpreted as morphisms $(V \otimes W) \otimes Z \to V \otimes (W \otimes Z)$ which define the map $(v_i \otimes w_j) \otimes z_k \mapsto v_i \otimes (w_j \otimes z_k)$.

julia

A vector space in TensorCategories.jl is of type `VSObject <: VectorSpaceObject` and defined by a basis and a ground field. We define some example vector spaces over $\mathbb{Q}$. There are two main constructors: One takes a `Field` and a vector with any kind of objects as basis, the other a `Field` and the dimension.

```
julia> V = VectorSpaceObject(QQ,2)
Vector space of dimension 2 over Rational Field.

julia> W = VectorSpaceObject(QQ,["a", "b"])
Vector space of dimension 2 over Rational Field.

julia> basis(V ⊕ W)
4-element Vector{Tuple{Int64, String}}:
(1, "v1")
(1, "v2")
(2, "a")
(2, "b")

julia> basis(V⊗W)
4-element Vector{Tuple{String, String}}:
("v1", "a")
("v1", "b")
("v2", "a")
("v2", "b")
```

The category of finite dimensional vector spaces can be constructed as a `VectorSpaces <: Category` object like so

```
julia> Vec = VectorSpaces(QQ)
Category of finite dimensional VectorSpaces over Rational Field

julia> parent(V) == Vec
true
```

Vector space morphisms are of type `VSMorphism <: Morphism` and are defined by their domain, codomain and representing matrix. Hom spaces are of type `VSHomSpace <: HomSpace` which is a subtype of `VectorSpaceObject`.

```
julia> H = Hom(V,W)
Vector space of dimension 4 over Rational Field.

julia> f1,f2,f3,f4 = basis(H);

julia> f = 2*f1 + f3 - f4
Vector space morphism with
Domain:Vector space of dimension 2 over Rational Field.
Codomain:Vector space of dimension 2 over Rational Field.

julia> g = f2 - 3*f3 + 3*f4
Vector space morphism with
```

```
Domain:Vector space of dimension 2 over Rational Field.
Codomain:Vector space of dimension 2 over Rational Field.

julia> matrix(f)
[2    1]
[0   -1]

julia> matrix(g)
[0   -3]
[1    3]

julia> matrix(f⊕g)
[2    1   0    0]
[0   -1   0    0]
[0    0   0   -3]
[0    0   1    3]

julia> matrix(f⊗g)
[0    0   -6   -3]
[0    0    0    3]
[2    1    6    3]
[0   -1    0   -3]
```

Let V be any finite dimensional k-vector space. Then the *dual* vector space of V is $V^* = \mathrm{Hom}(V, k)$. To see that this is in fact the categorical (left) dual of V we need to provide evaluation and coevaluation maps. Let $v_1, ..., v_n$ be a basis of V. Then there is a dual basis in V^* given by $f_1, ..., f_n$ where $f_i : V \to k,\ v_j \mapsto \delta_{ij}$. So we define

$$\begin{array}{rccccccc} \mathrm{ev}_V : & V^* \otimes V & \to & k; & f \otimes v & \mapsto & f(v) \\ \mathrm{coev}_V : & k & \to & V \otimes V^*; & t & \mapsto & t \cdot \sum_{i=1}^{n} f_i \otimes v_i \end{array}$$

Expressing those maps in the standard bases of V, V^*, $V^* \otimes V$ and $V \otimes V^*$ we can explicitly write down the matrix representations

$$\mathrm{ev}_V = \begin{pmatrix} e_1^t & \dots & e_n^t \end{pmatrix}$$

$$\mathrm{coev}_V = \begin{pmatrix} e_1 \\ \vdots \\ e_n \end{pmatrix}$$

where e_i is the i-th standard basis vector in k^n.

julia

Duals of vector spaces in TensorCategories.jl are constructed by defining a vector space whose basis are the projection maps.

```
julia> V = VectorSpaceObject(QQ,3)
Vector space of dimension 3 over Rational Field.
```

```
julia> basis(dual(V))
3-element Vector{VectorSpaceMorphism{fmpq}}:
Vector space morphism with
Domain:Vector space of dimension 3 over Rational Field.
Codomain:Vector space of dimension 1 over Rational Field.
Vector space morphism with
Domain:Vector space of dimension 3 over Rational Field.
Codomain:Vector space of dimension 1 over Rational Field.
Vector space morphism with
Domain:Vector space of dimension 3 over Rational Field.
Codomain:Vector space of dimension 1 over Rational Field.

julia> matrix(ev(V))
[1]
[0]
[0]
[0]
[1]
[0]
[0]
[0]
[1]

julia> matrix(coev(V))
[1   0   0   0   1   0   0   0   1]
```

And we can check for the condition

```
julia> matrix((id(V)⊗ev(V)) ∘ associator(V, dual(V), V) ∘ (coev(V)⊗id(V)))
[1   0   0]
[0   1   0]
[0   0   1]
```

4.3. Graded Vector Spaces

It becomes more interesting as soon as we consider vector spaces endowed with a grading:

Definition 4.3. Let G be a finite group. A G-*graded* vector space is a vector space V together with a direct sum decomposition $V \cong \bigoplus_{g\in G} V_g$. A morphism f between G-graded vector spaces V, W is called G-*graded* if $f(V_g) \subset W_g$ for all $g \in G$.

The finite dimensional G-graded vector spaces together with graded vector space morphisms form the category $\mathrm{Vec_G}$.

For computational reasons we assume that the basis of a vector space is G-graded, i.e. every basis vector belongs to one simple graded component. This formally leads to an equivalent full subcategory of $\mathrm{Vec_G}$. The assumption allows us to efficiently implement the objects as vector spaces for which we specify a grading for every simple subobject, i.e. a grading for every basis vector.

Let V, W be finite dimensional G-graded vector spaces with graded bases $v_1, ..., v_r$ and $w_1, ..., w_s$. The gradings of the bases is stored as sequences $(g_1, ..., g_r)$ and $(h_1, ..., h_s)$ where $g_i, h_j \in G$. Thus the direct sum of $V \oplus W$ is the inherited from Vec with grading sequence $(g_1, ..., g_r, h_1, ..., h_s)$. The direct sum of two graded vector space morphisms is again clearly graded. Simple objects in Vec are given by the one dimensional vector spaces. They are isomorphic if and only if they are graded by the same group element, whence the set of simple objects is $\{k_g \mid g \in G\}$ where k_g is the one dimensional space graded with g.

Let $f : V \to W$ be a graded morphism. Then $\ker f = (k^l, \phi)$ of f in Vec can be naturally endowed with a grading making it a graded kernel. Consider the graded component maps $f_g : V_g \to W_g$. Then $\ker f \cong \bigoplus \ker f_g$ implies a grading on $\ker f$ by grading $\phi^{-1}(V_g)$ with g. Similarly the cokernel is G-graded.

The tensor product $V \otimes W$ also comes naturally with a grading. The component vector space with basis vector $v_i \otimes v_j$ is graded by $g_i h_j$ defining the direct sum decomposition

$$V \otimes W \cong \bigoplus_{g \in G} (V \otimes W)_g, \quad (V \otimes W)_g \cong \bigoplus_{xy=g} V_x \otimes V_y.$$

The tensor product of graded morphisms therefore is naturally graded. The unit object is the one dimensinal vector space k graded with $1 \in G$. Associativity is also inherited from Vec since the associativity constraints are clearly G-graded.

julia

In TensorCategories.jl graded vector spaces are of type `GVSObject <: VectorSpaceObject`. All finite groups of type `GAPGroup` are supported as base groups. Graded vector space objects store a `VectorSpaceObject` and a vector assigning each basis element a grading element from G.

```
julia> G = symmetric_group(3); g,s = gens(G);

julia> V = VectorSpaceObject(g => VectorSpaceObject(QQ,2), s =>
 ↪  VectorSpaceObject(QQ,3))
Graded vector space of dimension 5 with grading
PermGroupElem[(1,2,3), (1,2,3), (1,2), (1,2), (1,2)]

julia> W = VectorSpaceObject(s*g => VectorSpaceObject(QQ,2),g =>
 ↪  VectorSpaceObject(QQ,2), s => VectorSpaceObject(QQ,2))
Graded vector space of dimension 6 with grading
PermGroupElem[(1,3), (1,3), (1,2,3), (1,2,3), (1,2), (1,2)]

julia> V ⊕ W
Graded vector space of dimension 11 with grading
PermGroupElem[(1,2,3), (1,2,3), (1,2), (1,2), (1,2), (1,3), (1,3), (1,2,3),
 ↪  (1,2,3), (1,2), (1,2)]


julia> V ⊗ W
Graded vector space of dimension 30 with grading
```

```
PermGroupElem[(1,2), (1,2), (1,2,3), (1,2,3), (1,2,3), (1,2), (1,2), (1,2,3),
 ↪  (1,2,3), (1,2,3), (1,3,2), (1,3,2), (1,3), (1,3), (1,3), (1,3,2), (1,3,2),
 ↪  (1,3), (1,3), (1,3), (2,3), (2,3), (), (), (), (2,3), (2,3), (), (), ()]
```

The category of graded vector spaces is of type `GradedVectorSpaces <: Category` and is constructed as follows.

```
julia> VecG = GradedVectorSpaces(QQ,G)
Category of G-graded vector spaces over Rational Field where G is Sym( [ 1 .. 3 ]
 ↪  )
```

Graded vector space morphisms can be constructed with type `GVSMorphism <: VectorSpaceMorphism`. It is easiest to build them from the bases of Hom-spaces to ensure they are graded. But there is also a constructor taking a `MatElem` and checking for well-definition.

```
julia> H = Hom(V,W)
Vector space of dimension 10 over Rational Field.

julia> B = basis(H);

julia> f = B[1] - B[2] + B[3] - B[6] + 3*B[8] + 2*B[9]
Vector space morphism with
Domain:Graded vector space of dimension 5 with grading
PermGroupElem[(1,2,3), (1,2,3), (1,2), (1,2), (1,2)]

Codomain:Graded vector space of dimension 6 with grading
PermGroupElem[(1,3), (1,3), (1,2,3), (1,2,3), (1,2), (1,2)]


julia> H = Hom(V,W)
Vector space of dimension 10 over Rational Field.

julia> B = basis(H);

julia> f = B[1] - B[2] + B[3] - B[6] + 3*B[8] + 2*B[9]
Vector space morphism with
Domain:Graded vector space of dimension 5 with grading
PermGroupElem[(1,2,3), (1,2,3), (1,2), (1,2), (1,2)]
Codomain:Graded vector space of dimension 6 with grading
PermGroupElem[(1,3), (1,3), (1,2,3), (1,2,3), (1,2), (1,2)]

julia> g = -B[1] + B[2] + 2*B[4];

julia> kernel(f⊗g)
(Graded vector space of dimension 17 with grading
PermGroupElem[(), (), (), (), (), (2,3), (2,3), (1,3,2), (1,3,2), (1,3,2),
 ↪  (1,3,2), (1,3), (1,3), (1,3), (1,3), (1,3), (1,3)], Vector space morphism
 ↪  with
Domain:Graded vector space of dimension 17 with grading
PermGroupElem[(), (), (), (), (), (2,3), (2,3), (1,3,2), (1,3,2), (1,3,2),
 ↪  (1,3,2), (1,3), (1,3), (1,3), (1,3), (1,3), (1,3)]
```

```
Codomain:Graded vector space of dimension 25 with grading
PermGroupElem[(1,3,2), (1,3,2), (1,3), (1,3), (1,3), (1,3,2), (1,3,2), (1,3),
 ↪  (1,3), (1,3), (2,3), (2,3), (), (), (), (2,3), (2,3), (), (), (), (2,3),
 ↪  (2,3), (), (), ()])
```

The category of finite dimensional graded vector spaces is also rigid. Let V be a finite dimensional vector space with grading $(g_1, ..., g_n)$. Then we can grade $V^* = \mathrm{Hom}(V, k)$ with $(g_1^{-1}, ..., g_n^{-1})$. On simple objects its easy to see that the identity $\mathrm{id} : k_{g^{-1}} \otimes k_g = k_1 \to k_g \otimes k_{g^{-1}} = k_1$ is a coevaluation map and also an evaluation map. Therefore in spirit of Lemma 1.15 the (co)evaluation maps of V are given by the maps

$$\mathrm{ev} : V^* \otimes V \to k_1, \quad f_i \otimes v_j \mapsto \delta_{ij} \tag{4.1}$$

$$\mathrm{coev} : k_1 \to V \otimes V^*, \quad t \mapsto t \sum_{i=1}^{n} v_i \otimes f_i \tag{4.2}$$

and thus coinciding with the (co)evaluation in Vec.

4.3.1. Twisted Graded Vector Spaces

The restriction to graded vector space morphisms allows the definition of alternative associators on Vec_G.

Definition 4.4. Let $\omega : G \times G \times G \to k^*$ be a 3-cocycle of G. The category of *twisted G-graded vector spaces* Vec_G^ω is given by the objects and morphisms of Vec_G with associativity constraints $a_{k_g,k_h,k_i} = \omega(g, h, i)$.

That ω is a three 3-cocycle ensures that the associativity constraints are well defined. Indeed for G-graded vector spaces the 3-cocycle condition is equivalent to the pentagon axiom. Morphisms, direct sums, (co)kernels and tensor products are all inherited from Vec_G. Only duals have a slight twist. As object the dual is still the same, but evaluation and coevaluation have to be adjusted. Considering the condition for duals

$$k_g \to (k_g \otimes k_{g^{-1}}) \otimes k_g \xrightarrow{a_{k_g,k_{g^{-1}},k_g}} k_g \otimes (k_{g^{-1}} \otimes k_g) \to k_g$$

we see that we need to adjust either the evaluation or the coevaluation by the factor $\omega(g, g^{-1}, g)^{-1} = \omega(g^{-1}, g, g^{-1})$.

To take a look at an example we need a specific 3-cocycle for some group. The following result can be found in [Eti+16, Example 2.6.4].

Proposition 4.5. *Let $G = \langle g \mid g^n = 1\rangle$ be cyclic of order m and let ξ_m be a primitive m-th root of unity. Then the maps*

$$\omega_a : G^3 \to k^*, \quad (g^{i_1}, g^{i_2}, g^{i_3}) \mapsto \xi_m^{\frac{a i_1 (i_2 + i_3 - (i_2+i_3)')}{m}}, \quad a = 1, ..., m$$

where $(-)'$ is the remainder after division with remainder by m.

julia

We construct Vec_G^ω where G is cyclic of order 5 and ω is a non-trivial 3-cocycle as seen in Proposition 4.5.
The cocycle can be generated with the function `cyclic_group_3cocycle`:

```
function cyclic_group_3cocycle(G::GAPGroup, F::Field, ξ::FieldElem)
    g = G[1]
    n = order(G)
    D = Dict((g^i,g^j,g^k) => ξ^(div(i*(j+k - rem(j+k,n)),n)) for i ∈ 1:n, j ∈
     ↪  1:n, k ∈ 1:n)
    return Cocycle(G,D)
end
```

Now the category and some vector spaces can be defined. It is important to define the category over a cyclotomic field (or something bigger) that contains a primitive fifth root of unity, since otherwise there are no non-trivial cocycles.

```
julia> G = cyclic_group(5);

julia> F,ξ = CyclotomicField(5, "ξ")
(Cyclotomic field of order 5, ξ)

julia> c = cyclic_group_3cocycle(G,F,ξ)
3-Cocycle of <pc group of size 5 with 1 generators>

julia> VecG = GradedVectorSpaces(F,G,c)
Category of G-graded vector spaces over Cyclotomic field of order 5 where G is
 ↪  <pc group of size 5 with 1 generators>

julia> s = simples(VecG);

julia> matrix(associator(s[2],[3],s[4]))
[ξ]

julia> V = s[2]^2 ⊕ s[3] ⊕ s[4]^2
Graded vector space of dimension 5 with grading
PcGroupElem[f1, f1, f1^2, f1^3, f1^3]

julia> matrix(ev(V))
[-ξ^3 - ξ^2 - ξ - 1]
[                 0]
[                 0]
[                 0]
[                 0]
[                 0]
[-ξ^3 - ξ^2 - ξ - 1]
[                 0]
[                 0]
[                 0]
[                 0]
[                 0]
```

```
[               ξ^3]
[                 0]
[                 0]
[                 0]
[                 0]
[                 0]
[               ξ^2]
[                 0]
[                 0]
[                 0]
[                 0]
[                 0]
[               ξ^2]

julia> matrix((id(V)⊗ev(V))∘associator(V,dual(V),V)∘(coev(V)⊗id(V)))
[1   0   0   0   0]
[0   1   0   0   0]
[0   0   1   0   0]
[0   0   0   1   0]
[0   0   0   0   1]
```

4.3.2. Forgetful Functors

There is an intuitive forgetful functor $F : \mathrm{Vec}_G \to \mathrm{Vec}$ dropping the graded structure.

julia

```
julia> G = symmetric_group(3); g,s = gens(G);

julia> VecG = GradedVectorSpaces(QQ,G)
Category of G-graded vector spaces over Rational Field where G is Sym( [ 1 .. 3 ]
 ↪  )

julia> F = Forgetful(VecG, VectorSpaces(QQ))
Forgetful functor from Category of G-graded vector spaces over Rational Field
 ↪  where G is Sym( [ 1 .. 3 ] ) to Category of finite dimensional VectorSpaces
 ↪  over Rational Field
```

This is a functor so we can apply it to objects and morphisms.

```
julia> V = s[1]^3 ⊕ s[3]^2;

julia> F(V)
Vector space of dimension 5 over Rational Field.

julia> F(id(V))
Vector space morphism with
Domain:Vector space of dimension 5 over Rational Field.
Codomain:Vector space of dimension 5 over Rational Field.
```

4.4. Representation Categories of Finite Groups over Finite Fields

In this section we consider the category of finite dimensional representations of finite groups over k. The implementation relies heavily on the MeatAxe algorithm [Rin] which computes irreducible representations of finite groups over finite fields. Therefore most functionality in TensorCategories.jl is currently only available for finite fields. This is in not too much of a restriction since for large enough characteristic the results can be might be lifted to $\mathbb{C}$ for many examples (see for example [Fei68]).

Definition 4.6. Let G be a finite group. The category $\text{Rep}(G) := \text{Rep}_k(G)$ consists of all finite dimensional representations $\rho : G \to \text{GL}(V)$. A Morphism $\rho \to \tau$ between representations $\rho : G \to \text{GL}(V)$ and $\tau : G \to \text{GL}(W)$ is given by a linear map $f : V \to W$ such that for all $g \in G$ the diagram

$$\begin{tikzcd} V \arrow[r, "f"] \arrow[d, "\rho(g)"'] & W \arrow[d, "\tau(g)"] \\ V \arrow[r, "f"] & W \end{tikzcd}$$

commutes. We denote this map by $f : \rho \to \tau$.

Group representations in TensorCategories.jl are of type `GroupRepresentation <: Object` and the category is of type `GroupRepresentationCategory <: Category`. A group representation has additional fields for a `GAPHomomorphism` which encodes a morphism with domain G and codomain $\text{GL}(V)$. The reason for this is the use of the GAP.jl and the MeatAxe to compute decompositions, Hom-spaces and others.

For morphisms there is the type `GroupRepresentationMorphism <: VectorSpaceMorphism`. It stores the (co)domain and a linear map in form of a matrix.

4.4.1. Direct Sums, Kernels and Cokernels

The category $\text{Rep}(G)$ is in abelian. The direct sum of two representations $\rho : G \to \text{GL}(V)$, $\tau : G \to \text{GL}(W)$ is given by

$$(\rho \oplus \tau) : G \to \text{GL}(V \oplus W), \quad (\rho \oplus \tau)(g) := \begin{pmatrix} \rho(g) & 0 \\ 0 & \tau(g) \end{pmatrix}$$

and the direct sum of morphisms is given by the direct sum of vector space morphisms.

Let $f : \rho \to \tau$ be a morphism of representations. Kernel and cokernel of f are given (κ, ϕ) and (ζ, ψ) such that the diagram

$$\begin{tikzcd} 0 \arrow[r] \arrow[d] & k^l \arrow[r, "\phi"] \arrow[d, "\kappa(g)"] & V \arrow[r, "f"] \arrow[d, "\rho(g)"] & W \arrow[r, "\psi"] \arrow[d, "\tau(g)"] & k^m \arrow[r] \arrow[d, "\zeta(g)"] & 0 \arrow[d] \\ 0 \arrow[r] & k^l \arrow[r, "\phi"] & V \arrow[r, "f"] & W \arrow[r, "\psi"] & k^m \arrow[r] & 0 \end{tikzcd}$$

commutes for all $g \in G$ and is exact. Let M be the matrix representing f, $v_1, \dots, v_l$ a basis of $\ker M$ and $w_1, \dots, w_m$ a basis for $\operatorname{coker} M$. Therefore $\phi = (v_1 \cdots v_l)$ and $\psi = (w_1 \cdots w_m)^t$ make the rows exact. It remains to define κ and ζ accordingly. Since ϕ is injective it admits a left inverse ϕ' and similarly ψ admits a right inverse ψ'. We define

$$\kappa(g) := \phi' \circ \rho(g) \circ \phi$$
$$\zeta(g) := \psi \circ \tau(g) \circ \psi'.$$

The five-lemma ensures that $\kappa(g)$ and $\zeta(g)$ are indeed automorphisms.

julia

The category $\mathrm{Rep}(G)$ is constructed in the following example.

```
julia> G = symmetric_group(5); g,s = gens(G)
2-element Vector{PermGroupElem}:
(1,2,3,4,5)
(1,2)

julia> RepG = RepresentationCategory(G,FiniteField(23)[1])
Representation Category of Sym( [ 1 .. 5 ] ) over Galois field with
 ↪  characteristic 23

julia> S = simples(RepG)
7-element Vector{GroupRepresentation{gfp_elem, PermGroup}}:
1-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 5 ] ))
1-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 5 ] ))
4-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 5 ] ))
4-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 5 ] ))
5-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 5 ] ))
5-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 5 ] ))
6-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 5 ] ))

julia> ρ = S[2]^2 ⊕ S[4]
6-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 5 ] ))

julia> ρ(g)
[1   0    0    0    0   0]
[0   1    0    0    0   0]
[0   0    2   17    0   0]
[0   0   22    8    1   9]
[0   0    0    2   11   0]
[0   0    0    0   22   1]
```

```
julia> τ = S[2] ⊕ S[4]^2 ⊕ S[6]
14-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 5 ] ))

julia> B = basis(Hom(ρ,τ));

julia> f = B[1] - B[2] + B[3];

julia> kernel(f)
(1-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 5 ] )), Group representation Morphism with defining matrix
[1 1 0 0 0 0])

julia> cokernel(f)
(9-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 5 ] )), Group representation Morphism with defining matrix
[0 0 0 0 0 0 0 0 0; 0 0 0 0 0 0 0 0 0; 0 0 0 0 0 0 0 0 0; 0 0 0 0 0 0 0 0 0; 0 0
 ↪  0 0 0 0 0 0 0; 1 0 0 0 0 0 0 0 0; 0 1 0 0 0 0 0 0 0; 0 0 1 0 0 0 0 0 0; 0 0 0
 ↪  1 0 0 0 0 0; 0 0 0 0 1 0 0 0 0; 0 0 0 0 0 1 0 0 0; 0 0 0 0 0 0 1 0 0; 0 0 0 0
 ↪  0 0 0 1 0; 0 0 0 0 0 0 0 0 1])
```

Again we see that is might be useful to compute the simple objects. Those are all irreducible representations. Thankfully this is taken care of by the implementation of the MeatAxe algorithm in GAP. Also most other queries for representations like `isisomorphic`, `isireducible` or `decompose` are fulfilled by the MeatAxe module in GAP.

Remark 4.7. The category $\mathrm{Rep}(G)$ is semisimple if and only if $\mathrm{char}(k) \nmid |G|$. This is well known as Maschke's Theorem. For a proof see for example [Thi21b, Theorem 3.4.3].

4.4.2. Tensor Product and Duals

The category $\mathrm{Rep}(G)$ is a rigid monoidal category. The tensor product of two representations ρ, τ is defined as

$$(\rho \otimes \tau) : G \to \mathrm{GL}(V \otimes W), \quad (\rho \otimes \tau)(g) := \rho(g) \otimes \tau(g).$$

The tensor product of morphisms is given by the Kronecker product of the defining matrices. It follows that the unit object is the trivial representation $\mathbb{1} : G \to \mathrm{GL}(k)$, $\mathbb{1}(g) = 1$. The associativity constraints are the canonical isomorphisms $(V \otimes W) \otimes Z \to V \otimes (W \otimes Z)$.

The dual of a representation $\rho : G \to \mathrm{GL}(V)$ is given by

$$\rho^* : G \to \mathrm{GL}(V^*), \quad \rho^*(g) := (f \mapsto f \circ \rho(g^{-1}))$$

This is clearly a representation of G. Now the categorical notion of a dual requires the existence of evaluation and coevaluation which are provided by

$$\mathrm{coev}_\rho : \mathbb{1} \to \rho \otimes \rho^*, \text{ defined by the matrix of } \mathrm{coev}_{k^{\dim \rho}}$$

and
$$\mathrm{ev}_\rho : \rho^* \otimes \rho \to \mathbb{1}, \text{ defined by the matrix of } \mathrm{ev}_{k^{\dim \rho}}$$

These morphisms are both well defined since G acts transitively on itself. Consider a basis (v_i) of $k^{\dim \rho}$ with dual basis (f_i) of $(k^{\dim \rho})^*$. Then

$$g \cdot \mathrm{coev}_{k^{\dim \rho}}(x) = x(g \cdot \sum v_i \otimes f_i) = x \sum v_i \otimes f_1 = \mathrm{coev}_{k^{\dim \rho}}(x)$$

as well as

$$\mathrm{ev}_{k^{\dim \rho}}(g \cdot (f_i \otimes v_i)) = \mathrm{ev}_{k^{\dim \rho}}(f_i \circ \rho(g^{-1}) \otimes gv_i) = f_i(\rho(g^{-1}\rho(g)v_i)) = f_i(v_i) = 1.$$

For the implementation we only consider the abstract space $k^{\dim \rho}$ and not $V^* = \mathrm{Hom}(V, k)$. We define the action $\rho^*(g) = \rho(g^{-1})^t$ which is canonically isomorphic.

julia

We construct the dual of representations.

```
julia> dual(ρ)
6-dimensional group representation over Galois field with characteristic 23 of
↪  Sym( [ 1 .. 5 ] ))

julia> ev(ρ)
Group representation Morphism with defining matrix
[1; 0; 0; 0; 0; 0; 0; 1; 0; 0; 0; 0; 0; 0; 1; 0; 0; 0; 0; 0; 0; 1; 0; 0; 0; 0; 0;
↪  0; 1; 0; 0; 0; 0; 0; 0; 1]

julia> (id(ρ)⊗ev(ρ))∘associator(ρ,dual(ρ),ρ)∘(coev(ρ)⊗id(ρ)) == id(ρ)
true
```

4.4.3. Induction and Restriction of Representations

We want to take a closer look on some functors we will need in sections 4.5 and 4.6.

Definition 4.8. Let $\rho : G \to \mathrm{GL}(V)$ be a representation. The *restriction* of ρ to $H \leq G$ is given by the representation
$$\mathrm{Res}_H^G(\rho) := \rho|_H, \quad \rho|_H(g) := \rho(g)|_H$$

Let $H \leq G$ and $\tau : H \to \mathrm{GL}(V)$ a representation. Let V_τ be the associated kH module. Then the *induction* of τ to H is given by the kG-module

$$\mathrm{Ind}_H^G(\tau) = kG \otimes_{kH} V_\tau$$

It is a standard result that the same operations on morphisms are well defined such that restriction and induction form functors $\mathrm{Ind}_H^G : \mathrm{Rep}(H) \to \mathrm{Rep}(G)$ and $\mathrm{Res}_H^G : \mathrm{Rep}(G) \to \mathrm{Rep}(H)$.

The implementation of restriction functors is obvious. For the induction we need to compute the action of G on $kG \otimes_{kH} V_\tau$ explicitly. To obtain this we need the action of G on representatives of the left cosets of H in G. If $g_1, ..., g_n$ are representatives of the left cosets then for each $g \in G$

there exist $h_1, ..., h_n \in H$ and $j(i)$ such that $gg_i = g_{j(i)}h_i$. Clearly $kG \otimes V_\tau \cong \bigoplus g_i V_\tau$ where $g_i V_\tau$ are disjoint isomorphic copies of V_τ. For computational purposes we fix an order of for the summands. Then the action of $g \in G$ on an element of $kG \otimes_{kH} V_\tau$ is given by

$$g \cdot \sum g_i v_i = \sum g_{j(i)} \tau(h_i) v_i.$$

From this we build a matrix representation by considering the permutation matrix of g acting on the cosets and replacing the non-zero entry in the i-th row by the action matrix $\tau(h_i)$.

