QPA

Quivers and Path Algebras

Version 1.18

April 2015

The QPA-team

The QPA-team Email: oyvind.solberg@math.ntnu.no
Homepage: http://sourceforge.net/projects/quiverspathalg/
Address: Department of Mathematical Sciences
NTNU
N-7491 Trondheim
Norway

Abstract

The GAP4 undeposited package QPA extends the GAP functionality for computations with finite dimensional quotients of path algebras. QPA has data structures for quivers, quotients of path algebras, representations of quivers with relations and complexes of modules. Basic operations on representations of quivers are implemented as well as contructing minimal projective resolutions of modules (using using linear algebra). A not necessarily minimal projective resolution constructed by using Groebner basis theory and a paper by Green-Solberg-Zacharia, "Minimal projective resolutions", has been implemented. A goal is to have a test for finite representation type. This work has started, but there is a long way left. Part of this work is to implement/port the functionality and data structures that was available in CREP.

Copyright

Acknowledgements

The system design of QPA was initiated by Edward L. Green, Lenwood S. Heath, and Craig A. Struble. It was continued and completed by Randall Cone and Edward Green. We would like to thank the following people for their contributions:

Chain complexes	Kristin Krogh Arnesen and Øystein Skartsæterhagen
Degeneration order for modules in finite type	Andrzej Mroz
GBNP interface (for Groebner bases)	Randall Cone
Homomorphisms of modules	Øyvind Solberg and Anette Wraalsen
Koszul duals	Stephen Corwin
Matrix representations of path algebras	Øyvind Solberg and George Yuhasz
Opposite algebra and tensor products of algebras	Øystein Skartsæterhagen
Predefined classes of algebras	Andrzej Mroz and Øyvind Solberg
Projective resolutions (using Groebnar basis)	Randall Cone and Øyvind Solberg
Projective resolutions (using linear algebra)	Øyvind Solberg
Quickstart	Kristin Krogh Arnesen
Quivers, path algebras	Gerard Brunick
The bounded derived category	Kristin Krogh Arnesen and Øystein Skartsæterhagen
Unitforms	Øyvind Solberg

Contents

Chapter 1

Introduction

1.1 General aims

The overall aim of QPA is to provide computational tools in basic research in mathematics (algebra). It seeks to furnish users within and outside the area of representation theory of finite dimensional algebras with a computational software package that will help in exploring a problem and testing conjectures. As with all software development, new ideas and question will arise. The ability to study and compute examples will help in proving or disproving well-established questions/conjectures. Furthermore, it will enable us to consider new examples which were not accessible by hand or other means before. In this way, we hope that QPA will aid in the development, not only of the area of representation theory of finite dimensional algebras, but also of a broad variety of other areas of mathematics, where such structures occur. In addition we aspire to create a research environment for international cooperation on computational representation theory of finite dimensional algebras.

1.2 Installation and system requirements

QPA does not use external binaries and, therefore, works without restrictions on the type of the operating system. This version of the package is designed for GAP >= 4.5 and no compatibility with previous releases of GAP 4 is guaranteed. However, QPA depend on the Groebner basis GAP-package GBNP.

To use the QPA online help it is necessary to install the GAP 4 package GAPDoc by Frank Lübeck and Max Neunhöffer, which is available from the GAP site or from `http://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc/`.

QPA is distributed in standard format (tar.gz) and can be obtained from `http://sourceforge.net/projects/quiverspathalg/`. To install the package, unpack its archive in the pkg subdirectory of your GAP installation. In a similar way install the GAP-package GBNP which is available from the GAP site.

For more detailed installation information and an alternative way for downloading QPA see `http://www.math.ntnu.no/~oyvinso/QPA/`.

Chapter 2

Quickstart

This chapter is intended for those who would like to get started with QPA right away by playing with a few examples. We assume that the user is familiar with GAP syntax, for instance the different ways to display various GAP objects: `View`, `Print` and `Display`. These features are all implemented for the objects defined in QPA, and by using `Display` on an object, you will get a complete description of it.

The following examples show how to create the most fundamental algebraic structures featured in QPA, namely quivers, path algebras and quotients of path algebras, modules and module homomorphisms. Sometimes, there is more than one way of constructing such objects. See their respective chapter in the documentation for more on this. The code from the examples can be found in the `examples/` directory of the distribution of QPA.

2.1 Example 1 – quivers, path algebras and quotients of path algebras

We construct a quiver Q, i.e. a finite directed graph, with one vertex and two loops:

Example
```
gap> Q := Quiver( 1, [ [1,1,"a"], [1,1,"b"] ] );
<quiver with 1 vertices and 2 arrows>
gap> Display(Q);
Quiver( ["v1"], [["v1","v1","a"],["v1","v1","b"]] )
```

When displaying Q, we observe that the vertex has been named `v1`, and that this name is used when describing the arrows. (The "Display" style of viewing a quiver can also be used in construction, i.e., we could have written `Q := Quiver( ["v1"], [["v1","v1","a"],["v1","v1","b"]] )` to get the same object.)

If we want to know the number and names of the vertices and arrows, without getting the structure of Q, we can request this information as shown below. We can also access the vertices and arrows directly.

Example
```
gap> VerticesOfQuiver(Q);
[ v1 ]
gap> ArrowsOfQuiver(Q);
[ a, b ]
gap> Q.a;
a
```

The next step is to create the path algebra kQ from Q, where k is the rational numbers (in general, one can chose any field implemented in GAP).

Example
```
gap> kQ := PathAlgebra(Rationals, Q);
<Rationals[<quiver with 1 vertices and 2 arrows>]>
gap> Display(kQ);
<Path algebra of the quiver <quiver with 1 vertices and 2 arrows>
over the field Rationals>
```

We know that this algebra has three generators, with the vertex `v_1` as the identity. This can be verified by QPA. For convenience, we introduce new variables `v1`, `a` and `b` to get easier access to the generators.

Example
```
gap> AssignGeneratorVariables(kQ);
#I  Assigned the global variables [ v1, a, b ]
gap> v1; a; b;
(1)*v1
(1)*a
(1)*b
gap> id := One(kQ);
(1)*v1
gap> v1 = id;
true
```

Now, we want to construct a finite dimensional algebra, by dividing out some ideal. The generators of the ideal (the relations) are given in terms of paths, and it is important to know the convention of writing paths used in QPA. If we first go the arrow a and then the arrow b, the path is written as $a * b$.

Say that we want our ideal to be generated by the relations `\{a^2, a*b - b*a, b^2\}`. Then we make a list `relations` consisting of these relations and to construct the quotient we say: `A := kQ/relations;` on the command line in GAP.

Example
```
gap> relations := [a^2,a*b-b*a, b*b];
[ (1)*a^2, (1)*a*b+(-1)*b*a, (1)*b^2 ]
gap> A := kQ/relations;
<Rationals[<quiver with 1 vertices and 2 arrows>]/<two-sided ideal in
<Rationals[<quiver with 1 vertices and 2 arrows>]>, (3 generators)>>
```

See 4.6 for further remarks on constructing quotients of path algebras.

2.2 Example 2 – Introducing modules

In representation theory, there are several conventions for expressing modules of path algebras, and again it is useful to comment on the convention used in QPA. A module (or representation) of an algebra $A = kQ/I$ is, briefly explained, a picture of Q where the vertices are finite dimensional k-vectorspaces, and the arrows are linear transformations between the vector spaces respecting the relations of I. The modules are *right* modules, and a linear transformation from k^n to k^m is represented by a $n \times m$-matrix.

There are several ways of constructing modules in *QPA*. First, we will explore some modules which *QPA* gives us for free, namely the indecomposable projectives. We start by constructing a new algebra. The underlying quiver has three vertices and three arrows and looks like an A_3 quiver with both arrows pointing to the right, and one additional loop in the final vertex. The only relation is to go this loop twice.

Example
```
gap> Q := Quiver( 3, [ [1,2,"a"], [2,3,"b"], [3,3,"c"] ]);
<quiver with 3 vertices and 3 arrows>
gap> kQ := PathAlgebra(Rationals, Q);
<Rationals[<quiver with 3 vertices and 3 arrows>]>
gap> relations := [kQ.c*kQ.c];
[ (1)*c^2 ]
gap> A := kQ/relations;
<Rationals[<quiver with 3 vertices and 3 arrows>]/
<two-sided ideal in <Rationals[<quiver with 3 vertices and 3 arrows>]>,
  (1 generators)>>
```

The indecomposable projectives are easily created with one command. We use `Display` to explore the modules.

Example
```
gap> projectives := IndecProjectiveModules(A);
[ <[ 1, 1, 2 ]>, <[ 0, 1, 2 ]>, <[ 0, 0, 2 ]> ]
gap> proj1 := projectives[1];
<[ 1, 1, 2 ]>
gap> Display(proj1);
<Module over <Rationals[<quiver with 3 vertices and 3 arrows>]/
<two-sided ideal in <Rationals[<quiver with 3 vertices and 3 arrows>]>,
  (1 generators)>> with dimension vector
[ 1, 1, 2 ]> and linear maps given by
for arrow a:
[ [  1 ] ]
for arrow b:
[ [  1,  0 ] ]
for arrow c:
[ [  0,  1 ],
  [  0,  0 ] ]
```

If we, for some reason, want to use the maps of this module, we can get the matrices directly by using the command `MatricesOfPathAlgebraModule(proj1)`:

Example
```
gap> M := MatricesOfPathAlgebraModule(proj1);
[ [ [ 1 ] ], [ [ 1, 0 ] ], [ [ 0, 1 ], [ 0, 0 ] ] ]
gap> M[1];
[ [ 1 ] ]
```

Naturally, the indecomposable injective modules are just as easily constructed, and so are the simple modules.

Example
```
gap> injectives := IndecInjectiveModules(A);
[ <[ 1, 0, 0 ]>, <[ 1, 1, 0 ]>, <[ 2, 2, 2 ]> ]
gap> simples := SimpleModules(A);
[ <[ 1, 0, 0 ]>, <[ 0, 1, 0 ]>, <[ 0, 0, 1 ]> ]
```

We know for a fact that the simple module in vertex 1 and the indecomposable injective module in vertex 1 coincide. Let us look at this relationship in QPA:

Example
```
gap> s1 := simples[1];
<[ 1, 0, 0 ]>
gap> inj1 := injectives[1];
<[ 1, 0, 0 ]>
gap> IsIdenticalObj(s1,inj1);
false
gap> s1 = inj1;
true
gap> IsomorphicModules(s1,inj1);
true
```

We observe that QPA recognizes the modules as "the same" (that is, isomorphic); however, they are *not* the same instance and hence the simplest test for equality fails. This is important to bear in mind – objects which are isomorphic and regarded as the same in the "real world", are not necessarily the same in GAP.

2.3 Example 3 – Constructing modules and module homomorphisms

Assume we want to construct the following A-module M, where A is the same algebra as in the previous example $\begin{tikzcd} 0 \arrow[r, "0"] & \mathbb{Q} \arrow[r, "1"] & \mathbb{Q} \arrow[loop right, "0"] \end{tikzcd}$. This module is neither indecomposable projective or injective, nor simple, so we need to do the dirty work ourselves. Usually, the easiest way to construct a module is to state the dimension vector and the non-zero maps. Here, there is only one non-zero map, and we write

Example
```
gap> M := RightModuleOverPathAlgebra( A, [0,1,1], [ ["b", [[1]] ] ] );
<[ 0, 1, 1 ]>
```

To make sure we got everything right, we can use `Display(M)` to view the maps. The most tricky thing is usually to get the correct numbers of brackets. Here is a slightly bigger example: $\begin{tikzcd} \mathbb{Q} \arrow[r, "(0\ 0)"] & \mathbb{Q}^2 \arrow[r, "\begin{pmatrix} 1 & 0 \\ -1 & 0 \end{pmatrix}"] & \mathbb{Q}^2 \arrow[loop right, "\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}"] \end{tikzcd}$.

Example
```
gap> N := RightModuleOverPathAlgebra( A, [1,2,2], [ ["a",[[1,1]] ],
    ["b", [[1,0], [-1,0]] ], ["c", [[0,0],[1,0]] ] ] );
<[ 1, 2, 2 ]>
```

Now we want to construct a map between the two modules, say $f : M \to N$, which is non-zero only in vertex 2. This is done by

Example
```
gap> f := RightModuleHomOverAlgebra(M,N, [ [[0]], [[1,1]], NullMat(1,2,Rationals)]);
<<[ 0, 1, 1 ]> ---> <[ 1, 2, 2 ]>>
gap> Display(f);
<<Module over <Rationals[<quiver with 3 vertices and 3 arrows>]/
<two-sided ideal in <Rationals[<quiver with 3 vertices and 3 arrows>]>,
  (1 generators)>> with dimension vector
[ 0, 1, 1 ]> ---> <Module over <Rationals[<quiver with 3 vertices and 3 arrows>]/
```

```
<two-sided ideal in <Rationals[<quiver with 3 vertices and 3 arrows>]>,
  (1 generators)>> with dimension vector [ 1, 2, 2 ]>>
with linear map for vertex number 1:
[ [  0 ] ]
linear map for vertex number 2:
[ [  1,   1 ] ]
linear map for vertex number 3:
[ [  0,   0 ] ]
```

Note the two different ways of writing zero maps. Again, we can retrieve the matrices describing f:

Example
```
gap> MatricesOfPathAlgebraMatModuleHomomorphism(f);
[ [ [ 0 ] ], [ [ 1, 1 ] ], [ [ 0, 0 ] ] ]
```

Chapter 3

Quivers

3.1 Information class, Quivers

A quiver Q is a set derived from a labeled directed multigraph with loops Γ. An element of Q is called a *path*, and falls into one of three classes. The first class is the set of *vertices* of Γ. The second class is the set of *walks* in Γ of length at least one, each of which is represented by the corresponding sequence of *arrows* in Γ. The third class is the singleton set containing the distinguished *zero path*, usually denoted 0. An associative multiplication is defined on Q.

This chapter describes the functions in QPA that deal with paths and quivers. The functions for constructing paths in Section 4.2 are normally not useful in isolation; typically, they are invoked by the functions for constructing quivers in Section 3.2.

3.1.1 InfoQuiver

▷ `InfoQuiver` (info class)

is the info class for functions dealing with quivers.

3.2 Constructing Quivers

3.2.1 Quiver (no. of vertices, list of arrows)

▷ `Quiver(`*`N, arrows`*`)` (function)
▷ `Quiver(`*`vertices, arrows`*`)` (function)
▷ `Quiver(`*`adjacencymatrix`*`)` (function)

Arguments: First construction: *`N`* – number of vertices, *`arrows`* – a list of arrows to specify the graph Γ. Second construction: *`vertices`* – a list of vertex names, *`arrows`* – a list of arrows. Third construction: takes an adjacency matrix for the graph Γ.

Returns: a quiver, which is an object from the category `IsQuiver` (3.3.1).

In the first and third constructions, the vertices are named ‘v1, v2, ...’. In the second construction, unique vertex names are given as strings in the list that is the first parameter. Each arrow is a list consisting of a source vertex and a target vertex, followed optionally by an arrow name as a string.

Vertices and arrows are referenced as record components using the dot (‘.’) operator.

Example

```
gap> q1 := Quiver(["u","v"],[["u","u","a"],["u","v","b"],
>                 ["v","u","c"],["v","v","d"]]);
<quiver with 2 vertices and 4 arrows>
gap> VerticesOfQuiver(q1);
[ u, v ]
gap> ArrowsOfQuiver(q1);
[ a, b, c, d ]
gap> q2 := Quiver(2,[[1,1],[2,1],[1,2]]);
<quiver with 2 vertices and 3 arrows>
gap> ArrowsOfQuiver(q2);
[ a1, a2, a3 ]
gap> VerticesOfQuiver(q2);
[ v1, v2 ]
gap> q3 := Quiver(2,[[1,1,"a"],[2,1,"b"],[1,2,"c"]]);
<quiver with 2 vertices and 3 arrows>
gap> ArrowsOfQuiver(q3);
[ a, b, c ]
gap> q4 := Quiver([[1,1],[2,1]]);
<quiver with 2 vertices and 5 arrows>
gap> VerticesOfQuiver(q4);
[ v1, v2 ]
gap> ArrowsOfQuiver(q4);
[ a1, a2, a3, a4, a5 ]
gap> SourceOfPath(q4.a2);
v1
gap> TargetOfPath(q4.a2);
v2
```

3.2.2 DynkinQuiver (DynkinQuiver)

▷ `DynkinQuiver(Delta, n, orientation)` (operation)

Arguments: `Delta`, `n`, `orientation` – a character (A,D,E), a positive integer, and a list giving the orientation.

Returns: a Dynkin quiver of type `Delta` ("A", "D", or "E") with index `n` and orientation of the arrows given by the list `orientation`.