On morphisms the implementation of the restriction is again trivial. In case of the induction consider a morphism $f : \tau \to \rho$ of representations for H. We have $\mathrm{Ind}(f) = \mathrm{id}(kG) \otimes f$ which means the induced morphism is given by copying the matrix of f onto the diagonal.

julia

Induction and restriction functors have their own types.

```
julia> G = symmetric_group(5); H = subgroups(G)[140]
Group([ (2,5), (1,4,3), (3,4) ])

julia> RepG = RepresentationCategory(G,FiniteField(23)[1])
Representation Category of Sym( [ 1 .. 5 ] ) over Galois field with
 ↪  characteristic 23

julia> RepH = RepresentationCategory(H,FiniteField(23)[1])
Representation Category of Group([ (2,5), (1,4,3), (3,4) ]) over Galois field
 ↪  with characteristic 23

julia> S = simples(RepG);

julia> ρ = S[2]^2 ⊕ S[4]
6-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 5 ] ))

julia> R = simples(RepH);

julia> τ = R[2] ⊗ R[3]
1-dimensional group representation over Galois field with characteristic 23 of
 ↪  Group([ (2,5), (1,4,3), (3,4) ]))

julia> Res = Restriction(RepG,RepH)
Restriction functor from Representation Category of Sym( [ 1 .. 5 ] ) over Galois
 ↪  field with characteristic 23 to Representation Category of Group([ (2,5),
 ↪  (1,4,3), (3,4) ]) over Galois field with characteristic 23.

julia> Ind = Induction(RepH, RepG)
Induction functor from Representation Category of Group([ (2,5), (1,4,3), (3,4)
 ↪  ]) over Galois field with characteristic 23 to Representation Category of
 ↪  Sym( [ 1 .. 5 ] ) over Galois field with characteristic 23.

julia> Res(ρ)
```

```
6-dimensional group representation over Galois field with characteristic 23 of
 ↪  Group([ (2,5), (1,4,3), (3,4) ]))

julia> Ind(τ)
10-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 5 ] ))

julia> f = sum(basis(End(ρ))[[1,2,4]])
Group representation Morphism with defining matrix
[1 1 0 0 0 0; 0 1 0 0 0 0; 0 0 0 0 0 0; 0 0 0 0 0 0; 0 0 0 0 0 0; 0 0 0 0 0 0]

julia> g = sum(basis(End(τ^2))[[1,2,3]])
Group representation Morphism with defining matrix
[1 1; 1 0]

julia> Res(f)
Group representation Morphism with defining matrix
[1 1 0 0 0 0; 0 1 0 0 0 0; 0 0 0 0 0 0; 0 0 0 0 0 0; 0 0 0 0 0 0; 0 0 0 0 0 0]

julia> Ind(g)
Group representation Morphism with defining matrix
[1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0; 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0; 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0; 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0; 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0; 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
 ↪  0 0 0 0 0; 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0; 0 0 0 0 0 0 1 0 0 0 0 0 0
 ↪  0 0 0 0 0 0 0; 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0; 0 0 0 0 0 0 0 0 1 0 0
 ↪  0 0 0 0 0 0 0 0 0; 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0; 0 0 0 0 0 0 0 0 0
 ↪  0 1 0 0 0 0 0 0 0 0 0; 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0; 0 0 0 0 0 0 0
 ↪  0 0 0 0 0 1 0 0 0 0 0 0 0; 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0; 0 0 0 0 0
 ↪  0 0 0 0 0 0 0 0 0 1 0 0 0 0 0; 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0; 0 0 0
 ↪  0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0; 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1; 0
 ↪  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
```

4.5. Equivariant Coherent Sheaves on Finite Sets

As a more abstract example we want to consider the category of G-equivariant coherent sheaves on a set finite X. We will not cover the theory of sheaves in this thesis and only consider an equivalent category. The results used to implement this example are taken from [Rog21, Section 2.3].

Definition 4.9. Let G be a finite group and X a finite G-set. Let (x_i) be a full set of representatives for the orbits of X and denote by H_i the stabilizer subgroup of x_i. An *equivariant coherent sheaf on* X is given by a family of representations (ρ_{x_i}) for the stabilizer subgroups H_i.

The category $\mathrm{Coh_G(X)}$ of equivariant coherent sheaves on X has equivariant coherent sheaves as objects and families of representation morphisms (f_{x_i}) as morphisms.

Remark 4.10. The definition indeed coincides with the geometric notion of a sheaf if the G-action is trivial. Sheaves on finite sets are described entirely by their stalks, i.e. can be reconstructed from them. It can then be shown that this correspondence is indeed an equivalence of tensor categories (see [Rog21, Section 2.1]). The G-action restricts then to G-equivariant objects in the category of coherent sheaves. More on that ought to be found in [Eti+16, Section 2.7].

The category is implemented with fields for the group, base ring and G-set. Additionally the type `CohSheaves <: Category` stores arrays with the orbit representatives and stabilizers. This is done to fix an order of the representatives. Thus precisely speaking we again implement only a full subcategory.

Coherent sheaves are of type `CohSheaf <: Object` and store a parent and a vector containing the representations of the stabilizers in order according to the parent. Morphisms similarly store a vector of representation morphisms next to domain and codomain.

Thanks to our definition as a product of representation categories the direct sum, tensor product, kernel, and dual are obtained component-wise. The simple objects are given by families such that there is only one non-zero component which an irreducible representation of the corresponding stabilizer.

julia

As an example we construct some coherent sheaves on the G-set $X = \{1, 2, 3, 4\}$ where $G = S_3$ with the natural action.

```
julia> G = symmetric_group(3); X = gset(G,[1,2,3,4]);

julia> Coh = CohSheaves(X,FiniteField(23)[1])
Category of equivariant coherent sheaves on [1, 2, 3, 4] over Galois field with
 ↪ characteristic 23.

julia> S = simples(Coh);

julia> G = S[1]^2 ⊕ S[4]
Equivariant choherent sheaf on [1, 2, 3, 4] over Galois field with characteristic
 ↪ 23.
```

```
julia> F = S[2] ⊕ S[3]^2 ⊕ S[5]
Equivariant choherent sheaf on [1, 2, 3, 4] over Galois field with characteristic
 ↪  23.

julia> G = S[1]^2 ⊕ S[4]
Equivariant choherent sheaf on [1, 2, 3, 4] over Galois field with characteristic
 ↪  23.

julia> F⊗G
Equivariant choherent sheaf on [1, 2, 3, 4] over Galois field with characteristic
 ↪  23.

julia> stalks(F)
2-element Vector{GroupRepresentation{gfp_elem, PermGroup}}:
1-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 2 .. 3 ] ))
4-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 3 ] ))

julia> orbit_stabilizers(Coh)
2-element Vector{PermGroup}:
Sym( [ 2 .. 3 ] )
Sym( [ 1 .. 3 ] )

julia> H = Hom(F⊗G,F)
Vector space of dimension 3 over Galois field with characteristic 23.

julia> B = basis(H);

julia> f = 2*B[1] + B[3]
Morphism of equivariant choherent sheaves on [1, 2, 3, 4] over Galois field with
 ↪  characteristic 23.

julia> g = B[2] - B[3]
Morphism of equivariant choherent sheaves on [1, 2, 3, 4] over Galois field with
 ↪  characteristic 23.

julia> (f⊕g)⊗g
Morphism of equivariant choherent sheaves on [1, 2, 3, 4] over Galois field with
 ↪  characteristic 23.
```

4.5.1. Pullback and Pushforward

Consider two equivariant coherent sheaves $\mathcal{F} \in \mathrm{Coh}_G(X)$ and $\mathcal{G} \in \mathrm{Coh}_G(Y)$ and an equivariant map $f : X \to Y$.

Definition 4.11. The *pullback sheaf* $f^*(\mathcal{G})$ of $\mathcal{G}$ on X is defined by the stalks

$$(f^*(\mathcal{G}))_{x_i} = \mathrm{Res}_{G_{x_i}}^{G_{f(x_i)}}(\mathcal{G}_{f(x_i)})$$

The *pushforward sheaf* $f_*(\mathcal{F})$ of $\mathcal{F}$ on Y is defined by the stalks

$$(f_*(\mathcal{F}))_{y_i} = \bigoplus_{f(x_i)=y_i} \mathrm{Ind}_{G_{x_i}}^{G_{y_i}}(\mathcal{F}_{x_i})$$

Those are indeed well-defined since whenever f is equivariant orbits are mapped into orbits and $G_{x_i} \leq G_{f(x_i)}$.

julia

The implementations of the pushforward and pullback functors are again straightforward utilizing the induction and restriction functor from section 4.4.

```
julia> G = symmetric_group(3); X = gset(G,[1,2,3,4]); Y = gset(G, [1,2,3,4,5]);

julia> CohX = CohSheaves(X,FiniteField(23)[1]); CohY =
 ↪  CohSheaves(Y,FiniteField(23)[1]);

julia> f = identity

julia> PF = Pushforward(CohX,CohY,f)
Pushforward functor from Category of equivariant coherent sheaves on [1, 2, 3, 4]
 ↪  over Galois field with characteristic 23 to Category of equivariant coherent
 ↪  sheaves on [1, 2, 3, 4, 5] over Galois field with characteristic 23

julia> PB = Pullback(CohY,CohX,f)
Pullback functor from Category of equivariant coherent sheaves on [1, 2, 3, 4, 5]
 ↪  over Galois field with characteristic 23 to Category of equivariant coherent
 ↪  sheaves on [1, 2, 3, 4] over Galois field with characteristic 23


julia> F = dsum(simples(CohX)[2:4])
Equivariant choherent sheaf on [1, 2, 3, 4] over Galois field with characteristic
 ↪  23

julia> G = dsum(simples(CohY)[3:6])
Equivariant choherent sheaf on [1, 2, 3, 4, 5] over Galois field with
 ↪  characteristic 23

julia> PF(F)
Equivariant choherent sheaf on [1, 2, 3, 4, 5] over Galois field with
 ↪  characteristic 23

julia> PB(G)
Equivariant choherent sheaf on [1, 2, 3, 4] over Galois field with characteristic
 ↪  23
```

4.6. Convolution Category

Let G be a finite group and X a finite G-set. We can use the category of equivariant coherent sheaves on $X \times X$ to describe a new tensor category by defining another monoidal structure. The construction we introduce is due to [Lus87, Section 2]. If G acts trivially on X the Grothendieck ring of $\mathrm{Coh}_G(X)$ is given by the matrix space $\mathrm{Mat}_{n\times n}$ with pointwise multiplication. Thus $\mathrm{Coh}_G(X \times X)$ can be seen as a structure inducing the pointwise matrix multiplication. Now the idea is to define a monoidal product on $\mathrm{Coh}_G(X \times X)$ which induces the usual matrix multiplication in the Grothendieck ring.

4.6.1. The Convolution Product

Let X be a finite G-set. The products $X \times X$ and $X \times X \times X$ are naturally G-sets by componentwise action. Consider the projection maps from $X \times X \times X$ onto $X \times X$

$$
\begin{aligned}
p_{12} &: X \times X \times X \to X \times X, \quad (x_1, x_2, x_3) \mapsto (x_1, x_2) \\
p_{13} &: X \times X \times X \to X \times X, \quad (x_1, x_2, x_3) \mapsto (x_1, x_3) \\
p_{23} &: X \times X \times X \to X \times X, \quad (x_1, x_2, x_3) \mapsto (x_2, x_3).
\end{aligned}
$$

These maps are all clearly G-equivariant and hence define pushback and pullforward maps.

Definition 4.12. Let $\mathcal{F}, \mathcal{G} \in \mathrm{Coh}(X \times X)$. Then the *convolution product* is defined by

$$\mathcal{F} \otimes_{conv} \mathcal{G} := (p_{13})_* (p_{12}^*(F) \otimes p_{23}^*(G))$$

where the right hand side tensor product is the usual product in $\mathrm{Coh}(X \times X \times X)$. The product for morphisms is analogous. We refer to [Rog21, Section 2.4] for a more detailed introduction.

Define $\mathrm{Conv}_G(X) := \mathrm{Coh}(X \times X)$ as abelian categories with monoidal product $\otimes_{conv}$.

As abelian categories $\mathrm{Coh}(X \times X)$ can be decomposed into representation categories $\mathrm{Coh}(X_i \times X_j)$ where $X = \bigcup X_i$ is the decomposition into orbits. The convolution product of two simple objects $\mathcal{F} \in \mathrm{Coh}(X_i \times X_j)$ and $\mathcal{G} \in \mathrm{Coh}(X_l \times X_k)$ is only non-zero if and only if $j = l$. If $j = l$ then $\mathcal{F} \otimes \mathcal{G} \in \mathrm{Coh}(X_i \times X_k)$. From here one might see the connection to the matrix product when the action of G is trivial since the simple objects behave just like the basis vectors of the matrix space.

Let $x_i, ..., x_n$ be representatives for the orbits in X. Then the unit object in $\mathrm{Conv}_G(X)$ is given by the object

$$\mathbb{1} = \bigoplus_{i=1}^{n} \mathbb{1}_{\mathrm{Rep}(G_{(x_i,x_i)})}$$

where $G_{(x_i,x_i)}$ is the stabilizer of (x_i, x_i). This should be seen as an analogue to the identity matrix where all diagonal entries are the unit.

The category $\mathrm{Conv}_G(X)$ is rigid. Let $M = (\rho_{ij})_{i,j=1,\ldots,n} \in \mathrm{Conv}_G(X)$ with $\rho_{ij} \in \mathrm{Rep}(G_{(x_i,x_j)})$. Define

$$M^* = (M_i j)_{i,j=1,\ldots,n}, \quad M^*_{ij} = \rho^*_{ji}.$$

Let $V \in \mathrm{Conv}_G(X)$ be simple. Then we have $p12^*(V) \otimes p13^*(V) \in \mathrm{Coh}_G(X \times X \times X)$ has only one non-zero component $\mathrm{Res}^H_H(V) \otimes \mathrm{Res}^H_H(V^*)$ where H is the corresponding stabilizer. From here any (co)evaluation of $\mathrm{Res}_H^H(V)$ can be considered as a (co)evaluation in $\mathrm{Coh}_G(X \times X \times X)$. Finally after applying p_{13}^* this yields (co)evaluation maps in $\mathrm{Conv}_G(X)$ (see [Rog21, Proposition 2.27]).

julia

Objects in $\mathrm{Conv}_G(X)$ are of type `ConvolutionObject <: Object` which is a wrapper type for a coherent sheaf object. Thus all functionality apart from the tensor product falls back to the coherent sheaf underneath.
We consider an example for $G = S_3$ action on $\{1, 2, 3, 4\}$ in the natural way.

```
julia> G = symmetric_group(3); X = gset(G,[1,2,3,4]);

julia> Conv = ConvolutionCategory(X,FiniteField(23)[1])
Convolution category over G-set with 4 elements.

julia> S = simples(Conv);

julia> F = S[2] ⊕ S[7]^2;

julia> G = S[3] ⊕ S[8];

julia> stalks(F)
5-element Vector{GroupRepresentation{gfp_elem, PermGroup}}:
1-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 2 .. 3 ] ))
0-dimensional group representation over Galois field with characteristic 23 of
 ↪  Group(()))
0-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 2 .. 3 ] ))
2-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 2 .. 3 ] ))
0-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 3 ] ))

julia> stalks(G)
5-element Vector{GroupRepresentation{gfp_elem, PermGroup}}:
0-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 2 .. 3 ] ))
1-dimensional group representation over Galois field with characteristic 23 of
 ↪  Group(()))
0-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 2 .. 3 ] ))
0-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 2 .. 3 ] ))
1-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 3 ] ))

julia> stalks(F⊗G)
5-element Vector{GroupRepresentation{gfp_elem, PermGroup}}:
```

```
0-dimensional group representation over Galois field with characteristic 23 of
 ↪  Group([ (2,3) ]))
1-dimensional group representation over Galois field with characteristic 23 of
 ↪  Group(()))
0-dimensional group representation over Galois field with characteristic 23 of
 ↪  Group([ (2,3) ]))
2-dimensional group representation over Galois field with characteristic 23 of
 ↪  Group([ (2,3) ]))
0-dimensional group representation over Galois field with characteristic 23 of
 ↪  Sym( [ 1 .. 3 ] ))
```

For morphisms there is also a wrapper type `ConvolutionMorphism <: Morphism` which falls back to the coherent sheaf functionality for everything abelian. The tensor product is given by the same combination of functors.

```
julia> f = sum(basis(End(F))[[1,2,5]])
Morphism in Convolution category over G-set with 4 elements.

julia> g = sum(basis(End(G))[[1,2]])
Morphism in Convolution category over G-set with 4 elements.

julia> matrices(f)
5-element Vector{gfp_mat}:
[1]
0 by 0 empty matrix
0 by 0 empty matrix
[1 0; 0 1]
0 by 0 empty matrix

julia> matrices(g)
5-element Vector{gfp_mat}:
0 by 0 empty matrix
[1]
0 by 0 empty matrix
0 by 0 empty matrix
[1]

julia> matrices(f⊗g)
5-element Vector{gfp_mat}:
0 by 0 empty matrix
[1]
0 by 0 empty matrix
[1 0; 0 1]
0 by 0 empty matrix
```

4.7. Finite Semisimple Ring Categories

Let $\mathcal{C}$ be a finite semisimple ring category. We saw in Chapter 2 that $\mathcal{C}$ is equivalent to a skeletal ring category $\bar{\mathcal{C}} = \bigoplus \overline{\mathrm{Vec}}$ with the same $6j$-symbols, i.e. the 'same' associator and fusion rule. We want to provide a structure to easily work with such a skeleton. A finite semisimple ring category is basically defined by its finite set of simple objects $X_1, ..., X_n$, the fusion rule and the associator. The fusion rule is given by a 3-dimensional matrix (M_{ijk}) such that

$$X_i \otimes X_j = \bigoplus_k M_{ijk} X_k$$

for all simple objects X_{i1}, X_{i_2}, X_{i_3} and the associator by a 4-dimensional matrix $(A_{i_1 i_2 i_3 k})$ such that

$$A_{i_1 i_2 i_3 k} : \bigoplus_{m=1}^{r} H_{ij}^m \otimes H_{mk}^l \to \bigoplus_{n=1}^{r} H_{in}^l \otimes H_{jk}^n$$

is a matrix (see Section 1.3).

4.7.1. Objects and Morphisms

Let $X_1, ..., X_r$ denote the simple objects in $\overline{\mathcal{C}}$. Then an arbitrary object X in $\overline{\mathcal{C}}$ is specified by the quantities $\dim \mathrm{Hom}(X_i, X)$, i.e. the multiplicities in the direct sum decomposition. Thus we fix the order of the simple objects and store the multiplicities of $X_1, ..., X_r$ in an array `A`, such that `A[i]` $= \dim \mathrm{Hom}(X_i, X)$. Therefore we obtain the structure for objects.

```
struct RingObject <: Object
    parent::RingCategory  # The struct RingCategory will be discussed later
    components::Vector{Int}
end
```

As seen in Chapter 2 the Hom-spaces between objects $X, Y \in \overline{\mathcal{C}}$ are given by sums of matrix spaces

$$\mathrm{Hom}(X, Y) \cong \bigoplus_{i=1}^{r} \mathrm{Mat}_{k_i \times l_i}(k)$$

where $k_i = \dim \mathrm{Hom}(X_i, X)$ and $l_i = \dim \mathrm{Hom}(X_i, Y)$. Thus a morphism is given by an ordered family of matrices.

```
struct RingMorphism <: Morphism
    domain::RingObject
    codomain::RingObject
    m::Vector{<:MatElem}
end
```

The matrix structure provides the proper foundation for k-linearity and direct summation. The tensor product of morphism $f = (f_i)_{i=1,\dots,r}, g = (g_i)_{i=1,\dots,r}$ is reconstructed from the fusion rule as

$$(f \otimes g)_k = \bigoplus_{i,j=1}^{r} \bigoplus_{l=1}^{M_{ijk}} f_i \otimes g_j \tag{4.3}$$

where the tensor product on the right hand side is the Kronecker product of matrices.

4.7.2. Associators

Given the $6j$-symbols $A_{i_1 i_2 i_3 k}$ we can construct the associator for arbitrary objects. It is important to realise that even if the $6j$-symbols are trivial, i.e. the associators on simple objects are the identity, this does not imply that all associator isomorphisms are the identity. Recall that the associators in $\overline{\mathcal{C}}$ are defined by the associators from $\bigoplus \mathrm{Vec} \cong \mathcal{C}$.

$$((U_{i_1}) \otimes (V_{i_2})) \otimes (W_{i_3}) \cong \left(\bigoplus_{i_1,i_2,i_3 \in \mathcal{I}} \bigoplus_{j \in \mathcal{I}} H^j_{i_1 i_2} \otimes H^k_{j i_3} \otimes U_{i_1} \otimes V_{i_2} \otimes W_{i_3} \right)_{k=1,\dots,r}$$

$$(U_{i_1}) \otimes ((V_{i_2}) \otimes (W_{i_3})) \cong \left(\bigoplus_{i_1,i_2,i_3 \in \mathcal{I}} \bigoplus_{l \in \mathcal{I}} H^k_{i_1 l} \otimes H^l_{i_2 i_3} \otimes U_{i_1} \otimes V_{i_2} \otimes W_{i_3} \right)_{k=1,\dots,r}$$

For the explicit computation we must include also the isomorphisms above. Thus we take a look at the unaltered objects.

$$((U_{i_1}) \otimes (V_{i_2})) \otimes (W_{i_3}) = \left(\bigoplus_{j,i_3} H^k_{j i_3} \otimes \left(\left(\bigoplus_{i_1,i_2} H^j_{i_1 i_2} \otimes \left(U_{i_1} \otimes V_{i_2} \right) \right) \otimes W_{i_3} \right) \right)_{k=1,\dots,r} \tag{4.4}$$

$$(U_{i_1}) \otimes ((V_{i_2}) \otimes (W_{i_3})) = \left(\bigoplus_{i_1,l} H^k_{i_1 l} \otimes \left(U_{i_1} \otimes \left(\bigoplus_{i_2,i_3} H^l_{i_2 i_3} \otimes (V_{i_2} \otimes W_{i_3}) \right) \right) \right)_{k=1,\dots,r} \tag{4.5}$$

We can ignore all parenthesising and distributive isomorphisms, because on the level of objects in $\overline{\mathcal{C}}$ they equal the identity and matrix multiplication and tensor product are associative and distributive. Thus the two right hand sides (4.4) and (4.5) differ only non-trivially by the ordering of the summands. These are the transformations we must perform before and after stacking the associators.

4.7.3. Kernels and Cokernels

Kernels and cokernels transport via the equivalence $\overline{\mathcal{C}} \cong \bigoplus \mathrm{Vec}$. That is if $f = (f_i)$ is a morphism then the kernel of f is the collection $\ker f = (\overline{\ker f_i})$ where f_i is interpreted as a vector space morphism. Similarly $\mathrm{coker}\, f = (\overline{\mathrm{coker}\, f_i})$.

For our implementation this means the kernel object is given by the coefficient vector $(\dim \ker f_i)$ and the inclusion morphism is simply (ϕ_i) when ϕ_i is the inclusion of $\ker f_i$. The cokernel is obtained similarly.

4.7.4. An Example

To emphasise the use we will take a look at an example.

Definition 4.13. The *Ising category* is a fusion category with three objects $\mathbb{1}, \chi$ and X with fusion rule $\chi \otimes \chi = \mathbb{1}$, $\chi \otimes X = X \otimes \chi = X$ and $X \otimes X = 1 \oplus \chi$. Associativity is given by the following isomorphisms:

$$
\begin{aligned}
a_{\chi,X,\chi} &= (-1)\mathrm{id}_X \\
a_{X,\mathbb{1},X} &= \mathrm{id}_{\mathbb{1}} \oplus (-1)\mathrm{id}_\chi \\
a_{X,\chi,X} &= (-1)\mathrm{id}_{\mathbb{1}} \oplus \mathrm{id}_\chi \\
a_{X,X,X} &= \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \mathrm{id}_{2X}
\end{aligned}
$$

and all not listed ones are the identity.

Remark 4.14. The example of the Ising category is motivated in physics stemming from the Ising model. We dont want to discuss this here. The more general definition of an Ising category is the following: An Ising category ia a not pointed[1] fusion category with Frobenius-Perron dimension 4. How this is related to the three element category and why it is actually a fusion category is elaborated in [Dri+10, Appendix B].

In the $6j$-symbol notion the associators would look like the following.

$$
\begin{array}{lll}
\Phi^{\mathbb{1}}_{\chi,X,\chi} = 0, & \Phi^{\chi}_{\chi,X,\chi} = 0, & \Phi^{X}_{\chi,X,\chi} = 1, \\
\Phi^{\mathbb{1}}_{X,\mathbb{1},X} = 1, & \Phi^{\chi}_{X,\mathbb{1},X} = -1, & \Phi^{X}_{X,\mathbb{1},X} = 0, \\
\Phi^{\mathbb{1}}_{X,\chi,X} = -1, & \Phi^{\chi}_{X,\chi,X} = 1, & \Phi^{X}_{X,\chi,X} = 0, \\
\Phi^{\mathbb{1}}_{X,X,X} = 0, & \Phi^{\chi}_{X,X,X} = 0, & \Phi^{X}_{X,X,X} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}
\end{array}
$$

julia

We construct the Ising category in TensorCategories.jl. To do so we need to specify the multiplication table and the associativity constraints.

```julia
# We need the square root of 2 explicitely
Qx,x = QQ["x"]
F,a = NumberField(x^2-2, "√2")

I = RingCategory(F,["1", "χ", "X"])

# Define the mutliplication table
M = zeros(Int,3,3,3)

M[1,1,:] = [1,0,0]
M[1,2,:] = [0,1,0]
M[1,3,:] = [0,0,1]
M[2,1,:] = [0,1,0]
M[2,2,:] = [1,0,0]
M[2,3,:] = [0,0,1]
M[3,1,:] = [0,0,1]
M[3,2,:] = [0,0,1]
```

[1] A fusion category is called pointed if all simple objects are invertible

```
M[3,3,:] = [1,1,0]

set_tensor_product!(I,A)

# Define the associativity
set_associator!(C,2,3,2, matrices(-id(I[3])))
set_associator!(C,3,1,3, matrices(id(I[1])⊕(-id(I[2]))))
set_associator!(C,3,2,3, matrices((-id(I[1]))⊕id(I[2])))
z = zero(MatrixSpace(F,0,0))
set_associator!(C,3,3,3, [z, z, inv(a)*matrix(F,[1 1; 1 -1])])
```

All not explicitly defined associators are initialised as the identity morphisms. The operation `I[i]` returns the i-th simple object and `matrices(f::Morphism)` returns the matrices defining f in order.

```
julia> a,b,c = simples(I)
3-element Vector{RingObject}:
1
χ
X

julia> (a ⊕ b^2) ⊗ (c^2 ⊕ a)
1 ⊕ 2·χ ⊕ 6·X

julia> H = Hom(a^2 ⊕ b, a ⊕ b^2); B = basis(H);

julia> f = B[1] - 2*B[3] + B[4]
Morphism with
Domain: 2·1 ⊕ χ
Codomain: 1 ⊕ 2·χ
Matrices: [1; 0], [-2 1], 0 by 0 empty matrix

julia> g = -B[1] + B[2] + 2*B[4]
Morphism with
Domain: 2·1 ⊕ χ
Codomain: 1 ⊕ 2·χ
Matrices: [-1; 1], [0 2], 0 by 0 empty matrix

julia> f⊗g
Morphism with
Domain: 5·1 ⊕ 4·χ
Codomain: 5·1 ⊕ 4·χ
Matrices: [-1 0 0 0 0; 1 0 0 0 0; 0 0 0 0 0; 0 0 0 0 0; 0 0 -4 0 2], [0 2 0 0; 0
 ↪  0 0 0; 0 0 2 -1; 0 0 -2 1], 0 by 0 empty matrix
```

4.7.5. Duals

Let $X \in \mathcal{C}$ be simple. By Lemma 1.17 and Lemma 1.18 we know that if X admits a dual then it is simple and there is precisely one simple $Y \in \mathcal{C}$ such that $\dim \operatorname{Hom}(\mathbb{1}, X \otimes Y) \neq 0$.

If there exists a unique Y with that property It is actually already a dual to X. Since $\mathrm{Hom}(\mathbb{1}, X \otimes Y) \cong k \cong \mathrm{Hom}(Y \otimes X, \mathbb{1})$ the coevaluation and evaluation maps are skalar multiples of the identity on $\mathbb{1}$. Thus we can choose either the evaluation or the coevaluation to equal the identity on $\mathbb{1}$, i.e. $\mathrm{coev}_X = (\delta_{i1}[1])_i$. Then just compute the concatenation

$$f : X \xrightarrow{(\delta_{i1}[1])_i \otimes \mathrm{id}_X} (X \otimes Y) \otimes X \xrightarrow{a_{X,Y,X}} X \otimes (Y \otimes X) \xrightarrow{\mathrm{id}_X \otimes (\delta_{i1}[1])_i} X$$

which is an element of $\mathrm{Hom}(X, X) \cong k$. Therefore f is a multiple of the identity and we can define the evaluation by scaling. We obtain $\mathrm{ev}_X = f^{-1}(\delta_{i1}[1])_i$.

5. The Centre of a Fusion Category

In this chapter we want to describe an algorithmic approach on how to compute the centre of a fusion category.

5.1. The Centre Construction

We follow the construction from [Müg03, Chapter 3] and [Eti+16, Section 7.13].

Definition 5.1. Let $\mathcal{C}$ be a monoidal category. A *half-braiding* for $X \in \mathcal{C}$ is a natural transformation $\{e_X(Y) : X \otimes Y \to Y \otimes X\}$ such that

(i)
$$\begin{tikzcd} X \otimes Y \arrow[r, "\mathrm{id}_X \otimes t"] \arrow[d, "e_X(Y)"'] & X \otimes Z \arrow[d, "e_X(Z)"] \\ Y \otimes X \arrow[r, "t \otimes \mathrm{id}_X"] & Z \otimes X \end{tikzcd}$$

commutes for all $t \in \mathrm{Hom}(Y, Z)$.

(ii)
$$\begin{tikzcd} & X \otimes (Y \otimes Z) \arrow[r, "e_X(Y \otimes Z)"] & (Y \otimes Z) \otimes X \arrow[dr, "a_{Y,Z,X}"] & \\ (X \otimes Y) \otimes Z \arrow[ur, "a_{X,Y,Z}"] \arrow[dr, "e_X(Y) \otimes \mathrm{id}_Z"'] & & & Y \otimes (Z \otimes X) \\ & (Y \otimes X) \otimes Z \arrow[r, "a_{Y,X,Z}"] & Y \otimes (X \otimes Z) \arrow[ur, "\mathrm{id}_Z \otimes e_X(Z)"'] & \end{tikzcd}$$

commutes for all $X, Y, Z \in \mathcal{C}$.

(iii) All $e_X(Y)$ are isomorphisms

(iv) $e_X(\mathbb{1}) = \mathrm{id}_X$.

The following result is important for the algorithm.

Lemma 5.2 ([Müg03, Lemma 3.2])**.** *If e_X satisfies (i) and (ii) then (iii) $\Rightarrow$ (iv) and if Y has a dual then $e_X(Y)$ is an isomorphism.*

This lemma will allow us to omit the checking for isomorphisms when considering a fusion category (which is rigid).