If `Delta` is equal to "A" with index `n`, then the list `orientation` is of the form `["r", "l", "l", ...,"r", "l"]` of length `n-1`, where "l" or "r" in coordinate i means that the arrow a_i is oriented to the left or to the right, respectively. The vertices and the arrows are named as in the following diagram $1 \overset{a_1}{\text{——}} 2 \overset{a_2}{\text{——}} - - - \overset{a_{n-2}}{\text{——}} n-1 \overset{a_{n-1}}{\text{——}} n$

If `Delta` is equal to "D" with index `n` and `n` greater or equal to 4, then the list `orientation` is of the form `["r", "l", "l", ...,"r", "l"]` of length `n-1`, where "l" or "r" in coordinate i means that the arrow a_i is oriented to the left or to the right, respectively. The vertices and the arrows are named as in the following diagram

$$\begin{tikzcd}
1 \arrow[dr, dash, "a_1"] & & & & & \\
& 3 \arrow[r, dash, "a_3"] & - - - \arrow[r, dash, "a_{n-2}"] & n-1 \arrow[r, dash, "a_{n-1}"] & n \\
2 \arrow[ur, dash, "a_2"'] & & & &
\end{tikzcd}$$

If *Delta* is equal to "E" with index *n* and *n* in $[6,7,8]$, then the list *orientation* is of the form `["r", "l", "l", ...,"r", "l","d"]` of length *n-1*, where "l" or "r" in the *n - 2* first coordinates and at coordinate i means that the arrow a_i is oriented to the left or to the right, respectively, and the last orientation parameter is "d" or "u" indicating if the arrow a_{n-1} is oriented down or up. The vertices and the arrows are named as in the following diagram

$$\begin{tikzcd}
& & n & & & \\
1 \arrow[r, dash, "a_1"] & 2 \arrow[r, dash, "a_2"] & 3 \arrow[u, dash, "a_{n-1}"'] \arrow[r, dash, "a_3"] & - - - \arrow[r, dash] & n-2 \arrow[r, dash, "a_{n-2}"] & n-1
\end{tikzcd}$$

3.2.3 OrderedBy

▷ `OrderedBy(`*quiver, ordering*`)` (function)

Returns: a copy of *quiver* whose elements are ordered by *ordering*. The default ordering of a quiver is length left lexicographic. See Section 3.4 for more information.

3.3 Categories and Properties of Quivers

3.3.1 IsQuiver

▷ `IsQuiver(`*object*`)` (category)

Returns: true when *object* is a quiver.

3.3.2 IsAcyclicQuiver

▷ `IsAcyclicQuiver(`*quiver*`)` (property)

Returns: true when *quiver* is a quiver with no oriented cycles.

3.3.3 IsUAcyclicQuiver

▷ `IsUAcyclicQuiver(`*quiver*`)` (property)

Returns: true when *quiver* is a quiver with no unoriented cycles. Note: an oriented cycle is also an unoriented cycle!

3.3.4 IsConnectedQuiver

▷ `IsConnectedQuiver(`*quiver*`)` (property)

Returns: true when *quiver* is a connected quiver (i.e. each pair of vertices is connected by an unoriented path in *quiver*).

3.3.5 IsTreeQuiver

▷ IsTreeQuiver(*quiver*) (property)

Returns: true when *quiver* is a tree as a graph (i.e. it is connected and contains no unoriented cycles).

Example
```
gap> q1 := Quiver(2,[[1,2]]);
<quiver with 2 vertices and 1 arrows>
gap> IsQuiver("v1");
false
gap> IsQuiver(q1);
true
gap> IsAcyclicQuiver(q1); IsUAcyclicQuiver(q1);
true
true
gap> IsConnectedQuiver(q1); IsTreeQuiver(q1);
true
true
gap> q2 := Quiver(["u","v"],[["u","v"],["v","u"]]);
<quiver with 2 vertices and 2 arrows>
gap> IsAcyclicQuiver(q2); IsUAcyclicQuiver(q2);
false
false
gap> IsConnectedQuiver(q2); IsTreeQuiver(q2);
true
false
gap> q3 := Quiver(["u","v"],[["u","v"],["u","v"]]);
<quiver with 2 vertices and 2 arrows>
gap> IsAcyclicQuiver(q3); IsUAcyclicQuiver(q3);
true
false
gap> IsConnectedQuiver(q3); IsTreeQuiver(q3);
true
false
gap> q4 := Quiver(2, []);
<quiver with 2 vertices and 0 arrows>
gap> IsAcyclicQuiver(q4); IsUAcyclicQuiver(q4);
true
true
gap> IsConnectedQuiver(q4); IsTreeQuiver(q4);
false
false
```

3.3.6 IsDynkinQuiver

▷ IsDynkinQuiver(*quiver*) (property)

Returns: true when *quiver* is a Dynkin quiver (more precisely, when underlying undirected graph of *quiver* is a Dynkin diagram).

This function prints an additional information. If it returns true, it prints the Dynkin type of *quiver*, i.e. A_n, D_m, E_6, E_7 or E_8. Moreover, in case *quiver* is not connected or contains an unoriented cycle, the function also prints a respective info.

```
Example
gap> q1 := Quiver(4,[[1,4],[4,2],[3,4]]);
<quiver with 4 vertices and 3 arrows>
gap> IsDynkinQuiver(q1);
D_4
true
gap> q2 := Quiver(2,[[1,2],[1,2]]);
<quiver with 2 vertices and 2 arrows>
gap> IsDynkinQuiver(q2);
Quiver contains an (un)oriented cycle.
false
gap> q3 := Quiver(5,[[1,5],[2,5],[3,5],[4,5]]);
<quiver with 5 vertices and 4 arrows>
```

3.4 Orderings of paths in a quiver

The only supported ordering on the paths in a quiver is length left lexicographic ordering. The reason for this is that QPA does not have its own functions for computing Groebner basis. Instead they are computed using the GAP-package GBNP. The interface with this package, which is provided by the QPA, only supports the length left lexicographic ordering, even though GBNP supports more orderings.

For constructing a quiver, there are three different methods. TODO: Explain how the vertices and arrows are ordered.

3.5 Attributes and Operations for Quivers

3.5.1 . (for quiver)

▷ `.(`*`Q, element`*`)` (operation)

Arguments: *`Q`* – a quiver, and *`element`* – a vertex or an arrow.

The operation `.` allows access to generators of the quiver. If you have named your vertices and arrows then the access looks like '*`Q.name of element`*'. If you have not named the elements of the quiver, then the default names are v1, v2, ... and a1, a2, ... in the order they are created.

3.5.2 VerticesOfQuiver

▷ `VerticesOfQuiver(`*`quiver`*`)` (attribute)

Returns: a list of paths that are vertices in *`quiver`*.

3.5.3 ArrowsOfQuiver

▷ `ArrowsOfQuiver(`*`quiver`*`)` (attribute)

Returns: a list of paths that are arrows in *`quiver`*.

3.5.4 AdjacencyMatrixOfQuiver

▷ `AdjacencyMatrixOfQuiver(quiver)` (attribute)

Returns: the adjacency matrix of *quiver*.

3.5.5 GeneratorsOfQuiver

▷ `GeneratorsOfQuiver(quiver)` (attribute)

Returns: a list of the vertices and the arrows in *quiver*.

3.5.6 NumberOfVertices

▷ `NumberOfVertices(quiver)` (attribute)

Returns: the number of vertices in *quiver*.

3.5.7 NumberOfArrows

▷ `NumberOfArrows(quiver)` (attribute)

Returns: the number of arrows in *quiver*.

3.5.8 OrderingOfQuiver

▷ `OrderingOfQuiver(quiver)` (attribute)

Returns: the ordering used to order elements in *quiver*. See Section 3.4 for more information.

3.5.9 OppositeQuiver

▷ `OppositeQuiver(quiver)` (attribute)

Returns: the opposite quiver of *quiver*, where the vertices are labelled "name in original quiver" + "_op" and the arrows are labelled "name in orginal quiver" + "_op".

This attribute contains the opposite quiver of a quiver, that is, a quiver which is the same except that every arrow goes in the opposite direction.

The operation `OppositePath` (4.15.1) takes a path in a quiver to the corresponding path in the opposite quiver.

The opposite of the opposite of a quiver Q is isomorphic to Q. In QPA, we regard these two quivers to be the same, so the call `OppositeQuiver(OppositeQuiver(Q))` returns the object `Q`.

Example

```
gap> q1 := Quiver(["u","v"],[["u","u","a"],["u","v","b"],
>               ["v","u","c"],["v","v","d"]]);
<quiver with 2 vertices and 4 arrows>
gap> q1.a;
a
gap> q1.v;
v
gap> VerticesOfQuiver(q1);
[ u, v ]
gap> ArrowsOfQuiver(q1);
[ a, b, c, d ]
gap> AdjacencyMatrixOfQuiver(q1);
[ [ 1, 1 ], [ 1, 1 ] ]
```

```
gap> GeneratorsOfQuiver(q1);
[ u, v, a, b, c, d ]
gap> NumberOfVertices(q1);
2
gap> NumberOfArrows(q1);
4
gap> OrderingOfQuiver(q1);
<length left lexicographic ordering>
gap> q1_op := OppositeQuiver(q1);
<quiver with 2 vertices and 4 arrows>
gap> VerticesOfQuiver(q1_op);
[ u_op, v_op ]
gap> ArrowsOfQuiver(q1_op);
[ a_op, b_op, c_op, d_op ]
```

3.5.10 FullSubquiver

▷ `FullSubquiver(`*`quiver, list`*`)` (operation)

Returns: This function returns a quiver which is a full subquiver of a *`quiver`* induced by the *`list`* of its vertices.

The names of vertices and arrows in resulting (sub)quiver remain the same as in original one. The function checks if *`list`* consists of vertices of *`quiver`*.

3.5.11 ConnectedComponents

▷ `ConnectedComponents(`*`quiver`*`)` (operation)

Returns: This function returns a list of quivers which are all connected components of a *`quiver`*.

The names of vertices and arrows in resulting (sub)quiver remain the same as in original one. The function sets the property `IsConnectedQuiver` (3.3.4) to true for all the components.

Example
```
gap> Q := Quiver(6, [ [1,2],[1,1],[3,2],[4,5],[4,5] ]);
<quiver with 6 vertices and 5 arrows>
gap> VerticesOfQuiver(Q);
[ v1, v2, v3, v4, v5, v6 ]
gap> FullSubquiver(Q, [Q.v1, Q.v2]);
<quiver with 2 vertices and 2 arrows>
gap> ConnectedComponents(Q);
[ <quiver with 3 vertices and 3 arrows>,
  <quiver with 2 vertices and 2 arrows>,
  <quiver with 1 vertices and 0 arrows> ]
```

3.5.12 SeparatedQuiver

▷ `SeparatedQuiver(`*`quiver`*`)` (attribute)

Arguments: *`quiver`* – a quiver.

Returns: the separated quiver of *`quiver`*.

The vertices in the separated quiver are labelled v and v' for each vertex v in *`quiver`*, and for each arrow $a\colon v \to w$ in *`quiver`* the arrow $v \to w'$ is labelled a.

3.6 Categories and Properties of Paths

3.6.1 IsPath

▷ `IsPath(object)` (category)

All path objects are in this category.

3.6.2 IsVertex

▷ `IsVertex(object)` (category)

All vertices are in this category.

3.6.3 IsArrow

▷ `IsArrow(object)` (category)

All arrows are in this category.

3.6.4 IsZeroPath

▷ `IsZeroPath(object)` (property)

is true when `object` is the zero path.

Example
```
gap> q1 := Quiver(["u","v"],[["u","u","a"],["u","v","b"],
>                 ["v","u","c"],["v","v","d"]]);
<quiver with 2 vertices and 4 arrows>
gap> IsPath(q1.b);
true
gap> IsPath(q1.u);
true
gap> IsVertex(q1.c);
false
gap> IsZeroPath(q1.d);
false
```

3.7 Attributes and Operations of Paths

3.7.1 SourceOfPath

▷ `SourceOfPath(path)` (attribute)

Returns: the source (first) vertex of `path`.

3.7.2 TargetOfPath

▷ `TargetOfPath(path)` (attribute)

Returns: the target (last) vertex of `path`.

3.7.3 LengthOfPath

▷ `LengthOfPath(path)` (attribute)

Returns: the length of `path`.

3.7.4 WalkOfPath

▷ `WalkOfPath(path)` (attribute)

Returns: a list of the arrows that constitute `path` in order.

3.7.5 *

▷ `*(p, q)` (operation)

Arguments: `p` and `q` – two paths in the same quiver.

Returns: the multiplication of the paths. If the paths are not in the same quiver an error is returned. If the target of `p` differs from the source of `q`, then the result is the zero path. Otherwise, if either path is a vertex, then the result is the other path. Finally, if both are paths of length at least 1, then the result is the concatenation of the walks of the two paths.

Example

```
gap> q1 := Quiver(["u","v"],[["u","u","a"],["u","v","b"],
>                ["v","u","c"],["v","v","d"]]);
<quiver with 2 vertices and 4 arrows>
gap> SourceOfPath(q1.v);
v
gap> p1:=q1.a*q1.b*q1.d*q1.d;
a*b*d^2
gap> TargetOfPath(p1);
v
gap> p2:=q1.b*q1.b;
0
gap> WalkOfPath(p1);
[ a, b, d, d ]
gap> WalkOfPath(q1.a);
[ a ]
gap> LengthOfPath(p1);
4
gap> LengthOfPath(q1.v);
0
```

3.7.6 =

▷ `=(p, q)` (operation)

Arguments: `p` and `q` – two paths in the same quiver.

Returns: true if the two paths are equal. Two paths are equal if they have the same source and the same target and if they have the same walks.

3.7.7 < (for two paths in a quiver)

▷ `<(p, q)` (operation)

Arguments: *p* and *q* – two paths in the same quiver.
Returns: a comparison of the two paths with respect to the ordering of the quiver.

3.8 Attributes of Vertices

3.8.1 IncomingArrowsOfVertex

▷ `IncomingArrowsOfVertex(vertex)` (attribute)

Returns: a list of arrows having *vertex* as target. Only meaningful if *vertex* is in a quiver.

3.8.2 OutgoingArrowsOfVertex

▷ `OutgoingArrowsOfVertex(vertex)` (attribute)

Returns: a list of arrows having *vertex* as source.

3.8.3 InDegreeOfVertex

▷ `InDegreeOfVertex(vertex)` (attribute)

Returns: the number of arrows having *vertex* as target. Only meaningful if *vertex* is in a quiver.

3.8.4 OutDegreeOfVertex

▷ `OutDegreeOfVertex(vertex)` (attribute)

Returns: the number of arrows having *vertex* as source.

3.8.5 NeighborsOfVertex

▷ `NeighborsOfVertex(vertex)` (attribute)

Returns: a list of neighbors of *vertex*, that is, vertices that are targets of arrows having *vertex* as source.

Example
```
gap> q1 := Quiver(["u","v"],[["u","u","a"],["u","v","b"],
>                ["v","u","c"],["v","v","d"]]);
<quiver with 2 vertices and 4 arrows>
gap> OutgoingArrowsOfVertex(q1.u);
[ a, b ]
gap> InDegreeOfVertex(q1.u);
2
gap> NeighborsOfVertex(q1.v);
[ u, v ]
```

Chapter 4

Path Algebras

4.1 Introduction

A path algebra is an algebra constructed from a field F (see Chapter 56 and 57 in the GAP manual for information about fields) and a quiver Q. The path algebra FQ contains all finite linear combinations of paths of Q. This chapter describes the functions in QPA that deal with path algebras and quotients of path algebras. Path algebras are algebras, so see Chapter 60: Algebras in the GAP manual for functionality such as generators, basis functions, and mappings.

The only supported ordering of elements in a path algebra is length left lexicographic ordering. See 3.4 for more information.

4.2 Constructing Path Algebras

4.2.1 PathAlgebra

▷ `PathAlgebra(F, Q)` (function)

Arguments: `F` – a field, `Q` – a quiver.

Returns: the path algebra `FQ` of `Q` over the field `F`.

For construction of fields, see the GAP documentation. The elements of the path algebra `FQ` will be ordered by left length-lexicographic ordering.

Example
```
gap> Q := Quiver( ["u","v"] , [ ["u","u","a"], ["u","v","b"],
> ["v","u","c"], ["v","v","d"] ] );
<quiver with 2 vertices and 4 arrows>
gap> F := Rationals;
Rationals
gap> FQ := PathAlgebra(F,Q);
<Rationals[<quiver with 2 vertices and 4 arrows>]>
```

4.3 Categories and Properties of Path Algebras

4.3.1 IsPathAlgebra

▷ `IsPathAlgebra(`*`object`*`)` (property)

Arguments: *`object`* – any object in GAP.
Returns: true whenever *`object`* is a path algebra.