Definition 5.3. Let $\mathcal{C}$ be a monoidal category. Define $\mathcal{Z}(\mathcal{C})$ as the category with objects (X, γ_X) where X is an object in $\mathcal{C}$ and γ_X is a half-braiding. A morphism $f : (X, \gamma_X) \to (Y, \gamma_Y)$ is given by a morphism $f : X \to Y$ such that

$$\begin{tikzcd} X \otimes Z \arrow[r, "f\otimes \mathrm{id}_Z"] \arrow[d, "\gamma_X(Z)"'] & Y \otimes Z \arrow[d, "\gamma_Y(Z)"] \\ Z \otimes X \arrow[r, "\mathrm{id}_Z \otimes f"] & Z \otimes Y \end{tikzcd} \tag{5.1}$$

Let $\mathcal{C}$ be a multiring category. Let $(X, \gamma_X), (Y, \gamma_Y) \in \mathcal{Z}(\mathcal{C})$, then the direct sum of two objects $(X, \gamma_X), (Y, \gamma_Y) \in \mathcal{Z}(\mathcal{C})$ is given by

$$(X, \gamma_X) \oplus (Y, \gamma_Y) = (X \oplus Y, \gamma_X \oplus \gamma_Y).$$

We can also define a tensor product by

$$(X, \gamma_X) \otimes (Y, \gamma_Y) = (X \otimes Y, \gamma_{X \otimes Y})$$

where $\gamma_{X\otimes Y}(Z)$ is defined by the following diagram.

$$\begin{tikzcd} (X \otimes Y) \otimes Z \arrow[r, "a_{X,Y,Z}"] \arrow[d, "\gamma_{X\otimes Y}(Z)"] & X \otimes (Y \otimes Z) \arrow[r, "\mathrm{id}_X \otimes \gamma_Y(Z)"] & X \otimes (Z \otimes Y) \arrow[d, "a^{-1}_{X,Z,Y}"] \\ Z \otimes (X \otimes Y) & (Z \otimes X) \otimes Y \arrow[l, "a_{Z,X,Y}"] & (X \otimes Z) \otimes Y \arrow[l, "\gamma_X(Z)\otimes \mathrm{id}_Y"] \end{tikzcd}$$

Clearly direct sums and tensor products of morphisms are again satisfying (5.1). Moreover $\mathcal{Z}(\mathcal{C})$ is naturally braided. A braiding is given by

$$c_{(X,\gamma_X),(Y,\gamma_Y)} = \gamma_X(Y).$$

Kernel and cokernel of a morphism $f : (X, \gamma_X) \to (Y, \gamma_Y)$ are given by $((K, \gamma_K), \phi)$ and $((C, \gamma_C), \psi)$ where (K, ϕ) and (C, ψ) are kernel respectively cokernel of f considered as morphism $f : X \to Y$. The braidings γ_K and γ_Y have to make the diagram

$$\begin{tikzcd} K \otimes Z \arrow[r, "\phi\otimes \mathrm{id}_Z"] \arrow[d, "\gamma_k(Z)"] & X \otimes Z \arrow[r, "f"] \arrow[d, "\gamma_X(Z)"] & Y \otimes Z \arrow[r, "\psi\otimes \mathrm{id}_Z"] \arrow[d, "\gamma_Y(Z)"] & C \otimes Z \arrow[d, "\gamma_C(Z)"] \\ Z \otimes K \arrow[r, "\mathrm{id}_Z\otimes\phi"] & Z \otimes X \arrow[r, "f"] & Z \otimes Y \arrow[r, "\mathrm{id}_Z\otimes\psi"] & Z \otimes C \end{tikzcd}$$

commute for all $Z \in \mathcal{C}$. Let ϕ' be a left inverse to ϕ and ψ' a right inverse to ψ. Then clearly

$$\gamma_K(Z) = (\mathrm{id}_Z \otimes \phi') \circ \gamma_X(Z) \circ (\phi \otimes \mathrm{id}_Z)$$
$$\gamma_C(Z) = (\mathrm{id}_Z \otimes \psi) \circ \gamma_Y(Z) \circ (\psi' \otimes \mathrm{id}_Z)$$

define appropriate half braidings on K and C.

Let $X \in \mathcal{C}$ with dual X^* and $\mathrm{ev}_X, \mathrm{coev}_X$. Then if $(X, \gamma_X) \in \mathcal{Z}(\mathcal{C})$ there is a dual object

$$(X, \gamma_X)^* = (X^*, \gamma_{X^*})$$

with $\gamma_{X^*}(Z)$ being defined by the commutative diagram

$$
\begin{tikzcd}
X^* \otimes Z \arrow[r, "\mathrm{id}_{X^*\otimes Z}\otimes \mathrm{coev}_X"] \arrow[dddd, "\gamma_{X^*}"] & (X^* \otimes Z) \otimes (X \otimes X^*) \arrow[r, "a_{X^*,Z,X\otimes X^*}"] & X^* \otimes (Z \otimes (X \otimes X^*)) \arrow[d, "\mathrm{id}_{X^*}\otimes a^{-1}_{Z,X,X^*}"] \\
 & & X^* \otimes ((Z \otimes X) \otimes X^*) \arrow[d, "\mathrm{id}_{X^*}\otimes\gamma_X(Z)^{-1}\otimes\mathrm{id}_{X^*}"] \\
 & & X^* \otimes ((X \otimes Z) \otimes X^*) \arrow[d, "\mathrm{id}_{X^*}\otimes a_{X,Z,X^*}"] \\
 & & X^* \otimes (X \otimes (Z \otimes X^*)) \arrow[dll, phantom] \\
Z \otimes X^* & (X^* \otimes X) \otimes (Z \otimes X^*) \arrow[l, "\mathrm{ev}_X\otimes\mathrm{id}_{Z\otimes X^*}"'] & X^* \otimes (X \otimes (Z \otimes X^*)) \arrow[l, "a^{-1}_{X^*,X,Z\otimes X^*}"']
\end{tikzcd}
$$

and $\mathrm{ev}_{(X,\gamma_X)} = \mathrm{ev}_X$, $\mathrm{coev}_{(X,\gamma_X)} = \mathrm{coev}_X$.

Theorem 5.4 ([Müg03, Theorem 1.2]). *The centre of a fusion category is a fusion category.*

This statement is important, because it allows us to build $\mathcal{Z}(\mathcal{C})$ from simple objects.

5.2. Computing Hom-Spaces

Not all morphisms between two objects X, Y that admit half-braidings satisfy property (5.1). Clearly the space of morphisms $\mathrm{Hom}_{\mathcal{Z}(\mathcal{C})}((X,\gamma_X),(Y,\gamma_Y))$ between central objects can be interpreted as a subspace of $\mathrm{Hom}_{\mathcal{C}}(X,Y)$.

Lemma 5.5. *Let $\mathcal{C}$ be a fusion category with simple objects $X_1, ..., X_n$ and spherical structure ψ such that $\dim \mathcal{C} \neq 0$. Let $(X,\gamma_X),(Y,\gamma_Y) \in \mathcal{Z}(\mathcal{C})$. The map $E_{X,Y} : \mathrm{Hom}_{\mathcal{C}}(X,Y) \to \mathrm{Hom}_{\mathcal{C}}(X,Y)$ given by*

$$E_{X,Y}(t) = \frac{1}{\dim \mathcal{C}} \sum_{i=1}^{n} \dim X_i \phi_i(t)$$

where $\phi_i(t)$ is given by

$$
\begin{tikzcd}
X \arrow[r, "\mathrm{id}_X\otimes\mathrm{coev}(X_i)"] \arrow[dd, "\phi_i(t)"] & X \otimes (X_i \otimes X_i^*) \arrow[r, "a^{-1}_{X,X_i,X_i^*}"] & (X \otimes X_i) \otimes X_i^* \arrow[r, "\gamma_X(X_i)\otimes\mathrm{id}_{X_i^*}"] & (X_i \otimes X) \otimes X_i^* \arrow[d, "\mathrm{id}_{X_i}\otimes t\otimes\mathrm{id}_{X_i^*}"] \\
 & & & (X_i \otimes Y) \otimes X_i^* \arrow[d, "a_{X_i,Y,X_i^*}"] \\
Y & (X_i^{**} \otimes X_i^*) \otimes Y \arrow[l, "\mathrm{ev}_{X_i^*}\otimes\mathrm{id}_Y"'] & X_i^{**} \otimes (X_i^* \otimes Y) \arrow[l, "a^{-1}_{X_i^{**},X_i^*,Y}"'] & X_i \otimes (Y \otimes X_i^*) \arrow[l, "\psi_{X_i}\otimes\gamma_Y(X_i^*)"']
\end{tikzcd}
$$

is a projection from $\mathrm{Hom}_{\mathcal{C}}(X,Y)$ onto $\mathrm{Hom}_{\mathcal{Z}(\mathcal{C})}((X,\gamma_X),(Y,\gamma_Y))$.

A proof of this lemma is to find in [Müg03, Lemma 3.10] for the strict case. Thus we get Hom-spaces between objects in the centre by applying the projection on the basis of $\mathrm{Hom}(X,Y)$ and choosing a generating set for the image.

Remark 5.6. This result is of particular interest since it allows to determine whether objects in the centre are simple.

5.3. Finding Half-Braidings

For arbitrary monoidal categories the half-braiding condition is not feasibly checkable. In the case of fusion categories we are able to restrict to simple objects.

Lemma 5.7 ([Müg03, Lemma 3.3]). *Let $\mathcal{C}$ be a fusion category with simple objects $\{X_i\}$. Let $Z \in \mathcal{C}$. There is a bijection between half-braidings for Z and families of morphisms $\{\gamma_Z(X_i) \in \mathrm{Hom}(Z \otimes X_i, X_i \otimes Z)\}$ such that for all i, j, k and $t \in \mathrm{Hom}(X_k, X_i \otimes X_j)$ the diagram*

$$\begin{tikzcd}
Z \otimes X_k \arrow[r, "\gamma_Z(X_k)"] \arrow[d, "\mathrm{id}_Z \otimes t"'] & X_k \otimes Z \arrow[r, "t \otimes \mathrm{id}_Z"] & (X_i \otimes X_j) \otimes Z \arrow[r, "a_{X_i,X_j,Z}"] & X_i \otimes (X_j \otimes Z) \\
Z \otimes (X_i \otimes X_j) \arrow[r, "a^{-1}_{Z,X_i,X_j}"] & (Z \otimes X_i) \otimes X_j \arrow[r, "\gamma_Z(X_i) \otimes \mathrm{id}_{X_j}"] & (X_i \otimes Z) \otimes X_j \arrow[r, "a_{X_i,Z,X_j}"] & X_i \otimes (Z \otimes X_j) \arrow[u, "\mathrm{id}_{X_i} \otimes \gamma_Z(X_j)"']
\end{tikzcd}$$

commutes and $\gamma_Z(\mathbb{1}) = \mathrm{id}_Z$.

The lemma provides us an equality between two morphisms in $\mathrm{Hom}(Z \otimes X_k, X_i \otimes (X_j \otimes Z))$ for each choice of i, j, k and t. Fixing a combination of i, j and k it becomes clear that whenever a family $\{\gamma_Z(X_i)\}$ satisfies the condition for t_1 and t_2 then it also satisfies it for $t_1 + t_2$. Therefore we can restrict even further and only consider a basis of $\mathrm{Hom}(X_k, X_i \otimes X_j)$.

Each morphism $e_Z(X_i)$ can be expressed in the corresponding basis. To get equations which are algebraically handleable we want to compute both sides of the equation and express the resulting morphisms in a common basis. After that we can compare coefficients. So consider the maps

$$\phi : \mathrm{Hom}(Z \otimes X_k, X_k \otimes Z) \to \mathrm{Hom}(Z \otimes X_k, X_i \otimes (X_j \otimes Z))$$

sending a morphism $\gamma_Z(X_k)$ to the top row of the diagram in the lemma and

$$\psi : \mathrm{Hom}(Z \otimes X_j, X_j \otimes Z) \times \mathrm{Hom}(Z \otimes X_i, X_i \otimes Z) \to \mathrm{Hom}(Z \otimes X_k, X_i \otimes (X_j \otimes Z))$$

sending a tuple of morphisms $(\gamma_Z(X_i), \gamma_Z(X_j))$ to the other composition of morphism in the diagram. We immediately get linearity of ϕ and bilinearity of ψ.

Let $f_{i1}, ..., f_{ir_i}$ be a basis of $\mathrm{Hom}(Z \otimes X_i, X_i \otimes Z)$. Write

$$\gamma_Z(X_i) = a_1 f_{i1} + \dots + a_{r_i} f_{ir_i},\ \gamma_Z(X_j) = b_1 f_{j1} + \dots + b_{r_j} f_{jr_j},\ \gamma_Z(X_k) = c_1 f_{k1} + \dots + c_{r_k} f_{kr_k}$$

and plug them in the equation to obtain

$$\sum_{x=1}^{r_k} c_k \phi(f_{kx}) = \sum_{y=1}^{r_i} \sum_{z=1}^{r_j} a_y b_z \psi(f_{jz}, f_{iy}). \tag{5.2}$$

Next we can compute the values for ϕ and ψ and collect the equations from comparing the coefficients on both sides. The resulting set of quadratic equations has non-empty vanishing set whenever there exists a half-braiding. Thus the task is to find these solutions.

There are multiple issues with this task: If the isomorphism class of a central object contains infinitely many objects with non-equal half-braidings the ideal generated by the equations is of positive dimension. There are no generic solvers for such problems at the moment. Therefore by now we are restricted to the case where the solution set is already finite. We may guess parts of solutions to find more half-braidings. A second issues comes with the field k. Whenever our equations contain non-rational coefficients the solving of the ideal becomes much harder, since we need algebraically precise solutions and approximations are not desirable. Thus at the moment we are only able to solve for coefficients in $\mathbb{Q}$. We use the project msolve[1] available via Oscar.jl which uses real isolation to find solutions in the case where the solution set is finite. Important is that msolve also computes a rational parametrisation whence we can reconstruct all solutions symbolically.

5.4. The Algorithm

From the last section we have (in theory) a way to find all possible half-braidings for an object. The main idea is now to check potential simple objects for centrality. That requires a boundary on the objects which is given by the following result

Lemma 5.8. *Let $\mathcal{C}$ be a fusion category. Then*

$$\dim \mathcal{Z}(\mathcal{C}) = (\dim \mathcal{C})^2$$

This can be shown by establishing that $\mathcal{Z}$ is weakly Morita equivalent to $C \boxtimes C^{op}$ like in [Eti+16, Section 7.16] or [Müg03, Chapter 4]. In conclusion there are only finitely many non-isomorphic objects in $\mathcal{C}$ which may have a simple object in $\mathcal{Z}(\mathcal{C})$ lying over them.

We get three main methods: `iscentral(Z::Object)` to find checking whether half-braidings exist, `half_braidings(Z::Object)` to compute the half braidings and `simples(C::CenterCategory)` to collect simple objects in the centre.

iscentral

Input: $X \in \mathcal{C}$
Output: `true` if a half braiding for X exists, else `false`.

```
S = simples(C)
equations = []
for k,i,j in 1:n
    for t in basis(Hom(S[k], S[i]⊗S[j]))
        E = equations from (5.2)
        push!(equations, E)
    end
end
push!(equations, [equations for γ_X(1) = id_X])

return dim(ideal(equations)) >= 0
```

Next the method to compute the half braidings. This boils down to finding solutions for the generated ideal. In general those ideals may be of positive dimension.

[1] `https://msolve.lip6.fr/index.html`

half_braidings

Input: $X \in \mathcal{C}$ such that half-braidings exist.
Output: Objects (X, γ) in $\mathcal{Z}(\mathcal{C})$

```
I = Ideal build in iscentral(X)
coefficients = finitely many zeros of I
S = simples(parent(X))
centrals = [CentralObject(Z, c.* [basis of all Hom(S[k], S[i]⊗S[j])]) for c in
 ↪  coefficients]
return non-isomorphic objects from centrals
```

We can ensure that there is only one object for each isomorphism class by testing whether $\mathrm{Hom}(s, t) = 0$ for all $s \neq t$ in the list. Also all solutions can only be found deterministically when $\dim(I) = 0$ otherwise we will try to guess some solutions and work from there.

We combine the above to compute (not necessarily all) simple objects in the centre. Let $X = \bigoplus k_i X_i$ be the direct sum decomposition into the simple objects $X_1, ..., X_n$ in $\mathcal{C}$. We refer to $(k_1, ..., k_n)$ as the coefficient vector. The key is to iterate over objects in $\mathcal{C}$ by iterating the possible combinations of coefficient vectors. This is performed in such a way, that only those coefficient vectors are checked for which all smaller ones already terminated negatively. Smaller in this case means all entries in the coefficient vector are smaller or equal. By doing this we ensure that every times we find a central object it is already simple, since if $(X, \gamma) \cong (X_1, \gamma_1) \oplus (X_2, \gamma_2)$ is not simple, then clearly $X \cong X_1 \oplus X_2$ is not simple. In this way we can assure that the found objects are simple. There might be simple objects $(X \oplus Y, \gamma)$ for which also X and Y admit half-braidings. Therefore we lose some simple objects along the way. This is unfortunate but necessary to avoid enormous computation times. Otherwise we will compute also direct sums of known braidings and still have no guaranty to find something new. There is some potential to obtain further simple objects by computing subobjects of products of the known simple objects as shown in the next section.

simples

Input: A Fusion category $\mathcal{C}$
Output: A vector with some simple objects of $\mathcal{Z}(\mathcal{C})$

```
d = dim(C)^2
S = simples(C)
coefficients = order [1,...,dim(C)]^length(S) by <

central = []

for c in coefficients
    if c a combination of already positively checked objects continue end
    is_central, objects = iscentral(dsum(c .* S))
    if is_central
        push!(central, objects)
     end
end

return central
```

5.5. The Centre in TensorCategories.jl

The centre category of a fusion category $\mathcal{C}$ is implemented with objects of type `CenterObject{T} <: Object` which wrap an object of $\mathcal{C}$ together with the parent category and a vector containing the half-braiding γ on simple objects, i.e. the $e_Z(X_i)$. This is actually enough since we only deal with semisimple categories and thus the half-braidings extend. Morphisms also have a wrapper type `CenterMorphism{T} <: Morphism` with the mandatory fields for domain and codomain next to a morphism from $\mathcal{C}$ which satisfies the central condition (5.1). These morphisms can be obtained by using the projection Lemma 5.5 to build the Hom-spaces.

julia

We want to examine the category Vec_G where $G = S_3$.

```
julia> G = symmetric_group(3);

julia> VecG = GradedVectorSpaces(F,G);

julia> C = Center(VecG);
```

Note, that in theory we usually work over the complex numbers. Since that is not feasible in this setting we use the smallest field that allows us to obtain all half-braidings. If the the field is not big enough a warning with the necessary extension will be displayed.

```
julia> S = simples(C)
┌ Warning: Not all halfbraidings found
└ @ TensorCategories
 ↪  ~/.julia/dev/TensorCategories/src/structures/Center/Center.jl:258
7-element Vector{CenterObject}:
Central object: Graded vector space of dimension 1 with grading
PermGroupElem[()]
Central object: Graded vector space of dimension 1 with grading
PermGroupElem[()]
Central object: Graded vector space of dimension 2 with grading
PermGroupElem[(1,3,2), (1,2,3)]
Central object: Graded vector space of dimension 2 with grading
PermGroupElem[(1,3,2), (1,2,3)]
Central object: Graded vector space of dimension 2 with grading
PermGroupElem[(1,3,2), (1,2,3)]
Central object: Graded vector space of dimension 3 with grading
PermGroupElem[(2,3), (1,3), (1,2)]
Central object: Graded vector space of dimension 3 with grading
PermGroupElem[(2,3), (1,3), (1,2)]
```

We see a warning that not all simple objects could be found. We can compute the sum of squared dimensions to see How many are missing.

```
julia> sum([dim(s)^2 for s ∈ S])
32
```

There is either one object of dimension two or four objects of dimension one left. We can easily see over which object the missing one lies by computing some products and decomposing them.

```
julia> X = S[3]⊗S[4]
Central object: Graded vector space of dimension 4 with grading
PermGroupElem[(1,2,3), (), (), (1,3,2)]

julia> [dim(Hom(X,s)) for s ∈ S]
7-element Vector{Int64}:
0
0
0
0
1
0
0
```

We find the missing object has to lie over $k^2_{()}$. In this case we can reconstruct the missing simple object by considering the image of the projection morphism.

```
julia> matrix.(basis(End(X)))
2-element Vector{AbstractAlgebra.Generic.MatSpaceElem{nf_elem}}:
[1 0 0 0; 0 0 0 0; 0 0 0 0; 0 0 0 1]
[0 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 0]

julia> I,i = image(basis(End(X))[2])
(Central object: Graded vector space of dimension 2 with grading
PermGroupElem[(), ()], Morphism in Drinfeld center of Category of G-graded vector
 ↪  spaces over Cyclotomic field of order 3 where G is Sym( [ 1 .. 3 ] ))

julia> [dim(Hom(I, s)) for s ∈ S]
7-element Vector{Int64}:
0
0
0
0
0
0
0

julia> End(I)
Vector space of dimension 1 over Cyclotomic field of order 3.
```

From the last two queries we conclude that I is indeed the missing simple object. We finish with pushing I to the simple objects of $\mathcal{Z}(\mathcal{C})$.

```
julia> add_simple!(C,I);

julia> simples(C)
8-element Vector{CenterObject}:
Central object: Graded vector space of dimension 1 with grading
PermGroupElem[()]
Central object: Graded vector space of dimension 1 with grading
PermGroupElem[()]
```

```
Central object: Graded vector space of dimension 2 with grading
PermGroupElem[(1,3,2), (1,2,3)]
Central object: Graded vector space of dimension 2 with grading
PermGroupElem[(1,3,2), (1,2,3)]
Central object: Graded vector space of dimension 2 with grading
PermGroupElem[(1,3,2), (1,2,3)]
Central object: Graded vector space of dimension 3 with grading
PermGroupElem[(2,3), (1,3), (1,2)]
Central object: Graded vector space of dimension 3 with grading
PermGroupElem[(2,3), (1,3), (1,2)]
Central object: Graded vector space of dimension 2 with grading
PermGroupElem[(), ()]
```

Now that we have a full set of simple objects we might compute the S-matrix.

```
julia> smatrix(C1)
[ 1   1    2    2    2   -3   -3    2]
[ 1   1    2    2    2    3    3    2]
[ 2   2    4   -2   -2    0    0   -2]
[ 2   2   -2   -2    4    0    0   -2]
[ 2   2   -2    4   -2    0    0   -2]
[-3   3    0    0    0    3   -3    0]
[-3   3    0    0    0   -3    3    0]
[ 2   2   -2   -2   -2    0    0    4]

julia> rank(ans)
8
```

we conclude that $\mathcal{Z}(\mathrm{Vec}_G)$ is indeed modular.

Finally we want to collect and visualize the information gathered above. The category $\mathcal{Z}(\mathcal{C})$ where $\mathcal{C}$ is the category of S_3-graded vector spaces has eight simple objects. Two of them lie over k_{id}, one over k_{id}^2, three over $V = k_{(123)} \oplus k_{(132)}$ and two over $W = k_{(12)} \oplus k_{(13)} \oplus k_{(23)}$. We denote them by the corresponding tuples and obtain the following symmetric multiplication table.

$\otimes$	$(k_{\mathrm{id}}, \mathrm{id})$	$(k_{\mathrm{id}}, \gamma)$	$(k_{\mathrm{id}}^2, \tau)$	(V, σ_1)	(V, σ_2)	(V, σ_3)	(W, η_1)	(W, η_2)
$(k_{\mathrm{id}}, \mathrm{id})$	$(k_{\mathrm{id}}, \mathrm{id})$	$(k_{\mathrm{id}}, \gamma)$	$(k_{\mathrm{id}}^2, \tau)$	(V, σ_1)	(V, σ_2)	(V, σ_3)	(W, η_1)	(W, η_2)
$(k_{\mathrm{id}}, \gamma)$		$(k_{\mathrm{id}}, \mathrm{id})$	$(k_{\mathrm{id}}^2, \tau)$	(V, σ_1)	(V, σ_2)	(V, σ_3)	(W, η_2)	(W, η_1)
$(k_{\mathrm{id}}^2, \tau)$			$(k_{\mathrm{id}}, \mathrm{id})$ $\oplus(k_{\mathrm{id}}, \gamma)$ $\oplus(k_{\mathrm{id}}^2, \tau)$	(V, σ_2) $\oplus(V, \sigma_3)$	(V, σ_1) $\oplus(V, \sigma_3)$	(V, σ_1) $\oplus(V, \sigma_2)$	(W, η_1) $\oplus(W, \eta_2)$	(W, η_1) $\oplus(W, \eta_2)$
(V, σ_1)				$(k_{\mathrm{id}}, \mathrm{id})$ $\oplus(k_{\mathrm{id}}, \gamma)$ $\oplus(V, \sigma_1)$	$(k_{\mathrm{id}}^2, \tau)$ $\oplus(V, \sigma_3)$	$(k_{\mathrm{id}}^2, \tau)$ $\oplus(V, \sigma_2)$	(W, η_1) $\oplus(W, \eta_2)$	(W, η_1) $\oplus(W, \eta_2)$
(V, σ_2)					$(k_{\mathrm{id}}, \mathrm{id})$ $\oplus(k_{\mathrm{id}}, \gamma)$ $\oplus(V, \sigma_2)$	$(k_{\mathrm{id}}^2, \tau)$ $\oplus(V, \sigma_1)$	(W, η_1) $\oplus(W, \eta_2)$	(W, η_1) $\oplus(W, \eta_2)$
(V, σ_3)						$(k_{\mathrm{id}}, \mathrm{id})$ $\oplus(k_{\mathrm{id}}, \gamma)$ $\oplus(V, \sigma_3)$	(W, η_1) $\oplus(W, \eta_2)$	(W, η_1) $\oplus(W, \eta_2)$
(W, η_1)							$(k_{\mathrm{id}}, \mathrm{id})$ $\oplus(k_{\mathrm{id}}^2, \tau)$ $\oplus(V, \sigma_1)$ $\oplus(V, \sigma_2)$ $\oplus(V, \sigma_3)$	$(k_{\mathrm{id}}, \gamma)$ $\oplus(k_{\mathrm{id}}^2, \tau)$ $\oplus(V, \sigma_1)$ $\oplus(V, \sigma_2)$ $\oplus(V, \sigma_3)$
(W, η_2)								$(k_{\mathrm{id}}, \mathrm{id})$ $\oplus(k_{\mathrm{id}}^2, \tau)$ $\oplus(V, \sigma_1)$ $\oplus(V, \sigma_2)$ $\oplus(V, \sigma_3)$

Bibliography

[Dri+10] Vladimir Drinfeld et al. "On braided fusion categories I". In: *Selecta Mathematica* 16.1 (2010), pp. 1–119.

[Eti+16] Pavel Etingof et al. *Tensor categories*. Vol. 205. American Mathematical Soc., 2016.

[Fei68] Burton Fein. "Lifting modular representations of finite groups". In: *Proceedings of the American Mathematical Society* 19.1 (1968), pp. 217–221.

[HLY14] Hua-Lin Huang, Gongxiang Liu and Yu Ye. "The braided monoidal structures on a class of linear Gr-categories". In: *Algebras and Representation Theory* 17.4 (2014), pp. 1249–1265.

[Lus87] George Lusztig. "Leading coefficients of character values of Hecke algebras". In: *Proc. Symp. Pure Math.* Vol. 47. 2. 1987, pp. 235–262.

[Mac71] Saunders Mac Lane. *Categories for the working mathematician*. Springer Science & Business Media, 1971.

[Müg03] Michael Müger. "From subfactors to categories and topology II: The quantum double of tensor categories and subfactors". In: *Journal of Pure and Applied Algebra* 180.1 (2003), pp. 159–219. DOI: https://doi.org/10.1016/S0022-4049(02)00248-7. URL: https://www.sciencedirect.com/science/article/pii/S0022404902002487.

[nLaa] nLab. *adjoints preserve (co-)limits*. URL: https://ncatlab.org/nlab/show/adjoints+preserve+%28co-%29limits (visited on 23/03/2022).

[nLab] nLab. *skeleton*. URL: https://ncatlab.org/nlab/show/skeleton (visited on 23/03/2022).

[Rin] Michael Ringe. *The MeatAxe - Computing with Modular Representations*. URL: http://www.math.rwth-aachen.de/MTX/ (visited on 12/04/2022).

[Rog21] Liam Rogel. "Monoidal Categories of Equivariant Coherent Sheaves on Finite Sets". MA thesis. Technische Universität Kaiserslautern, 2021.

[TY98] Daisuke Tambara and Shigeru Yamagami. "Tensor Categories with Fusion Rules of Self-Duality for Finite Abelian Groups". In: *Journal of Algebra* 209.2 (1998), pp. 692–707. DOI: https://doi.org/10.1006/jabr.1998.7558. URL: https://www.sciencedirect.com/science/article/pii/S0021869398975585.

[Thi21a] Ulrich Thiel. *Comments on the book "Tensor Categories" by Etingof–Gelaki–Nikshych–Ostrik*. 2021. URL: https://ulthiel.com/math/teaching/lecture-notes/comments-egno/#Chapter_4_Tensor_categories (visited on 16/03/2022).

[Thi21b] Ulrich Thiel. *Introduction to Categorical Thinking and Categorification*. https://ulthiel.com/math/wp-content/uploads/lecture-notes/Tensor-Categories.pdf. Online; v0.1. 2021.

A. Source Code

A.1. TensorCategories.jl and Abstracts

```
module TensorCategories

import Base: show,^,==, getindex, in, issubset, iterate, length,*,+,-, iterate,
                getproperty

import Oscar.AbstractAlgebra.Integers

import Oscar: VectorSpace, Field, elem_type, QQ, FieldElem,
                dim, base_ring, MatrixSpace, GAPGroup, GroupElem,
                ModuleIsomorphism, parent, matrix, basis, MatElem, ∘, gens,
                ⊕, compose, ⊗, tensor_product, Map, MatrixElem, kronecker_product,
                id, domain, one, zero, MatrixSpace, size, AbstractSet, inv, product,
                Ring, RingElem, base_field, MPolyQuo, iscommutative, isinvertible,
                MatrixGroup, hom, GAPGroupHomomorphism, GL, MatrixSpace, matrix,
                codomain, GAP, characteristic, degree, julia_to_gap, GSet, gset,
                FinField,gen, GSet, gset, orbits, stabilizer, orbit,
                isisomorphic, issubgroup, left_transversal, ComplexField, order,
                elements, index, symmetric_group, gap_to_julia, multiplication_table,
                issemisimple, AlgAss, AlgAssElem, FiniteField, abelian_closure,
                irreducible_modules, action, decompose,+, dual, tr, iscentral, rank,
                ZZ, solve_left, PolynomialRing, groebner_basis, ideal, roots,
                splitting_field, change_base_ring, isconstant, coeff, isindependent,
                coefficients, isabelian, leading_monomial, gcd, msolve, fmpz, fmpq,
                rref, NumberField, nf_elem, kernel, cokernel, primary_decomposition,
                Ideal, minpoly, image

export Category, TensorCategory, Morphism, Object, VectorSpaces, base_ring, hom,
        GradedVectorSpaces, VectorSpaceObject, simples,
        VectorSpaceMorphism, parent, dsum,⊕, domain, codomain, compose, ∘, ^, ⊗,
        tensor_product,==, associator, basis, id, getindex, one, zero, Forgetful,
        Functor, Sets, SetObject, SetMorphism, inv, product, coproduct,
        features, issemisimple, isabelian, ismonoidal, ×, □, RepresentationCategory,
        GroupRepresentationCategory, ismultiring, ismultifusion, isring, ismultitensor,
        istensor,
         FusionCategory, VSHomSpace,HomSpace,
        Hom, GVSHomSpace, HomFunctor, VSObject, GVSObject, GVSMorphism, SetHomSet,
        HomSet, Cocycle, trivial_3_cocylce,*,
        +,-, GroupRepresentation, GroupRepresentationMorphism,
        GroupRepresentationCategory,
        isinvertible, Representation, isequivariant, matrix, GRHomSpace,
        OppositeCategory, OppositeMorphism, OppositeObject, ProductCategory,
```

```
        ProductObject, ProductMorphism, CohSheaves, CohSheaf, CohSheafMorphism,
        stalks, PullbackFunctor, Pullback, PushforwardFunctor, Pushforward,
        CohSfHomSpace, ConvolutionCategory, ConvolutionObject, ConvolutionMorphism,
        ConvHomSpace,stalk, induction, restriction, orbit_index, dsum_morphisms,
        decompose, multiplication_table, print_multiplication_table, grothendieck_ring,
        dual, left_dual, right_dual, ev, coev, left_trace, right_trace, tr, braiding,
        drinfeld_morphism, smatrix, End, CenterCategory, CenterObject, CenterMorphism,
        spherical, iscentral, center_simples, RingCategory, set_tensor_product!,
        set_braiding!, Ising, zero_morphism, express_in_basis, solve_groebner,
        Center, CenterCategory, CenterObject, CenterMorphism, ev_coev, matrices,
        orbit_stabilizers, GRepRestriction, GRepInduction, Restriction, Induction,
        print_multiplication_table, RingObject, RingMorphism, kernel, cokernel,
        image, isgraded, cyclic_group_3cocycle, decompose_morphism,
        central_objects, half_braiding, half_braidings, left_inverse, right_inverse,
        simple_subobjects, add_simple!

include("Utility/FFE_to_FinField.jl")
include("Utility/SolveGroebner.jl")
include("structures/abstracts.jl")
include("structures/MISC/ProductCategory.jl")
include("structures/MISC/OppositeCategory.jl")
include("structures/VectorSpaces/VectorSpaces.jl")
include("structures/VectorSpaces/Cocycles.jl")
include("structures/VectorSpaces/GradedVectorSpaces.jl")
include("structures/set.jl")
include("structures/Representations/Representations.jl")
include("structures/Representations/GroupRepresentations.jl")
include("structures/Functors.jl")
include("structures/ConvolutionCategory/CoherentSheaves.jl")
include("structures/ConvolutionCategory/ConvolutionCategory.jl")
include("structures/MultiFusionCategories/FusionCategory.jl")
include("structures/MultiFusionCategories/Duals.jl")
include("structures/MISC/multiplication_table.jl")
include("structures/GrothendieckRing.jl")
include("structures/Center/Center.jl")

end

#-------------------------------------------------------------------------
#   Structs for categories
#-------------------------------------------------------------------------

abstract type Category end

abstract type Object end

abstract type Morphism end


"""
    VectorSpaceObject
```

```
An object in the category of finite dimensional vector spaces.
"""
abstract type VectorSpaceObject <: Object end

"""
    VectorSpaceMorphism

A morphism in the category of finite dimensional vector spaces.
"""
abstract type VectorSpaceMorphism <: Morphism end

abstract type HomSet end

abstract type HomSpace <: VectorSpaceObject end

domain(m::Morphism) = m.domain
codomain(m::Morphism) = m.codomain

"""
    parent(X::Object)

Return the parent category of the object X.
"""
parent(X::Object) = X.parent

"""
    base_ring(X::Object)

Return the base ring ```k``` of the ```k```-linear parent category of ```X```.
"""
base_ring(X::Object) = parent(X).base_ring

base_ring(X::Morphism) = parent(domain(X)).base_ring
"""
    base_ring(C::Category)

Return the base ring ```k```of the ```k```-linear category ```C```.
"""
base_ring(C::Category) = C.base_ring

base_group(C::Category) = C.base_group
base_group(X::Object) = parent(X).base_group

#-----------------------------------------------------------
#   Direct Sums, Products, Coproducts
#-----------------------------------------------------------

function ⊕(T::Tuple{S,Vector{R},Vector{R2}},X::S1) where {S <: Object,S1 <: Object, R <:
 ↪  Morphism, R2 <: Morphism}
    Z,ix,px = dsum(T[1],X)
    incl = vcat([ix[1] ∘ t for t in T[2]], ix[2:2])
    proj = vcat([t ∘ px[1] for t in T[3]], px[2:2])
```

```
    return Z, incl, proj
end

⊕(X::S1,T::Tuple{S,Vector{R}, Vector{R2}}) where {S <: Object,S1 <: Object, R <:
 ↪  Morphism, R2 <: Morphism} = ⊕(T,X)

function dsum(X::Object...)
    if length(X) == 0 return nothing end
    Z = X[1]
    for Y ∈ X[2:end]
        Z = dsum(Z,Y)
    end
    return Z
end

function dsum_morphisms(X::Object...)
    if length(X) == 1
        return X[1], [id(X[1]),id(X[1])]
    end
    Z,ix,px = dsum(X[1],X[2],true)
    for Y in X[3:end]
        Z,ix,px = ⊕((Z,ix,px),Y)
    end
    return Z,ix,px
end

function dsum(f::Morphism...)
    g = f[1]

    for h ∈ f[2:end]
        g = g ⊕ h
    end
    return g
end

function ×(T::Tuple{S,Vector{R}},X::S1) where {S <: Object,S1 <: Object, R <: Morphism}
    Z,px = product(T[1],X)
    m = vcat([t ∘ px[1] for t in T[2]], px[2])
    return Z, m
end

×(X::S1,T::Tuple{S,Vector{R}}) where {S <: Object,S1 <: Object, R <: Morphism} = ×(T,X)

function product(X::Object...)
    if length(X) == 0 return nothing end
    Z = X[1]
    for Y ∈ X[2:end]
        Z = product(Z,Y)
    end
    return Z
end
```

```
function product_morphisms(X::Object...)
    if length(X) == 1
        return X[1], [id(X[1])]
    end
    Z,px = product(X[1],X[2], true)
    for Y in X[3:end]
        Z,px = ×((Z,px),Y)
    end
    return Z,px
end

function ∐(T::Tuple{S,Vector{R}},X::S1) where {S <: Object,S1 <: Object, R <: Morphism}
    Z,px = coproduct(T[1],X)
    m = vcat([px[1] ∘ t for t in T[2]], px[2])
    return Z, m
end

∐(X::S1,T::Tuple{S,Vector{R}}) where {S <: Object,S1 <: Object, R <: Morphism} = ∐(T,X)

function coproduct(X::Object...)
    if length(X) == 0 return nothing end
    Z = X[1]
    for Y in X[2:end]
        Z = coproduct(Z,Y)
    end
    return Z
end

function coproduct_morphisms(X::Object...)
    if length(X) == 1
        return X[1], [id(X[1])]
    end
    Z,ix = coproduct(X[1],X[2])
    for Y in X[3:end]
        Z,ix = ∐((Z,ix),Y)
    end
    return Z,ix
end

"""
    ×(X::Object...)

Return the product Object and an array containing the projection morphisms.
"""
×(X::Object...) = product(X...)

"""
    ∐(X::Object...)

Return the coproduct Object and an array containing the injection morphisms.
"""
∐(X::Object...) = coproduct(X...)
```

```
"""
    ⊕(X::Object...)

Return the direct sum Object and arrays containing the injection and projection
morphisms.
"""

⊕(X::Object...) = dsum(X...)

⊕(X::Morphism...) = dsum(X...)

"""
    ⊗(X::Object...)

Return the tensor product object.
"""
⊗(X::Object...) = tensor_product(X...)

"""
    ^(X::Object, n::Integer)

Return the n-fold product object ```X^n```.
"""
^(X::Object,n::Integer) = n == 0 ? zero(parent(X)) : product([X for i in 1:n]...)

^(X::Morphism,n::Integer) = n == 0 ? zero_morphism(zero(parent(domain(X))),
 ↪  zero(parent(domain(X)))) : dsum([X for i in 1:n]...)
"""
    ⊗(f::Morphism, g::Morphism)

Return the tensor product morphism of ```f```and ```g```.
"""
⊗(f::Morphism, g::Morphism) where {T} = tensor_product(f,g)


dsum(X::T) where T <: Union{Vector,Tuple} = dsum(X...)
product(X::T) where T <: Union{Vector,Tuple} = product(X...)
coproduct(X::T) where T <: Union{Vector,Tuple} = coproduct(X...)

product(X::Object,Y::Object) = dsum(X,Y)
coproduct(X::Object, Y::Object) = dsum(X,Y)
#-------------------------------------------------------
#   tensor_product
#-------------------------------------------------------

function tensor_product(X::Object...)
    if length(X) == 1 return X end

    Z = X[1]
    for Y ∈ X[2:end]
        Z = Z⊗Y
```

```
    end
    return Z
end

tensor_product(X::T) where T <: Union{Vector,Tuple} = tensor_product(X...)
#-------------------------------------------------------
#   Abstract Methods
#-------------------------------------------------------
isfusion(C::Category) = false
ismultifusion(C::Category) = isfusion(C)

istensor(C::Category) = isfusion(C)
ismultitensor(C::Category) = ismultifusion(C) || istensor(C)

isring(C::Category) = istensor(C)
ismultiring(C::Category) = ismultitensor(C)

ismonoidal(C::Category) = ismultitensor(C)

isabelian(C::Category) = ismultitensor(C)

isadditive(C::Category) = isabelian(C)

islinear(C::Category) = isabelian(C)

issemisimple(C::Category) = ismultitensor(C)

function image(f::Morphism)
    C,c = cokernel(f)
    return kernel(c)
end

∘(f::Morphism...) = compose(reverse(f)...)

-(f::Morphism, g::Morphism) = f + (-1)*g
-(f::Morphism) = (-1)*f

#--------------------------------------------------------
# Hom Spaces
#--------------------------------------------------------

dim(V::HomSpace) = length(basis(V))

End(X::Object) = Hom(X,X)


#--------------------------------------------------------
# Duals
#--------------------------------------------------------

left_dual(X::Object) = dual(X)
right_dual(X::Object) = dual(X)
```

```
dual(f::Morphism) = left_dual(f)

function left_dual(f::Morphism)
    X = domain(f)
    Y = codomain(f)
    a = ev(Y)⊗id(dual(X))
    b = id(dual(Y)⊗f)⊗id(dual(X))
    c = inv(associator(dual(Y),X,dual(X)))
    d = id(dual(Y)⊗coev(X))
    (a)∘(b)∘(c)∘(d)
end

tr(f::Morphism) = left_trace(f)

function left_trace(f::Morphism)
    V = domain(f)
    W = codomain(f)
    C = parent(V)
    if V == zero(C) || W == zero(C) return zero_morphism(one(C),one(C)) end

    if V == W
        return ev(left_dual(V)) ∘ ((spherical(V)∘f) ⊗ id(left_dual(V))) ∘ coev(V)
    end
    return ev(left_dual(V)) ∘ (f ⊗ id(left_dual(V))) ∘ coev(V)
end

function right_trace(f::Morphism)
    V = domain(f)
    W = codomain(f)
    dV = right_dual(V)
    _,i = isisomorphic(left_dual(dV),V)
    _,j = isisomorphic(right_dual(V), left_dual(right_dual(dV)))
    return (ev(right_dual(dV))) ∘ (j⊗(f∘i)) ∘ coev(right_dual(V))
end

#-------------------------------------------------------
# Spherical structure
#-------------------------------------------------------

function drinfeld_morphism(X::Object)
     (ev(X)⊗id(dual(dual(X)))) ∘ (braiding(X,dual(X))⊗id(dual(dual(X)))) ∘
      ↪  (id(X)⊗coev(dual(X)))
 end

dim(X::Object) = base_ring(X)(tr(spherical(X)))

dim(C::Category) = sum(dim(s)^2 for s ∈ simples(C))
#-------------------------------------------------------
# S-Matrix
#-------------------------------------------------------
```

```
function smatrix(C::Category, simples = simples(C))
    @assert issemisimple(C) "Category has to be semisimple"
    F = base_ring(C)
    m = [tr(braiding(s,t)∘braiding(t,s)) for s ∈ simples, t ∈ simples]
    try
        return matrix(F,[F(n) for n ∈ m])
    catch
    end
    return matrix(F,m)
end

#-------------------------------------------------------
# decomposition morphism
#-------------------------------------------------------

function decompose(X::Object, S = simples(parent(X)))
    C = parent(X)
    @assert issemisimple(C) "Category not semisimple"
    dimensions = [dim(Hom(X,s)) for s ∈ S]
    return [(s,d) for (s,d) ∈ zip(S,dimensions) if d > 0]
end

function decompose_morphism(X::Object)
    C = parent(X)
    @assert issemisimple(C) "Semisimplicity required"

    S = simples(C)

    proj = [basis(Hom(X,s)) for s ∈ S]
    dims = [length(p) for p ∈ proj]

    Z = dsum([s^d for (s,d) ∈ zip(S,dims)])
    incl = [basis(Hom(s,Z)) for s ∈ S]

    f = zero_morphism(X,Z)

    for (pk,ik) ∈ zip(proj, incl)
        for (p,i) ∈ zip(pk,ik)
            g = i∘p
            f = f + g
        end
    end
    return f
end




#-------------------------------------------------------
# Semisimple: Subobjects
#-------------------------------------------------------
```

```
function unique_simples(simples::Vector{<:Object})
    unique_simples = simples[1:1]
    for s ∈ simples[2:end]
        if sum([dim(Hom(s,u)) for u ∈ unique_simples]) == 0
            unique_simples = [unique_simples; s]
        end
    end
    return unique_simples
end
```

A.2. Vector Spaces

```
"""
    VectorSpaces{T}(K::S) where T <: FieldElem

The category of finite dimensional vector spaces over K.
"""
struct VectorSpaces <: Category
    base_ring::Field
end


struct VSObject<: VectorSpaceObject
    basis::Vector
    parent::VectorSpaces
end

struct VSMorphism <: VectorSpaceMorphism
    m::MatElem
    domain::VectorSpaceObject
    codomain::VectorSpaceObject
end

isfusion(::VectorSpaces) = true

#-----------------------------------------------------------------
#   Constructors
#-----------------------------------------------------------------

# function (Vec::VectorSpaces{T})(V::FreeModule{T}) where T <: FieldElem
#     return VectorSpaceObject{T,FreeModule{T}}(V,Vec)
# end
#

"""
    VectorSpaceObject(Vec::VectorSpaces, n::Int64)
    VectorSpaceObject(K::Field, n::Int)
    VectorSpaceObject(Vec::VectorSpaces, basis::Vector)
    VectorSpaceObject(K::Field, basis::Vector)
```

```
The n-dimensional vector space with basis v1,..,vn (or other specified basis)
"""
function VectorSpaceObject(Vec::VectorSpaces, n::Int)
    basis = ["v$i" for i ∈ 1:n]
    return VSObject(basis,Vec)
end


function VectorSpaceObject(K::Field,n::Int)
    Vec = VectorSpaces(K)
    return VectorSpaceObject(Vec,n)
end

function VectorSpaceObject(Vec::VectorSpaces, basis::Vector)
    return VSObject(basis, Vec)
end

function VectorSpaceObject(K::Field, basis::Vector)
    Vec = VectorSpaces(K)
    return VSObject(basis, Vec)
end
"""
    Morphism(X::VectorSpaceObject, Y::VectorSpaceObject, m::MatElem)

Return a morphism in the category of vector spaces defined by m.
"""
function Morphism(X::VectorSpaceObject, Y::VectorSpaceObject, m::MatElem)
    if parent(X) != parent(Y)
        throw(ErrorException("Missmatching parents."))
    elseif size(m) != (dim(X),dim(Y))
        throw(ErrorException("Mismatching dimensions"))
    else
        return VSMorphism(m,X,Y)
    end
end

"""
    Morphism(m::MatElem)

Vector space morphisms defined by m.
"""
function Morphism(m::MatElem)
    l,n = size(m)
    F = base_ring(m)
    dom = VectorSpaceObject(F,l)
    codom = VectorSpaceObject(F,n)
    return Morphism(dom,codom,m)
end
#
# function VectorSpaceMorphism(X::VectorSpaceObject{T}, Y::VectorSpaceObject{T}, m::U)
 ↪  where {T,U <: MatrixElem}
```

```
#     if parent(X) == parent(Y)
#         f = ModuleHomomorphism(X.V,Y.V,m)
#         return VectorSpaceMorphism{T}(f,X,Y)
#     else
#         throw(ErrorException("Missmatching parents."))
#     end
# end
#
# #-------------------------------------------------------------------
# #   Pretty Printing
# #-------------------------------------------------------------------
function Base.show(io::IO, C::VectorSpaces)
    print(io,"Category of finite dimensional VectorSpaces over $(C.base_ring)")
end

function Base.show(io::IO, V::VectorSpaceObject)
    print(io, "Vector space of dimension $(dim(V)) over $(base_ring(V)).")
end

function Base.show(io::IO, m::VectorSpaceMorphism)
    print(io, """
Vector space morphism with
Domain:$(domain(m))
Codomain:$(codomain(m))""")
end
#-------------------------------------------------------------------
#   Functionality
#-------------------------------------------------------------------

base_ring(V::VectorSpaceObject) = parent(V).base_ring
base_ring(Vec::VectorSpaces) = Vec.base_ring

"""
    dim(V::VectorSpaceObject) = length(V.basis)

Return the vector space dimension of ``V``.
"""
dim(V::VectorSpaceObject) = length(basis(V))

basis(V::VectorSpaceObject) = V.basis

simples(Vec::VectorSpaces) = [VectorSpaceObject(base_ring(Vec),1)]

decompose(V::VSObject) = [(one(parent(V)),dim(V))]

matrix(f::VectorSpaceMorphism) = f.m
"""
    one(Vec::VectorSpaces) = VectorSpaceObject(base_ring(Vec),1)

Return the one-dimensional vector space.
"""
one(Vec::VectorSpaces) = VectorSpaceObject(base_ring(Vec),1)
```

```julia
"""
    zero(Vec::VectorSpaces) = VectorSpaceObject(base_ring(Vec), 0)

Return the zero-dimensional vector space.
"""
zero(Vec::VectorSpaces) = VectorSpaceObject(base_ring(Vec), 0)

==(V::VectorSpaces,W::VectorSpaces) = V.base_ring == W.base_ring

function ==(X::VectorSpaceObject, Y::VectorSpaceObject) where T
    basis(X) == basis(Y) && base_ring(X) == base_ring(Y)
end

"""
    isisomorphic(V::VSObject, W::VSObject)

Check whether ``V`` and ``W``are isomorphic. Return the isomorphisms if existent.
"""
function isisomorphic(V::VectorSpaceObject, W::VectorSpaceObject)
    if parent(V) != parent(W) return false, nothing end
    if dim(V) != dim(W) return false, nothing end

    return true, Morphism(V,W,one(MatrixSpace(base_ring(V),dim(V),dim(V))))
end


dual(V::VectorSpaceObject) = Hom(V,one(parent(V)))

function ev(V::VectorSpaceObject)
    dom = dual(V)⊗V
    cod = one(parent(V))
    m = [matrix(f)[i] for f ∈ basis(dual(V)), i ∈ 1:dim(V)]
    Morphism(dom,cod, matrix(base_ring(V), reshape(m,dim(dom),1)))
end

function coev(V::VectorSpaceObject)
    dom = one(parent(V))
    cod = V ⊗ dual(V)
    m = [Int(i==j) for i ∈ 1:dim(V), j ∈ 1:dim(V)][:]
    Morphism(dom,cod, transpose(matrix(base_ring(V), reshape(m,dim(cod),1))))
end

spherical(V::VectorSpaceObject) = Morphism(V,dual(dual(V)), id(V).m)

#-----------------------------------------------------------------
#   Functionality: Direct Sum
#-----------------------------------------------------------------

"""
    dsum(X::VectorSpaceObject{T}, Y::VectorSpaceObject{T}, morphisms = false) where {T}
```

```
Direct sum of vector spaces together with the embedding morphisms if morphisms = true.
"""
function dsum(X::VectorSpaceObject, Y::VectorSpaceObject, morphisms::Bool = false)
    if parent(X) != parent(Y)
        throw(ErrorException("Mismatching parents."))
    end

    if dim(X) == 0 return morphisms ? (Y,[zero_morphism(X,Y), id(Y)],
     ↪  [zero_morphism(Y,X), id(Y)]) : Y end
    if dim(Y) == 0 return morphisms ? (X,[id(X), zero_morphism(Y,X)], [id(X),
     ↪  zero_morphism(X,Y), ]) : X end

    F = base_ring(X)
    b = [(1,x) for x in basis(X)] ∪ [(2,y) for y in basis(Y)]

    V = VectorSpaceObject(parent(X),b)

    if !morphisms return V end

    ix = Morphism(X,V, matrix(F,[i == j ? 1 : 0 for i ∈ 1:dim(X), j ∈ 1:dim(V)]))
    iy = Morphism(Y,V, matrix(F,[i == j - dim(X) for i ∈ 1:dim(Y), j ∈ 1:dim(V)]))

    px = Morphism(V,X, transpose(matrix(ix)))
    py = Morphism(V,Y, transpose(matrix(iy)))

    return V,[ix,iy], [px,py]
end

product(X::VectorSpaceObject, Y::VectorSpaceObject, projections::Bool = false) =
 ↪  projections ? dsum(X,Y, projections)[[1,3]] : dsum(X,Y)
coproduct(X::VectorSpaceObject, Y::VectorSpaceObject, injections::Bool = false) =
 ↪  injections ? dsum(X,Y, injections)[[1,2]] : dsum(X,Y)

"""
    dsum(f::VectorSpaceMorphism{T},g::VectorSpaceMorphism{T}) where T

Return the direct sum of morphisms of vector spaces.
"""
function dsum(f::VectorSpaceMorphism,g::VectorSpaceMorphism)
    F = base_ring(domain(f))
    mf,nf = size(f.m)
    mg,ng = size(g.m)
    z1 = zero(MatrixSpace(F,mf,ng))
    z2 = zero(MatrixSpace(F,mg,nf))
    m = vcat(hcat(f.m,z1), hcat(z2,g.m))
    return VSMorphism(m,dsum(domain(f),domain(g)),dsum(codomain(f),codomain(g)))
end

#-------------------------------------------------------------------
#   Functionality: (Co)Kernel
#-------------------------------------------------------------------
```

```julia
function kernel(f::VSMorphism)
    F = base_ring(domain(f))
    d,k = kernel(f.m, F, side = :left)
    k = k[1:d,:]
    K = VectorSpaceObject(parent(domain(f)), d)
    return K, Morphism(K,domain(f),k)
end

function cokernel(f::VSMorphism)
    F = base_ring(domain(f))
    d,k = kernel(f.m, F)
    k = k[:,1:d]
    K = VectorSpaceObject(parent(domain(f)), d)
    return K, Morphism(codomain(f), K, k)
end
#-----------------------------------------------------------------
#   Functionality: Tensor Product
#-----------------------------------------------------------------

"""
    tensor_product(X::VectorSpaceObject{T}, Y::VectorSpaceObject{T}) where {T,S1,S2}

Return the tensor product of vector spaces.
"""
function tensor_product(X::VectorSpaceObject, Y::VectorSpaceObject)
    if parent(X) != parent(Y)
        throw(ErrorException("Mismatching parents."))
    end
    b = [[(x,y) for y ∈ basis(Y), x ∈ basis(X)]...]
    return VectorSpaceObject(parent(X),b)
end

"""
    tensor_product(f::VectorSpaceMorphism, g::VectorSpaceMorphism)

Return the tensor product of vector space morphisms.
"""
function tensor_product(f::VectorSpaceMorphism, g::VectorSpaceMorphism)
    D = tensor_product(domain(f),domain(g))
    C = tensor_product(codomain(f),codomain(g))
    m = kronecker_product(f.m, g.m)
    return Morphism(D,C,m)
end
#


#-----------------------------------------------------------------
#   Functionality: Morphisms
#-----------------------------------------------------------------

function compose(f::VectorSpaceMorphism...)
```

```
    if [isisomorphic(domain(f[i]), codomain(f[i-1]))[1] for i ∈ 2:length(f)] !=
     ↪  trues(length(f)-1)
        throw(ErrorException("Morphisms not compatible"))
    end
    return VSMorphism(*([g.m for g ∈ f]...),domain(f[1]),codomain(f[end]))
end


function ==(f::VectorSpaceMorphism, g::VectorSpaceMorphism)
    a = domain(f) == domain(g)
    b = codomain(f) == codomain(g)
    c = f.m == g.m
    return a && b && c
end

function +(f::VectorSpaceMorphism, g::VectorSpaceMorphism)
    @assert isisomorphic(domain(f),domain(g))[1] &&
     ↪  isisomorphic(codomain(f),codomain(g))[1]
    return Morphism(domain(f),codomain(f), f.m + g.m)
end

"""
    id(X::VectorSpaceObject{T}) where T

Return the identity on the vector space ``X``.
"""
function id(X::VectorSpaceObject)
    n = dim(X)
    m = matrix(base_ring(X), [i == j ? 1 : 0 for i ∈ 1:n, j ∈ 1:n])
    return Morphism(X,X,m)
end

inv(f::VectorSpaceMorphism)= Morphism(codomain(f), domain(f), inv(matrix(f)))

*(λ,f::VectorSpaceMorphism)  =
 ↪  Morphism(domain(f),codomain(f),parent(domain(f)).base_ring(λ)*f.m)

isinvertible(f::VectorSpaceMorphism) = rank(f.m) == dim(domain(f)) ==
 ↪  dimension(codomain(f))

function left_inverse(f::VectorSpaceMorphism)
    k = matrix(f)
    d = rank(k)
    F = base_ring(f)
    k_inv = transpose(solve_left(transpose(k), one(MatrixSpace(F,d,d))))
    return Morphism(codomain(f), domain(f), k_inv)
end

function right_inverse(f::VectorSpaceMorphism)
    k = matrix(f)
    d = rank(k)
    F = base_ring(f)
```

```
    c_inv = solve_left(k, one(MatrixSpace(F,d,d)))
    return Morphism(codomain(f),domain(f), c_inv)
end
#-------------------------------------------------------------------------------
#   Associators
#-------------------------------------------------------------------------------
#

"""
    associator(X::VectorSpaceObject, Y::VectorSpaceObject, Z::VectorSpaceObject)

Return the associator isomorphism a::(X⊗Y)⊗Z -> X⊗(Y⊗Z).
"""
function associator(X::VectorSpaceObject, Y::VectorSpaceObject, Z::VectorSpaceObject)
    if !(parent(X) == parent(Y) == parent(Z))
        throw(ErrorException("Mismatching parents"))
    end
    n = *(dim.([X,Y,Z])...)
    F = base_ring(X)
    m = matrix(F, [i == j ? 1 : 0 for i ∈ 1:n, j ∈ 1:n])
    return Morphism((X⊗Y)⊗Z, X⊗(Y⊗Z), m)
end


#--------------------------------------------------------------------------------
#   Hom Spaces
#--------------------------------------------------------------------------------

struct VSHomSpace <: HomSpace
    X::VectorSpaceObject
    Y::VectorSpaceObject
    basis::Vector{VectorSpaceMorphism}
    parent::VectorSpaces
end

"""
    Hom(X::VectorSpaceObject, Y::VectorSpaceObject)

Return the Hom(``X,Y```) as a vector space.
"""
function Hom(X::VectorSpaceObject, Y::VectorSpaceObject)
    n1,n2 = (dim(X),dim(Y))
    mats = [matrix(base_ring(X), [i==k && j == l ? 1 : 0 for i ∈ 1:n1, j ∈ 1:n2]) for k
     ↪  ∈ 1:n1, l ∈ 1:n2]
    basis = [[Morphism(X,Y,m) for m ∈ mats]...]
    return VSHomSpace(X,Y,basis,VectorSpaces(base_ring(X)))
end

basis(V::VSHomSpace) = V.basis

zero(V::VSHomSpace) = Morphism(V.X,V.Y,matrix(base_ring(V.X), [0 for i ∈ 1:dim(V.X), j ∈
 ↪  1:dim(V.Y)]))
```

```
zero_morphism(V::VectorSpaceObject,W::VectorSpaceObject) = Morphism(V,W,
 ↪  zero(MatrixSpace(base_ring(V), dim(V),dim(W))))

function express_in_basis(f::VectorSpaceMorphism, B::Vector{<:VectorSpaceMorphism})
    F = base_ring(f)
    B_mat = matrix(F,hcat([[x for x ∈ b.m][:] for b ∈ B]...))
    f_mat = matrix(F, 1, *(size(f.m)...), [x for x ∈ f.m][:])

    return [x for x ∈ solve_left(transpose(B_mat),f_mat)][:]
end

(F::Field)(f::VectorSpaceMorphism) = F(matrix(f)[1,1])


struct Cocycle{N}
    group::GAPGroup
    F::Field
    m::Union{Nothing,Dict{NTuple{N,G},T}} where {G<:GroupElem,T<:FieldElem}
end

"""
    Cocylce(G::GAPGroup, m::Dict{NTuple{N,G}, T})

Return a ```N```-cocylce of ```G```. By now the condition is not checked.
"""
function Cocycle(G::GAPGroup, m::Dict{NTuple{N,S},T}) where
 ↪  {S<:GroupElem,T<:FieldElem,N}
    return Cocycle(G,parent(collect(values(m))[1]),m)
end

function Cocycle(G::GAPGroup, N::Int, f::Function)
    Cocycle(G,Dict(x => f(x...) for x ∈ Base.product([G for i ∈ 1:N]...)))
end

trivial_3_cocycle(G,F) = Cocycle{3}(G,F,nothing)

(c::Cocycle{N})(x...) where N = c.m == nothing ? c.F(1) : c.m[x]

function cyclic_group_3cocycle(G::GAPGroup, F::Field, ξ::FieldElem)
    g = G[1]
    n = order(G)
    D = Dict((g^i,g^j,g^k) => ξ^(div(i*(j+k - rem(j+k,n)),n)) for i ∈ 1:n, j ∈ 1:n, k ∈
     ↪  1:n)
    return Cocycle(G,D)
end

function show(io::IO, c::Cocycle{N}) where N
    print(io, "$N-Cocycle of $(c.group)")
end


struct GradedVectorSpaces <: Category
```

```
    base_ring::Field
    base_group::GAPGroup
    twist::Cocycle{3}
end

struct GVSObject <: VectorSpaceObject
    parent::GradedVectorSpaces
    V::VectorSpaceObject
    grading::Vector{<:GroupElem}
end

struct GVSMorphism <: VectorSpaceMorphism
    domain::GVSObject
    codomain::GVSObject
    m::MatElem
end

function GradedVectorSpaces(F::Field, G::GAPGroup)
    elems = elements(G)
    GradedVectorSpaces(F,G,trivial_3_cocycle(G,F))
end

function VectorSpaceObject(V::Pair{<:GroupElem, <:VectorSpaceObject}...)
    W = dsum([v for (_,v) ∈ V])
    G = parent(V[1][1])
    elems = elements(G)
    grading = vcat([[g for _ ∈ 1:dim(v)] for (g,v) ∈ V]...)
    C = GradedVectorSpaces(base_ring(W), G)
    return GVSObject(C, W, grading)
end

isfusion(C::GradedVectorSpaces) = true

dim(X::GVSObject) = dim(X.V)
basis(X::GVSObject) = basis(X.V)

function Morphism(X::GVSObject, Y::GVSObject, m::MatElem)
    if !isgraded(X,Y,m)
        throw(ErrorException("Matrix does not define graded morphism"))
    end
    return GVSMorphism(X,Y,m)
end

one(C::GradedVectorSpaces) = GVSObject(C,VectorSpaceObject(base_ring(C),1),
 ↪  [one(base_group(C))])
zero(C::GradedVectorSpaces) = GVSObject(C,VectorSpaceObject(base_ring(C),0),
 ↪  elem_type(base_group(C))[])

function isisomorphic(X::GVSObject, Y::GVSObject)
    b,f = isisomorphic(X.V,Y.V)
```

```
    return b && Set(X.grading) == Set(Y.grading) ? (true,Morphism(X,Y,matrix(f))) :
     ↪  (false,nothing)
end
#----------------------------------------------------------------
#   Functionality: Direct Sums
#----------------------------------------------------------------

function dsum(X::GVSObject, Y::GVSObject, morphisms::Bool = false)
    W,(ix,iy),(px,py) = dsum(X.V, Y.V, true)
    m,n = dim(X), dim(Y)
    F = base_ring(X)
    grading = [X.grading; Y.grading]
    j = 1

    Z = GVSObject(parent(X), W, grading)

    if morphisms
        ix = Morphism(X,Z,matrix(ix))
        iy = Morphism(Y,Z,matrix(iy))

        px = Morphism(Z,X,matrix(px))
        py = Morphism(Z,Y,matrix(py))

        return Z, [ix,iy], [px,py]
    end

    return Z
end

# function dsum(f::GVSMorphism, g::GVSMorphism)
#     dom = domain(f)⊕domain(g)
#     cod = codomain(f)⊕codomain(g)
#     F = base_ring(f)
#     m1,n1 = size(f.m)
#     m2,n2 = size(g.m)
#     m = [f.m zero(MatrixSpace(F,m1,n2)); zero(MatrixSpace(F,m2,n1)) g.m]
# end

#----------------------------------------------------------------
#   Functionality: Tensor Product
#----------------------------------------------------------------

function tensor_product(X::GVSObject, Y::GVSObject)
    W = X.V ⊗ Y.V
    G = base_group(X)
    elems = elements(G)
    grading = vcat([[i*j for i ∈ Y.grading] for j ∈ X.grading]...)
    return GVSObject(parent(X), W, length(grading) == 0 ? elem_type(G)[] : grading)
end

#----------------------------------------------------------------
#   Functionality: Simple Objects
```

```
#-----------------------------------------------------------------

function simples(C::GradedVectorSpaces)
    K = VectorSpaceObject(base_ring(C),1)
    G = base_group(C)
    n = Int(order(G))
    return [GVSObject(C,K,[g]) for g ∈ G]
end

function decompose(V::GVSObject)
    simpls = simples(parent(V))
    return filter(e -> e[2] > 0, [(s, dim(Hom(s,V))) for s ∈ simpls])
end
#-----------------------------------------------------------------
#   Functionality: (Co)Kernel
#-----------------------------------------------------------------

function kernel(f::GVSMorphism)
    F = base_ring(f)
    G = base_group(domain(f))
    X,Y = domain(f),codomain(f)
    n = dim(X) - rank(f.m)
    m = zero(MatrixSpace(F, n, dim(X)))
    l = 1
    grading = elem_type(G)[]

    for x ∈ unique(domain(f).grading)
        i = findall(e -> e == x, X.grading)
        j = findall(e -> e == x, Y.grading)

        if length(i) == 0 continue end
        if length(j) == 0
            grading = [grading; [x for _ ∈ i]]
            for k in i
                m[l,k] = F(1)
                l = l +1
            end
            continue
        end

        mx = f.m[i,j]

        d,k = kernel(mx, side = :left)
        k = k[1:d,:]

        m[l:l+d-1 ,i] = k
        l = l+d
        grading = [grading; [x for _ ∈ 1:d]]
    end
    K = GVSObject(parent(X), VectorSpaceObject(F,n), grading)
    return K, GVSMorphism(K,domain(f), m)
end
```

```
function cokernel(f::GVSMorphism)
    g = GVSMorphism(codomain(f),domain(f),transpose(f.m))
    C,c = kernel(g)
    return C, GVSMorphism(codomain(f),C, transpose(c.m))
end

#-----------------------------------------------------------------
#   Functionality: Associators
#-----------------------------------------------------------------

function associator(X::GVSObject, Y::GVSObject, Z::GVSObject)
    C = parent(X)
    twist = C.twist
    elems = elements(base_group(C))

    dom = (X⊗Y)⊗Z
    cod = X⊗(Y⊗Z)

    m = one(MatrixSpace(base_ring(X),dim(dom),dim(cod)))

    j = 1
    for x ∈ X.grading, y ∈ Y.grading, z ∈ Z.grading
        m[j,j] = twist(x,y,z)
        j = j+1
    end
    return Morphism(dom,cod,m)
end

#-----------------------------------------------------------------
#   Functionality: Duals
#-----------------------------------------------------------------

function dual(V::GVSObject)
    W = dual(V.V)
    G = base_group(V)
    grading = [inv(j) for j ∈ V.grading]
    return GVSObject(parent(V), W, grading)
end

function ev(V::GVSObject)
    dom = dual(V)⊗V
    cod = one(parent(V))
    elems = elements(base_group(V))
    twist = parent(V).twist
    m = [i == j ? inv(twist(g,inv(g),g)) : 0 for (i,g) ∈ zip(1:dim(V), V.grading), j ∈
     ↪  1:dim(V)][:]
    Morphism(dom,cod, matrix(base_ring(V), reshape(m,dim(dom),1)))
end

#-----------------------------------------------------------------
#   Functionality: Hom-Spaces
```

```julia
#-------------------------------------------------------------------

struct GVSHomSpace <: HomSpace
    X::GVSObject
    Y::GVSObject
    basis::Vector{VectorSpaceMorphism}
    parent::VectorSpaces
end

function Hom(V::GVSObject, W::GVSObject)
    G = base_group(V)
    B = VSMorphism[]

    zero_M = MatrixSpace(base_ring(V), dim(V), dim(W))

    for x ∈ unique(V.grading)
        V_grading = findall(e -> e == x, V.grading)
        W_grading = findall(e -> e == x, W.grading)


        for i ∈ V_grading, j ∈ W_grading
            m = zero(zero_M)
            m[i,j] = 1
            B = [B; GVSMorphism(V,W,m)]
        end
    end
    return GVSHomSpace(V,W,B,VectorSpaces(base_ring(V)))
end

function isgraded(X::GVSObject, Y::GVSObject, m::MatElem)
    G = base_group(X)
    for k ∈ 1:order(G)
        i = findall(e -> e == k, X.grading)
        j = findall(e -> e == k, Y.grading)
        for t ∈ i, s ∈ 1:length(j)
            if m[t,s] != 0 && !(s ∈ j)
                return false
            end
        end
    end
    true
end

id(X::GVSObject) = Morphism(X,X,one(MatrixSpace(base_ring(X),dim(X),dim(X))))
#-------------------------------------------------------------------
#   Pretty Printing
#-------------------------------------------------------------------

function show(io::IO, C::GradedVectorSpaces)
    print(io, "Category of G-graded vector spaces over $(base_ring(C)) where G is
     ↪  $(base_group(C))")
end
```

```
function show(io::IO, V::GVSObject)
    elems = elements(base_group(V))
    print(io, "Graded vector space of dimension $(dim(V)) with grading\n$(V.grading)")
end
```