Example
```
gap> IsPathAlgebra(FQ);
true
```

4.4 Attributes and Operations for Path Algebras

4.4.1 AssociatedMonomialAlgebra

▷ `AssociatedMonomialAlgebra(`*`A`*`)` (attribute)

Arguments: *`A`* – a quiver algebra.
Returns: the associated monomial algebra of *`A`* with respect to the Groebner basis the path algebra is endoved with.

4.4.2 QuiverOfPathAlgebra

▷ `QuiverOfPathAlgebra(`*`FQ`*`)` (attribute)

Arguments: *`FQ`* – a path algebra.
Returns: the quiver from which *`FQ`* was constructed.

Example
```
gap> QuiverOfPathAlgebra(FQ);
<quiver with 2 vertices and 4 arrows>
```

4.4.3 OrderingOfAlgebra

▷ `OrderingOfAlgebra(`*`FQ`*`)` (attribute)

Arguments: *`FQ`* – a path algebra.
Returns: the ordering of the quiver of the path algebra.
Note: As of the current version of QPA, only left length lexicographic ordering is supported.

4.4.4 . (for a path algebra)

▷ `.(`*`FQ, generator`*`)` (operation)

Arguments: *`FQ`* – a path algebra, *`generator`* – a vertex or an arrow in the quiver *`Q`*.
Returns: the *`generator`* as an element of the path algebra.
Other elements of the path algebra can be constructed as linear combinations of the generators. For further operations on elements, see below.

Example
```
gap> FQ.a;
(1)*a
gap> FQ.v;
(1)*v
gap> elem := 2*FQ.a - 3*FQ.v;
(-3)*v+(2)*a
```

4.5 Operations on Path Algebra Elements

4.5.1 ElementOfPathAlgebra

▷ `ElementOfPathAlgebra(PA, path)` (operation)

Arguments: `PA` – a path algebra, `path` – a path in the quiver from which `PA` was constructed.
Returns: The embedding of `path` into the path algebra `PA`, or it returns false if `path` is not an element of the quiver from which `PA` was constructed.

4.5.2 < (for two elements in a path algebra)

▷ `<(a, b)` (operation)

Arguments: `a` and `b` – two elements of the same path algebra.
Returns: True whenever `a` is smaller than `b`, according to the ordering of the path algebra.

4.5.3 IsLeftUniform

▷ `IsLeftUniform(element)` (operation)

Arguments: `element` – an element of the path algebra.
Returns: true if each monomial in `element` has the same source vertex, false otherwise.

4.5.4 IsRightUniform

▷ `IsRightUniform(element)` (operation)

Arguments: `element` – an element of the path algebra.
Returns: true if each monomial in `element` has the same target vertex, false otherwise.

4.5.5 IsUniform

▷ `IsUniform(element)` (operation)

Arguments: `element` – an element of the path algebra.
Returns: true whenever `element` is both left and right uniform.

Example
```
gap> IsLeftUniform(elem);
false
```

```
gap> IsRightUniform(elem);
false
gap> IsUniform(elem);
false
gap> another := FQ.a*FQ.b + FQ.b*FQ.d*FQ.c*FQ.b*FQ.d;
(1)*a*b+(1)*b*d*c*b*d
gap> IsLeftUniform(another);
true
gap> IsRightUniform(another);
true
gap> IsUniform(another);
true
```

4.5.6 LeadingTerm

▷ `LeadingTerm(element)` (operation)
▷ `Tip(element)` (operation)

Arguments: *element* – an element of the path algebra.

Returns: the term in *element* whose monomial is largest among those monomials that have nonzero coefficients (known as the "tip" of *element*).

Note: The two operations are equivalent.

4.5.7 LeadingCoefficient

▷ `LeadingCoefficient(element)` (operation)
▷ `TipCoefficient(element)` (operation)

Arguments: *element* – an element of the path algebra.

Returns: the coefficient of the tip of *element* (which is an element of the field).

Note: The two operations are equivalent.

4.5.8 LeadingMonomial

▷ `LeadingMonomial(element)` (operation)
▷ `TipMonomial(element)` (operation)

Arguments: *element* – an element of the path algebra.

Returns: the monomial of the tip of *element* (which is an element of the underlying quiver, not of the path algebra).

Note: The two operations are equivalent.

Example
```
gap> elem := FQ.a*FQ.b*FQ.c + FQ.b*FQ.d*FQ.c+FQ.d*FQ.d;
(1)*d^2+(1)*a*b*c+(1)*b*d*c
gap> LeadingTerm(elem);
(1)*b*d*c
gap> LeadingCoefficient(elem);
1
gap> mon := LeadingMonomial(elem);
```

```
b*d*c
gap> mon in FQ;
false
gap> mon in Q;
true
```

4.5.9 MakeUniformOnRight

▷ MakeUniformOnRight(*elems*) (operation)

Arguments: *elems* – a list of elements in a path algebra.
Returns: a list of right uniform elements generated by each element of *elems*.

4.5.10 MappedExpression

▷ MappedExpression(*expr, gens1, gens2*) (operation)

Arguments: *expr* – element of a path algebra, *gens1* and *gens2* – equal-length lists of generators for subalgebras.
Returns: *expr* as an element of the subalgebra generated by *gens2*.
The element *expr* must be in the subalgebra generated by *gens1*. The lists define a mapping of each generator in *gens1* to the corresponding generator in *gens2*. The value returned is the evaluation of the mapping at *expr*.

4.5.11 VertexPosition

▷ VertexPosition(*element*) (operation)

Arguments: *element* – an element of the path algebra on the form $k*v$, where v is a vertex of the underlying quiver and k is an element of the field.
Returns: the position of the vertex `v` in the list of vertices of the quiver.

4.6 Constructing Quotients of Path Algebras

In the introduction we saw already one way of constructing a quotient of a path algebra. In addition to this there are at least two other ways of constructing a quotient of a path algebra; one with factoring out an ideal and one where a Groebner basis is attached to the quotient. We discuss these two next.

For several functions in QPA to function properly one needs to have a Groebner basis attached to the quotient one wants to construct, or equivalently a Groebner basis for the ideal one is factoring out. For this to work the ideal must admit a finite Groebner basis. However, to our knowlegde there is no algorithm for determining if an ideal has a finite Groebner basis. On the other hand, it is known that if the factor algebra is finite dimensional, then the ideal has a finite Groebner basis (independent of the ordering of the elements, see [Gre00]). In addition to having a finite Groebner basis, several functions also need that the factoring ideal is admissible. A quotient of a path algebra by an admissible ideal belongs to the category IsAdmissibleQuotientOfPathAlgebra (4.11.1). The method used in the introduction constructs a quotient in this category. However, there are situations where it is interesting to analyze quotients of path algebras by a non-admissible ideal, so we provide also additional methods.

In the example below, we construct a factor of a path algebra purely with commands in GAP (cf. also Chapter 60: Algebras in the GAP manual on how to construct an ideal and a quotient of an algebra). Functions which use Groebner bases like `IsFiniteDimensional` (4.11.3), `Dimension` (4.12.6), `IsSpecialBiserialAlgebra` (4.11.19) or a membership test `\in` (4.7.5) will work properly (they simply compute the Groebner basis if it is necessary). But some "older" functions (like `IndecProjectiveModules` (6.5.3)) can fail or give an incorrect answer! This way of constructing a quotient of a path algebra can be useful e.g. if we know that computing a Groebner basis will take a long time and we do not need this because we want to deal only with modules.

Example
```
gap> Q := Quiver( 1, [ [1,1,"a"], [1,1,"b"] ] );
<quiver with 1 vertices and 2 arrows>
gap> kQ := PathAlgebra(Rationals, Q);
<Rationals[<quiver with 1 vertices and 2 arrows>]>
gap> gens := GeneratorsOfAlgebra(kQ);
[ (1)*v1, (1)*a, (1)*b ]
gap> a := gens[2];
(1)*a
gap> b := gens[3];
(1)*b
gap> relations := [a^2,a*b-b*a, b*b];
[ (1)*a^2, (1)*a*b+(-1)*b*a, (1)*b^2 ]
gap> I := Ideal(kQ,relations);
<two-sided ideal in <Rationals[<quiver with 1 vertices and 2 arrows>]>
    , (3 generators)>
gap> A := kQ/I;
<Rationals[<quiver with 1 vertices and 2 arrows>]/
<two-sided ideal in <Rationals[<quiver with 1 vertices and 2 arrows>]>
    , (3 generators)>>
gap> IndecProjectiveModules(A);
Compute a Groebner basis of the ideal you are factoring out with befor\
e you form the quotient algebra, or you have entered an algebra which \
is not finite dimensional.
fail
```

To resolve this matter, we need to compute the Gröbner basis of the ideal generated by the relations in *kQ* (yes, it seems like we are going in circles here. Remember, then, that an ideal in the "mathematical sense" may exist independently of the a corresponding `Ideal` object in GAP. Also, Gröbner bases in QPA are handled by the GBNP package, with constructor methods not dependent on `Ideal` objects. After creating the ideal *I*, we need to perform yet another Gröbner basis operation which just set a respective attribute for *I*, see `GroebnerBasis` (5.1.2).

Example
```
gap> gb := GBNPGroebnerBasis(relations,kQ);
[ (1)*a^2, (-1)*a*b+(1)*b*a, (1)*b^2 ]
gap> I := Ideal(kQ,gb);
<two-sided ideal in <Rationals[<quiver with 1 vertices and 2 arrows>]>,
  (3 generators)>
gap> GroebnerBasis(I,gb);
<complete two-sided Groebner basis containing 3 elements>
gap> IndecProjectiveModules(A);
fail
gap> A := kQ/I;
```

```
<Rationals[<quiver with 1 vertices and 2 arrows>]/
<two-sided ideal in <Rationals[<quiver with 1 vertices and 2 arrows>]>,
  (3 generators)>>
gap> IndecProjectiveModules(A);
[ <[ 4 ]> ]
```

Note that the instruction `A := kQ/relations;` used in Introduction is exactly an abbreviation for a sequence of instructions with Groebner basis as in above example.

Most QPA operations working on algebras handle path algebras and quotients of path algebras in the same way (when this makes sense). However, there are still a few operations which does not work properly when given a quotient of a path algebra. When constructing a quotient of a path algebra one needs define the ideal one is factoring out. Above this has been done with the commands

Example
```
gap> gens := GeneratorsOfAlgebra(kQ);
[ (1)*v1, (1)*a, (1)*b ]
gap> a := gens[2];
(1)*a
gap> b := gens[3];
(1)*b
```

The following command makes this process easier.

4.6.1 AssignGeneratorVariables

▷ `AssignGeneratorVariables(A)` (operation)

Arguments: `A` – a quiver algebra.

Returns: Takes a quiver algebra `A` as an argument and creates variables, say $v_1,\ldots,v_n$ for the vertices, and $a_1,\ldots,a_t$ for the arrows for the corresponding elements in `A`, whenever the quiver for the quiver algebra `A` is was constructed with the vertices being named $v_1,\ldots,v_n$ and the arrows being named $a_1,\ldots,a_t$.

Here is an example of its use.

Example
```
gap> AssignGeneratorVariables(kQ);
#I  Assigned the global variables [ v1, a, b ]
gap> v1; a; b;
(1)*v1
(1)*a
(1)*b
```

4.7 Ideals and operations on ideals

4.7.1 Ideal

▷ `Ideal(FQ, elems)` (function)

Arguments: `FQ` – a path algebra, `elems` – a list of elements in `FQ`.

Returns: the ideal of `FQ` generated by `elems` with the property `IsIdealInPathAlgebra` (4.8.2).

For more on ideals, see the GAP reference manual (Chapter 60.6).
Technical info: Ideal is a synonym for a global GAP function TwoSidedIdeal which calls an operation TwoSidedIdealByGenerators (synonym IdealByGenerators) for an algebra (FLMLOR).

Example
```
gap> gb := GBNPGroebnerBasis(relations,kQ);
[ (1)*a^2, (-1)*a*b+(1)*b*a, (1)*b^2 ]
gap> I := Ideal(kQ,gb);
<two-sided ideal in <Rationals[<quiver with 1 vertices and 2 arrows>]>
    , (3 generators)>
gap> GroebnerBasis(I,gb);
<complete two-sided Groebner basis containing 3 elements>
gap> IndecProjectiveModules(A);
[ <[ 4 ]> ]
gap> A := kQ/I;
<Rationals[<quiver with 1 vertices and 2 arrows>]/
<two-sided ideal in <Rationals[<quiver with 1 vertices and 2 arrows>]>
    , (3 generators)>>
gap> IndecProjectiveModules(A);
[ <[ 4 ]> ]
true
```

4.7.2 PathsOfLengthTwo

▷ `PathsOfLengthTwo(Q)` (operation)

Arguments: *Q* – a quiver.
Returns: a list of all paths of length two in *Q*, sorted by <. Fails with error message if *Q* is not a Quiver object.

4.7.3 NthPowerOfArrowIdeal

▷ `NthPowerOfArrowIdeal(FQ, n)` (operation)

Arguments: *FQ* – a path algebra, *n* – a positive integer.
Returns: the ideal generated all the paths of length *n* in *FQ*.

4.7.4 AddNthPowerToRelations

▷ `AddNthPowerToRelations(FQ, rels, n)` (operation)

Arguments: *FQ* – a path algebra, *rels* – a (possibly empty) list of elements in *FQ*, *n* – a positive integer.
Returns: the list *rels* with the paths of length *n* of *FQ* appended (will change the list *rels*).

4.7.5 \in (elt. in path alg. and ideal)

▷ `\in(elt, I)` (operation)

Arguments: `elt` - an element in a path algebra, `I` - an ideal in the same path algebra (i.e. with `IsIdealInPathAlgebra` (4.8.2) property).

Returns: true, if `elt` belongs to `I`.

It performs the membership test for an ideal in path algebra using completely reduced Groebner bases machinery.

Technical info: For the efficiency reasons, it computes Groebner basis for `I` only if it has not been computed yet. Similarly, it performs CompletelyReduceGroebnerBasis only if it has not been reduced yet. The method can change the existing Groebner basis.

Remark: It works only in case `I` is in the arrow ideal.

4.8 Categories and properties of ideals

4.8.1 IsAdmissibleIdeal

▷ `IsAdmissibleIdeal(`*`I`*`)` (property)

Arguments: `I` – an IsIdealInPathAlgebra object.

Returns: true whenever `I` is an *admissible* ideal in a path algebra, i.e. `I` is a subset of J^2 and `I` contains J^n for some n, where J is the arrow ideal.

Technical note: The second condition is checked by the nilpotency index of the radical and checking if the ideal generated by the arrows to one plus this index is in the ideal of the relations (this uses Groebner bases machinery).

4.8.2 IsIdealInPathAlgebra

▷ `IsIdealInPathAlgebra(`*`I`*`)` (property)

Arguments: `I` – an IsFLMLOR object.

Returns: true whenever `I` is an ideal in a path algebra.

4.8.3 IsMonomialIdeal

▷ `IsMonomialIdeal(`*`I`*`)` (property)

Arguments: `I` – an IsIdealInPathAlgebra object.

Returns: true whenever `I` is a *monomial* ideal in a path algebra, i.e. `I` is generated by a set of monomials (= "zero-relations").

Technical note: It uses the observation: `I` is a monomial ideal iff Groebner basis of `I` is a set of monomials. It computes Groebner basis for `I` only in case it has not been computed yet and a usual set of generators (GeneratorsOfIdeal) is not a set of monomials.

4.8.4 IsQuadraticIdeal

▷ `IsQuadraticIdeal(`*`rels`*`)` (operation)

Arguments: `rels` – a list of elements in a path algebra.

Returns: true whenever `rels` is a list of elements in the linear span of degree two elements of a path algebra. It returns false whenever `rels` is a list of elements in a path algebra, but not in the linear span of degree two of a path algebra. Otherwise it returns fail.

4.9 Operations on ideals

4.9.1 ProductOfIdeals

▷ `ProductOfIdeals(I, J)` (operation)

Arguments: `I, J` – two ideals in a path algebra `KQ`.

Returns: the ideal formed by the product of the ideals `I` and `J`, whenever the ideal `J` admits finitely many nontips in `KQ`.

The function checks if the two ideals are ideals in the same path algebra and that `J` admits finitely many nontips in `KQ`.

4.9.2 QuadraticPerpOfPathAlgebraIdeal

▷ `QuadraticPerpOfPathAlgebraIdeal(rels)` (operation)

Arguments: `rels` – a list of elements in a path algebra.

Returns: fail if `rels` is not a list of elements in the linear span of degree two elements of a path algebra `KQ`. Otherwise it returns a list of length two, where the first element is a set of generators for the ideal `rels`$^{\perp}$ in opposite algebra of `KQ` and the second element is the opposite algebra of `KQ`.

4.10 Attributes of ideals

For many of the functions related to quotients, you will need to compute a Groebner basis of the ideal. This is done with the GBNP package. The following example shows how to set a Groebner basis for an ideal (note that this must be done before the quotient is constructed). See the next two chapters for more on Groebner bases.

Example
```
gap> rels := [FQ.a - FQ.b*FQ.c, FQ.d*FQ.d];
[ (1)*a+(-1)*b*c, (1)*d^2 ]
gap> gb := GBNPGroebnerBasis(rels, FQ);
[ (-1)*a+(1)*b*c, (1)*d^2 ]
gap> I := Ideal(FQ, gb);
<two-sided ideal in <Rationals[<quiver with 2 vertices and 4 arrows>]>
    , (2 generators)>
gap> GroebnerBasis(I, gb);
<complete two-sided Groebner basis containing 2 elements>
gap> quot := FQ/I;
<Rationals[<quiver with 2 vertices and 4 arrows>]/
<two-sided ideal in <Rationals[<quiver with 2 vertices and 4 arrows>]>
    , (2 generators)>>
```

4.10.1 GroebnerBasisOfIdeal

▷ `GroebnerBasisOfIdeal(I)` (attribute)

Arguments: `I` – an ideal in path algebra.
Returns: a Groebner basis of ideal `I` (if it has been already computed!).
This attribute is set only by an operation `GroebnerBasis` (5.1.2).

4.11 Categories and Properties of Quotients of Path Algebras

4.11.1 IsAdmissibleQuotientOfPathAlgebra

▷ `IsAdmissibleQuotientOfPathAlgebra(A)` (filter)

Arguments: `A` – any object.
Returns: true whenever `A` is a quotient of a path algebra by an admissible ideal constructed by the command `\/` with arguments a path algebra and a list of relations, `KQ/rels`, where `rels` is a list of relations. Otherewise it returns an error message.

4.11.2 IsQuotientOfPathAlgebra

▷ `IsQuotientOfPathAlgebra(object)` (property)

Argument: *object* – any object in GAP.
Returns: true whenever *object* is a quotient of a path algebra.

Example
```
gap> quot := FQ/I;
<Rationals[<quiver with 2 vertices and 4 arrows>]/
<two-sided ideal in <Rationals[<quiver with 2 vertices and 4 arrows>]>
    , (2 generators)>>
gap> IsQuotientOfPathAlgebra(quot);
true
gap> IsQuotientOfPathAlgebra(FQ);
false
```

4.11.3 IsFiniteDimensional

▷ `IsFiniteDimensional(A)` (property)

Arguments: `A` – a path algebra or a quotient of a path algebra.
Returns: true whenever `A` is a finite dimensional algebra.
Technical note: For a path algebra it uses a standard GAP method. For a quotient of a path algebra it uses Groebner bases machinery (it computes Groebner basis for the ideal only in case it has not been computed yet).

4.11.4 IsCanonicalAlgebra

▷ `IsCanonicalAlgebra(A)` (property)

Arguments: `A` – a path algebra or a quotient of a path algebra.
Returns: true if `A` has been constructed by the operation `CanonicalAlgebra` (4.14.1), otherwise "Error, no method found".

4.11.5 IsDistributiveAlgebra

▷ `IsDistributiveAlgebra(A)` (property)

Arguments: `A` – a path algebra or a quotient of a path algebra.
Returns: true if `A` is finite dimensional and distributive. Otherwise it returns false.

4.11.6 IsFiniteGlobalDimensionAlgebra

▷ `IsFiniteGlobalDimensionAlgebra(A)` (property)

Arguments: `A` - an algebra over a field.
Returns: true if it is known that the entered algebra `A` has finite global dimension.
There is no method associated to this, so if it is not known that the algebra has finite global dimension, then an error message saying "no method found!" is return.

4.11.7 IsGentleAlgebra

▷ `IsGentleAlgebra(A)` (property)

Arguments: `A` – a path algebra or a quotient of a path algebra.
Returns: true if the algebra `A` is a gentle algebra. Otherwise false.

4.11.8 IsGorensteinAlgebra

▷ `IsGorensteinAlgebra(A)` (property)

Arguments: `A` – an algebra.
Returns: true if it is known that `A` is a Gorenstein algebra. If unknown it returns an error message saying "no method found!".
There is no method installed for this yet.

4.11.9 IsHereditaryAlgebra

▷ `IsHereditaryAlgebra(A)` (property)

Arguments: `A` – an admissible quotient of a path algebra.
Returns: true if `A` is a hereditary algebra and false otherwise.

4.11.10 IsKroneckerAlgebra

▷ `IsKroneckerAlgebra(A)` (property)

Arguments: `A` – a path algebra or a quotient of a path algebra.
Returns: true if `A` has been constructed by the operation `KroneckerAlgebra` (4.14.2), otherwise "Error, no method found".

4.11.11 IsMonomialAlgebra

▷ `IsMonomialAlgebra(A)` (property)

Arguments: *A* – a quiver algebra.
Returns: true when *A* is given as `kQ/I` and `I` is a monomial ideal in `kQ`, otherwise it returns false.

4.11.12 IsNakayamaAlgebra

▷ `IsNakayamaAlgebra(A)` (property)

Arguments: *A* – a path algebra or a quotient of a path algebra.
Returns: true if *A* has been constructed by the operation `NakayamaAlgebra` (4.14.3), otherwise "Error, no method found".

4.11.13 IsQuiverAlgebra

▷ `IsQuiverAlgebra(A)` (filter)

Arguments: *A* – an algebra.
Returns: true if *A* is a path algebra or a quotient of a path algebra algebra, otherwise false.

4.11.14 IsRadicalSquareZeroAlgebra

▷ `IsRadicalSquareZeroAlgebra(A)` (property)

Arguments: *A* – an algebra.
Returns: true if *A* is a radical square zero algebra, otherwise false.

4.11.15 IsSchurianAlgebra

▷ `IsSchurianAlgebra(A)` (property)

Arguments: *A* – a path algebra or a quotient of a path algebra.
Returns: true if *A* is a schurian algebra. By definition it means that: for all $x,y \in Q_0$ we have $\dim A(x,y) \leq 1$.
Note: This method fail when a Groebner basis for ideal has not been computed before creating a quotient!

4.11.16 IsSelfinjectiveAlgebra

▷ `IsSelfinjectiveAlgebra(A)` (property)

Arguments: *A* – a path algebra or a quotient of a path algebra.
Returns: fail if *A* is not finite dimensional. Otherwise it returns true or false according to whether *A* is selfinjective or not.

4.11.17 IsSemicommutativeAlgebra

▷ `IsSemicommutativeAlgebra(A)` (property)

Arguments: `A` – a path algebra or a quotient of a path algebra.

Returns: true if `A` is a semicommutative algebra. By definition it means that:
1. `A` is schurian (cf. `IsSchurianAlgebra` (4.11.15)).
2. Quiver Q of `A` is acyclic (cf. `IsAcyclicQuiver` (3.3.2)).
3. For all pairs of vertices (x,y) the following condition is satisfied: for every two paths P,P' from x to y: $P \in I \Leftrightarrow P' \in I$.

Note: This method fail when a Groebner basis for ideal has not been computed before creating a quotient!

4.11.18 IsSemisimpleAlgebra

▷ `IsSemisimpleAlgebra(A)` (property)

Arguments: `A` - an algebra over a field.

Returns: true if the entered algebra `A` is semisimple, false otherwise.

Checks if the algebra is finite dimensional. If it is an admissible quotients of a path algebra, it only checks if the underlying quiver has any arrows or not. Otherwise, it computes the radical of the algebra and checks if it is zero.

4.11.19 IsSpecialBiserialAlgebra

▷ `IsSpecialBiserialAlgebra(A)` (property)

Arguments: `A` – a path algebra or a quotient of a path algebra.

Returns: true whenever `A` is a *special biserial algebra*, i.e. `A=KQ/I`, where `Q` is `IsSpecialBiserialQuiver` (4.14.5), `I` is an admissible ideal (`IsAdmissibleIdeal` (4.8.1)) and `I` satisfies the "special biserial" conditions, i.e.:

- for any arrow a there exists at most one arrow b such that ab does not belong to `I`
- there exists at most one arrow c such that ca does not belong to I.

Note: e.g. a path algebra of one loop IS NOT special biserial, but one loop IS special biserial quiver (see `IsSpecialBiserialQuiver` (4.14.5) for examples).

4.11.20 IsStringAlgebra

▷ `IsStringAlgebra(A)` (property)

Arguments: `A` – a path algebra or a quotient of a path algebra.

Returns: true whenever `A` is a *string* (special biserial) algebra, i.e. `A=KQ/I` is a special biserial algebra (`IsSpecialBiserialAlgebra` (4.11.19) and `I` is generated by monomials (= "zero-relations") (cf. `IsMonomialIdeal` (4.8.3)). See `IsSpecialBiserialQuiver` (4.14.5) for examples.

4.11.21 IsSymmetricAlgebra

▷ IsSymmetricAlgebra(*A*) (property)

Arguments: *A* – a path algebra or a quotient of a path algebra.

Returns: fail if *A* is not finite dimensional or does not have a Groebner basis. Otherwise it returns true or false according to whether *A* is symmetric or not.

4.11.22 IsWeaklySymmetricAlgebra

▷ IsWeaklySymmetricAlgebra(*A*) (property)

Arguments: *A* – a path algebra or a quotient of a path algebra.

Returns: fail if *A* is not finite dimensional or does not have a Groebner basis. Otherwise it returns true or false according to whether *A* is weakly symmetric or not.

4.11.23 IsFiniteTypeAlgebra

▷ IsFiniteTypeAlgebra(*A*) (property)

Arguments: *A* – a path algebra or a quotient of a path algebra.

Returns: Returns true if *A* is of finite representation type. Returns false if *A* is of infinite representation type. Returns fail if we can not determine the representation type (i.e. it impossible from theoretical/algorithmic point of view or a suitable criterion has not been implemented yet; the implementation is in progress). Note: in case *A* is a path algebra the function is completely implemented.

Example
```
gap> Q := Quiver(5, [ [1,2,"a"], [2,4,"b"], [3,2,"c"], [2,5,"d"] ]);
<quiver with 5 vertices and 4 arrows>
gap> A := PathAlgebra(Rationals, Q);
<Rationals[<quiver with 5 vertices and 4 arrows>]>
gap> IsFiniteTypeAlgebra(A);
Infinite type!
Quiver is not a (union of) Dynkin quiver(s).
false
gap> quo := A/[A.a*A.b, A.c*A.d];;
gap> IsFiniteTypeAlgebra(quo);
Finite type!
Special biserial algebra with no unoriented cycles in Q.
true
```

4.12 Attributes and Operations (for Quotients) of Path Algebras

4.12.1 CartanMatrix

▷ CartanMatrix(*A*) (operation)

Arguments: *A* – a path algebra or a quotient of a path algebra.

Returns: the Cartan matrix of the algebra *A*, after having checked that *A* is a finite dimensional quotient of a path algebra.

4.12.2 Centre/Center

▷ `Centre/Center(A)` (operation)

Arguments: `A` – a path algebra or a quotient of a path algebra.

Returns: the centre of the algebra `A` as a subalgebra.

The function checks if `A` is a finite dimensional quotient of a path algebra. The function is not yet implemented for path algebras. The answer in this case is on copy of the field for each connected component of the quiver of that path algebra, given that it is finite dimensional (ie. no oriented cycles).

4.12.3 ComplexityOfAlgebra

▷ `ComplexityOfAlgebra(A, n)` (operation)

Arguments: `A` – a path algebra or a quotient of a path algebra, `n` – a positive integer.

Returns: an estimate of the complexity of the algebra `A`.

The function checks if the algebra `A` is known to have finite global dimension. If so, it returns complexity zero. Otherwise it tries to estimate the complexity in the following way. Recall that if a function $f(x)$ is a polynomial in x, the degree of $f(x)$ is given by $\lim_{n\to\infty} \frac{\log|f(n)|}{\log n}$. So then this function computes an estimate of the maximal complexity of the simple modules over `A` by approximating the complexity of each simple module S by considering the limit $\lim_{m\to\infty} \log \frac{\dim(P(S)(m))}{\log m}$ where $P(S)(m)$ is the m-th projective in a minimal projective resolution of `S` at stage m. This limit is estimated by $\frac{\log \dim(P(S)(n))}{\log n}$.

4.12.4 CoxeterMatrix

▷ `CoxeterMatrix(A)` (attribute)

Arguments: `A` – a path algebra or a quotient of a path algebra.

Returns: the Coxeter matrix of the algebra `A`, after having checked that `A` is a finite dimensional quotient of a path algebra.

4.12.5 CoxeterPolynomial

▷ `CoxeterPolynomial(A)` (attribute)

Arguments: `A` – a path algebra or a quotient of a path algebra.

Returns: the Coxeter polynomial of the algebra `A`, after having checked that `A` is a finite dimensional quotient of a path algebra.

4.12.6 Dimension

▷ `Dimension(A)` (attribute)

Arguments: `A` – a path algebra or a quotient of a path algebra.

Returns: the dimension of the algebra `A` or *infinity* in case `A` is an infinite dimensional algebra.

For a quotient of a path algebra it uses Groebner bases machinery (it computes Groebner basis for the ideal only in case it has not been computed yet).

4.12.7 GlobalDimension

▷ `GlobalDimension(A)` (attribute)

Arguments: `A` – an algebra.

Returns: the global dimension of the algebra `A` if it is known. Otherwise it returns an error message saying "no method found!".

There is no method installed for this yet.

4.12.8 LoewyLength

▷ `LoewyLength(A)` (attribute)

Arguments: `A` – a path algebra or a quotient of a path algebra.

Returns: fail if `A` is not finite dimensional. Otherwise it returns the Loewy length of the algebra `A`.

4.12.9 NakayamaAutomorphism

▷ `NakayamaAutomorphism(A)` (attribute)

Arguments: `A` – a path algebra or a quotient of a path algebra.

Returns: false if `A` is not selfinjective algebra. Otherwise it returns the Nakayama automorphism of `A`.

4.12.10 NakayamaPermutation

▷ `NakayamaPermutation(A)` (attribute)

Arguments: `A` – a path algebra or a quotient of a path algebra.

Returns: false if `A` is not selfinjective algebra. Otherwise it returns a list of two elements where the first is the Nakayama permutation on the simple modules and the second is the Nakayama permutation on the index set of the simple modules of `A`.

4.12.11 OrderOfNakayamaAutomorphism

▷ `OrderOfNakayamaAutomorphism(A)` (attribute)

Arguments: `A` – a path algebra or a quotient of a path algebra.

Returns: false if `A` is not selfinjective algebra. Otherwise it returns the order of the Nakayama autormorphism of `A`.

4.12.12 RadicalSeriesOfAlgebra

▷ `RadicalSeriesOfAlgebra(A)` (attribute)

Arguments: `A` – an algebra.

Returns: the radical series of the algebra `A` in a list, where the first element is the algebra `A` itself, then radical of `A`, radical square of `A`, and so on.

4.13 Attributes and Operations on Elements of Quotients of Path Algebra

4.13.1 IsElementOfQuotientOfPathAlgebra

▷ `IsElementOfQuotientOfPathAlgebra(`*object*`)` (property)

Arguments: *object* – any object in GAP.
Returns: true whenever *object* is an element of some quotient of a path algebra.

Example
```
gap> elem := quot.a*quot.b;
[(1)*a*b]
gap> IsElementOfQuotientOfPathAlgebra(elem);
true
gap> IsElementOfQuotientOfPathAlgebra(FQ.a*FQ.b);
false
```

4.13.2 Coefficients

▷ `Coefficients(`*B, element*`)` (operation)

Arguments: *B, element* – a basis for a quotient of a path algebra and element thereof.
Returns: the coefficients of the *element* in terms of the canonical basis *B* of the quotient of a path algebra in which *element* is an element.

4.13.3 IsNormalForm

▷ `IsNormalForm(`*element*`)` (operation)

Arguments: *element* – an element of a path algebra.
Returns: true if *element* is known to be in normal form.

Example
```
 gap> IsNormalForm(elem);
 true
```

4.13.4 < (for two elements of a path algebra)

▷ `<(a, `*b*`)` (operation)

Arguments: `a` and *b* – elements from a path algebra.
Returns: true whenever `a` < *b*.

4.13.5 ElementOfQuotientOfPathAlgebra

▷ `ElementOfQuotientOfPathAlgebra(`*family, element, computenormal*`)` (operation)

Arguments: *family* – a family of elements, *element* – an element of a path algebra, *computenormal* – true or false.

Returns: The projection of `element` into the quotient given by `family`. If `computenormal` is true, then the normal form of the projection of `element` is returned.

`family` is the ElementsFamily of the family of the algebra `element` is projected into.

4.13.6 OriginalPathAlgebra

▷ `OriginalPathAlgebra(algebra)` (attribute)

Arguments: `algebra` – an algebra.

Returns: a path algebra.