A.3. Representations

```
abstract type RepresentationCategory <: Category end
#abstract type AlgebraRepresentationCategory{T} <: RepresentationCategory{T} end
#abstract type HopfAlgebraRepresentationCategory <: AlgebraRepresentationCategory end

abstract type Representation <: Object end
# #abstract type GroupRepresentation <: Representation end
# abstract type AlgebraRepresentation <: Representation end
# abstract type HopfAlgebraRepresentation <: Representation end

abstract type RepresentationMorphism <: VectorSpaceMorphism end


dim(ρ::Representation) = ρ.dim
base_ring(ρ::Representation) = parent(ρ).base_ring
base_ring(Rep::RepresentationCategory) = Rep.base_ring

struct GroupRepresentationCategory <: RepresentationCategory
    group::GAPGroup
    base_ring::Field
end

struct GroupRepresentation <: Representation
    parent::GroupRepresentationCategory
    group::GAPGroup
    m
    base_ring::Ring
    dim::Int64
end

struct GroupRepresentationMorphism <: RepresentationMorphism
    domain::GroupRepresentation
    codomain::GroupRepresentation
    map::MatElem
end

istensor(::GroupRepresentationCategory) = true
isfusion(C::GroupRepresentationCategory) =
 ↪  mod(order(C.group),characteristic(base_ring(C))) != 0
#-------------------------------------------------------------------------
#   Constructors
#-------------------------------------------------------------------------
"""
    RepresentationCategory(G::GAPGroup, F::Field)
```

```
Category of finite dimensonal group representations of \\G\\.
"""
function RepresentationCategory(G::GAPGroup, F::Field)
    return GroupRepresentationCategory(G,F)
end

function RepresentationCategory(G::GAPGroup)
    return RepresentationCategory(G, abelian_closure(QQ)[1])
end

"""
    Representation(G::GAPGroup, pre_img::Vector, img::Vector)

Group representation defined by the images of generators of G.
"""
function Representation(G::GAPGroup, pre_img::Vector, img::Vector)
    F = base_ring(img[1])
    d = size(img[1])[1]
    H = GL(d, F)
    m = hom(G,H, pre_img,H.(img))
    return GroupRepresentation(RepresentationCategory(G,F),G,m,F,d)
end


"""
    Representation(G::GAPGroup, m::Function)

Group representation defined by m:G -> Mat_n.
"""
function Representation(G::GAPGroup, m::Function)
    F = order(G) == 1 ? base_ring(parent(m(elements(G)[1]))) :
     ↪  base_ring(parent(m(G[1])))
    d = order(G) == 1 ? size(m(elements(G)[1]))[1] : size(m(G[1]))[1]
    H = GL(d,F)
    m = hom(G,H,g -> H(m(g)))
    return GroupRepresentation(RepresentationCategory(G,F),G,m,F,d)
end


"""
    Morphism(ρ::GroupRepresentation, τ::GroupRepresentation, m::MatElem; check = true)

Morphism between representations defined by ``m``. If check == false equivariancy
will not be checked.
"""
function Morphism(ρ::GroupRepresentation, τ::GroupRepresentation, m::MatElem; check =
 ↪  true)
    if size(m) != (dim(ρ), dim(τ)) throw(ErrorException("Mismatching dimensions")) end
    if check
        if !isequivariant(m,ρ,τ) throw(ErrorException("Map has to be equivariant")) end
    end
```

```
    return GroupRepresentationMorphism(ρ,τ,m)
end

#-----------------------------------------------------------------------
#   Functionality
#-----------------------------------------------------------------------
"""
    issemisimple(C::GroupRepresentationCategory)

Return true if C is semisimple else false.
"""
issemisimple(C::GroupRepresentationCategory) = gcd(characteristic(base_ring(C)),
 ↪  order(base_group(C))) == 1

function (ρ::GroupRepresentation)(x)
    if ρ.m == 0
        F = base_ring(ρ)
        return GL(0,F)(zero(MatrixSpace(F,0,0)))
    elseif order(ρ.group) == 1
        return one(codomain(ρ.m))
    else
        return ρ.m(x)
    end
end
matrix(f::GroupRepresentationMorphism) = f.map

base_group(Rep::GroupRepresentationCategory) = Rep.group
base_group(ρ::GroupRepresentation) = ρ.group

"""
    parent(ρ::GroupRepresentation)

Return the parent representation category of ρ.
"""
parent(ρ::GroupRepresentation) = ρ.parent

"""
    zero(Rep::GroupRepresentationCategory)

Return the zero reprensentation.
"""
function zero(Rep::GroupRepresentationCategory)
    grp = base_group(Rep)
    F = base_ring(Rep)
    GroupRepresentation(Rep,grp,0,F,0)
end

"""
    one(Rep::GroupRepresentationCategory)

Return the trivial representation.
"""
```

```
function one(Rep::GroupRepresentationCategory)
    grp = base_group(Rep)
    F = base_ring(Rep)
    if order(grp) == 1 return Representation(grp,x -> one(MatrixSpace(F,1,1))) end
    Representation(grp, gens(grp), [one(MatrixSpace(F,1,1)) for _ ∈ gens(grp)])
end

"""
    id(ρ::GroupRepresentation)

Return the identity on ρ.
"""
function id(ρ::GroupRepresentation)
    return GroupRepresentationMorphism(ρ,ρ,one(MatrixSpace(base_ring(ρ),dim(ρ),dim(ρ))))
end

function ==(ρ::GroupRepresentation, τ::GroupRepresentation)
    if ρ.m == 0 || τ.m == 0
        return ρ.m == 0 && τ.m == 0
    elseif order(ρ.group) == 1
        if order(τ.group) == 1
            return dim(τ) == dim(ρ)
        end
        return false
    end
    return *([ρ.m(g) == τ.m(g) for g ∈ gens(base_group(ρ))]...)
end

function ==(C::RepresentationCategory, D::RepresentationCategory)
    return C.group == D.group && C.base_ring == D.base_ring
end

function ==(f::GroupRepresentationMorphism, g::GroupRepresentationMorphism)
    return domain(f) == domain(g) && codomain(f) == codomain(g) && f.map == g.map
end

"""
    isisomorphic(σ::GroupRepresentation, τ::GroupRepresentation)

Check whether σ and τ are isomorphic. If true return the isomorphism.
"""
function isisomorphic(σ::GroupRepresentation, τ::GroupRepresentation)
    @assert parent(σ) == parent(τ) "Mismatching parents"

    if dim(σ) != dim(τ) return false, nothing end
    if dim(σ) == 0 return true, zero_morphism(σ,τ) end

    F = base_ring(σ)
    grp = σ.group

    if order(grp) == 1 return true, Morphism(σ,τ,one(MatrixSpace(F,dim(σ),dim(τ)))) end
```

```
    gap_F = GAP.Globals.FiniteField(Int(characteristic(F)), degree(F))

    #Build the modules from σ and τ
    mats_σ = GAP.GapObj([GAP.julia_to_gap(σ(g)) for g ∈ gens(grp)])
    mats_τ = GAP.GapObj([GAP.julia_to_gap(τ(g)) for g ∈ gens(grp)])


    Mσ = GAP.Globals.GModuleByMats(mats_σ, gap_F)
    Mτ = GAP.Globals.GModuleByMats(mats_τ, gap_F)


    iso = GAP.Globals.MTX.IsomorphismModules(Mσ,Mτ)

    if iso == GAP.Globals.fail return false,nothing end

    m = matrix(F,[F(iso[i,j]) for i ∈ 1:dim(σ), j ∈ 1:dim(τ)])
    return true, Morphism(σ,τ,m)
end

function dual(ρ::GroupRepresentation)
    G = base_group(ρ)
    F = base_ring(ρ)
    if dim(ρ) == 0 return ρ end
    generators = order(G) == 1 ? elements(G) : gens(G)
    return Representation(G, generators, [transpose(matrix(ρ(inv(g)))) for g ∈
     ↪  generators])
end

function ev(ρ::GroupRepresentation)
    dom = dual(ρ) ⊗ ρ
    cod = one(parent(ρ))
    F = base_ring(ρ)
    m = matrix(ev(VectorSpaceObject(F,dim(ρ))))
    return Morphism(dom,cod,m)
end

function coev(ρ::GroupRepresentation)
    dom = one(parent(ρ))
    cod = ρ ⊗ dual(ρ)
    F = base_ring(ρ)
    m = matrix(coev(VectorSpaceObject(F,dim(ρ))))
    return Morphism(dom,cod, m)
end
#-------------------------------------------------------------------------
#   Functionality: Morphisms
#-------------------------------------------------------------------------

function compose(f::GroupRepresentationMorphism, g::GroupRepresentationMorphism)
    return GroupRepresentationMorphism(domain(f),codomain(g), matrix(f)*matrix(g))
end
```

```
associator(σ::GroupRepresentation, τ::GroupRepresentation, ρ::GroupRepresentation) =
 ↪  id(σ⊗τ⊗ρ)

*(x, f::GroupRepresentationMorphism) = Morphism(domain(f),codomain(f),x*f.map)

function +(f::GroupRepresentationMorphism, g::GroupRepresentationMorphism)
    @assert domain(f) == domain(g) && codomain(f) == codomain(g) "Not compatible"
    return Morphism(domain(f), codomain(f), f.map + g.map)
end

function (F::Field)(f::GroupRepresentationMorphism)
    D = domain(f)
    C = codomain(f)
    if dim(D) == dim(C) == 1
        return F(f.map[1,1])
    else
        throw(ErrorException("Cannot coerce"))
    end
end

#-----------------------------------------------------------------------
#   Functionality: (Co)Kernel
#-----------------------------------------------------------------------

function kernel(f::GroupRepresentationMorphism)
    ρ = domain(f)
    G = base_group(ρ)
    F = base_ring(ρ)

    d,k = kernel(f.map, side = :left)
    k = k[1:d,:]

    if d == 0
        return zero(parent(ρ)), zero_morphism(zero(parent(ρ)), ρ)
    end

    k_inv = transpose(solve_left(transpose(k), one(MatrixSpace(F,d,d))))

    generators = order(G) == 1 ? elements(G) : gens(G)

    images = [k*matrix(ρ(g))*k_inv for g ∈ generators]

    K = Representation(G, generators, images)

    return K, Morphism(K,ρ,k)
end

function cokernel(f::GroupRepresentationMorphism)
    ρ = codomain(f)
    G = base_group(ρ)
    F = base_ring(ρ)
    d,c = kernel(f.map, side = :right)
```

```
    c = c[:,1:d]

    if d == 0
        return zero(parent(ρ)), zero_morphism(ρ,zero(parent(ρ)))
    end

    c_inv = solve_left(c, one(MatrixSpace(F,d,d)))

    generators = order(G) == 1 ? elements(G) : gens(G)

    images = [c_inv*matrix(ρ(g))*c for g ∈ generators]
    C = Representation(G, generators, images)
    return C, Morphism(ρ,C,c)
end
#------------------------------------------------------------------------
#   Necessities
#------------------------------------------------------------------------

function isequivariant(m::MatElem, ρ::GroupRepresentation, τ::GroupRepresentation)
    if dim(ρ)*dim(τ) == 0 return true end
    for g ∈ gens(ρ.group)
        if matrix(ρ(g))*m != m*matrix(τ(g))
            return false
        end
    end
    return true
end


#------------------------------------------------------------------------
#   Tensor Products
#------------------------------------------------------------------------

"""
    tensor_product(ρ::GroupRepresentation, τ::GroupRepresentation)

Return the tensor product of representations.
"""
function tensor_product(ρ::GroupRepresentation, τ::GroupRepresentation)
    @assert ρ.group == τ.group "Mismatching groups"

    if ρ.m == 0 || τ.m == 0 return zero(parent(ρ)) end

    G = ρ.group

    if order(G) == 1
        g = elements(G)[1]
        return Representation(G, x -> kronecker_product(matrix(ρ(g)),matrix(τ(g))))
    end
    generators = gens(G)
    return Representation(G, generators, [kronecker_product(matrix(ρ(g)),matrix(τ(g)))
     ↪  for g ∈ generators])
```

```
end

"""
    tensor_product(f::GroupRepresentationMorphism, g::GroupRepresentationMorphism)

Return the tensor product of morphisms of representations.
"""
function tensor_product(f::GroupRepresentationMorphism, g::GroupRepresentationMorphism)
    dom = domain(f) ⊗ domain(g)
    codom = codomain(f) ⊗ codomain(g)

    m = kronecker_product(matrix(f),matrix(g))
    return Morphism(dom,codom, m)
end

function braiding(X::GroupRepresentation, Y::GroupRepresentation)
    F = base_ring(X)
    n,m = dim(X),dim(Y)
    map = zero(MatrixSpace(F,n*m,n*m))
    for i ∈ 1:n, j ∈ 1:m
        v1 = matrix(F,transpose([k == i ? 1 : 0 for k ∈ 1:n]))
        v2 = matrix(F,transpose([k == j ? 1 : 0 for k ∈ 1:m]))
        map[(j-1)*n + i, :] = kronecker_product(v1,v2)
    end
    return Morphism(X⊗Y, Y⊗X, transpose(map))
end

spherical(X::GroupRepresentation) = id(X)
#-------------------------------------------------------------------------
#   Direct Sum
#-------------------------------------------------------------------------

"""
    dsum(ρ::GroupRepresentation, τ::GroupRepresentation, morphisms::Bool = false)

Return the direct sum of representations. If morphisms is set true inclusion and
projection morphisms are also returned.
"""
function dsum(ρ::GroupRepresentation, τ::GroupRepresentation, morphisms::Bool = false)
    @assert ρ.group == τ.group "Mismatching groups"

    grp = ρ.group
    F = base_ring(ρ)

    if ρ.m == 0
        if !morphisms return τ end
        return τ,[GroupRepresentationMorphism(ρ,τ,zero(MatrixSpace(F,0,dim(τ)))),
         ↪  id(τ)], [GroupRepresentationMorphism(τ,ρ,zero(MatrixSpace(F,dim(τ),0))),
         ↪  id(τ)]
    elseif τ.m == 0
        if !morphisms return ρ end
```

```
        return ρ,[id(ρ), GroupRepresentationMorphism(τ,ρ,zero(MatrixSpace(F,0,dim(ρ)))),
         ↪  id(τ)], [id(ρ),
         ↪  GroupRepresentationMorphism(ρ,τ,zero(MatrixSpace(F,dim(ρ),0)))]
    end

    M1 = MatrixSpace(F,dim(ρ),dim(ρ))
    M2 = MatrixSpace(F,dim(ρ),dim(τ))
    M3 = MatrixSpace(F,dim(τ),dim(ρ))
    M4 = MatrixSpace(F,dim(τ),dim(τ))

    generators = order(grp) == 1 ? elements(grp) : gens(grp)

    S = Representation(grp, generators, [[matrix(ρ(g)) zero(M2); zero(M3) matrix(τ(g))]
     ↪  for g ∈ generators])

    if !morphisms return S end

    incl_ρ = Morphism(ρ, S, [one(M1) zero(M2)])
    incl_τ = Morphism(τ, S, [zero(M3) one(M4)])
    proj_ρ = Morphism(S, ρ, [one(M1); zero(M3)])
    proj_τ = Morphism(S, τ, [zero(M2); one(M4)])

    return S, [incl_ρ, incl_τ], [proj_ρ, proj_τ]
end

"""
    dsum(f::GroupRepresentationMorphism, g::GroupRepresentationMorphism)

Direct sum of morphisms of representations.
"""
function dsum(f::GroupRepresentationMorphism, g::GroupRepresentationMorphism)

    dom = domain(f)⊕domain(g)
    codom = codomain(f)⊕codomain(g)
    F = base_ring(domain(f))

    z1 = zero(MatrixSpace(F, dim(domain(f)), dim(codomain(g))))
    z2 = zero(MatrixSpace(F, dim(domain(g)), dim(codomain(f))))

    m = [matrix(f) z1; z2 matrix(g)]

    return Morphism(dom,codom, m)
end

# product(X::GroupRepresentation,Y::GroupRepresentation, morphisms = false) = morphisms
 ↪  ? dsum(X,Y, true)[[1,3]] : dsum(X,Y)
# coproduct(X::GroupRepresentation,Y::GroupRepresentation, morphisms = false) =
 ↪  morphisms ? dsum(X,Y, true)[[1,2]] : dsum(X,Y)


#-------------------------------------------------------------------------
#   Simple Objects
```

```
#-----------------------------------------------------------------------

"""
    simples(Rep::GroupRepresentationCategory)

Return a list of the simple objects in Rep.
"""
function simples(Rep::GroupRepresentationCategory)
    grp = base_group(Rep)
    F = base_ring(Rep)

    if order(grp) == 1 return [one(Rep)] end

    if characteristic(F) == 0
        mods = irreducible_modules(grp)
        reps = [Representation(grp,gens(grp),[matrix(x) for x ∈ action(m)]) for m ∈
         ↪  mods]
        return reps
    else

        gap_field = GAP.Globals.FiniteField(Int(characteristic(F)), degree(F))
        gap_reps = GAP.Globals.IrreducibleRepresentations(grp.X,gap_field)

        dims = [GAP.Globals.DimensionOfMatrixGroup(GAP.Globals.Range(m)) for m ∈
         ↪  gap_reps]

        oscar_reps = [GAPGroupHomomorphism(grp, GL(dims[i],F), gap_reps[i]) for i ∈
         ↪  1:length(gap_reps)]
        reps = [GroupRepresentation(Rep,grp,m,F,d) for (m,d) ∈ zip(oscar_reps,dims)]

        return reps
    end
end

"""
    decompose(σ::GroupRepresentation)

Decompose the representation into a direct sum of simple objects. Return a
list of tuples with simple objects and multiplicities.
"""
function decompose(σ::GroupRepresentation)
    F = base_ring(σ)
    if dim(σ) == 0 return [] end
    G = σ.group

    if order(G) == 1 return [(one(parent(σ)),dim(σ))] end

    M = to_gap_module(σ,F)
    ret = []
    facs = GAP.Globals.MTX.CollectedFactors(M)
    d = dim(σ)
    for m ∈ facs
```

```
        imgs = [matrix(F,[F(n[i,j]) for i ∈ 1:length(n), j ∈ 1:length(n)]) for n ∈
         ↪  m[1].generators]
        ret = [ret;(Representation(G,gens(G),imgs),GAP.gap_to_julia(m[2]))]
    end
    ret
end

#----------------------------------------------------------------------------
#   Hom Spaces
#----------------------------------------------------------------------------

struct GRHomSpace<: HomSpace
    X::GroupRepresentation
    Y::GroupRepresentation
    basis::Vector{GroupRepresentationMorphism}
    parent::VectorSpaces
end

"""
    Hom(σ::GroupRepresentation, τ::GroupRepresentation)

Return the hom-space of the representations as a vector space.
"""
function Hom(σ::GroupRepresentation, τ::GroupRepresentation)
    grp = base_group(σ)
    F = base_ring(σ)

    if dim(σ)*dim(τ) == 0 return
     ↪  GRHomSpace(σ,τ,GroupRepresentationMorphism[],VectorSpaces(F)) end

    gap_F = GAP.Globals.FiniteField(Int(characteristic(F)), degree(F))
    generators = order(grp) == 1 ? elements(grp) : gens(grp)

    #Build the modules from σ and τ
    mats_σ = GAP.GapObj([GAP.julia_to_gap(σ(g)) for g ∈ generators])
    mats_τ = GAP.GapObj([GAP.julia_to_gap(τ(g)) for g ∈ generators])
    Mσ = GAP.Globals.GModuleByMats(mats_σ, gap_F)
    Mτ = GAP.Globals.GModuleByMats(mats_τ, gap_F)

    # Use GAPs Meat Axe to calculate a basis
    gap_homs = GAP.Globals.MTX.BasisModuleHomomorphisms(Mσ,Mτ)

    dims_m,dims_n = dim(σ), dim(τ)
    mat_homs = [matrix(F,[F(m[i,j]) for i ∈ 1:dims_m, j ∈ 1:dims_n]) for m ∈ gap_homs]

    rep_homs = [Morphism(σ,τ,m,check = false) for m ∈ mat_homs]

    return GRHomSpace(σ,τ, rep_homs, VectorSpaces(F))
end

function zero(H::GRHomSpace)
    dom = H.X
```

```
    codom = H.Y
    m = zero(MatrixSpace(base_ring(dom),dim(dom),dim(codom)))
    return Morphism(dom,codom,m)
end

function zero_morphism(X::GroupRepresentation, Y::GroupRepresentation)
    m = zero(MatrixSpace(base_ring(X),dim(X),dim(Y)))
    return Morphism(X,Y,m)
end

#-------------------------------------------------------------------------
#   Restriction and Induction Functor
#-------------------------------------------------------------------------

function restriction(ρ::GroupRepresentation, H::GAPGroup)
    b,f = issubgroup(ρ.group, H)
    RepH = RepresentationCategory(H,base_ring(ρ))
    if b == false throw(ErrorException("Not a subgroup")) end
    if ρ.m == 0 return zero(RepH) end
    h = hom(H,codomain(ρ.m), gens(H), [ρ(f(g)) for g ∈ gens(H)])
    return GroupRepresentation(RepH, H, h, base_ring(ρ), dim(ρ))
end

function restriction(f::GroupRepresentationMorphism, H::GAPGroup)
    if domain(f).group == H return f end
    return Morphism(restriction(domain(f),H), restriction(codomain(f),H), matrix(f))
end

function induction(ρ::GroupRepresentation, G::GAPGroup)
    H = ρ.group

    if H == G return ρ end

    if !issubgroup(G, H)[1] throw(ErrorException("Not a supergroup")) end

    if ρ.m == 0 return zero(RepresentationCategory(G,base_ring(ρ))) end

    transversal = left_transversal(G,H)

    g = order(G) == 1 ? elements(G) : gens(G)

    ji = [[findfirst(x -> g[k]*t ∈ orbit(gset(H, (y,g) -> y*g, G), x), transversal) for
     ↪  t ∈ transversal] for k ∈ 1:length(g)]
    g_ji = [[transversal[i] for i ∈ m] for m ∈ ji]

    hi = [[inv(g_ji[k][i])*g[k]*transversal[i] for i ∈ 1:length(transversal)] for k ∈
     ↪  1:length(g)]

    images = []
    d = dim(ρ)
    n = length(transversal)*d
    for i ∈ 1:length(g)
```

```
        m = zero(MatrixSpace(base_ring(ρ), n, n))

        for j ∈ 1:length(transversal)
            m[ (ji[i][j]-1)*d+1:ji[i][j]*d, (j-1)*d+1:j*d] = matrix(ρ(hi[i][j]))
        end
        images = [images; m]
    end
    return Representation(G, g, images)
end

function induction(f::GroupRepresentationMorphism, G::GAPGroup)
    dom = induction(domain(f), G)
    codom = induction(codomain(f), G)
    return Morphism(dom,codom, dsum([Morphism(matrix(f)) for i ∈
    ↪  1:Int64(index(G,domain(f).group))]).m)
end

#-----------------------------------------------------------------------
#   Pretty Printing
#-----------------------------------------------------------------------


function show(io::IO, Rep::GroupRepresentationCategory)
    print(io, """Representation Category of $(Rep.group) over $(Rep.base_ring)""")
end

function show(io::IO, ρ::GroupRepresentation)
    print(io,"$(dim(ρ))-dimensional group representation over $(base_ring(ρ)) of
    ↪  $(ρ.group))")
end

function show(io::IO, f::GroupRepresentationMorphism)
    println(io, "Group representation Morphism with defining matrix")
    print(io,f.map)
end

#-----------------------------------------------------------------------
#   Utility
#-----------------------------------------------------------------------


function to_gap_module(σ::GroupRepresentation,F::Field)
    grp = σ.group
    gap_F = GAP.Globals.FiniteField(Int(characteristic(F)), degree(F))
    mats_σ = GAP.GapObj([GAP.julia_to_gap(σ(g)) for g ∈ gens(grp)])
    Mσ = GAP.Globals.GModuleByMats(mats_σ, gap_F)
end

function express_in_basis(f::GroupRepresentationMorphism,
↪  basis::Vector{GroupRepresentationMorphism})
    o = one(base_group(domain(f)))
```

```
    express_in_basis(Morphism(o => Morphism(f.map)), [Morphism(o => Morphism(g.map)) for
     ↪  g in basis])
end
```