If `algebra` is a quotient of a path algebra or just a path algebra itself, the returned algebra is the path algebra it was constructed from. Otherwise it returns an error saying that the algebra entered was not given as a quotient of a path algebra.

4.14 Predefined classes and classes of (quotients of) path algebras

4.14.1 CanonicalAlgebra

▷ `CanonicalAlgebra(field, weights[, relcoeff])` (operation)

Arguments: `field` – a field, `weights` – a list of positive integers, [, `relcoeff` – a list of non-zero elements in the field.

Returns: the canonical algebra over the `field` with the quiver given by the weight sequence `weights` and the relations given by the coefficients `relcoeff`.

It function checks if all the `weights` are greater or equal to two, the number of weights is at least two, the number of coefficients is the number of `weights` - 2, the coefficients for the relations are in field and non-zero. If only the two first arguments are given, then the number of weights must be two.

4.14.2 KroneckerAlgebra

▷ `KroneckerAlgebra(field, n)` (operation)

Arguments: `field` – a field, `n` – a positive integer.

Returns: the `n`-Kronecker algebra over the field `field`.

It function checks if the number `n` of arrows is greater or equal to two and returns an error message if not.

4.14.3 NakayamaAlgebra

▷ `NakayamaAlgebra(admiss-seq, field)` (function)

Arguments: `field` – a field, `admiss-seq` – a list of positive integers.

Returns: the Nakayama algebra corresponding to the admissible sequence `admiss-seq` over the field `field`. If the entered sequence is not an admissible sequence, the sequence is returned.

The `admiss-seq` consists of the dimensions of the projective representations.

Example
```
gap> alg := NakayamaAlgebra([2,1], Rationals);
<Rationals[<quiver with 2 vertices and 1 arrows>]>
gap> QuiverOfPathAlgebra(alg);
<quiver with 2 vertices and 1 arrows>
```

4.14.4 TruncatedPathAlgebra

▷ `TruncatedPathAlgebra(F, Q, n)` (operation)

Arguments: F – a field, Q – a quiver, n – a positive integer.

Returns: the truncated path algebra KQ/I, where I is the ideal generated by all paths of length n in KQ.

4.14.5 IsSpecialBiserialQuiver

▷ `IsSpecialBiserialQuiver(Q)` (property)

Arguments: Q – a quiver.

Returns: true whenever Q is a *"special biserial"* quiver, i.e. every vertex in Q is a source (resp. target) of at most 2 arrows.

Note: e.g. a path algebra of one loop IS NOT special biserial, but one loop IS special biserial quiver (cf. `IsSpecialBiserialAlgebra` (4.11.19) and also an Example below).

Example
```
gap> Q := Quiver(1, [ [1,1,"a"], [1,1,"b"] ]);;
gap> A := PathAlgebra(Rationals, Q);;
gap> IsSpecialBiserialAlgebra(A); IsStringAlgebra(A);
false
false
gap> rel1 := [A.a*A.b, A.a^2, A.b^2];
[ (1)*a*b, (1)*a^2, (1)*b^2 ]
gap> quo1 := A/rel1;;
gap> IsSpecialBiserialAlgebra(quo1); IsStringAlgebra(quo1);
true
true
gap> rel2 := [A.a*A.b-A.b*A.a, A.a^2, A.b^2];
[ (1)*a*b+(-1)*b*a, (1)*a^2, (1)*b^2 ]
gap> quo2 := A/rel2;;
gap> IsSpecialBiserialAlgebra(quo2); IsStringAlgebra(quo2);
true
false
gap> rel3 := [A.a*A.b+A.b*A.a, A.a^2, A.b^2, A.b*A.a];
[ (1)*a*b+(1)*b*a, (1)*a^2, (1)*b^2, (1)*b*a ]
gap> quo3 := A/rel3;;
gap> IsSpecialBiserialAlgebra(quo3); IsStringAlgebra(quo3);
true
true
gap> rel4 := [A.a*A.b, A.a^2, A.b^3];
[ (1)*a*b, (1)*a^2, (1)*b^3 ]
gap> quo4 := A/rel4;;
gap> IsSpecialBiserialAlgebra(quo4); IsStringAlgebra(quo4);
false
false
```

4.15 Opposite algebras

4.15.1 OppositePath

▷ OppositePath(*p*) (operation)

Arguments: p – a path.
Returns: the path corresponding to p in the opposite quiver.
The following example illustrates the use of OppositeQuiver (3.5.9) and OppositePath (4.15.1).

Example
```
gap> Q := Quiver( [ "u", "v" ], [ [ "u", "u", "a" ],
>               [ "u", "v", "b" ] ] );
<quiver with 2 vertices and 2 arrows>
gap> Qop := OppositeQuiver(Q);
<quiver with 2 vertices and 2 arrows>
gap> VerticesOfQuiver( Qop );
[ u_op, v_op ]
gap> ArrowsOfQuiver( Qop );
[ a_op, b_op ]
gap> OppositePath( Q.a * Q.b );
b_op*a_op
gap> IsIdenticalObj( Q, OppositeQuiver( Qop ) );
true
gap> OppositePath( Qop.b_op * Qop.a_op );
a*b
```

4.15.2 OppositePathAlgebra

▷ OppositePathAlgebra(*A*) (attribute)

Arguments: A – a path algebra or quotient of path algebra.
Returns: the opposite algebra A^{op}.
This attribute contains the opposite algebra of an algebra.
The opposite algebra of a path algebra is the path algebra over the opposite quiver (as given by OppositeQuiver (3.5.9)). The opposite algebra of a quotient of a path algebra has the opposite quiver and the opposite relations of the original algebra.
The function OppositePathAlgebraElement (4.15.3) takes an algebra element to the corresponding element in the opposite algebra.
The opposite of the opposite of an algebra A is isomorphic to A. In QPA, we regard these two algebras to be the same, so the call OppositePathAlgebra(OppositePathAlgebra(A)) returns the object A.

4.15.3 OppositePathAlgebraElement

▷ `OppositePathAlgebraElement(x)` (function)

Arguments: *x* – a path.

Returns: the element corresponding to *x* in the opposite algebra.

The following example illustrates the use of `OppositePathAlgebra` (4.15.2) and `OppositePathAlgebraElement` (4.15.3).

Example
```
gap> Q := Quiver( [ "u", "v" ], [ [ "u", "u", "a" ],
>               [ "u", "v", "b" ] ] );
<quiver with 2 vertices and 2 arrows>
gap> A := PathAlgebra( Rationals, Q );
<Rationals[<quiver with 2 vertices and 2 arrows>]>
gap> OppositePathAlgebra( A );
<Rationals[<quiver with 2 vertices and 2 arrows>]>
gap> OppositePathAlgebraElement( A.u + 2*A.a + 5*A.a*A.b );
(1)*u_op+(2)*a_op+(5)*b_op*a_op
gap> IsIdenticalObj( A,
>         OppositePathAlgebra( OppositePathAlgebra( A ) ) );
true
```

4.16 Tensor products of path algebras

If Λ and Γ are quotients of path algebras over the same field F, then their tensor product $\Lambda \otimes_F \Gamma$ is also a quotient of a path algebra over F.

The quiver for the tensor product path algebra is the `QuiverProduct` (4.16.1) of the quivers of the original algebras.

The operation `TensorProductOfAlgebras` (4.16.6) computes the tensor products of two quotients of path algebras as a quotient of a path algebra.

4.16.1 QuiverProduct

▷ `QuiverProduct(Q1, Q2)` (operation)

Arguments: *Q1* and *Q2* – quivers.

Returns: the product quiver $Q1 \times Q2$.

A vertex in $Q1 \times Q2$ which is made by combining a vertex named `u` in *Q1* with a vertex `v` in *Q2* is named `u_v`. Arrows are named similarly (they are made by combining an arrow from one quiver with a vertex from the other).

4.16.2 QuiverProductDecomposition

▷ `QuiverProductDecomposition(Q)` (attribute)

Arguments: *Q* – a quiver.

Returns: the original quivers *Q* is a product of, if *Q* was created by the `QuiverProduct` (4.16.1) operation.

The value of this attribute is an object in the category `IsQuiverProductDecomposition` (4.16.3).

4.16.3 IsQuiverProductDecomposition

▷ `IsQuiverProductDecomposition(object)` (category)

Arguments: *object* – any object in GAP.

Category for objects containing information about the relation between a product quiver and the quivers it is a product of. The quiver factors can be extracted from the decomposition object by using the [] notation (like accessing elements of a list). The decomposition object is also used by the operations `IncludeInProductQuiver` (4.16.4) and `ProjectFromProductQuiver` (4.16.5).

4.16.4 IncludeInProductQuiver

▷ `IncludeInProductQuiver(L, Q)` (operation)

Arguments: *L* – a list containing the paths q_1 and q_2, Q – a product quiver.

Returns: a path in *Q*.

Includes paths q_1 and q_2 from two quivers into the product of these quivers, *Q*. If at least one of q_1 and q_2 is a vertex, there is exactly one possible inclusion. If they are both non-trivial paths, there are several possibilities. This operation constructs the path which is the inclusion of q_1 at the source of q_2 multiplied with the inclusion of q_2 at the target of q_1.

4.16.5 ProjectFromProductQuiver

▷ `ProjectFromProductQuiver(i, p)` (operation)

Arguments: *i* – a positive integer, *p* – a path in the product quiver.

Returns: the projection of the product quiver path p to one of the factors. Which factor it should be projected to is specified by the argument *i*.

The following example shows how the operations related to quiver products are used.

Example
```
gap> q1 := Quiver( [ "u1", "u2" ], [ [ "u1", "u2", "a" ] ] );
<quiver with 2 vertices and 1 arrows>
gap> q2 := Quiver( [ "v1", "v2", "v3" ],
                      [ [ "v1", "v2", "b" ],
                        [ "v2", "v3", "c" ] ] );
<quiver with 3 vertices and 2 arrows>
gap> q1_q2 := QuiverProduct( q1, q2 );
<quiver with 6 vertices and 7 arrows>
gap> q1_q2.u1_b * q1_q2.a_v2;
u1_b*a_v2
gap> IncludeInProductQuiver( [ q1.a, q2.b * q2.c ], q1_q2 );
a_v1*u2_b*u2_c
gap> ProjectFromProductQuiver( 2, q1_q2.a_v1 * q1_q2.u2_b * q1_q2.u2_c );
b*c
gap> q1_q2_dec := QuiverProductDecomposition( q1_q2 );
<object>
gap> q1_q2_dec[ 1 ];
```

```
<quiver with 2 vertices and 1 arrows>
gap> q1_q2_dec[ 1 ] = q1;
true
```

4.16.6 TensorProductOfAlgebras

▷ `TensorProductOfAlgebras(FQ1, FQ2)` (operation)

Arguments: *FQ1* and *FQ2* – (quotients of) path algebras.
Returns: The tensor product of *FQ1* and *FQ2*.

The result is a quotient of a path algebra, whose quiver is the `QuiverProduct` (4.16.1) of the quivers of the operands.

4.16.7 SimpleTensor

▷ `SimpleTensor(L, T)` (operation)

Arguments: *L* – a list containing two elements x and y of two (quotients of) path algebras, *T* – the tensor product of these algebras.
Returns: the simple tensor $x \otimes y$.

$x \otimes y$ is in the tensor product *T* (produced by `TensorProductOfAlgebras` (4.16.6)).

4.16.8 TensorProductDecomposition

▷ `TensorProductDecomposition(T)` (attribute)

Arguments: *T* – a tensor product of path algebras.
Returns: a list of the factors in the tensor product.

T should be produced by `TensorProductOfAlgebras` (4.16.6)).

The following example shows how the operations for tensor products of quotients of path algebras are used.

Example
```
gap> q1 := Quiver( [ "u1", "u2" ], [ [ "u1", "u2", "a" ] ] );
<quiver with 2 vertices and 1 arrows>
gap> q2 := Quiver( [ "v1", "v2", "v3", "v4" ],
>                  [ [ "v1", "v2", "b" ],
>                    [ "v1", "v3", "c" ],
>                    [ "v2", "v4", "d" ],
>                    [ "v3", "v4", "e" ] ] );
<quiver with 4 vertices and 4 arrows>
gap> fq1 := PathAlgebra( Rationals, q1 );
<Rationals[<quiver with 2 vertices and 1 arrows>]>
gap> fq2 := PathAlgebra( Rationals, q2 );
<Rationals[<quiver with 4 vertices and 4 arrows>]>
gap> I := Ideal( fq2, [ fq2.b * fq2.d - fq2.c * fq2.e ] );
<two-sided ideal in <Rationals[<quiver with 4 vertices and 4 arrows>]>
    , (1 generators)>
gap> quot := fq2 / I;
<Rationals[<quiver with 4 vertices and 4 arrows>]/
<two-sided ideal in <Rationals[<quiver with 4 vertices and 4 arrows>]>
```

```
    , (1 generators)>>
gap> t := TensorProductOfAlgebras( fq1, quot );
<Rationals[<quiver with 8 vertices and 12 arrows>]/
<two-sided ideal in <Rationals[<quiver with 8 vertices and
    12 arrows>]>, (6 generators)>>
gap> SimpleTensor( [ fq1.a, quot.b ], t );
[(1)*u1_b*a_v2]
gap> t_dec := TensorProductDecomposition( t );
[ <Rationals[<quiver with 2 vertices and 1 arrows>]>,
  <Rationals[<quiver with 4 vertices and 4 arrows>]/
    <two-sided ideal in <Rationals[<quiver with 4 vertices and
        4 arrows>]>, (1 generators)>> ]
gap> t_dec[ 1 ] = fq1;
true
```

4.16.9 EnvelopingAlgebra

▷ `EnvelopingAlgebra(A)` (attribute)

Arguments: `A` – a (quotient of) a path algebra.
Returns: the enveloping algebra $A^{e} = A^{op} \otimes A$ of `A`

4.16.10 IsEnvelopingAlgebra

▷ `IsEnvelopingAlgebra(A)` (property)

Arguments: `A` – an algebra.
Returns: true if and only if `A` is the result of a call to `EnvelopingAlgebra` (4.16.9).

4.16.11 AlgebraAsModuleOverEnvelopingAlgebra

▷ `AlgebraAsModuleOverEnvelopingAlgebra(A)` (attribute)

Arguments: `A` – a (quotient of a) path algebra `A`.
Returns: the algebra `A` as a right module over the enveloping algebra of `A`.

4.16.12 DualOfAlgebraAsModuleOverEnvelopingAlgebra

▷ `DualOfAlgebraAsModuleOverEnvelopingAlgebra(A)` (attribute)

Arguments: `A` – a finite dimensional (admissible quotient of) a path algebra `A`.
Returns: the algebra `A` as a right module over the enveloping algebra of `A`.

4.16.13 TrivialExtensionOfQuiverAlgebra

▷ `TrivialExtensionOfQuiverAlgebra(A)` (attribute)

Arguments: `A` – a finite dimensional (admissible quotient of) a path algebra `A`.
Returns: the trivial extension algebra $T(A) = A \oplus D(A)$ of the entered algebra `A`.

4.17 Finite dimensional algebras over finite fields

4.17.1 IsBasicAlgebra

▷ `IsBasicAlgebra(A)` (property)

Arguments: `A` - a finite dimensional algebra over a finite field.

Returns: true if the entered algebra `A` is a (finite dimensional) basic algebra and false otherwise. This method only applies to algebras over finite fields.

4.17.2 IsElementaryAlgebra

▷ `IsElementaryAlgebra(A)` (property)

Arguments: `A` - a finite dimensional algebra over a finite field.

Returns: true if the entered algebra `A` is a (finite dimensional) elementary algebra and false otherwise. This method only applies to algebras over finite fields.

The algebra `A` need not to be an elementary algebra over the field which it is defined, but be an elementary algebra over a field extension.

4.17.3 PrimitiveIdempotents

▷ `PrimitiveIdempotents(A)` (operation)

Arguments: `A` - a finite dimensional simple algebra over a finite field.

Returns: a complete set of primitive idempotents $\{e_i\}$ such that $A \simeq Ae_1 + ... + Ae_n$.

TODO: Understand what this function actually does.

Chapter 5

Groebner Basis

This chapter contains the declarations and implementations needed for Groebner basis. Currently, we do not provide algorithms to actually compute Groebner basis; instead, the declarations and implementations are provided here for GAP objects and the actual elements of Groebner basis are computed by the GBNP package.

5.1 Constructing a Groebner Basis

5.1.1 InfoGroebnerBasis

▷ `InfoGroebnerBasis` (info class)

is the info class for functions dealing with Groebner basis.

5.1.2 GroebnerBasis

▷ `GroebnerBasis(I, rels)` (operation)

Arguments: *I* – an ideal, *rels* – a list of relations generating *I*.

Returns: an object *GB* in the `IsGroebnerBasis` (5.2.3) category with `IsCompleteGroebnerBasis` (5.2.2) property set on true.

Sets also *GB* as a value of the attribute `GroebnerBasisOfIdeal` (4.10.1) for *I* (so one has an access to it by calling GroebnerBasisOfIdeal(*I*)).
There are absolutely no computations and no checks for correctness in this function. Giving a set of relations that does not form a Groebner basis may result in incorrect answers or unexpected errors. This function is intended to be used by packages providing access to external Groebner basis programs and should be invoked before further computations on Groebner basis or ideal I (cf. also `IsCompleteGroebnerBasis` (5.2.2)).

5.2 Categories and Properties of Groebner Basis

5.2.1 IsCompletelyReducedGroebnerBasis

▷ `IsCompletelyReducedGroebnerBasis(gb)` (property)

Arguments: *GB* – a Groebner basis.
Returns: true when *GB* is a Groebner basis which is completely reduced.

5.2.2 IsCompleteGroebnerBasis

▷ IsCompleteGroebnerBasis(*gb*) (property)

Arguments: *GB* – a Groebner basis.
Returns: true when *GB* is a complete Groebner basis.

While philosophically something that isn't a complete Groebner basis isn't a Groebner basis at all, this property can be used in conjuction with other properties to see if the the Groebner basis contains enough information for computations. An example of a system that creates incomplete Groebner bases is 'Opal'.
Note: The current package used for creating Groebner bases is GBNP, and this package does not create incomplete Groebner bases.

5.2.3 IsGroebnerBasis

▷ IsGroebnerBasis(*object*) (category)

Arguments: *object* – any object in GAP.
Returns: true when *object* is a Groebner basis and false otherwise.

The function only returns true for Groebner bases that has been set as such using the `GroebnerBasis` function, as illustrated in the following example.

5.2.4 IsHomogeneousGroebnerBasis

▷ IsHomogeneousGroebnerBasis(*gb*) (property)

Arguments: *GB* – a Groebner basis.
Returns: true when *GB* is a Groebner basis which is homogenous.

Example
```
gap> Q := Quiver( 3, [ [1,2,"a"], [2,3,"b"] ] );
<quiver with 3 vertices and 2 arrows>
gap> PA := PathAlgebra( Rationals, Q );
<Rationals[<quiver with 3 vertices and 2 arrows>]>
gap> rels := [ PA.a*PA.b ];
[ (1)*a*b ]
gap> gb := GBNPGroebnerBasis( rels, PA );
[ (1)*a*b ]
gap> I := Ideal( PA, gb );
<two-sided ideal in <Rationals[<quiver with 3 vertices and 2 arrows>]>
    , (1 generators)>
gap> grb := GroebnerBasis( I, gb );
<complete two-sided Groebner basis containing 1 elements>
gap> alg := PA/I;
<Rationals[<quiver with 3 vertices and 2 arrows>]/
<two-sided ideal in <Rationals[<quiver with 3 vertices and 2 arrows>]>
    , (1 generators)>>
gap> IsGroebnerBasis(gb);
```

```
false
gap> IsGroebnerBasis(grb);
true
```

5.2.5 IsTipReducedGroebnerBasis

▷ `IsTipReducedGroebnerBasis(gb)` (property)

Arguments: *GB* – a Groebner Basis.
Returns: true when *GB* is a Groebner basis which is tip reduced.

5.3 Attributes and Operations for Groebner Basis

5.3.1 AdmitsFinitelyManyNontips

▷ `AdmitsFinitelyManyNontips(GB)` (operation)

Arguments: *GB* – a complete Groebner basis.
Returns: true if the Groebner basis admits only finitely many nontips and false otherwise.

5.3.2 CompletelyReduce

▷ `CompletelyReduce(GB, a)` (operation)

Arguments: *GB* – a Groebner basis, `a` – an element in a path algebra.
Returns: `a` reduced by *GB*.
If `a` is already completely reduced, the original element `a` is returned.

5.3.3 CompletelyReduceGroebnerBasis

▷ `CompletelyReduceGroebnerBasis(GB)` (operation)

Arguments: *GB* – a Groebner basis.
Returns: the completely reduced Groebner basis of the ideal generated by *GB*.
The operation modifies a Groebner basis *GB* such that each relation in *GB* is completely reduced. The `IsCompletelyReducedGroebnerBasis` and `IsTipReducedGroebnerBasis` properties are set as a result of this operation. The resulting relations will be placed in sorted order according to the ordering of *GB*.

5.3.4 Enumerator

▷ `Enumerator(GB)` (operation)

Arguments: *GB* – a Groebner basis.
Returns: an enumerat that enumerates the relations making up the Groebner basis.
These relations should be enumerated in ascending order with respect to the ordering for the family the elements are contained in.

5.3.5 IsPrefixOfTipInTipIdeal

▷ `IsPrefixOfTipInTipIdeal(GB, R)` (operation)

Arguments: `GB` – a Groebner basis, `R` – a relation.
Returns: true if the tip of the relation `R` is in the tip ideal generated by the tips of `GB`.
This is used mainly for the construction of right Groebner basis, but is made available for general use in case there are other unforseen applications.

5.3.6 Iterator

▷ `Iterator(GB)` (operation)

Arguments: `GB` – a Groebner basis.
Returns: an iterator (in the IsIterator category, see the GAP manual, chapter 28.7).
Creates an iterator that iterates over the relations making up the Groebner basis. These relations are iterated over in ascending order with respect to the ordering for the family the elements are contained in.

5.3.7 Nontips

▷ `Nontips(GB)` (attribute)

Arguments: `GB` – a Groebner basis.
Returns: a list of nontip elements for `GB`.
In order to compute the nontip elements, the Groebner basis must be complete and tip reduced, and there must be a finite number of nontips. If there are an infinite number of nontips, the operation returns ‘fail’.

5.3.8 NontipSize

▷ `NontipSize(GB)` (operation)

Arguments: `GB` – a complete Groebner basis.
Returns: the number of nontips admitted by `GB`.

5.3.9 TipReduce

▷ `TipReduce(GB, a)` (operation)

Arguments: `GB` – a Groebner basis, `a` - an element in a path algebra.
Returns: the element `a` tip reduced by the Groebner basis.
If `a` is already tip reduced, then the original `a` is returned.

5.3.10 TipReduceGroebnerBasis

▷ `TipReduceGroebnerBasis(GB)` (operation)

Arguments: *GB* – a Groebner basis.

Returns: a tip reduced Groebner basis.

The returned Groebner basis is equivalent to *GB* If *GB* is already tip reduced, this function returns the original object *GB*, possibly with the addition of the ‘IsTipReduced‘’ property set.

5.4 Right Groebner Basis

In this section we support right Groebner basis for two-sided ideals with Groebner basis. More general cases may be supported in the future.

5.4.1 IsRightGroebnerBasis

▷ `IsRightGroebnerBasis(`*object*`)` (property)

Arguments: *object* – any object in GAP.

Returns: true when *object* is a right Groebner basis.

5.4.2 RightGroebnerBasis

▷ `RightGroebnerBasis(`*I*`)` (operation)

Arguments: *I* – a right ideal.

Returns: a right Groebner basis for *I*, which must support a right Groebner basis theory. Right now, this requires that *I* has a complete Groebner basis.

5.4.3 RightGroebnerBasisOfIdeal

▷ `RightGroebnerBasisOfIdeal(`*I*`)` (attribute)

Arguments: *I* – a right ideal.

Returns: a right Groebner basis of a right ideal, *I*, if one has been computed.

Chapter 6

Right Modules over Path Algebras

There are two implementations of right modules over path algebras. The first type are matrix modules that are defined by vector spaces and linear transformations. The second type is presentations defined by vertex projective modules (see 6.7).

6.1 Modules of matrix type

The first implementation of right modules over path algebras views them as a collection of vector spaces and linear transformations. Each vertex in the path algebra is associated with a vector space over the field of the algebra. For each vertex v of the algebra there is a vector space V. Arrows of the algebra are then associated with linear transformations which map the vector space of the source vertex to the vector space of the target vertex. For example, if a is an arrow from v to w, then there is a transformation from vector space V to W. Given the dimension vector of the module we want to construct, the information we need to provide is the non-zero linear transformations. The size of the matrices for the zero linear transformation are given when we know the dimension vector. Alternatively, if we enter all the transformations, we can create the vector spaces of the correct dimension, and check to make sure the dimensions all agree. We can create a module in this way as follows.

6.1.1 RightModuleOverPathAlgebra (with dimension vector)

▷ `RightModuleOverPathAlgebra(`*`A, dim_vector, gens`*`)` (operation)
▷ `RightModuleOverPathAlgebra(`*`A, mats`*`)` (operation)
▷ `RightModuleOverPathAlgebraNC(`*`A, mats`*`)` (operation)

Arguments: *`A`* – a (quotient of a) path algebra and *`dim_vector`* – the dimension vector of the module, *`gens`* or *`mats`* – a list of matrices. For further explanations, see below.

Returns: a module over a path algebra or over a qoutient of a path algebra.

In the first function call, the second argument *`dim_vector`* is the dimension vector of the module, and the last argument *`gens`* (maybe an empty list `[]`) is a list of elements of the form `["label",matrix]`. This function constructs a right module over a (quotient of a) path algebra *`A`* with dimension vector *`dim_vector`*, and where the generators/arrows with a non-zero action is given in the list *`gens`*. The format of the list *`gens`* is [["a",[matrix_a]],["b",[matrix_b]],...], where "a" and "b" are labels of arrows used when the underlying quiver was created and matrix_? is the action of the algebra element corresponding to the arrow with label "?". The action of the arrows can be entered in any order. The function checks (i) if the algebra `A` is a (quotient of a) path algebra, (ii) if the matrices of the action of the arrows have the correct size according to the dimension vector entered, (iii) also whether or not the relations of the algebra are satisfied and (iv) if all matrices are over the correct field.

In the second function call, the list of matrices `mats` can take on three different forms. The function checks (i), (ii), (iii) and (iv) as above.

1) The argument `mats` can be a list of blocks of matrices where each block is of the form, '["name of arrow",matrix]'. So if you named your arrows when you created the quiver, then you can associate a matrix with that arrow explicitly.

2) The argument `mats` is just a list of matrices, and the matrices will be associated to the arrows in the order of arrow creation. If when creating the quiver, the arrow *a* was created first, then *a* would be associated with the first matrix.

3) The method is very much the same as the second method. If `arrows` is a list of the arrows of the quiver (obtained for instance through `arrows := ArrowsOfQuiver(Q);`), the argument `mats` can have the format `[[arrows[1],matrix_1],[arrows[2],matrix_2],.... ].`

If you would like the trivial vector space at any vertex, then for each incoming arrow "a", associate it with a list of the form `["a",[n,0]]` where n is the dimension of the vector space at the source vertex of the arrow. Likewise for all outgoing arrows "b", associate them to a block of form `["b",[0,n]]` where n is the dimension of the vector space at the target vertex of the arrow.

The third function call is the same as the second except that the check (iv) is not performed.

A warning though, the function assumes that you do not mix the styles of inputting the matrices/linear transformations associated to the arrows in the quiver. Furthermore in the two last versions, each arrow needs to be assigned a matrix, otherwise an error will be returned.

6.1.2 RightAlgebraModuleToPathAlgebraMatModule

▷ `RightAlgebraModuleToPathAlgebraMatModule(M)` (operation)

Arguments: `M` – a right module over an algebra.

Returns: a module over a (qotient of a) path algebra.

This function constructs a right module over a (quotient of a) path algebra *A* from a RightAlgebraModule over the same algebra *A*. The function checks if *A* actually is a quotient of a path algebra and if the module *M* is finite dimensional and if not, it returns an error message.

6.1.3 \= (for two path algebra matrix modules)

▷ `\=(M, N)` (operation)

Arguments: `M, N` – two path algebra matrix modules.

Returns: true if `M` and `N` has the same dimension vectors and the same matrices defining the module structure.

Example
```
gap> Q := Quiver(2, [[1, 2, "a"], [2, 1, "b"],[1, 1, "c"]]);
<quiver with 2 vertices and 3 arrows>
gap> P := PathAlgebra(Rationals, Q);
<Rationals[<quiver with 2 vertices and 3 arrows>]>
gap> matrices := [["a", [[1,0,0],[0,1,0]]],
>  ["b", [[0,1],[1,0],[0,1]]],
>  ["c", [[0,0],[1,0]]]];
[ [ "a", [ [ 1, 0, 0 ], [ 0, 1, 0 ] ] ],
  [ "b", [ [ 0, 1 ], [ 1, 0 ], [ 0, 1 ] ] ],
  [ "c", [ [ 0, 0 ], [ 1, 0 ] ] ] ]
gap> M := RightModuleOverPathAlgebra(P,matrices);
<[ 2, 3 ]>
gap> mats := [ [[1,0,0], [0,1,0]], [[0,1],[1,0],[0,1]],
>           [[0,0],[1,0]] ];;
gap> N := RightModuleOverPathAlgebra(P,mats);
<[ 2, 3 ]>
gap> arrows := ArrowsOfQuiver(Q);
[ a, b, c ]
gap> mats := [[arrows[1], [[1,0,0],[0,1,0]]],
>          [arrows[2], [[0,1],[1,0],[0,1]]],
>          [arrows[3], [[0,0],[1,0]]]];;
gap> N := RightModuleOverPathAlgebra(P,mats);
<[ 2, 3 ]>
gap> # Next we give the vertex simple associate to vertex 1.
gap> M := RightModuleOverPathAlgebra(P,[["a",[1,0]],["b",[0,1]],
>               ["c",[[0]]]]);
<[ 1, 0 ]>
gap> # The zero module.
gap> M := RightModuleOverPathAlgebra(P,[["a",[0,0]],["b",[0,0]],
>               ["c",[0,0]]]);
<[ 0, 0 ]>
gap> Dimension(M);
0
gap> Basis(M);
Basis( <[ 0, 0 ]>, ... )
gap> matrices := [["a", [[1,0,0],[0,1,0]]], ["b",
>  [[0,1],[1,0],[0,1]]], ["c", [[0,0],[1,0]]]];
[ [ "a", [ [ 1, 0, 0 ], [ 0, 1, 0 ] ] ],
  [ "b", [ [ 0, 1 ], [ 1, 0 ], [ 0, 1 ] ] ],
  [ "c", [ [ 0, 0 ], [ 1, 0 ] ] ] ]
gap> M := RightModuleOverPathAlgebra(P,[2,3],matrices);
<[ 2, 3 ]>
gap> M := RightModuleOverPathAlgebra(P,[2,3],[]);
<[ 2, 3 ]>
gap> A := P/[P.c^2 - P.a*P.b, P.a*P.b*P.c, P.b*P.c];
<Rationals[<quiver with 2 vertices and 3 arrows>]/
<two-sided ideal in <Rationals[<quiver with 2 vertices and 3 arrows>]>
    , (4 generators)>>
gap> Dimension(A);
9
gap> Amod := RightAlgebraModule(A,\*,A);
<9-dimensional right-module over <Rationals[<quiver with
2 vertices and 3 arrows>]/
<two-sided ideal in <Rationals[<quiver with 2 vertices and 3 arrows>]>
    , (4 generators)>>>
gap> RightAlgebraModuleToPathAlgebraMatModule(Amod);
<[ 4, 5 ]>
```

6.2 Categories Of Matrix Modules

6.2.1 IsPathAlgebraMatModule

▷ `IsPathAlgebraMatModule(object)` (filter)

Returns: true or false depending on whether *object* belongs to the category `IsPathAlgebraMatModule`.

These matrix modules fall under the category 'IsAlgebraModule' with the added filter of 'IsPathAlgebraMatModule'. Operations available for algebra modules can be applied to path algebra modules. See **Reference: Representations of Algebras** for more details. These modules are also vector spaces over the field of the path algebra. So refer to **Reference: Vector Spaces** for descriptions of the basis and elementwise operations available.

6.3 Acting on Module Elements

6.3.1 ^ (a PathAlgebraMatModule element and a PathAlgebra element)

▷ `^(m, p)` (operation)

Arguments: *m* – an element in a module, *p* – an element in a quiver algebra.

Returns: the element *m* multiplied with *p*.

When you act on an module element m by an arrow a from v to w, the component of m from V is acted on by L the transformation associated to a and placed in the component W. All other components are given the value 0.