A.4. Coherent Sheaves and Convolution

```
struct CohSheaves <: Category
    group::GAPGroup
    base_ring::Field
    GSet::GSet
    orbit_reps
    orbit_stabilizers
end

struct CohSheaf <: Object
    parent::CohSheaves
    stalks::Vector{GroupRepresentation}
end

struct CohSheafMorphism <: Morphism
    domain::CohSheaf
    codomain::CohSheaf
    m::Vector{GroupRepresentationMorphism}
end

ismultitensor(::CohSheaves) = true
ismultifusion(C::CohSheaves) = mod(order(C.group),characteristic(base_ring(C))) != 0

#-------------------------------------------------------------------
#   Constructors
#-------------------------------------------------------------------
"""
    CohSheaves(X::GSet,F::Field)

The category of ``G``-equivariant coherent sheafes on ``X``.
"""
function CohSheaves(X::GSet, F::Field)
    G = X.group
    orbit_reps = [O.seeds[1] for O ∈ orbits(X)]
    orbit_stabilizers = [stabilizer(G,x,X.action_function)[1] for x ∈ orbit_reps]
    return CohSheaves(G, F, X, orbit_reps, orbit_stabilizers)
end

"""
    CohSheaves(X, F::Field)

The category of coherent sheafes on ``X``.
"""
function CohSheaves(X,F::Field)
    G = symmetric_group(1)
```

```
    return CohSheaves(gset(G,X), F)
end

Morphism(X::CohSheaf, Y::CohSheaf, m::Vector) = CohSheafMorphism(X,Y,m)


#----------------------------------------------------------------
#   Functionality
#----------------------------------------------------------------
"""
    issemisimple(C::CohSheaves)

Return whether ``C``is semisimple.
"""
issemisimple(C::CohSheaves) = gcd(order(C.group), characteristic(base_ring(C))) == 1

"""
    stalks(X::CohSheaf)

Return the stalks of ``X``.
"""
stalks(X::CohSheaf) = X.stalks

orbit_stabilizers(Coh::CohSheaves) = Coh.orbit_stabilizers

function orbit_index(X::CohSheaf, y)
    i = findfirst(x -> y ∈ x, orbits(parent(X).GSet))
end

function orbit_index(C::CohSheaves, y)
    i = findfirst(x -> y ∈ x, orbits(C.GSet))
end

function stalk(X::CohSheaf,y)
    return stalks(X)[orbit_index(X,y)]
end

"""
    zero(C::CohSheaves)

Return the zero sheaf on the ``G``-set.
"""
zero(C::CohSheaves) = CohSheaf(C,[zero(RepresentationCategory(H,base_ring(C))) for H ∈
 ↪  C.orbit_stabilizers])

"""
    zero_morphism(X::CohSheaf, Y::CohSheaf)

Return the zero morphism ```0:X → Y```.
"""
zero_morphism(X::CohSheaf, Y::CohSheaf) = CohSheafMorphism(X,Y,[zero(Hom(x,y)) for (x,y)
 ↪  ∈ zip(stalks(X),stalks(Y))])
```

```
function ==(X::CohSheaf, Y::CohSheaf)
    if parent(X) != parent(Y) return false end
    for (s,r) ∈ zip(stalks(X),stalks(Y))
        if s != r return false end
    end
    return true
end

"""
    isisomorphic(X::CohSheaf, Y::CohSheaf)

Check whether ``X``and ``Y`` are isomorphic and the isomorphism if possible.
"""
function isisomorphic(X::CohSheaf, Y::CohSheaf)
    m = GroupRepresentationMorphism[]
    for (s,r) ∈ zip(stalks(X),stalks(Y))
        b, iso = isisomorphic(s,r)
        if !b return false, nothing end
        m = [m; iso]
    end
    return true, CohSheafMorphism(X,Y,m)
end

==(f::CohSheafMorphism, g::CohSheafMorphism) = f.m == f.m

"""
    id(X::CohSheaf)

Return the identity on ```X```.
"""
id(X::CohSheaf) = CohSheafMorphism(X,X,[id(s) for s ∈ stalks(X)])

"""
    associator(X::CohSheaf, Y::CohSheaf, Z::CohSheaf)

Return the associator isomorphism ```(X⊗Y)⊗Z → X⊗(Y⊗Z)```.
"""
associator(X::CohSheaf, Y::CohSheaf, Z::CohSheaf) = id(X⊗Y⊗Z)

"""
    dual(X::CohSheaf)

Return the dual object of ```X```.
"""
dual(X::CohSheaf) = CohSheaf(parent(X),[dual(s) for s ∈ stalks(X)])

"""
    ev(X::CohSheaf)

Return the evaluation morphism ```X∗⊗X → 1```.
"""
```

```
function ev(X::CohSheaf)
    dom = dual(X)⊗X
    cod = one(parent(X))
    return CohSheafMorphism(dom,cod, [ev(s) for s ∈ stalks(X)])
end

"""
    coev(X::CohSheaf)

Return the coevaluation morphism ```1 → X⊗X*```.
"""
function coev(X::CohSheaf)
    dom = one(parent(X))
    cod = X ⊗ dual(X)
    return CohSheafMorphism(dom,cod, [coev(s) for s ∈ stalks(X)])
end

"""
    spherical(X::CohSheaf)

Return the spherical structure isomorphism ```X → X**```.
"""
spherical(X::CohSheaf) = Morphism(X,X,[spherical(s) for s ∈ stalks(X)])

"""
    braiding(X::cohSheaf, Y::CohSheaf)

Return the braiding isomoephism ```X⊗Y → Y⊗X```.
"""
braiding(X::CohSheaf, Y::CohSheaf) = Morphism(X⊗Y, Y⊗X, [braiding(x,y) for (x,y) ∈
↪   zip(stalks(X),stalks(Y))])

#-----------------------------------------------------------------
#   Functionality: Direct Sum
#-----------------------------------------------------------------

"""
    dsum(X::CohSheaf, Y::CohSheaf, morphisms::Bool = false)

Return the direct sum of sheaves. Return also the inclusion and projection if
morphisms = true.
"""
function dsum(X::CohSheaf, Y::CohSheaf, morphisms::Bool = false)
    sums = [dsum(x,y,true) for (x,y) ∈ zip(stalks(X), stalks(Y))]
    Z = CohSheaf(parent(X), [s[1] for s ∈ sums])

    if !morphisms return Z end

    ix = [CohSheafMorphism(x,Z,[s[2][i] for s ∈ sums]) for (x,i) ∈ zip([X,Y],1:2)]
    px = [CohSheafMorphism(Z,x,[s[3][i] for s ∈ sums]) for (x,i) ∈ zip([X,Y],1:2)]
    return Z,ix,px
end
```

```
"""
    dsum(f::CohSheafMorphism, g::CohSheafMorphism)

Return the direct sum of morphisms of sheaves.
"""
function dsum(f::CohSheafMorphism, g::CohSheafMorphism)
    dom = dsum(domain(f), domain(g))
    codom = dsum(codomain(f), codomain(g))
    mors = [dsum(m,n) for (m,n) ∈ zip(f.m,g.m)]
    return CohSheafMorphism(dom,codom, mors)
end

product(X::CohSheaf,Y::CohSheaf,projections = false) = projections ?
 ↪  dsum(X,Y,projections)[[1,3]] : dsum(X,Y)
coproduct(X::CohSheaf,Y::CohSheaf,projections = false) = projections ?
 ↪  dsum(X,Y,projections)[[1,2]] : dsum(X,Y)

#-------------------------------------------------------------------
#   Functionality: (Co)Kernel
#-------------------------------------------------------------------

"""
    kernel(f::CohSheafMorphism)

Return a tuple ```(K,k)``` where ```K``` is the kernel object and ```k``` is the
 ↪  inclusion.
"""
function kernel(f::CohSheafMorphism)
    kernels = [kernel(g) for g ∈ f.m]
    K = CohSheaf(parent(domain(f)), [k for (k,_) ∈ kernels])
    return K, Morphism(K, domain(f), [m for (_,m) ∈ kernels])
end

"""
    cokernel(f::CohSheafMorphism)

Return a tuple ```(C,c)``` where ```C``` is the kernel object and ```c``` is the
 ↪  projection.
"""
function cokernel(f::CohSheafMorphism)
    cokernels = [cokernel(g) for g ∈ f.m]
    C = CohSheaf(parent(domain(f)), [c for (c,_) ∈ cokernels])
    return C, Morphism(codomain(f), C, [m for (_,m) ∈ cokernels])
end

#-------------------------------------------------------------------
#   Functionality: Tensor Product
#-------------------------------------------------------------------

"""
    tensor_product(X::CohSheaf, Y::CohSheaf)
```

```
Return the tensor product of equivariant coherent sheaves.
"""
function tensor_product(X::CohSheaf, Y::CohSheaf)
    @assert parent(X) == parent(Y) "Mismatching parents"
    return CohSheaf(parent(X), [x⊗y for (x,y) ∈ zip(stalks(X), stalks(Y))])
end

"""
    tensor_product(f::CohSheafMorphism, g::CohSheafMorphism)

Return the tensor product of morphisms of equivariant coherent sheaves.
"""
function tensor_product(f::CohSheafMorphism, g::CohSheafMorphism)
    dom = tensor_product(domain(f), domain(g))
    codom = tensor_product(codomain(f), codomain(g))

    mors = [tensor_product(m,n) for (m,n) ∈ zip(f.m,g.m)]
    return CohSheafMorphism(dom,codom,mors)
end

"""
    one(C::CohSheaves)

Return the one object in ``C``.
"""
function one(C::CohSheaves)
    return CohSheaf(C,[one(RepresentationCategory(H,base_ring(C))) for H ∈
     ↪  C.orbit_stabilizers])
end

#-------------------------------------------------------------------
#   Functionality: Morphisms
#-------------------------------------------------------------------

"""
    compose(f::CohSheafMorphism, g::CohSheafMorphism)

Return the composition ```g∘f```.
"""
function compose(f::CohSheafMorphism, g::CohSheafMorphism)
    dom = domain(f)
    codom = codomain(f)
    mors = [compose(m,n) for (m,n) ∈ zip(f.m,g.m)]
    return CohSheafMorphism(dom,codom,mors)
end

function +(f::CohSheafMorphism, g::CohSheafMorphism)
    #@assert domain(f) == domain(g) && codomain(f) == codomain(g) "Not compatible"
    return Morphism(domain(f), codomain(f), [fm + gm for (fm,gm) ∈ zip(f.m,g.m)])
end
```

```
function *(x,f::CohSheafMorphism)
    Morphism(domain(f),codomain(f),x .* f.m)
end

matrices(f::CohSheafMorphism) = matrix.(f.m)

"""
    inv(f::CohSheafMorphism)

Retrn the inverse morphism of ```f```.
"""
function inv(f::CohSheafMorphism)
    return Morphism(codomain(f), domain(f), [inv(g) for g in f.m])
end

#-------------------------------------------------------------------
#   Simple Objects
#-------------------------------------------------------------------
"""
    simples(C::CohSheaves)

Return the simple objects of ``C``.
"""
function simples(C::CohSheaves)

    simple_objects = CohSheaf[]

    #zero modules
    zero_mods = [zero(RepresentationCategory(H,C.base_ring)) for H ∈
     ↪  C.orbit_stabilizers]

    for k ∈ 1:length(C.orbit_stabilizers)

        #Get simple objects from the corresponding representation categories
        RepH = RepresentationCategory(C.orbit_stabilizers[k], C.base_ring)

        RepH_simples = simples(RepH)
        for i ∈ 1:length(RepH_simples)
            Hsimple_sheaves = [CohSheaf(C,[k == j ? RepH_simples[i] : zero_mods[j] for j
             ↪  ∈ 1:length(C.orbit_stabilizers)])]
            simple_objects = [simple_objects; Hsimple_sheaves]
        end
    end
    return simple_objects
end

"""
    decompose(X::CohSheaf)

Decompose ``X`` into a direct sum of simple objects with multiplicity.
"""
function decompose(X::CohSheaf)
```

```
    ret = []
    C = parent(X)
    zero_mods = [zero(RepresentationCategory(H,C.base_ring)) for H ∈
     ↪  C.orbit_stabilizers]

    for k ∈ 1:length(C.orbit_reps)
        X_H_facs = decompose(stalks(X)[k])

        ret = [ret; [(CohSheaf(C,[k == j ? Y : zero_mods[j] for j ∈
         ↪  1:length(C.orbit_reps)]),d) for (Y,d) ∈ X_H_facs]]
    end
    return ret
end
#-------------------------------------------------------------------
#   Hom Spaces
#-------------------------------------------------------------------

struct CohSfHomSpace <: HomSpace
    X::CohSheaf
    Y::CohSheaf
    basis::Vector{CohSheafMorphism}
    parent::VectorSpaces
end

"""
    Hom(X::CohSheaf, Y::CohSheaf)

Return Hom(``X,Y``) as a vector space.
"""
function Hom(X::CohSheaf, Y::CohSheaf)
    @assert parent(X) == parent(Y) "Missmatching parents"

    b = CohSheafMorphism[]
    H = [Hom(stalks(X)[i],stalks(Y)[i]) for i ∈ 1:length(stalks(X))]
    for i ∈ 1:length(stalks(X))
        for ρ ∈ basis(H[i])
            reps = [zero(H[j]) for j ∈ 1:length(stalks(X))]
            reps[i] = ρ
            b = [b; CohSheafMorphism(X,Y,reps)]
        end
    end
    return CohSfHomSpace(X,Y,b,VectorSpaces(base_ring(X)))
end

zero(H::CohSfHomSpace) = zero_morphism(H.X,H.Y)


#-------------------------------------------------------------------
#   Pretty Printing
#-------------------------------------------------------------------

function show(io::IO, C::CohSheaves)
```

```
    print(io, "Category of equivariant coherent sheaves on $(C.GSet.seeds) over
     ↪  $(C.base_ring)")
end

function show(io::IO, X::CohSheaf)
    print(io, "Equivariant choherent sheaf on $(X.parent.GSet.seeds) over
     ↪  $(base_ring(X))")
end

function show(io::IO, X::CohSheafMorphism)
    print(io, "Morphism of equivariant coherent sheaves on
     ↪  $(domain(X).parent.GSet.seeds) over $(base_ring(X))")
end


#-----------------------------------------------------------------
#   Functors
#-----------------------------------------------------------------

struct PullbackFunctor <: Functor
    domain::Category
    codomain::Category
    obj_map
    mor_map
end

pullb_obj_map(CY,CX,X,f) = CohSheaf(CX, [restriction(stalk(X,f(x)), H) for (x,H) ∈
 ↪  zip(CX.orbit_reps, CX.orbit_stabilizers)])

function pullb_mor_map(CY,CX,m,f)
    dom = pullb_obj_map(CY,CX,domain(m),f)
    codom = pullb_obj_map(CY,CX,codomain(m),f)
    maps = [restriction(m.m[orbit_index(CY,f(x))], H) for (x,H) ∈ zip(CX.orbit_reps,
     ↪  CX.orbit_stabilizers)]
    CohSheafMorphism(dom, codom, maps)
end

"""
    Pullback(C::CohSheaves, D::CohSheaves, f::Function)

Return the pullback functor ```C → D``` defined by the ```G```-set map ```f::X → Y```.
"""
function Pullback(CY::CohSheaves, CX::CohSheaves, f::Function)
    @assert isequivariant(CX.GSet, CY.GSet, f) "Map not equivariant"

    obj_map = X -> pullb_obj_map(CY,CX,X,f)
    mor_map = m -> pullb_mor_map(CY,CX,m,f)

    return PullbackFunctor(CY, CX, obj_map, mor_map)
end

struct PushforwardFunctor <: Functor
```

```
    domain::Category
    codomain::Category
    obj_map
    mor_map
end

function pushf_obj_map(CX,CY,X,f)
    stlks = [zero(RepresentationCategory(H,base_ring(CY))) for H ∈ CY.orbit_stabilizers]
    for i ∈ 1:length(CY.orbit_reps)
        y = CY.orbit_reps[i]
        Gy = CY.orbit_stabilizers[i]

        if length([x for x ∈ CX.GSet.seeds if f(x) == y]) == 0 continue end
        fiber = gset(Gy, CX.GSet.action_function, [x for x ∈ CX.GSet.seeds if f(x) ==
         ↪  y])

        orbit_reps = [O.seeds[1] for O ∈ orbits(fiber)]

        for j ∈ 1:length(orbit_reps)
            stlks[i] = dsum(stlks[i], induction(stalk(X,orbit_reps[j]),
             ↪  CY.orbit_stabilizers[i]))
        end
    end
    return CohSheaf(CY, stlks)
end

function pushf_mor_map(CX,CY,m,f)
    mor = GroupRepresentationMorphism[]

    for i ∈ 1:length(CY.orbit_reps)
        y = CY.orbit_reps[i]
        Gy = CY.orbit_stabilizers[i]

        if length([x for x ∈ CX.GSet.seeds if f(x) == y]) == 0 continue end
        fiber = gset(Gy, CX.GSet.action_function, [x for x ∈ CX.GSet.seeds if f(x) ==
         ↪  y])

        orbit_reps = [O.seeds[1] for O ∈ orbits(fiber)]

        mor = [mor; dsum([induction(m.m[orbit_index(CX,y)], Gy) for y ∈ orbit_reps]...)]
    end
    return CohSheafMorphism(pushf_obj_map(CX,CY,domain(m),f),
     ↪  pushf_obj_map(CX,CY,codomain(m),f), mor)
end

"""
    Pushforward(C::CohSheaves, D::CohSheaves, f::Function)

Return the push forward functor ```C → D``` defined by the ```G```-set map ```f::X →
 ↪  Y```.
"""
function Pushforward(CX::CohSheaves, CY::CohSheaves, f::Function)
```

```
    @assert isequivariant(CX.GSet, CY.GSet, f) "Map not equivariant"

    return PushforwardFunctor(CX,CY,X -> pushf_obj_map(CX,CY,X,f),m ->
     ↪  pushf_mor_map(CX,CY,m,f))
end

#dummy
function isequivariant(X::GSet, Y::GSet, f::Function)
    true
end

function show(io::IO,F::PushforwardFunctor)
    print(io, "Pushforward functor from $(domain(F)) to $(codomain(F))")
end

function show(io::IO,F::PullbackFunctor)
    print(io, "Pullback functor from $(domain(F)) to $(codomain(F))")
end

struct ConvolutionCategory <: Category
    group::GAPGroup
    base_ring::Field
    GSet::GSet
    squaredGSet::GSet
    cubedGSet::GSet
    squaredCoh::CohSheaves
    cubedCoh::CohSheaves
    projectors::Vector
end

struct ConvolutionObject <: Object
    sheaf::CohSheaf
    parent::ConvolutionCategory
end

struct ConvolutionMorphism <: Morphism
    domain::ConvolutionObject
    codomain::ConvolutionObject
    m::CohSheafMorphism
end
istensor(::ConvolutionCategory) = true
isfusion(C::ConvolutionCategory) = mod(order(C.group),characteristic(base_ring(C))) != 0
"""
    ConvolutionCategory(X::GSet, K::Field)

Return the category of equivariant coherent sheaves on ``X`` with convolution product.
"""
function ConvolutionCategory(X::GSet, K::Field)
    G = X.group
    sqX = gset(G,(x,g) -> Tuple(X.action_function(xi,g) for xi ∈ x), [(x,y) for x ∈
     ↪  X.seeds, y ∈ X.seeds][:])
```

```
    cuX = gset(G,(x,g) -> Tuple(X.action_function(xi,g) for xi ∈ x), [(x,y,z) for x ∈
    ↪  X.seeds, y ∈ X.seeds, z ∈ X.seeds][:])
    sqCoh = CohSheaves(sqX,K)
    cuCoh = CohSheaves(cuX,K)

    p12 = x -> (x[1],x[2])
    p13 = x -> (x[1],x[3])
    p23 = x -> (x[2],x[3])

    P12 = Pullback(sqCoh, cuCoh, p12)
    P13 = Pushforward(cuCoh, sqCoh, p13)
    P23 = Pullback(sqCoh, cuCoh, p23)
    return ConvolutionCategory(G,K,X,sqX,cuX,sqCoh,cuCoh, [P12, P13, P23])
end

"""
    ConvolutionCategory(X, K::Field)

Return the category of coherent sheaves on ``X`` with convolution product.
"""
function ConvolutionCategory(X, K::Field)
    G = symmetric_group(1)
    return ConvolutionCategory(gset(G,X), K)
end

Morphism(D::ConvolutionObject, C::ConvolutionObject, m:: CohSheafMorphism) =
↪  ConvolutionMorphism(D,C,m)
#-------------------------------------------------------------------
#   Functionality
#-------------------------------------------------------------------

"""
    issemisimple(C::ConvolutionCategory)

Check whether ``C`` semisimple.
"""
issemisimple(C::ConvolutionCategory) = gcd(order(C.group), characteristic(base_ring(C)))
↪  == 1

"""
    orbit_stabilizers(C::ConvolutionCategory)

Return the stabilizers of representatives of the orbits.
"""
orbit_stabilizers(C::ConvolutionCategory) = C.squaredCoh.orbit_stabilizers
orbit_index(X::ConvolutionObject, y) = orbit_index(X.sheaf, y)
orbit_index(X::ConvolutionCategory, y) = orbit_index(X.squaredCoh, y)

"""
    stalks(X::ConvolutionObject)

Return the stalks of the sheaf ``X``.
```

```
"""
stalks(X::ConvolutionObject) = stalks(X.sheaf)
stalk(X::ConvolutionObject, x) = stalk(X.sheaf,x)

==(X::ConvolutionObject,Y::ConvolutionObject) = X.sheaf == Y.sheaf

==(f::ConvolutionMorphism, g::ConvolutionMorphism) = f.m == f.m

"""
    isisomorphic(X::ConvolutionObject, Y::ConvolutionObject)

Check whether ``X``and ``Y``are isomorphic. Return the isomorphism if true.
"""
function isisomorphic(X::ConvolutionObject, Y::ConvolutionObject)
    b, iso = isisomorphic(X.sheaf, Y.sheaf)
    if !b return false, nothing end
    return true, ConvolutionMorphism(X,Y,iso)
end

id(X::ConvolutionObject) = ConvolutionMorphism(X,X,id(X.sheaf))

function associator(X::ConvolutionObject, Y::ConvolutionObject, Z::ConvolutionObject)
    dom = (X⊗Y)⊗Z
    cod = X⊗(Y⊗Z)
    return inv(decompose_morphism(cod))∘decompose_morphism(dom)
end
#-------------------------------------------------------------------
#   Functionality: Direct Sum
#-------------------------------------------------------------------

"""
    dsum(X::ConvolutionObject, Y::ConvolutionObject, morphisms::Bool = false)

documentation
"""
function dsum(X::ConvolutionObject, Y::ConvolutionObject, morphisms::Bool = false)
    @assert parent(X) == parent(Y) "Mismatching parents"
    Z,ix,px = dsum(X.sheaf,Y.sheaf,true)
    Z = ConvolutionObject(Z,parent(X))

    if !morphisms return Z end

    ix = [ConvolutionMorphism(x,Z,i) for (x,i) ∈ zip([X,Y],ix)]
    px = [ConvolutionMorphism(Z,x,p) for (x,p) ∈ zip([X,Y],px)]
    return Z, ix, px
end

"""
    dsum(f::ConvolutionMorphism, g::ConvolutionMorphism)

Return the direct sum of morphisms of coherent sheaves (with convolution product).
"""
```

```
function dsum(f::ConvolutionMorphism, g::ConvolutionMorphism)
    dom = dsum(domain(f), domain(g))
    codom = dsum(codomain(f), codomain(g))
    m = dsum(f.m,g.m)
    return ConvolutionMorphism(dom,codom,m)
end

product(X::ConvolutionObject,Y::ConvolutionObject,projections::Bool = false) =
 ↪  projections ? dsum(X,Y,projections)[[1,3]] : dsum(X,Y)
coproduct(X::ConvolutionObject,Y::ConvolutionObject,projections::Bool = false) =
 ↪  projections ? dsum(X,Y,projections)[[1,2]] : dsum(X,Y)

"""
    zero(C::ConvolutionCategory)

Return the zero object in Conv(``X``).
"""
zero(C::ConvolutionCategory) = ConvolutionObject(zero(C.squaredCoh),C)


#-----------------------------------------------------------------
#   Functionality: (Co)Kernel
#-----------------------------------------------------------------

function kernel(f::ConvolutionMorphism)
    K,k = kernel(f.m)
    return ConvolutionObject(K,parent(domain(f))), Morphism(K, domain(f), k)
end

function cokernel(f::ConvolutionMorphism)
    C,c = cokernel(f.m)
    return ConvolutionObject(C, parent(domain(f))), Morphism(codomain(f), C, c)
end

#-----------------------------------------------------------------
#   Functionality: Tensor Product
#-----------------------------------------------------------------

"""
    tensor_product(X::ConvolutionObject, Y::ConvolutionObject)

Return the convolution product of coherent sheaves.
"""
function tensor_product(X::ConvolutionObject, Y::ConvolutionObject)
    @assert parent(X) == parent(Y) "Mismatching parents"
    p12,p13,p23 = parent(X).projectors

    return ConvolutionObject(p13(p12(X.sheaf)⊗p23(Y.sheaf)),parent(X))
end

"""
    tensor_product(f::ConvolutionMorphism, g::ConvolutionMorphism)
```

```
Return the convolution product of morphisms of coherent sheaves.
"""
function tensor_product(f::ConvolutionMorphism, g::ConvolutionMorphism)
    dom = domain(f)⊗domain(g)
    codom = codomain(f)⊗codomain(g)

    p12,p13,p23 = parent(domain(f)).projectors
    return ConvolutionMorphism(dom,codom, p13(p12(f.m)⊗p23(g.m)))
end

"""
    one(C::ConvolutionCategory)

Return the one object in Conv(``X``).
"""
function one(C::ConvolutionCategory)
    F = base_ring(C)

    stlks = [zero(RepresentationCategory(H,F)) for H ∈ orbit_stabilizers(C)]
    diag = [(x,x) for x ∈ C.GSet.seeds]

    for i ∈ [orbit_index(C,d) for d ∈ diag]
        stlks[i] = one(RepresentationCategory(orbit_stabilizers(C)[i], F))
    end
    return ConvolutionObject(CohSheaf(C.squaredCoh, stlks), C)
end

function dual(X::ConvolutionObject)
    orbit_reps = parent(X).squaredCoh.orbit_reps
    GSet = parent(X).squaredCoh.GSet
    perm = [findfirst(e -> e ∈ orbit(GSet, (y,x)), orbit_reps) for (x,y) ∈ orbit_reps]
    reps = [dual(ρ) for ρ ∈ stalks(X)][perm]
    return ConvolutionObject(CohSheaf(parent(X.sheaf), reps), parent(X))
end
#-------------------------------------------------------------------
#   Functionality: Morphisms
#-------------------------------------------------------------------

function compose(f::ConvolutionMorphism,g::ConvolutionMorphism)
    return ConvolutionMorphism(domain(f),codomain(g),compose(f.m,g.m))
end

function zero_morphism(X::ConvolutionObject, Y::ConvolutionObject)
    return ConvolutionMorphism(X,Y,zero_morphism(X.sheaf,Y.sheaf))
end

function +(f::ConvolutionMorphism, g::ConvolutionMorphism)
    Morphism(domain(f),codomain(f), f.m + g.m)
end

function *(x, f::ConvolutionMorphism)
```

```
    Morphism(domain(f), codomain(f), x * f.m)
end

function matrices(f::ConvolutionMorphism)
    matrices(f.m)
end

function inv(f::ConvolutionMorphism)
    return Morphism(codomain(f), domain(f), inv(f.m))
end
#-------------------------------------------------------------------
#   Simple Objects
#-------------------------------------------------------------------

"""
    simples(C::ConvolutionCategory)

Return a list of simple objects in Conv(``X``).
"""
function simples(C::ConvolutionCategory)
    return [ConvolutionObject(sh,C) for sh ∈ simples(C.squaredCoh)]
end

"""
    decompose(X::ConvolutionObject)

Decompose ``X`` into a direct sum of simple objects with multiplicities.
"""
function decompose(X::ConvolutionObject)
    facs = decompose(X.sheaf)
    return [(ConvolutionObject(sh,parent(X)),d) for (sh,d) ∈ facs]
end

#-------------------------------------------------------------------
#   Hom Space
#-------------------------------------------------------------------

struct ConvHomSpace <: HomSpace
    X::ConvolutionObject
    Y::ConvolutionObject
    basis::Vector{ConvolutionMorphism}
    parent::VectorSpaces
end

"""
    Hom(X::ConvolutionObject, Y::ConvolutionObject)

Return Hom(``X,Y``) as a vector space.
"""
function Hom(X::ConvolutionObject, Y::ConvolutionObject)
    @assert parent(X) == parent(Y) "Missmatching parents"
    b = basis(Hom(X.sheaf,Y.sheaf))
```

```
    conv_b = [ConvolutionMorphism(X,Y,m) for m ∈ b]

    return ConvHomSpace(X,Y,conv_b, VectorSpaces(base_ring(X)))
end

zero(H::ConvHomSpace) = zero_morphism(H.X,H.Y)


#-----------------------------------------------------------------
#   pretty printing
#-----------------------------------------------------------------

function show(io::IO, C::ConvolutionCategory)
    print(io, """Convolution category over G-set with $(length(C.GSet)) elements.""")
end

function show(io::IO, X::ConvolutionObject)
    print(io, "Object in $(parent(X))")
end

function show(io::IO, f::ConvolutionMorphism)
    print(io, "Morphism in $(parent(domain(f)))")
end
```