Example
```
gap> # Using the path algebra P from the above example.
gap> matrices := [["a", [[1,0,0],[0,1,0]]],
> ["b", [[0,1],[1,0],[0,1]]], ["c", [[0,0],[1,0]]]];
[ [ "a", [ [ 1, 0, 0 ], [ 0, 1, 0 ] ] ],
  [ "b", [ [ 0, 1 ], [ 1, 0 ], [ 0, 1 ] ] ],
  [ "c", [ [ 0, 0 ], [ 1, 0 ] ] ] ]
gap> M := RightModuleOverPathAlgebra(P,matrices);
<[ 2, 3 ]>
gap> B:=BasisVectors(Basis(M));
[ [ [ 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 ] ] ]
gap> B[1] + B[3];
[ [ 1, 0 ], [ 1, 0, 0 ] ]
gap> 4*B[2];
[ [ 0, 4 ], [ 0, 0, 0 ] ]
gap> m := 5*B[1] + 2*B[4]+B[5];
[ [ 5, 0 ], [ 0, 2, 1 ] ]
gap> m^(P.a*P.b-P.c);
[ [ 0, 5 ], [ 0, 0, 0 ] ]
gap> B[1]^P.a;
[ [ 0, 0 ], [ 1, 0, 0 ] ]
gap> B[2]^P.b;
[ [ 0, 0 ], [ 0, 0, 0 ] ]
```

```
gap> B[4]^(P.b*P.c);
[ [ 0, 0 ], [ 0, 0, 0 ] ]
```

6.4 Operations on representations

Example

```
gap> Q  := Quiver(3,[[1,2,"a"],[1,2,"b"],[2,2,"c"],[2,3,"d"],
> [3,1,"e"]]);
<quiver with 3 vertices and 5 arrows>
gap> KQ := PathAlgebra(Rationals, Q);
<Rationals[<quiver with 3 vertices and 5 arrows>]>
gap> gens := GeneratorsOfAlgebra(KQ);
[ (1)*v1, (1)*v2, (1)*v3, (1)*a, (1)*b, (1)*c, (1)*d, (1)*e ]
gap> u := gens[1];; v := gens[2];;
gap> w := gens[3];; a := gens[4];;
gap> b := gens[5];; c := gens[6];;
gap> d := gens[7];; e := gens[8];;
gap> rels := [d*e,c^2,a*c*d-b*d,e*a];;
gap> A := KQ/rels;
<Rationals[<quiver with 3 vertices and 5 arrows>]/
<two-sided ideal in <Rationals[<quiver with 3 vertices and 5 arrows>]>
    , (5 generators)>>
gap> mat := [["a",[[1,2],[0,3],[1,5]]],["b",[[2,0],[3,0],[5,0]]],
> ["c",[[0,0],[1,0]]],["d",[[1,2],[0,1]]],["e",[[0,0,0],[0,0,0]]]];;
gap> N := RightModuleOverPathAlgebra(A,mat);
<[ 3, 2, 2 ]>
```

6.4.1 AnnihilatorOfModule

▷ AnnihilatorOfModule(*M*) (operation)

Arguments: *M* – a path algebra module.

Returns: a basis of the annihilator of the module *M* in the finite dimensional algebra over which *M* is a module.

6.4.2 BasicVersionOfModule

▷ BasicVersionOfModule(*M*) (operation)

Arguments: *M* – a path algebra module.

Returns: a basic version of the entered module *M*, that is, if $M \simeq M_1^{n_1} \oplus \cdots \oplus M_t^{n_t}$, where M_i is indecomposable, then $M_1 \oplus \cdots \oplus M_t$ is returned. At present, this function only work at best for finite dimensional (quotients of a) path algebra over a finite field. If *M* is zero, then *M* is returned.

6.4.3 BlockDecompositionOfModule

▷ BlockDecompositionOfModule(*M*) (operation)

Arguments: `M` – a path algebra module.

Returns: a set of modules $\{M_1,...,M_t\}$ such that $M \simeq M_1 \oplus \cdots \oplus M_t$, where each M_i is isomorphic to $X_i^{n_i}$ for some indecomposable module X_i and positive integer n_i for all i, where $X_i \not\simeq X_j$ for $i \neq j$.

6.4.4 BlockSplittingIdempotents

▷ `BlockSplittingIdempotents(M)` (operation)

Arguments: `M` – a path algebra module.

Returns: a set $\{e_1,...,e_t\}$ of idempotents in the endomorphism of `M` such that $M \simeq \operatorname{Im} e_1 \oplus \cdots \oplus \operatorname{Im} e_t$, where each $\operatorname{Im} e_i$ is isomorphic to $X_i^{n_i}$ for some module X_i and positive integer n_i for all i.

6.4.5 CommonDirectSummand

▷ `CommonDirectSummand(M, N)` (operation)

Arguments: `M` and `N` – two path algebra modules.

Returns: a list of four modules [`X`,`U`,`X`, `V`], where `X` is one common non-zero direct summand of `M` and `N`, the sum of `X` and `U` is `M` and the sum of `X` and `V` is `N`, if such a non-zero direct summand exists. Otherwise it returns false.

The function checks if `M` and `N` are `PathAlgebraMatModules` over the same (quotient of a) path algebra.

6.4.6 ComplexityOfModule

▷ `ComplexityOfModule(M, n)` (operation)

Arguments: `M` – path algebdra module, `n` – a positive integer.

Returns: an estimate of the complexity of the module `M`.

The function checks if the algebra over which the module `M` lives is known to have finite global dimension. If so, it returns complexity zero. Otherwise it tries to estimate the complexity in the following way. Recall that if a function $f(x)$ is a polynomial in x, the degree of $f(x)$ is given by $\lim_{n\to\infty} \frac{\log|f(n)|}{\log n}$. So then this function computes an estimate of the complexity of `M` by approximating the complexity by considering the limit $\lim_{m\to\infty} \log \frac{\dim(P(M)(m))}{\log m}$ where $P(M)(m)$ is the m-th projective in a minimal projective resolution of `M` at stage m. This limit is estimated by $\frac{\log \dim(P(M)(n))}{\log n}$.

6.4.7 DecomposeModule

▷ `DecomposeModule(M)` (operation)

Arguments: `M` – a path algebra module.

Returns: a list of indecomposable modules whose direct sum is isomorphic to the module `M`.

Warning: the function is not properly tested and it at best only works properly over finite fields.

6.4.8 DecomposeModuleWithMultiplicities

▷ `DecomposeModuleWithMultiplicities(M)` (operation)

Arguments: *M* – a path algebra module.

Returns: a list of length two, where the first entry is a list of all indecomposable non-isomorphic direct summands of *M* and the second entry is the list of the multiplicities of these direct summand in the module *M*.

Warning: the function is not properly tested and it at best only works properly over finite fields.

6.4.9 Dimension (for a PathAlgebraMatModule)

▷ `Dimension(M)` (operation)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`).

Returns: the dimension of the representation *M*.

6.4.10 DimensionVector

▷ `DimensionVector(M)` (attribute)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`).

Returns: the dimension vector of the representation *M*.

6.4.11 DirectSumOfModules

▷ `DirectSumOfModules(L)` (operation)

Arguments: *L* – a list of `PathAlgebraMatModules` over the same (quotient of a) path algebra.

Returns: the direct sum of the representations contained in the list *L*.

In addition three attributes are attached to the result, `IsDirectSumOfModules` (6.4.16), `DirectSumProjections` (6.4.13) `DirectSumInclusions` (6.4.12).

6.4.12 DirectSumInclusions

▷ `DirectSumInclusions(M)` (attribute)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`).

Returns: the list of inclusions from the individual modules to their direct sum, when a direct sum has been constructed using `DirectSumOfModules` (6.4.11).

6.4.13 DirectSumProjections

▷ `DirectSumProjections(M)` (attribute)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`).

Returns: the list of projections from the direct sum to the individual modules used to construct direct sum, when a direct sum has been constructed using `DirectSumOfModules` (6.4.11).

6.4.14 IntersectionOfSubmodules

▷ `IntersectionOfSubmodules(list)` (operation)

Arguments: `f, g` or `list` – two homomorphisms of PathAlgebraMatModules or a list of such.

Returns: the subrepresentation given by the intersection of all the submodules given by the inclusions `f` and `g` or `list`.

The function checks if `list` is non-empty and if `f`: $M \to X$ and `g`: $N \to X$ or all the homomorphism in `list` have the same range and if they all are inclusions. If the function is given two arguments `f` and `g`, then it returns $[f', g', g' * f]$, where $f'\colon E \to N$, $g'\colon E \to M$, and E is the pullback of `f` and `g`. For a list of inclusions it returns a monomorphism from a module isomorphic to the intersection to X.

6.4.15 IsDirectSummand

▷ `IsDirectSummand(M, N)` (operation)

Arguments: `M, N` – two path algebra modules (`PathAlgebraMatModules`).

Returns: true if `M` is isomorphic to a direct summand of `N`, otherwise false.

The function checks if `M` and `N` are `PathAlgebraMatModules` over the same (quotient of a) path algebra.

6.4.16 IsDirectSumOfModules

▷ `IsDirectSumOfModules(M)` (attribute)

Arguments: `M` – a path algebra module (`PathAlgebraMatModule`).

Returns: true if `M` is constructed via the command `DirectSumOfModules` (6.4.11).

Using the example above.

Example
```
gap> N2 := DirectSumOfModules([N,N]);
<[ 6, 4, 4 ]>
gap> proj := DirectSumProjections(N2);
[ <<[ 6, 4, 4 ]> ---> <[ 3, 2, 2 ]>>
    , <<[ 6, 4, 4 ]> ---> <[ 3, 2, 2 ]>>
     ]
gap> inc := DirectSumInclusions(N2);
[ <<[ 3, 2, 2 ]> ---> <[ 6, 4, 4 ]>>
    , <<[ 3, 2, 2 ]> ---> <[ 6, 4, 4 ]>>
     ]
```

6.4.17 IsExceptionalModule

▷ `IsExceptionalModule(M)` (property)

Arguments: `M` – a path algebra module (`PathAlgebraMatModule`).

Returns: true if `M` is an exceptional module, otherwise false, if the field, over which the algebra `M` is defined over, is finite.

The module `M` is an exceptional module, if it is indecomposable and $\mathrm{Ext}^1(M,M) = (0)$.

6.4.18 IsIndecomposableModule

▷ `IsIndecomposableModule(M)` (property)

Arguments: `M` – a path algebra module (`PathAlgebraMatModule`).
Returns: true if `M` is an indecomposable module, if the field, over which the algebra `M` is defined over, is finite.

6.4.19 IsInAdditiveClosure

▷ `IsInAdditiveClosure(M, N)` (operation)

Arguments: `M, N` – two path algebra modules (`PathAlgebraMatModules`).
Returns: true if `M` is in the additive closure of the module `N`, otherwise false.
The function checks if `M` and `N` are `PathAlgebraMatModules` over the same (quotient of a) path algebra.

6.4.20 IsInjectiveModule

▷ `IsInjectiveModule(M)` (property)

Arguments: `M` – a path algebra module (`PathAlgebraMatModule`).
Returns: true if the representation `M` is injective.

6.4.21 IsomorphicModules

▷ `IsomorphicModules(M, N)` (operation)

Arguments: `M, N` – two path algebra modules (`PathAlgebraMatModules`).
Returns: true or false depending on whether `M` and `N` are isomorphic or not.
The function first checks if the modules `M` and `N` are modules over the same algebra, and returns fail if not. The function returns true if the modules are isomorphic, otherwise false.

6.4.22 IsProjectiveModule

▷ `IsProjectiveModule(M)` (property)

Arguments: `M` – a path algebra module (`PathAlgebraMatModule`).
Returns: true if the representation `M` is projective.

6.4.23 IsRigidModule

▷ `IsRigidModule(M)` (property)

Arguments: `M` – a path algebra module (`PathAlgebraMatModule`).
Returns: true if `M` is a rigid module, otherwise false.
The module `M` is a rigid module, if $\mathrm{Ext}^1(M,M) = (0)$.

6.4.24 IsSemisimpleModule

▷ `IsSemisimpleModule(M)` (property)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`).
Returns: true if the representation *M* is semisimple.

6.4.25 IsSimpleModule

▷ `IsSimpleModule(M)` (property)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`).
Returns: true if the representation *M* is simple.

6.4.26 IsTauRigidModule

▷ `IsTauRigidModule(M)` (property)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`).
Returns: true if *M* is a τ-rigid module, otherwise false.
The module *M* is a τ-rigid module, if $\mathrm{Hom}(M, \tau M) = (0)$.

6.4.27 LoewyLength (for a PathAlgebraMatModule)

▷ `LoewyLength(M)` (attribute)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`).
Returns: the Loewy length of the module *M*.
The function checks that the module *M* is a module over a finite dimensional quotient of a path algebra, and returns fail otherwise (This is not implemented yet).

6.4.28 MatricesOfPathAlgebraModule

▷ `MatricesOfPathAlgebraModule(M)` (operation)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`).
Returns: a list of the matrices that defines the representation *M* as a right module of the acting path algebra.
The list of matrices that are returned are not the same identical to the matrices entered to define the representation if there is zero vector space in at least one vertex. Then zero matrices of the appropriate size are returned.

6.4.29 MaximalCommonDirectSummand

▷ `MaximalCommonDirectSummand(M, N)` (operation)

Arguments: `M, N` – two path algebra modules (`PathAlgebraMatModules`).

Returns: a list of three modules [`X`,`U`,`V`], where `X` is a maximal common non-zero direct summand of `M` and `N`, the sum of `X` and `U` is `M` and the sum of `X` and `V` is `N`, if such a non-zero maximal direct summand exists. Otherwise it returns false.

The function checks if `M` and `N` are `PathAlgebraMatModules` over the same (quotient of a) path algebra.

6.4.30 NumberOfNonIsoDirSummands

▷ `NumberOfNonIsoDirSummands(M)` (operation)

Arguments: `M` – a path algebra modules (`PathAlgebraMatModules`).

Returns: a list with two elements: (1) the number of non-isomorphic indecomposable direct summands of the module `M` and (2) the dimensions of the simple blocks of the semisimple ring $\mathrm{End}(M)/\operatorname{rad}\mathrm{End}(M)$.

6.4.31 MinimalGeneratingSetOfModule

▷ `MinimalGeneratingSetOfModule(M)` (attribute)

Arguments: `M` – a path algebra module (`PathAlgebraMatModule`).

Returns: a minimal generator set of the module `M` as a module of the path algebra it is defined over.

6.4.32 RadicalOfModule

▷ `RadicalOfModule(M)` (operation)

Arguments: `M` – a path algebra module (`PathAlgebraMatModule`).

Returns: the radical of the module `M`.

This returns only the representation given by the radical of the module `M`. The operation `RadicalOfModuleInclusion` (7.3.19) computes the inclusion of the radical of `M` into `M`.

6.4.33 RadicalSeries

▷ `RadicalSeries(M)` (operation)

Arguments: `M` – a path algebra module (`PathAlgebraMatModule`).

Returns: the radical series of the module `M`.

The function gives the radical series as a list of vectors `[n_1,...,n_s]`, where the algebra has s isomorphism classes of simple modules and the numbers give the multiplicity of each simple. The first vector listed corresponds to the top layer, and so on.

6.4.34 SocleSeries

▷ `SocleSeries(M)` (operation)

Arguments: `M` – a path algebra module (`PathAlgebraMatModule`).

Returns: the socle series of the module `M`.

The function gives the socle series as a list of vectors `[n_1,...,n_s]`, where the algebra has s isomorphism classes of simple modules and the numbers give the multiplicity of each simple. The last vector listed corresponds to the socle layer, and so on backwards.

6.4.35 SocleOfModule

▷ `SocleOfModule(M)` (operation)

Arguments: `M` – a path algebra module (`PathAlgebraMatModule`).

Returns: the socle of the module `M`.

This operation only return the representation given by the socle of the module `M`. The inclusion the socle of `M` into `M` can be computed using `SocleOfModuleInclusion` (7.3.20).

6.4.36 SubRepresentation

▷ `SubRepresentation(M, gens)` (operation)

Arguments: `M` – a path algebra module (`PathAlgebraMatModule`), `gens` – elements in `M`.

Returns: the submodule of the module `M` generated by the elements `gens`.

The function checks if `gens` are elements in `M`, and returns an error message otherwise. The inclusion of the submodule generated by the elements `gens` into `M` can be computed using `SubRepresentationInclusion` (7.3.21).

6.4.37 SumOfSubmodules

▷ `SumOfSubmodules(list)` (operation)

Arguments: `f, g` or `list` – two inclusions of PathAlgebraMatModules or a list of such.

Returns: the subrepresentation given by the sum of all the submodules given by the inclusions `f, g` or `list`.

The function checks if `list` is non-empty and if $f : M \to X$ and $g : N \to X$ or all the homomorphism in `list` have the same range and if they all are inclusions. If the function is given two arguments `f` and g, then it returns $[h, f', g']$, where $h \colon M + N \to X$, $f' \colon M \to M + N$ and $g' \colon N \to M + N$. For a list of inclusions it returns a monomorphism from a module isomorphic to the sum of the subrepresentations to X.

6.4.38 SupportModuleElement

▷ `SupportModuleElement(m)` (operation)

Arguments: `m` – an element of a path algebra module.

Returns: the primitive idempotents `v` in the algebra over which the module containing the element `m` is a module, such that `m^v` is non-zero.