A.5. Fusion Categories

```
mutable struct RingCategory <: Category
    base_ring::Field
    simples::Int64
    simples_names::Vector{String}
    ass::Array{<:MatElem,4}
    braiding::Function
    tensor_product::Array{Int,3}
    spherical::Vector
    twist::Vector

    function RingCategory(F::Field, mult::Array{Int,3}, names::Vector{String} = ["X$i"
     ↪  for i ∈ 1:length(mult[1])])
        C = New(F, length(mult[1]), names)
        C.tensor_product = mult
        #C.ass = [id(⊗(X,Y,Z)) for X ∈ simples(C), Y ∈ simples(C), Z ∈ simples(C)]
        #C.dims = [1 for i ∈ 1:length(names)]
        return C
    end

    function RingCategory(F::Field, names::Vector{String})
        C = new(F,length(names), names)
        #C.dims = [1 for i ∈ 1:length(names)]
        return C
```

```
    end

end


struct RingObject <: Object
    parent::RingCategory
    components::Vector{Int}
end

struct RingMorphism <: Morphism
    domain::RingObject
    codomain::RingObject
    m::Vector{<:MatElem}
end


#-------------------------------------------------------------------------------
#   Constructors
#-------------------------------------------------------------------------------

RingCategory(x...) = RingCategory(x...)

Morphism(X::RingObject, Y::RingObject, m::Vector) = RingMorphism(X,Y,m)

#-------------------------------------------------------------------------------
#   Setters/Getters
#-------------------------------------------------------------------------------

function set_tensor_product!(F::RingCategory, tensor::Array{Int,3})
    F.tensor_product = tensor
    n = size(tensor,1)
    F.ass = Array{MatElem,4}(undef,n,n,n,n)
    for i ∈ 1:n, j ∈ 1:n, k ∈ 1:n
        F.ass[i,j,k,:] = matrices(id(F[i]⊗F[j]⊗F[k]))
    end
end

function set_braiding!(F::RingCategory, braiding::Function)
    F.braiding = braiding
end

function set_associator!(F::RingCategory, i::Int, j::Int, k::Int,
 ↪  ass::Vector{<:MatElem})
    F.ass[i,j,k,:] = ass
end

function set_ev!(F::RingCategory, ev::Vector)
    F.evals = ev
end

function set_coev!(F::RingCategory, coev::Vector)
```

```
    F.coevals = coev
end

function set_spherical!(F::RingCategory, sp::Vector)
    F.spherical = sp
end

function set_duals!(F::RingCategory, d::Vector)
    F.duals = d
end

function set_ribbon!(F::RingCategory, r::Vector)
    F.ribbon = r
end

function set_dims!(F::RingCategory, d::Vector)
    F.dims = d
end

function set_twist!(F::RingCategory, t::Vector)
    F.twist = t
end

# function set_ev!(F::RingCategory, ev::Vector)
#     F.ev = ev
# end
#
# function set_coev!(F::RingCategory, coev::Vector)
#     F.coev = coev
# end

dim(X::RingObject) = base_ring(X)(tr(id(X)))

(::Type{Int})(x::fmpq) = Int(numerator(x))


braiding(X::RingObject, Y::RingObject) = parent(X).braiding(X,Y)

function associator(X::RingObject, Y::RingObject, Z::RingObject)
    @assert parent(X) == parent(Y) == parent(Z) "Mismatching parents"
    C = parent(X)
    F = base_ring(C)
    n = C.simples
    dom = X⊗Y⊗Z
    m = zero_morphism(zero(C),zero(C))

    table = C.tensor_product
    associator = C.ass

    # Order of summands in domain
    dom_order_temp = [(k, m1*m2*table[i,j,k],[i,j]) for k ∈ 1:n, (i,m1) ∈
     ↪  zip(1:n,X.components), (j,m2) ∈ zip(1:n,Y.components)][:]
```

```
filter!(e -> e[2] != 0, dom_order_temp)
sort!(dom_order_temp, by = e -> e[1])
dom_order = [(k,m1*m2*table[i,j,k], [id; j]) for k ∈ 1:n, (i,m1,id) ∈
↪  dom_order_temp, (j,m2) ∈ zip(1:n,Z.components)][:]
filter!(e -> e[2] != 0, dom_order)
sort!(dom_order, by = e -> e[1])

# Order of summands in codomain
cod_order_temp = [(k, m1*m2*table[i,j,k], [i,j]) for k ∈ 1:n, (i,m1) ∈
↪  zip(1:n,Y.components), (j,m2) ∈ zip(1:n,Z.components)][:]
filter!(e -> e[2] != 0, cod_order_temp)
sort!(cod_order_temp, by = e -> e[1])
cod_order = [(k,m1*m2*table[i,j,k], [i; id]) for k ∈ 1:n, (i,m2) ∈
↪  zip(1:n,X.components), (j,m1, id) ∈ cod_order_temp][:]
filter!(e -> e[2] != 0, cod_order)
sort!(cod_order)

# Associator
for i ∈ 1:n, j ∈ 1:n, k ∈ 1:n
    for i2 ∈ 1:X[i], j2 ∈ 1:Y[j], k2 ∈ 1:Z[k]
        T = C[i]⊗C[j]⊗C[k]
        m = m ⊕ Morphism(T,T,associator[i,j,k,:])
    end
end

# Order of summands in associator
ass_order_temp = [(k, m1*m2*table[i,j,k],[i,j]) for k ∈ 1:n, (i,m1) ∈
↪  zip(1:n,X.components), (j,m2) ∈ zip(1:n,Y.components)][:]
filter!(e -> e[2] != 0, ass_order_temp)
ass_order = [(k,m1*m2*table[i,j,k], [id; j]) for k ∈ 1:n, (i,m1,id) ∈
↪  ass_order_temp, (j,m2) ∈ zip(1:n,Z.components)][:]
filter!(e -> e[2] != 0, ass_order)

comp_maps = matrices(m)

# Permutation matrices
for i ∈ 1:n
    dom_i = filter(e -> e[1] == i, dom_order)
    cod_i = filter(e -> e[1] == i, cod_order)
    ass_i = filter(e -> e[1] == i, ass_order)

    c_ass = vector_permutation(dom_i,ass_i)

    dom_dims = [k for (_,k,_) ∈ dom_i]
    ass_dims = [k for (_,k,_) ∈ ass_i]
    cod_dims = [k for (_,k,_) ∈ cod_i]

    # Permutation dom -> associator
    ass_perm = zero(MatrixSpace(F,sum(dom_dims),sum(dom_dims)))
    j = 0

    for (k,d) ∈ zip(c_ass,dom_dims)
```

```
            nk = sum(ass_dims[1:k-1])

            for i ∈ 1:d
                ass_perm[j+i,nk+i] = F(1)
            end
            j = j+d
        end

        # Permutation associator -> cod
        cod_perm = zero(MatrixSpace(F,sum(dom_dims),sum(dom_dims)))

        c_cod = vector_permutation(ass_i,cod_i)
        j = 0
        for (k,d) ∈ zip(c_cod,ass_dims)
            nk = sum(cod_dims[1:k-1])

            for i ∈ 1:d
                cod_perm[j+i,nk+i] = F(1)
            end
            j = j+d
        end
        comp_maps[i] = ass_perm*comp_maps[i]*cod_perm

    end

    return Morphism(dom,dom, comp_maps)
end

function vector_permutation(A::Vector,B::Vector)
    temp = deepcopy(B)
    perm = Int[]
    for a ∈ A
        i = findall(e -> e == a, temp)
        j = filter(e -> !(e ∈ perm), i)[1]
        perm = [perm; j]
    end
    return perm
end



#-------------------------------------------------------------------------------
#   Functionality
#-------------------------------------------------------------------------------
issemisimple(::RingCategory) = true

issimple(X::RingObject) = sum(X.components) == 1

==(X::RingObject, Y::RingObject) = parent(X) == parent(Y) && X.components ==
 ↪  Y.components
```

```
==(f::RingMorphism, g::RingMorphism) = domain(f) == domain(g) && codomain(f) ==
↪   codomain(g) && f.m == g.m

decompose(X::RingObject) = [(x,k) for (x,k) ∈ zip(simples(parent(X)), X.components) if k
↪   != 0]

inv(f::RingMorphism) = RingMorphism(codomain(f),domain(f), inv.(f.m))

id(X::RingObject) = RingMorphism(X,X, [one(MatrixSpace(base_ring(X),d,d)) for d ∈
↪   X.components])

function compose(f::RingMorphism, g::RingMorphism)
    @assert codomain(f) == domain(g) "Morphisms not compatible"
    return RingMorphism(domain(f), codomain(g), [m*n for (m,n) ∈ zip(f.m,g.m)])
end

function +(f::RingMorphism, g::RingMorphism)
    @assert domain(f) == domain(g) && codomain(f) == codomain(g) "Not compatible"
    RingMorphism(domain(f), codomain(f), [m + n for (m,n) ∈ zip(f.m,g.m)])
end

"""
    dual(X::RingObject)

Return the dual object of ``X``. An error is thrown if ``X`` is not rigid.
"""
function dual(X::RingObject)
    C = parent(X)

    # Dual of simple Object
    if issimple(X)
        # Check for rigidity
        i = findfirst(e -> e == 1, X.components)
        j = findall(e -> C.tensor_product[i,e,1] >= 1, 1:C.simples)
        if length(j) != 1
            throw(ErrorException("Object not rigid."))
        end
        return RingObject(C,[i == j[1] ? 1 : 0 for i ∈ 1:C.simples])
    end

    # Build dual from simple objects
    return dsum([dual(Y)^(X.components[i]) for (Y,i) ∈ zip(simples(C), 1:C.simples)])
end

function coev(X::RingObject) where T
    DX = dual(X)
    C = parent(X)
    F = base_ring(C)

    if sum(X.components) == 0 return zero_morphism(one(C), X) end

    m = []
```

```
    for (x,k) ∈ zip(simples(C),X.components), y ∈ simples(C)

        if x == dual(y)
            c = [F(a==b) for a ∈ 1:k, b ∈ 1:k][:]
            m = [m; c]
        else
            c = [0 for _ ∈ 1:(x⊗y).components[1]]
            m = [m; c]
        end
    end

    mats = matrices(zero_morphism(one(C), X⊗DX))
    M = parent(mats[1])
    mats[1] = M(F.(m))
    return Morphism(one(C), X⊗DX, mats)
end

function ev(X::RingObject)
    DX = dual(X)
    C = parent(X)
    F = base_ring(C)

    # Simple Objects
    if issimple(X)
        # If X is simple
        e = basis(Hom(DX⊗X, one(C)))[1]
        # Scale ev
        f = (id(X)⊗e)∘associator(X,DX,X)∘(coev(X)⊗id(X))
        return inv(F(f))*e
    end

    m = elem_type(F)[]
    #Arbitrary Objects
    for (x,k) ∈ zip(simples(C),DX.components), y ∈ simples(C)
        if x == dual(y)
            c = F(ev(y)[1]).*([F(a==b) for a ∈ 1:k, b ∈ 1:k][:])
            m = [m; c]
        else
            c = [0 for _ ∈ 1:(x⊗y).components[1]]
            m = [m; c]
        end
    end

    mats = matrices(zero_morphism(X⊗DX, one(C)))
    M = parent(mats[1])
    mats[1] = M(F.(m))
    return Morphism(X⊗DX,one(C),mats)
end

function spherical(X::RingObject)
    C = parent(X)
```

```
    sp = C.spherical
    return dsum([x^k for (x,k) ∈ zip(sp, X.components)])
end


*(λ,f::RingMorphism) = RingMorphism(domain(f), codomain(f), λ .*f.m)

# function tr(f::RingMorphism)
#     sum(tr.(f.m))
# end

# function smatrix(C::RingCategory)
#     θ = C.twist
#     #[inv(θ(i))*inv(θ(j))*sum() i ∈ simples(C), j ∈ simples(C)]
# end

function getindex(f::RingMorphism, i)
    m = zero_morphism(domain(f),codomain(f)).m
    m[i] = f.m[i]
    simple = simples(parent(domain(f)))
    dom = simple[i]^domain(f).components[i]
    cod = simple[i]^codomain(f).components[i]
    return RingMorphism(dom,cod,m)
end

getindex(X::RingObject, i) = X.components[i]

function matrices(f::RingMorphism)
    f.m
end


function (F::Field)(f::RingMorphism)
    if !(domain(f) == codomain(f) && issimple(domain(f)))
        throw(ErrorException("Cannot convert Morphism to $F"))
    end
    i = findfirst(e -> e == 1, domain(f).components)
    return F(f.m[i][1,1])
end
#-------------------------------------------------------------------------------
#   Tensor Product
#-------------------------------------------------------------------------------

function tensor_product(X::RingObject, Y::RingObject)
    @assert parent(X) == parent(Y) "Mismatching parents"
    C = parent(X)
    n = C.simples
    T = [0 for i ∈ 1:n]

    Xc = X.components
    Yc = Y.components
```

```
    for (i,j) ∈ Base.product(1:n, 1:n)
        if (c = Xc[i]) != 0 && (d = Yc[j]) != 0
            coeffs = C.tensor_product[i,j,:]
            T = T .+ ((c*d) .* coeffs)
        end
    end

    return RingObject(C,T)
end

function tensor_product(f::RingMorphism, g::RingMorphism)
    dom = domain(f) ⊗ domain(g)
    cod = codomain(f) ⊗ codomain(g)
    C = parent(dom)

    h = zero_morphism(zero(C), zero(C))

    table = C.tensor_product
    simpl = simples(C)

    for i ∈ 1:C.simples, j ∈ 1:C.simples
        A = kronecker_product(f.m[i],g.m[j])
        d1,d2 = size(A)
        #if d1*d2 == 0 continue end
        for k ∈ 1:C.simples
            if table[i,j,k] > 0
                m = zero_morphism(simpl[k]^d1,simpl[k]^d2).m
                m[k] = A

                for _ ∈ 1:table[i,j,k]
                    h = h ⊕ RingMorphism(simpl[k]^d1,simpl[k]^d2, m)
                end

            end
        end
    end
    #dom_left = dom.components - domain(h).components
    #cod_left = cod.components - codomain(h).components
    return h #⊕ zero_morphism(RingObject(C,dom_left), RingObject(C,cod_left))
end


one(C::RingCategory) = simples(C)[1]

#-------------------------------------------------------------------------------
#   Direct sum
#-------------------------------------------------------------------------------

function dsum(X::RingObject, Y::RingObject)
    @assert parent(X) == parent(Y) "Mismatching parents"
    return RingObject(parent(X), X.components .+ Y.components)
end
```

```
function dsum(f::RingMorphism, g::RingMorphism)
    dom = domain(f) ⊕ domain(g)
    cod = codomain(f) ⊕ codomain(g)
    F = base_ring(dom)
    m = zero_morphism(dom,cod).m
    for i ∈ 1:parent(dom).simples
        mf,nf = size(f.m[i])
        mg,ng = size(g.m[i])
        z1 = zero(MatrixSpace(F,mf,ng))
        z2 = zero(MatrixSpace(F,mg,nf))
        m[i] = [f.m[i] z1; z2 g.m[i]]
    end
    return RingMorphism(dom,cod, m)
end


zero(C::RingCategory) = RingObject(C,[0 for i ∈ 1:C.simples])

function zero_morphism(X::RingObject, Y::RingObject)
    return RingMorphism(X,Y,[zero(MatrixSpace(base_ring(X), cX, cY)) for (cX,cY) ∈
     ↪  zip(X.components, Y.components)])
end
#-------------------------------------------------------------------------------
#   Simple Objects
#-------------------------------------------------------------------------------

function simples(C::RingCategory)
    n = C.simples
    [RingObject(C, [i == j ? 1 : 0 for j ∈ 1:n]) for i ∈ 1:n]
end

function getindex(C::RingCategory, i)
    RingObject(C,[i == j ? 1 : 0 for j ∈ 1:C.simples])
end

#-------------------------------------------------------------------------------
#   Examples
#-------------------------------------------------------------------------------

function Ising()
    Qx,x = QQ["x"]
    F,a = NumberField(x^2-2, "√2")
    C = RingCategory(F,["1", "χ", "X"])
    M = zeros(Int,3,3,3)

    M[1,1,:] = [1,0,0]
    M[1,2,:] = [0,1,0]
    M[1,3,:] = [0,0,1]
    M[2,1,:] = [0,1,0]
    M[2,2,:] = [1,0,0]
    M[2,3,:] = [0,0,1]
```

```
    M[3,1,:] = [0,0,1]
    M[3,2,:] = [0,0,1]
    M[3,3,:] = [1,1,0]

    set_tensor_product!(C,M)

    set_associator!(C,2,3,2, matrices(-id(C[3])))
    set_associator!(C,3,1,3, matrices(id(C[1])⊕(-id(C[2]))))
    set_associator!(C,3,2,3, matrices((-id(C[1]))⊕id(C[2])))
    z = zero(MatrixSpace(F,0,0))
    set_associator!(C,3,3,3, [z, z, inv(a)*matrix(F,[1 1; 1 -1])])

    set_spherical!(C, [id(s) for s ∈ simples(C)])

    a,b,c = simples(C)

    return C
end

#-------------------------------------------------------------------------------
#   Hom Spaces
#-------------------------------------------------------------------------------

struct RingCatHomSpace<: HomSpace
    X::RingObject
    Y::RingObject
    basis::Vector{RingMorphism}
    parent::VectorSpaces
end

function Hom(X::RingObject, Y::RingObject)
    @assert parent(X) == parent(Y) "Mismatching parents"
    Xi, Yi = X.components, Y.components
    F = base_ring(X)

    d = sum([x*y for (x,y) ∈ zip(Xi,Yi)])

    if d == 0 return RingCatHomSpace(X,Y,RingMorphism[], VectorSpaces(F)) end

    basis = [zero_morphism(X,Y).m for i ∈ 1:d]
    next = 1
    for k ∈ 1:parent(X).simples

        for i ∈ 1:Xi[k], j ∈ 1:Yi[k]
            basis[next][k][i,j] = 1
            next = next + 1
        end
    end
    basis_mors = [RingMorphism(X,Y,m) for m ∈ basis]
    return RingCatHomSpace(X,Y,basis_mors, VectorSpaces(F))
end
```

```
function express_in_basis(f::RingMorphism, base::Vector)
    F = base_ring(domain(f))
    A = Array{elem_type(F),2}(undef,length(base),0)
    b = []
    for g ∈ base
        y = []
        for m ∈ g.m
            y = [y; [x for x ∈ m][:]]
        end
        A = [A y]
    end
    for m ∈ f.m
        b = [b; [x for x ∈ m][:]]
    end

    return [i for  i ∈ solve_left(transpose(matrix(F,A)),
     ↪  MatrixSpace(F,1,length(b))(F.(b)))][:]
end


#-------------------------------------------------------------------------------
#   Pretty Printing
#-------------------------------------------------------------------------------

function show(io::IO, C::RingCategory)
    print(io, "Fusion Category with $(C.simples) simple objects")
end

function show(io::IO, X::RingObject)
    coeffs = X.components

    if sum(coeffs) == 0
        print(io,"0")
        return
    end

    strings = parent(X).simples_names
    non_zero_coeffs = coeffs[coeffs .> 0]
    non_zero_strings = strings[coeffs .> 0]

    disp = non_zero_coeffs[1] == 1 ? "$(non_zero_strings[1])" :
     ↪  "$(non_zero_coeffs[1])⋅$(non_zero_strings[1])"

    for (Y,d) ∈ zip(non_zero_strings[2:end], non_zero_coeffs[2:end])
        disp = d == 1 ? disp*" ⊕ $Y" : disp*" ⊕ $(d)⋅$Y"
    end
    print(io,disp)
end

function show(io::IO, f::RingMorphism)
    print(io, """Morphism with
Domain: $(domain(f))
```

```
Codomain: $(codomain(f))
Matrices: """)
print(io, join(["$(m)" for m ∈ f.m], ", "))
end

#-------------------------------------------------------------------------------
#   Utility
#-------------------------------------------------------------------------------

function ev_coev(X::Object)
    Y = dual(X)
    ev_dom = X⊗Y
    coev_cod = Y⊗X
    C = parent(X)

    F = base_ring(X)

    H_ev = Hom(ev_dom, one(C))
    H_coev = Hom(one(C), coev_cod)

    r,s = dim(H_coev), dim(H_ev)

    R,x = PolynomialRing(F,r+s)

    f, g = basis(H_coev), basis(H_ev)
    EndX = End(X)
    base = basis(EndX)
    eqs = [zero(R) for b ∈ base]

    for i ∈ 1:r, j ∈ 1:s
        eqs = eqs .+ (x[i]*x[r+j])
         ↪  .*express_in_basis((g[i]⊗id(X))∘inv(associator(X,dual(X),X))∘(id(X)⊗f[j]),base)
    end

    I = ideal(eqs .- express_in_basis(id(X),base))

    @show dim(I)
    coeffs = recover_solutions(msolve(I),F)[1]
    return sum(coeffs[r+1:end] .* g), sum(coeffs[1:r] .* f)
end

ev(X::Object) = ev_coev(X)[1]
coev(X::Object) = ev_coev(X)[2]
```

A.6. Center

```
mutable struct CenterCategory <: Category
    base_ring::Field
    category::Category
    simples::Vector{O} where O <: Object
```

```
    function CenterCategory(F::Field, C::Category)
        Z = new()
        Z.base_ring = F
        Z.category = C
        return Z
    end
end

struct CenterObject <: Object
    parent::CenterCategory
    object::Object
    γ::Vector{M} where M <: Morphism
end

struct CenterMorphism <: Morphism
    domain::CenterObject
    codomain::CenterObject
    m::Morphism
end



#-------------------------------------------------------------------------------
#   Center Constructor
#-------------------------------------------------------------------------------
"""
    Center(C::Category)

Return the Drinfeld center of ```C```.
"""
function Center(C::Category)
    @assert issemisimple(C) "Semisimplicity required"
    return CenterCategory(base_ring(C), C)
end

function Morphism(dom::CenterObject, cod::CenterObject, m::Morphism)
    return CenterMorphism(dom,cod,m)
end

"""
    half_braiding(Z::CenterObject)

Return  a vector with half braiding morphisms ```Z⊗S → S⊗Z``` for all simple
objects ```S```.
"""
half_braiding(Z::CenterObject) = Z.γ

isfusion(C::CenterCategory) = true

"""
    add_simple!(C::CenterCategory, S::CenterObject)
```

```
Add the simple object ```S``` to the vector of simple objects.
"""
function add_simple!(C::CenterCategory, S::CenterObject)
    @assert dim(End(S)) == 1 "Not simple"
    C.simples = unique_simples([simples(C); S])
end

"""
    spherical(X::CenterObject)

Return the spherical structure ```X → X**``` of ```X```.
"""
spherical(X::CenterObject) = Morphism(X,dual(dual(X)), spherical(X.object))

(F::Field)(f::CenterMorphism) = F(f.m)
#-------------------------------------------------------------------------------
#   Direct Sum & Tensor Product
#-------------------------------------------------------------------------------

"""
    dsum(X::CenterObject, Y::CenterObject)

Return the direct sum object of ```X``` and ```Y```.
"""
function dsum(X::CenterObject, Y::CenterObject)
    S = simples(parent(X.object))
    Z,(ix,iy),(px,py) = dsum(X.object, Y.object,true)

    γZ = [(id(S[i])⊗ix)∘(X.γ[i])∘(px⊗id(S[i])) + (id(S[i])⊗iy)∘(Y.γ[i])∘(py⊗id(S[i]))
     ↪  for i ∈ 1:length(S)]
    return CenterObject(parent(X), Z, γZ)
end

"""
    dsum(f::CenterMorphism, g::CenterMorphism)

Return the direct sum of ```f``` and ```g```.
"""
function dsum(f::CenterMorphism, g::CenterMorphism)
    dom = domain(f) ⊕ domain(g)
    cod = codomain(f) ⊕ codomain(g)
    m = f.m ⊕ g.m
    return Morphism(dom,cod, m)
end

"""
    tensor_product(X::CenterObject, Y::CenterObject)

Return the tensor product of ```X``` and ```Y```.
"""
function tensor_product(X::CenterObject, Y::CenterObject)
    Z = X.object ⊗ Y.object
```

```
    γ = Morphism[]
    a = associator
    s = simples(parent(X.object))
    x,y = X.object, Y.object
    for (S, yX, yY) ∈ zip(s, X.γ, Y.γ)
        push!(γ, a(S,x,y)∘(yX⊗id(y))∘inv(a(x,S,y))∘(id(x)⊗yY)∘a(x,y,S))
    end
    return CenterObject(parent(X), Z, γ)
end

"""
    tensor_product(f::CenterMorphism,g::CenterMorphism)

Return the tensor product of ```f``` and ```g```.
"""
function tensor_product(f::CenterMorphism,g::CenterMorphism)
    dom = domain(f)⊗domain(g)
    cod = codomain(f)⊗codomain(g)
    return Morphism(dom,cod,f.m⊗g.m)
end

"""
    zero(C::CenterCategory)

Return the zero object of ```C```.
"""
function zero(C::CenterCategory)
    Z = zero(C.category)
    CenterObject(C,Z,[zero_morphism(Z,Z) for _ ∈ simples(C.category)])
end

"""
    one(C::CenterCategory)

Return the one object of ```C```.
"""
function one(C::CenterCategory)
    Z = one(C.category)
    CenterObject(C,Z,[id(s) for s ∈ simples(C.category)])
end
#-------------------------------------------------------------------------------
#   Induction
#-------------------------------------------------------------------------------

function induction(X::Object, simples::Vector = simples(parent(X)))
    @assert issemisimple(parent(X)) "Requires semisimplicity"
    Z = dsum([dual(s)⊗X⊗s for s ∈ simples])

    function γ(W)
        r = Morphism[]
        for i ∈ simples, j ∈ simples
            b1 = basis(Hom(W⊗dual(i),j))
```

```
            b2 = basis(Hom(i,j⊗W))
            if length(b1)*length(b2) == 0 continue end
            push!(r,dim(i)*dsum([ϕ ⊗ id(X) ⊗ ψ for (ϕ,ψ) ∈ zip(b1,b2)]))
        end
        return dsum(r)
    end
    return CenterObject(CenterCategory(base_ring(X),parent(X)),Z,γ)
end



#-------------------------------------------------------------------------------
#   Is central?
#-------------------------------------------------------------------------------

"""
    iscentral(Z::Object)

Return true if ```Z``` is in the categorical center, i.e. there exists a half-braiding
 ↪ on ```Z```.
"""
function iscentral(Z::Object, simples::Vector{<:Object} = simples(parent(Z)))
    if prod([isisomorphic(Z⊗s,s⊗Z)[1] for s ∈ simples]) == 0
        return false
    end
    return dim(build_center_ideal(Z,simples)) >= 0
end



function build_center_ideal(Z::Object, simples::Vector = simples(parent(Z)))
    @assert issemisimple(parent(Z)) "Not semisimple"

    Homs = [Hom(Z⊗Xi, Xi⊗Z) for Xi ∈ simples]
    n = length(simples)
    ks = [dim(Homs[i]) for i ∈ 1:n]

    var_count = sum([dim(H) for H ∈ Homs])

    R,x = PolynomialRing(QQ, var_count, ordering = :lex)

    # For convinience: build arrays with the variables xi
    vars = []
    q = 1
    for i ∈ 1:n
        m = dim(Homs[i])
        vars = [vars; [x[q:q+m-1]]]
        q = q + m
    end

    eqs = []
```

```julia
    for k ∈ 1:n, i ∈ 1:n, j ∈ 1:n
        base = basis(Hom(Z⊗simples[k], simples[i]⊗(simples[j]⊗Z)))

        for t ∈ basis(Hom(simples[k], simples[i]⊗simples[j]))
            e = [zero(R) for i ∈ base]

            l1 = [zero(R) for i ∈ base]
            l2 = [zero(R) for i ∈ base]

            for ai ∈ 1:dim(Homs[k])
                a = basis(Homs[k])[ai]
                l1 = l1 .+ (vars[k][ai] .*
                 ↪  QQ.(express_in_basis(associator(simples[i],simples[j],Z)∘(t⊗id(Z))∘a,
                 ↪  base)))
            end
            for bi ∈ 1:dim(Homs[j]), ci ∈ 1:dim(Homs[i])
                b,c = basis(Homs[j])[bi], basis(Homs[i])[ci]
                l2 = l2 .+ ((vars[j][bi]*vars[i][ci]) .*
                 ↪  QQ.(express_in_basis((id(simples[i])⊗b)∘associator(simples[i],Z,simples[j])
                 ↪  ∘ (c⊗id(simples[j])) ∘ inv(associator(Z,simples[i],simples[j])) ∘
                 ↪  (id(Z) ⊗ t), base)))
            end
            push!(eqs, l1 .-l2)
        end
    end
    ideal_eqs = []
    for p ∈ eqs
        push!(ideal_eqs, p...)
    end

    I = ideal([f for f ∈ unique(ideal_eqs) if f != 0])

    #Require e_Z(1) = id(Z)
    one_index = findfirst(e -> isisomorphic(one(parent(Z)), e)[1], simples)
    one_c = QQ.(express_in_basis(id(Z), basis(End(Z))))
    push!(ideal_eqs, (vars[one_index] .- one_c)...)

    I = ideal([f for f ∈ unique(ideal_eqs) if f != 0])
end

function braidings_from_ideal(Z::Object, I::Ideal, simples::Vector{<:Object}, C)
    Homs = [Hom(Z⊗Xi, Xi⊗Z) for Xi ∈ simples]
    coeffs = recover_solutions(msolve(I),base_ring(Z))
    ks = [dim(H) for H ∈ Homs]
    centrals = CenterObject[]

    for c ∈ coeffs
        k = 1
        ex = Morphism[]
        c = [k for k ∈ c]
        for i ∈ 1:length(simples)
            if ks[i] == 0 continue end
```

```
            e = sum(c[k:k + ks[i] - 1] .* basis(Homs[i]))
            ex = [ex ; e]
            k = k + ks[i]
        end
        centrals = [centrals; CenterObject(C, Z, ex)]
    end
    return centrals
end

"""
    half_braidings(Z::Object)

Return all objects in the center lying over ```Z```.
"""
function half_braidings(Z::Object; simples = simples(parent(Z)), parent =
↪   Center(parent(Z)))

    I = build_center_ideal(Z,simples)

    d = dim(I)

    if d < 0 return CenterObject[] end

    if d == 0 return braidings_from_ideal(Z,I,simples, parent) end

    solutions = guess_solutions(Z,I,simples,CenterObject[],gens(base_ring(I)),d, parent)

    if length(solutions) == 0
        return CenterObject[]
    end
    unique_sols = solutions[1:1]

    for s ∈ solutions[2:end]
        if sum([dim(Hom(s,u)) for u ∈ unique_sols]) == 0
            unique_sols = [unique_sols; s]
        end
    end
    return unique_sols
end

function guess_solutions(Z::Object, I::Ideal, simples::Vector{<:Object},
↪   solutions::Vector{CenterObject}, vars, d = dim(I), C = Center(parent(Z)))
    for y in vars
        J = I + ideal([y*(y^2-1)])
        d2 = dim(J)
        if d2 == 0
            return [solutions; braidings_from_ideal(Z,J,simples,C)]
        elseif d2 < 0
            return solutions
        else
            vars_new = filter(e -> e != y, vars)
            return [solutions; guess_solutions(Z,J,simples,solutions,vars_new,d2,C)]
```

```
        end
    end
end

function center_simples(C::CenterCategory, simples = simples(C.category))
    d = dim(C.category)^2

    simples_indices = []
    c_simples = CenterObject[]
    d_max = dim(C.category)
    d_rem = d
    k = length(simples)

    coeffs = [i for i ∈ Base.product([0:d_max for i ∈ 1:k]...)][:][2:end]

    for c ∈ sort(coeffs, by = t -> (sum(t),length(t) - length([i for i ∈ t if i != 0])))
        if sum((c .* dim.(simples)).^2) > d_rem continue end

        if simples_covered(c,simples_indices) continue end

        X = dsum([simples[j]^c[j] for j ∈ 1:k])

        ic = iscentral(X)

        if ic
            so = half_braidings(X, simples = simples, parent = C)
            c_simples = [c_simples; so]
            d_rem = d_rem - sum([dim(x)^2 for x in so])
            if d_rem == 0 return c_simples end
            push!(simples_indices, c)
        end
    end
    if d_rem > 0
        @warn "Not all halfbraidings found"
    end
    return c_simples
end

# function monoidal_completion(simples::Vector{CenterObject})
#     complete_simples = simples
#     for i ∈ 1:length(simples)
#         for j ∈ i:length(simples)
#             X,Y = simples[[i,j]]
#             complete_simples = [complete_simples; [x for (x,m) ∈
↪   simple_subobjects(X⊗Y)]]
#             @show complete_simples
#             complete_simples = unique_simples(complete_simples)
#         end
#     end
#     if length(complete_simples) > length(simples)
#         return monoidal_completion(complete_simples)
#     end
```

```
#     return complete_simples
# end

function simples_covered(c::Tuple, v::Vector)
    for w ∈ v
        if *((w .<= c)...)
            return true
        end
    end
    false
end

function isindependent(c::Vector,v::Vector...)
    if length(v) == 0 return true end
    m = matrix(ZZ, [vi[j] for vi ∈ v, j ∈ 1:length(v[1])])

    try
        x = solve(m,matrix(ZZ,c))
    catch
        return true
    end

    return !(*((x .>=0)...))
end

function find_centrals(simples::Vector{<:Object})
    c_simples = typeof(simples[1])[]
    non_central = typeof(simples[1])[]
    for s ∈ simples
        ic, so = iscentral(s)
        if ic
            c_simples = [c_simples; so]
        else
            non_central = [non_central; s]
        end
    end
    return c_simples, non_central
end

function partitions(d::Int64,k::Int64)
    parts = []
    for c ∈ Base.product([0:d for i ∈ 1:k]...)
        if sum([x for x ∈ c]) == d
            parts = [parts; [[x for x ∈ c]]]
        end
    end
    return parts
end

"""
    braiding(X::CenterObject, Y::CenterObject)
```

```
Return the braiding isomorphism ```X⊗Y → Y⊗X```.
"""
function braiding(X::CenterObject, Y::CenterObject)
    dom = X.object⊗Y.object
    cod = Y.object⊗X.object
    braid = zero_morphism(dom, cod)
    for (s,ys) ∈ zip(simples(parent(X).category), X.γ)
        proj = basis(Hom(Y.object,s))
        if length(proj) == 0 continue end
        incl = basis(Hom(s,Y.object))
        braid = braid + sum([(i⊗id(X.object))∘ys∘(id(X.object)⊗p) for i ∈ incl, p ∈
         ↪  proj][:])
    end
    return Morphism(X⊗Y,Y⊗X,braid)
end

function half_braiding(X::CenterObject, Y::Object)
    dom = X.object⊗Y
    cod = Y⊗X.object
    braid = zero_morphism(dom, cod)
    for (s,ys) ∈ zip(simples(parent(X).category), X.γ)
        proj = basis(Hom(Y,s))
        if length(proj) == 0 continue end
        incl = basis(Hom(s,Y))
        braid = braid + sum([(i⊗id(X.object))∘ys∘(id(X.object)⊗p) for i ∈ incl, p ∈
         ↪  proj][:])
    end
    return braid
end


#-------------------------------------------------------------------------------
#   Functionality
#-------------------------------------------------------------------------------

"""
    dim(X::CenterObject)

Return the categorical dimension of ```X```.
"""
dim(X::CenterObject) = dim(X.object)

"""
    simples(C::CenterCategory)

Return a vector containing the simple objects of ```C```. The list might be incomplete.
"""
function simples(C::CenterCategory)
    if isdefined(C, :simples) return C.simples end
    C.simples = center_simples(C)
    return C.simples
end
```

```
"""
    associator(X::CenterObject, Y::CenterObject, Z::CenterObject)

Return the associator isomorphism ```(X⊗Y)⊗Z → X⊗(Y⊗Z)```.
"""
function associator(X::CenterObject, Y::CenterObject, Z::CenterObject)
    dom = (X⊗Y)⊗Z
    cod = X⊗(Y⊗Z)
    return Morphism(dom,cod, associator(X.object, Y.object, Z.object))
end

matrices(f::CenterMorphism) = matrices(f.m)
matrix(f::CenterMorphism) = matrix(f.m)

"""
    compose(f::CenterMorphism, g::CenterMorphism)

Return the composition ```g∘f```.
"""
compose(f::CenterMorphism, g::CenterMorphism) = Morphism(domain(f), codomain(g),
 ↪  g.m∘f.m)

"""
    dual(X::CenterObject)

Return the (left) dual object of ```X```.
"""
function dual(X::CenterObject)
    a = associator
    e = ev(X.object)
    c = coev(X.object)
    γ = Morphism[]
    dX = dual(X.object)
    for (Xi,yXi) ∈ zip(simples(parent(X).category), X.γ)
        f =
         ↪  (e⊗id(Xi⊗dX))∘inv(a(dX,X.object,Xi⊗dX))∘(id(dX)⊗a(X.object,Xi,dX))∘(id(dX)⊗(inv(yXi)⊗id(dX)))∘(
        γ = [γ; f]
    end
    return CenterObject(parent(X),dX,γ)
end

"""
    ev(X::CenterObject)

Return the evaluation morphism ``` X⊗X → 1```.
"""
function ev(X::CenterObject)
    Morphism(dual(X)⊗X,one(parent(X)),ev(X.object))
end
```

```
"""
    coev(X::CenterObject)

Return the coevaluation morphism ```1 → X⊗X*```.
"""
function coev(X::CenterObject)
    Morphism(one(parent(X)),X⊗dual(X),coev(X.object))
end

"""
    id(X::CenterObject)

Return the identity on ```X```.
"""
id(X::CenterObject) = Morphism(X,X,id(X.object))

"""
    tr(f:::CenterMorphism)

Return the categorical trace of ```f```.
"""
function tr(f::CenterMorphism)
    C = parent(domain(f))
    return CenterMorphism(one(C),one(C),tr(f.m))
end

"""
    inv(f::CenterMorphism)

Return the inverse of ```f```if possible.
"""
function inv(f::CenterMorphism)
    return Morphism(codomain(f),domain(f), inv(f.m))
end

"""
    isisomorphic(X::CenterObject, Y::CenterObject)

Check if ```X≃Y```. Return ```(true, m)``` where ```m```is an isomorphism if true,
else return ```(false,nothing)```.
"""
function isisomorphic(X::CenterObject, Y::CenterObject)
    S = simples(parent(X))
    if [dim(Hom(X,s)) for s ∈ S] == [dim(Hom(Y,s)) for s ∈ S]
        return true, inv(decompose_morphism(Y))∘decompose_morphism(X)
    else
        return false, nothing
    end
end

function +(f::CenterMorphism, g::CenterMorphism)
    return Morphism(domain(f), codomain(f), g.m +f.m)
```

```
end

function *(x, f::CenterMorphism)
    return Morphism(domain(f),codomain(f),x*f.m)
end
#-------------------------------------------------------------------------------
#   Functionality: Image
#-------------------------------------------------------------------------------

"""
    kernel(f::CenterMoprhism)

Return a tuple ```(K,k)``` where ```K```is the kernel object and ```k```is the
 ↪  inclusion.
"""
function kernel(f::CenterMorphism)
    ker, incl = kernel(f.m)
    f_inv = left_inverse(incl)

    braiding = [(id(s)⊗f_inv)∘γ∘(incl⊗id(s)) for (s,γ) ∈
     ↪  zip(simples(parent(domain(f.m))), domain(f).γ)]

    Z = CenterObject(parent(domain(f)), ker, braiding)
    return Z, Morphism(Z,domain(f), incl)
end

"""
    cokernel(f::CenterMoprhism)

Return a tuple ```(C,c)``` where ```C```is the cokernel object and ```c```is the
 ↪  projection.
"""
function cokernel(f::CenterMorphism)
    coker, proj = cokernel(f.m)
    f_inv = right_inverse(proj)

    braiding = [(proj⊗id(s))∘γ∘(id(s)⊗f_inv) for (s,γ) ∈
     ↪  zip(simples(parent(domain(f.m))), codomain(f).γ)]

    Z = CenterObject(parent(domain(f)), coker, braiding)
    return Z, Morphism(codomain(f),Z, proj)
end



#-------------------------------------------------------------------------------
#   Hom Spaces
#-------------------------------------------------------------------------------

struct CenterHomSpace <: HomSpace
    X::CenterObject
    Y::CenterObject
```

```
    basis::Vector{CenterMorphism}
    parent::VectorSpaces
end

function Hom(X::CenterObject, Y::CenterObject)
    b = basis(Hom(X.object, Y.object))

    projs = [central_projection(X,Y,f) for f in b]

    proj_exprs = [express_in_basis(p,b) for p ∈ projs]

    M = zero(MatrixSpace(base_ring(X), length(b),length(b)))
    for i ∈ 1:length(proj_exprs)
        M[i,:] = proj_exprs[i]
    end
    r, M = rref(M)
    H_basis = CenterMorphism[]
    for i ∈ 1:r
        f = Morphism(X,Y,sum([m*bi for (m,bi) ∈ zip(M[i,:], b)]))
        H_basis = [H_basis; f]
    end
    return CenterHomSpace(X,Y,H_basis, VectorSpaces(base_ring(X)))
end

function central_projection(dom::CenterObject, cod::CenterObject, f::Morphism, simples =
 ↪  simples(parent(domain(f))))
    X = domain(f)
    Y = codomain(f)
    C = parent(X)
    D = dim(C)
    proj = zero_morphism(X, Y)
    a = associator

    for (Xi, yX) ∈ zip(simples, dom.γ)
        dXi = dual(Xi)
        yY = half_braiding(cod, dXi)

        ϕ =
         ↪  (ev(dXi)⊗id(Y))∘inv(a(dual(dXi),dXi,Y))∘(spherical(Xi)⊗yY)∘a(Xi,Y,dXi)∘((id(Xi)⊗f)⊗id(dXi))∘(yX

        proj = proj + dim(Xi)*ϕ
    end
    return inv(D*base_ring(dom)(1))*proj
end

"""
    zero_morphism(X::CenterObject, Y::CenterObject)

Return the zero morphism ```0:X → Y```.
"""
zero_morphism(X::CenterObject, Y::CenterObject) =
 ↪  Morphism(X,Y,zero_morphism(X.object,Y.object))
```

```
#------------------------------------------------------------------------------
#   Pretty Printing
#------------------------------------------------------------------------------

function show(io::IO, X::CenterObject)
    print(io, "Central object: $(X.object)")
end

function show(io::IO, C::CenterCategory)
    print(io, "Drinfeld center of $(C.category)")
end

function show(io::IO, f::CenterMorphism)
    print(io, "Morphism in $(parent(domain(f)))")
end
```

A.7. Misc

```
struct ProductCategory{N} <: Category
    factors::Tuple
end

struct ProductObject{N} <: Object
    parent::ProductCategory{N}
    factors::Tuple
end

struct ProductMorphism{N} <: Morphism
    domain::ProductObject{N}
    codomain::ProductObject{N}
    factors::Tuple
end


ProductCategory(C::Category...) = ProductCategory{length(C)}(C)
ProductObject(X::Object...) = ProductObject{length(X)}(ProductCategory(parent.(X)...),
 ↪  X)

×(C::Category, D::Category) = ProductCategory(C,D)

function Morphism(f::Morphism...)
    dom = ProductObject(domain.(f)...)
    cod = ProductObject(codomain.(f)...)
    ProductMorphism{length(X)}(dom,cod,f)
end

getindex(C::ProductCategory,x) = C.factors[x]
getindex(X::ProductObject,x) = X.factors[x]
getindex(f::ProductMorphism,x) = f.factors[x]
```

```
#-------------------------------------------------------------------
#   Functionality
#-------------------------------------------------------------------

function dsum(X::ProductObject, Y::ProductObject)
    return ProductObject([dsum(x,y) for x ∈ X.factors, y ∈ Y.factors]...)
end

function tensor_product(X::ProductObject, Y::ProductObject)
    return ProductObject([tensor_product(x,y) for x ∈ X.factors, y ∈ Y.factors]...)
end

function dsum(f::ProductMorphism, g::ProductMorphism)
    ProductMorphism([dsum(fi,gi) for fi ∈ f.factors, gi ∈ g.factors])
end

function tensor_product(f::ProductMorphism, g::ProductMorphism)
    ProductMorphism([tensor_product(fi,gi) for fi ∈ f.factors, gi ∈ g.factors])
end

function simples(C::ProductCategory{N}) where N
    zeros = [zero(Ci) for Ci ∈ C.factors]
    simpls = ProductObject{N}[]
    for i ∈ 1:N
        for s ∈ simples C.factors[i]
            so = zeros
            so[i] = s
            push!(simpls, ProductObject(so...))
        end
    end
    return simpls
end

struct OppositeCategory <: Category
    C::Category
end

struct OppositeObject <: Object
    parent::OppositeCategory
    X::Object
end


struct OppositeMorphism <: Morphism
    domain::OppositeObject
    codomain::OppositeObject
    m::Morphism
end


base_ring(C::OppositeCategory) = base_ring(C.C)
```

```
base_ring(X::OppositeObject) = base_ring(X.X)
parent(X::OppositeObject) = OppositeCategory(parent(X.X))

compose(f::OppositeMorphism, g::OppositeMorphism) = OppositeMorphism(compose(g.m,f.m))

function product(X::OppositeObject, Y::OppositeObject)
    Z,px = product(X.X,Y.X)
    return OppositeObject(Z), OppositeMorphism.(px)
end

function coproduct(X::OppositeObject, Y::OppositeObject)
    Z,ix = coproduct(X.X,Y.X)
    return OppositeObject(Z), OppositeMorphism.(ix)
end

function dsum(X::OppositeObject, Y::OppositeObject)
    Z,ix,px = coproduct(X.X,Y.X)
    return OppositeObject(Z), OppositeMorphism.(ix), OppositeMorphism.(px)
end

tensor_product(X::OppositeObject, Y::OppositeObject) =
 ↪  OppositeObject(tensor_product(X.X,Y.X))

#-------------------------------------------------------------------------------
#   Inversion
#-------------------------------------------------------------------------------

OppositeCategory(C::OppositeCategory) = C.C
OppositeObject(X::OppositeObject) = X.X
OppositeMorphism(f::OppositeMorphism) = f.m

function multiplication_table(C::Category, simples::Vector{<:Object} = simples(C))
    @assert issemisimple(C) "Category needs to be semi-simple"
    m = [s⊗t for s ∈ simples, t ∈ simples]
    coeffs = [coefficients(m,simples) for m ∈ m]
    return [c[k] for c ∈ coeffs, k ∈ 1:length(simples)]
end

function multiplication_table(simples::Vector{<:Object})
    @assert issemisimple(parent(simples[1])) "Category needs to be semi-simple"

    return multiplication_table(parent(simples[1]), simples)
end

function print_multiplication_table(simples::Vector{<:Object}, names::Vector{String} =
 ↪  ["v$i" for i ∈ 1:length(simples)])
    @assert length(simples) == length(names) "Invalid input"
    mult_table = multiplication_table(parent(simples[1]), simples)

    return [pretty_print_semisimple(s⊗t,simples,names) for s ∈ simples, t ∈ simples]
end
```

```
function
↪ pretty_print_semisimple(m::Object,simples::Vector{<:Object},names::Vector{String})
    facs = decompose(m, simples)

    if length(facs) == 0 return "0" end

    str = ""
    for (o,k) ∈ facs
        i = findfirst(x -> x == o, simples)
        if i == nothing i = findfirst(x -> isisomorphic(x,o)[1], simples) end

        str = length(str) > 0 ? str*"⊕"*"$(names[i])^$k" : str*"$(names[i])^$k"
    end
    return str
end


function coefficients(X::T, simples::Vector{T} = simples(parent(X))) where {T <: Object}
    facs = decompose(X)
    coeffs = [0 for i ∈ 1:length(simples)]
    for (x,k) ∈ facs
        i = findfirst(y -> y == x, simples)
        if i == nothing i = findfirst(y -> isisomorphic(y,x)[1], simples) end
        coeffs[i] = k
    end
    return coeffs
end


"""
    grothendieck_ring(C::Category)

Return the grothendieck ring of the multiring category ```C```.
"""
function grothendieck_ring(C::Category, simples = simples(C))
    @assert ismultiring(C) "C is required to be tensor"

    m = multiplication_table(C,simples)

    Z = Integers{Int64}()

    A = AlgAss(Z, m, coefficients(one(C),simples))
    function to_gd(X)
        coeffs = coefficients(X,simples)
        z = AlgAssElem{Int64, AlgAss{Int64}}(A)
        z.coeffs = coeffs
        return z
    end
    return A,to_gd
end

abstract type Functor end
```

```
domain(F::Functor) = F.domain
codomain(F::Functor) = F.codomain

#-------------------------------------------------------------------------------
#   Functors Functionality
#-------------------------------------------------------------------------------



#-------------------------------------------------------------------------------
#   Forgetful Functors
#-------------------------------------------------------------------------------

struct Forgetful <: Functor
    domain::Category
    codomain::Category
    obj_map
    mor_map
end

function Forgetful(C::GradedVectorSpaces, D::VectorSpaces)
    obj_map = x -> X.V
    mor_map = f -> Morphism(domain(f).V, codomain(f).V, f.m)
    return Forgetful(C,D,obj_map, mor_map)
end

(F::Functor)(x::T) where {T <: Object} = F.obj_map(x)
(F::Functor)(x::T) where {T <: Morphism} = F.mor_map(x)

function show(io::IO, F::Forgetful)
    print(io, "Forgetful functor from $(domain(F)) to $(codomain(F))")
end
#-------------------------------------------------------------------------------
#   Hom Functors
#-------------------------------------------------------------------------------

struct HomFunctor <: Functor
    domain::Category
    codomain::Category
    obj_map
    mor_map
end

function Hom(X::Object,::Colon)
    K = base_ring(parent(X))
    C = VectorSpaces(K)
    obj_map = Y -> Hom(X,Y)
    mor_map = f -> g -> g ∘ f
    return HomFunctor(parent(X),C,obj_map,mor_map)
end
```

```
function Hom(::Colon,X::Object)
    K = base_ring(parent(X))
    C = VectorSpaces(K)
    obj_map = Y -> Hom(Y,X)
    mor_map = g -> f -> g ∘ f
    return HomFunctor(OppositeCategory(parent(X)),C,obj_map,mor_map)
end

function Hom(X::SetObject,::Colon)
    obj_map = Y -> Hom(X,Y)
    mor_map = f -> g -> g ∘ f
    return HomFunctor(parent(X),Sets(),obj_map,mor_map)
end

function Hom(::Colon,X::SetObject)
    obj_map = Y -> Hom(Y,X)
    mor_map = g -> f -> g ∘ f
    return HomFunctor(parent(X),Sets(),obj_map,mor_map)
end

function show(io::IO, H::HomFunctor)
    print(io,"$(typeof(H.domain) == OppositeCategory ? "Contravariant" : "Covariant")
     ↪  Hom-functor in $(H.domain)")
end
#-------------------------------------------------------------------------------
#   Tensor Product Functors
#-------------------------------------------------------------------------------

struct TensorFunctor <: Functor
    domain::ProductCategory{2}
    codomain::Category
    obj_map
    mor_map
end

function TensorFunctor(C::Category)
    domain = ProductCategory(C,C)
    obj_map = X -> X[1]⊗X[2]
    mor_map = f -> f[1]⊗f[2]
    return TensorFunctor(domain, C, obj_map, mor_map)
end

⊗(C::Category) = TensorFunctor(C)


#-------------------------------------------------------------------------------
#   Restriction and Induction
#-------------------------------------------------------------------------------

struct GRepRestriction <: Functor
    domain::GroupRepresentationCategory
    codomain::GroupRepresentationCategory
```

```
    obj_map
    mor_map
end

function Restriction(C::GroupRepresentationCategory, D::GroupRepresentationCategory)
    @assert base_ring(C) == base_ring(D) "Not compatible"
    #@assert issubgroup(base_group(D), base_group(C))[1] "Not compatible"
    obj_map = X -> restriction(X, base_group(D))
    mor_map = f -> restriction(f, base_group(D))
    return GRepRestriction(C,D,obj_map,mor_map)
end

struct GRepInduction <: Functor
    domain::GroupRepresentationCategory
    codomain::GroupRepresentationCategory
    obj_map
    mor_map
end

function Induction(C::GroupRepresentationCategory, D::GroupRepresentationCategory)
    @assert base_ring(C) == base_ring(D) "Not compatible"
    #@assert issubgroup(base_group(C), base_group(D))[1] "Not compatible"
    obj_map = X -> induction(X, base_group(D))
    mor_map = f -> induction(f, base_group(D))
    return GRepInduction(C,D, obj_map, mor_map)
end

function show(io::IO, F::GRepRestriction)
    print(io,"Restriction functor from $(domain(F)) to $(codomain(F)).")
end

function show(io::IO, F::GRepInduction)
    print(io,"Induction functor from $(domain(F)) to $(codomain(F)).")
end

function (F::FinField)(x::GAP.FFE)
    if GAP.Globals.Characteristic(x) != Int(characteristic(F))
        throw(ErrorException("Mismatching characteristics"))
    end
    if x == GAP.Globals.Zero(x) return F(0) end
    if x == GAP.Globals.One(x) return F(1) end

    deg = degree(F)

    if deg == 1 return F(GAP.Globals.IntFFE(x)) end

    char = Int(characteristic(F))
    exponent = GAP.Globals.LogFFE(x,GAP.Globals.Z(char,deg))

    if exponent == GAP.Globals.fail throw(ErrorException("Conversion failed")) end

    a = gen(F)
```

```
    return a^exponent
end

function recover_solutions(p::Tuple, K::Field)
    p = p[1]
    f = p[1]
    g = p[2]
    v = p[3] .* p[4]
    F = splitting_field(f)

    if degree(F) > degree(K) @warn "would split over $F" end
    f = change_base_ring(K,f)
    rs = roots(f)
    solutions = []
    for r ∈ rs
        solutions = [solutions; Tuple([[vi(r)*inv(g(r)) for vi ∈ v]; [r]])]
    end
    return solutions
end


struct Sets <: Category end

struct SetObject <: Object
    set::T where T <: AbstractSet
end

struct SetMorphism <: Morphism
    m::Dict
    domain::SetObject
    codomain::SetObject
end

#-------------------------------------------------
#   Constructors
#-------------------------------------------------

SetObject(S::Array) = SetObject(Set(S))

function SetMorphism(D::SetObject, C::SetObject, m::Dict)
    if keys(m) ⊆ D && values(m) ⊆ C
        return SetMorphism(m,D,C)
    else
        throw(ErrorException("Mismatching (co)domain"))
    end
end

SetMorphism(D::SetObject, C::SetObject, m::Function) = SetMorphism(D,C, Dict(x => m(x)
↪  for x ∈ D))

#-------------------------------------------------
#   Functionality
#-------------------------------------------------
```

```julia
in(item,S::SetObject) = in(item,S.set)

issubset(item, S::SetObject) = issubset(item, S.set)

iterate(X::SetObject) = iterate(X.set)
iterate(X::SetObject,state) = iterate(X.set,state)

length(X::SetObject) = length(X.set)

(f::SetMorphism)(item) = f.m[item]

==(X::SetObject,Y::SetObject) = X.set == X.set

parent(X::SetObject) = Sets()

#-------------------------------------------------
#   Functionality: Morphisms
#-------------------------------------------------

function compose(f::SetMorphism...)
    if length(f) == 1 return f[1] end
    if [domain(f[i]) == codomain(f[i-1]) for i ∈ 2:length(f)] != trues(length(f)-1)
        throw(ErrorException("Morphisms not compatible"))
    end
    m = f[1]
    for g in f[2:end]
        m = Dict(x => g(m(x)) for x ∈ keys(m.m))
    end
    return SetMorphism(domain(f[1]), codomain(f[end]),m)
end


function inv(f::SetMorphism)
    if length(values(f.m)) == length(keys(f.m))
        SetMorphism( codomain(f),domain(f), Dict(v => k for (k,v) ∈ f.m))
    else
        throw(ErrorException("Not invertible"))
    end
end

id(X::SetObject) = SetMorphism(X,X, x->x)
==(f::SetMorphism, g::SetMorphism) = f.m == g.m

#-------------------------------------------------
#   Product
#-------------------------------------------------

function product(X::SetObject, Y::SetObject, projections = false)
    Z = SetObject(Set([(x,y) for x ∈ X, y ∈ Y]))
    pX = SetMorphism(Z,X, x -> x[1])
    pY = SetMorphism(Z,Y, x -> x[2])
```

```
    return projections ? (Z,[pX,pY]) : Z
end

function coproduct(X::SetObject, Y::SetObject, injections = false)
    if length(X.set ∩ Y.set) != 0
        Z = SetObject(union([(x,0) for x ∈ X],[(y,1) for y ∈ Y]))
        ix = SetMorphism(X,Z, x -> (x,0))
        iy = SetMorphism(Y,Z, y -> (y,1))
    else
        Z = SetObject(union(X.set,Y.set))
        ix = SetMorphism(X,Z, x -> x)
        iy = SetMorphism(Y,Z, y -> y)
    end
    return injections ? (Z, [ix,iy]) : Z
end

#--------------------------------------------------
#   HomSets
#--------------------------------------------------

struct SetHomSet <: HomSet
    X::SetObject
    Y::SetObject
end

Hom(X::SetObject, Y::SetObject) = SetHomSet(X,Y)

#--------------------------------------------------
#   Pretty printing
#--------------------------------------------------

function show(io::IO, X::Sets)
    print(io,"Category of finte sets")
end

function show(io::IO, X::SetObject)
    print(X.set)
end
```