The function checks if `m` is an element in a module over a (quotient of a) path algebra, and returns fail otherwise.

6.4.39 TopOfModule

▷ `TopOfModule(M)` (operation)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`).
Returns: the top of the module *M*.

This returns only the representation given by the top of the module *M*. The operation `TopOfModuleProjection` (7.3.22) computes the projection of the module *M* onto the top of the module *M*.

6.5 Special representations

Here we collect the predefined representations/modules over a finite dimensional quotient of a path algebra.

6.5.1 BasisOfProjectives

▷ `BasisOfProjectives(A)` (attribute)

Arguments: *A* – a finite dimensional (quotient of a) path algebra.
Returns: a list of bases for all the indecomposable projective representations over *A*. The basis for each indecomposable projective is given a list of elements in nontips in *A*.

The function checks if the algebra *A* is a finite dimensional (quotient of a) path algebra, and returns an error message otherwise.

6.5.2 IndecInjectiveModules

▷ `IndecInjectiveModules(A)` (attribute)

Arguments: *A* – a finite dimensional (quotient of a) path algebra.
Returns: a list of all the non-isomorphic indecomposable injective representations over *A*.

The function checks if the algebra *A* is a finite dimensional (quotient of a) path algebra, and returns an error message otherwise.

6.5.3 IndecProjectiveModules

▷ `IndecProjectiveModules(A)` (attribute)

Arguments: *A* – a finite dimensional (quotient of a) path algebra.
Returns: a list of all the non-isomorphic indecomposable projective representations over *A*.

The function checks if the algebra *A* is a finite dimensional (quotient of a) path algebra, and returns an error message otherwise.

6.5.4 SimpleModules

▷ `SimpleModules(A)` (attribute)

Arguments: `A` – a finite dimensional (quotient of a) path algebra.
Returns: a list of all the simple representations over `A` .
The function checks if the algebra `A` is a finite dimensional (quotient of a) path algebra, and returns an error message otherwise.

6.5.5 ZeroModule

▷ `ZeroModule(A)` (attribute)

Arguments: `A` – a finite dimensional (quotient of a) path algebra.
Returns: the zero representation over `A`.
The function checks if the algebra `A` is a finite dimensional (quotient of a) path algebra, and returns an error message otherwise.

6.6 Functors on representations

6.6.1 DualOfModule

▷ `DualOfModule(M)` (attribute)

Arguments: `M` – a `PathAlgebraMatModule`.
Returns: the dual of `M` over the opposite algebra A^{op}, if `M` is a module over A.

6.6.2 DualOfModuleHomomorphism

▷ `DualOfModuleHomomorphism(f)` (attribute)

Arguments: `f` – a map between two representations `M` and `N` over a path algebra A.
Returns: the dual of this map over the opposite path algebra `A^\op`.

6.6.3 DTr

▷ `DTr(M[, n])` (operation)
▷ `DualOfTranspose(M[, n])` (operation)

Arguments: `M` – a path algebra module, (optional) `n` – an integer.
Returns: the dual of the transpose of `M` when called with only one argument, while it returns the dual of the transpose applied to `M` `n` times otherwise. If `n` is negative, then powers of `TrD` are computed. `DualOfTranspose` is a synonym for `DTr`.

6.6.4 NakayamaFunctorOfModule

▷ `NakayamaFunctorOfModule(M)` (attribute)

Arguments: `M` – a `PathAlgebraMatModule`.
Returns: the module $\mathrm{Hom}_K(\mathrm{Hom}_A(M,A),K)$ over A, when `M` is a module over a K-algebra A.

6.6.5 NakayamaFunctorOfModuleHomomorphism

▷ `NakayamaFunctorOfModuleHomomorphism(f)` (attribute)

Arguments: `f` – a map between two modules `M` and `N` over a path algebra A.

Returns: the homomorphism induced by $f\colon M \to N$ from the module $\mathrm{Hom}_K(\mathrm{Hom}_A(M,A),K)$ to $\mathrm{Hom}_K(\mathrm{Hom}_A(N,A),K)$, when `f` is a module homomorphism over a K-algebra `A`.

6.6.6 StarOfModule

▷ `StarOfModule(M)` (attribute)

Arguments: `M` – a `PathAlgebraMatModule`.

Returns: the module $\mathrm{Hom}_A(M,A)$ over the opposite of A, when `M` is a module over an algebra A.

6.6.7 StarOfModuleHomomorphism

▷ `StarOfModuleHomomorphism(f)` (attribute)

Arguments: `f` – a map between two modules `M` and `N` over a path algebra A.

Returns: the homomorphism induced by $f\colon M \to N$ from the module $\mathrm{Hom}_A(N,A)$ to $\mathrm{Hom}_A(M,A)$, when `f` is a module homomorphism over an algebra `A`.

6.6.8 TrD

▷ `TrD(M[, n])` (operation)

▷ `TransposeOfDual(M[, n])` (operation)

Arguments: `M` – a path algebra module, (optional) `n` – an integer.

Returns: the transpose of the dual of `M` when called with only one argument, while it returns the transpose of the dual applied to `M` `n` times otherwise. If `n` is negative, then powers of `TrD` are computed. `TransposeOfDual` is a synonym for `TrD`.

6.6.9 TransposeOfModule

▷ `TransposeOfModule(M)` (attribute)

Arguments: `M` – a path algebra module.

Returns: the transpose of the module `M`.

6.7 Vertex projective modules and submodules thereof

In general, if R is a ring and e is an idempotent of R, then eR is a projective module of R. Then we can form a direct sum of these projective modules together to form larger projective module. One can construct more general modules by providing a `vertex projective presentation`. In this case, M is the cokernel as given by the following exact sequence: $\oplus_{j=1}^{r} w(j)R \to \oplus_{i=1}^{g} v(i)R \to M \to 0$ for some map between $\oplus_{j=1}^{r} w(j)R$ and $\oplus_{i=1}^{g} v(i)R$. The maps w and v map the integers to some idempotent in R.

6.7.1 RightProjectiveModule

▷ `RightProjectiveModule(A, verts)` (function)

Arguments: `A` – a (quotient of a) path algebra, `verts` – a list of vertices.

Returns: the right projective module over `A` which is the direct sum of projective modules of the form `vA` where the vertices are taken from `verts`.

The module created is in the category `IsPathAlgebraModule`. In this implementation the algebra can be a quotient of a path algebra. So if the list was $[v,w]$ then the module created will be the direct sum $vA \oplus wA$, in that order. Elements of the modules are vectors of algebra elements, and in each component, each path begins with the vertex in that position in the list of vertices. Right projective modules are implementated as algebra modules (see **Reference: Representations of Algebras**) and all operations for algebra modules are applicable to right projective modules. In particular, one can construct submodules using `SubAlgebraModule` (**Reference: SubAlgebraModule**).

Here we create the right projective module $P = vA \oplus vA \oplus wA$.

Example
```
gap> F := GF(11);
GF(11)
gap> Q := Quiver(["v","w", "x"],[["v","w","a"],["v","w","b"],
> ["w","x","c"]]);
<quiver with 3 vertices and 3 arrows>
gap> A := PathAlgebra(F,Q);
<GF(11)[<quiver with 3 vertices and 3 arrows>]>
gap> P := RightProjectiveModule(A,[A.v,A.v,A.w]);
<right-module over <GF(11)[<quiver with 3 vertices and 3 arrows>]>>
gap> Dimension(P);
12
```

6.7.2 CompletelyReduceGroebnerBasisForModule

▷ `CompletelyReduceGroebnerBasisForModule(GB)` (function)

Arguments: `GB` – an right Groebner basis for a (submodule of a) vertex projective module over a path algebra.

Returns: a completely reduced right Groebner basis from the entered Groebner basis `GB`.

This function takes as input an right Groebner basis for a vertex projective module or a submodule thereof, an constructs completely reduced right Groebner from it.

6.7.3 IsLeftDivisible

▷ `IsLeftDivisible(x, y)` (property)

Arguments: `x, y` – two path algebra vectors.

Returns: true if the tip of `y` left divides the tip of `x`. False otherwise.

Given two PathAlgebraVectors `x` and `y`, then `y` is said to left divide `x`, if the tip of `x` and the tip of `y` occur in the same coordinate, and the tipmonomial of the tip of `y` leftdivides the tipmonomial of the tip of `x`.

6.7.4 IsPathAlgebraModule

▷ `IsPathAlgebraModule(P)` (property)

Arguments: P – any object.
Returns: true if the argument P is in the category `IsPathAlgebraModule`.

6.7.5 IsPathAlgebraVector

▷ `IsPathAlgebraVector(v)` (property)

Arguments: v – a path algebra vector.
Returns: true if v has been constructed as a PathAlgebraVector. Otherwise it returns false.

6.7.6 LeadingCoefficient (of PathAlgebraVector)

▷ `LeadingCoefficient (of PathAlgebraVector)(x)` (operation)

Arguments: x – an element in a PathAlgebraModule.
Returns: the coefficient of the leading term/tip of a `PathAlgebraVector`.
The tip of the element x can by found by applying the command `LeadingTerm (of PathAlgebraVector)` (6.7.9).

6.7.7 LeadingComponent

▷ `LeadingComponent(v)` (operation)

Arguments: v – a path algebra vector.
Returns: `v[pos]`, where `pos` is the coordinate for the tip of the vector, whenever v is non-zero. That is, it returns the coordinate of the vector v where the tip occors. It returns zero otherwise.

6.7.8 LeadingPosition

▷ `LeadingPosition(v)` (operation)

Arguments: v – a path algebra vector.
Returns: the coordinate in which the tip of the vector occurs.

6.7.9 LeadingTerm (of PathAlgebraVector)

▷ `LeadingTerm (of PathAlgebraVector)(x)` (operation)

Arguments: x – an element in a PathAlgebraModule.
Returns: the leading term/tip of a `PathAlgebraVector`.
The tip of the element x is computed using the following order: the tip is computed for each coordinate, if the largest of these occur as a tip of several coordinates, then the coordinate with the smallest index from 1 to the length of vector is chosen. The position of the tip was computed when the `PathAlgebraVector` was created.

6.7.10 LeftDivision

▷ `LeftDivision(x, y)` (operation)

Arguments: `x, y` – two path algebra vectors.

Returns: a scalar multiple of a path, say λ such that the tips of $y * \lambda$ and x are the same, if the tip of y left divides the tip of `x`. False otherwise.

In the following example, we create two elements in P, perform some elementwise operations, and then construct a submodule using the two elements as generators.

Example
```
gap> p1 := Vectorize(P,[A.b*A.c,A.a*A.c,A.c]);
[ (Z(11)^0)*b*c, (Z(11)^0)*a*c, (Z(11)^0)*c ]
gap> p2 := Vectorize(P,[A.a,A.b,A.w]);
[ (Z(11)^0)*a, (Z(11)^0)*b, (Z(11)^0)*w ]
gap> 2*p1 + p2;
[ (Z(11)^0)*a+(Z(11))*b*c, (Z(11)^0)*b+(Z(11))*a*c,
  (Z(11)^0)*w+(Z(11))*c ]
gap> S := SubAlgebraModule(P,[p1,p2]);
<right-module over <GF(11)[<quiver with 3 vertices and 3 arrows>]>>
gap> Dimension(S);
3
```

6.7.11 ^ (a PathAlgebraModule element and a PathAlgebra element)

▷ `^(m, a)` (operation)

Arguments: `m` – an element of a path algebra module, `a` – an element of a path algebra.

Returns: the element `m` multiplied with `a`.

This action is defined by multiplying each component in `m` by `a` on the right.

Example
```
gap> p2^(A.c - A.w);
[ (Z(11)^5)*a+(Z(11)^0)*a*c, (Z(11)^5)*b+(Z(11)^0)*b*c,
  (Z(11)^5)*w+(Z(11)^0)*c ]
```

6.7.12 < (for two elements in a PathAlgebraModule)

▷ `<(m1, m2)` (operation)

Arguments: `m1, m2` – two elements of a PathAlgebraModule.

Returns: 'true' if `m1` is less than `m2` and false otherwise.

Elements are compared componentwise from left to right using the ordering of the underlying algebra. The element `m1` is less than `m2` if the first time components are not equal, the component of `m1` is less than the corresponding component of `m2`.

Example
```
gap> p1 < p2;
false
```

6.7.13 /

▷ /(*M*, *N*) (operation)

Arguments: *M*, *N* – two finite dimensional `PathAlgebraModules`.

Returns: the factor module M/N.

This module is again a right algebra module, and all applicable methods and operations are available for the resulting factor module. Furthermore, the resulting module is a vector space, so operations for computing bases and dimensions are also available.

Example
```
gap> PS := P/S;
<9-dimensional right-module over <GF(11)[<quiver with 3 vertices and
3 arrows>]>>
gap> Basis(PS);
Basis( <9-dimensional right-module over <GF(11)[<quiver with
3 vertices and 3 arrows>]>>,
[ [ [ <zero> of ..., (Z(11)^0)*v, <zero> of ... ] ],
  [ [ (Z(11)^0)*v, <zero> of ..., <zero> of ... ] ],
  [ [ <zero> of ..., <zero> of ..., (Z(11)^0)*w ] ],
  [ [ <zero> of ..., (Z(11)^0)*a, <zero> of ... ] ],
  [ [ (Z(11)^0)*a, <zero> of ..., <zero> of ... ] ],
  [ [ (Z(11)^0)*b, <zero> of ..., <zero> of ... ] ],
  [ [ <zero> of ..., <zero> of ..., (Z(11)^0)*c ] ],
  [ [ <zero> of ..., (Z(11)^0)*a*c, <zero> of ... ] ],
  [ [ (Z(11)^0)*a*c, <zero> of ..., <zero> of ... ] ] ] )
```

6.7.14 PathAlgebraVector

▷ PathAlgebraVector(*fam*, *components*) (operation)

Arguments: *fam* – a PathAlgebraVectorFamily, *components* – a homogeneous list of elements.

Returns: a PathAlgebraVector in the PathAlgebraVectorFamily *fam* with the components of the vector being equal to *components*.

This function is typically used when constructing elements of a module constructed by the command `RightProjectiveModule`. If `P` is constructed as say, `P := RightProjectiveModule(KQ, [KQ.v1, KQ.v1, KQ.v2])`, then `ExtRepOfObj(p)`, where `p` is an element if `P` is a `PathAlgebraVector`. The tip is computed using the following ordering: the tip is computed for each coordinate, if the largest of these occur as a tip of several coordinates, then the coordinate with the smallest index from 1 to the length of vector is chosen.

6.7.15 ProjectivePathAlgebraPresentation

▷ ProjectivePathAlgebraPresentation(*M*) (operation)

Arguments: *M* – a finite dimensional module over a (quotient of a) path algebra.

Returns: a projective presentation of the entered module *M* over a (qoutient of a) path algebra A. The projective presentation, or resolution is over the path algebra form which A was constructed.

This function takes as input a PathAlgebraMatModule and constructs a projetive presentation of this module over the path algebra over which it is defined, ie. a projetive resolution of length 1. It returns a list of five elements: (1) a projective module P over the path algebra, which modulo the relations induced the projective cover of `M`, (2) a submodule U of P such that P/U is isomorphic to `M`, (3) module generators of P, (4) module generators for U which forms a completely reduced right Groebner basis for U, and (5) a matrix with enteries in the path algebra which gives the map from U to P, if U were considered a direct sum of vertex projective modules over the path algebra.

6.7.16 RightGroebnerBasisOfModule

▷ `RightGroebnerBasisOfModule(M)` (attribute)

Arguments: `M` – a PathAlgebraModule.

Returns: a right Groebner basis for the module `M`.

It checks if the acting algebra on the module `M` is a path algebra, and it returns an error message otherwise. The elements in the right Groebner basis that is constructed, can be retrieved by the command `BasisVectors`. The underlying module is likewise returned by the command `UnderlyingModule`. The output of the function is satisfying the filter/category `IsRightPathAlgebraModuleGroebnerBasis`.

6.7.17 TargetVertex

▷ `TargetVertex(v)` (operation)

Arguments: `v` – a PathAlgebraVector.

Returns: a vertex w such that $v*w = v$, if such a vertex exists, and fail otherwise.

Given a PathAlgebraVector `v`, if `v` is right uniform, this function finds the vertex w such that $v*w = v$ whenever `v` is non-zero, and returns the zero path otherwise. If `v` is not right uniform it returns fail.

6.7.18 UniformGeneratorsOfModule

▷ `UniformGeneratorsOfModule(M)` (attribute)

Arguments: `M` – a `PathAlgebraModule`.

Returns: a set of right uniform generators of the mdoule `M`. If `M` is the zero module, then it returns an empty list.

6.7.19 Vectorize

▷ `Vectorize(M, components)` (function)

Arguments: `M` – a module over a path algebra, `components` – a list of elements of `M`.

Returns: a vector in `M` from a list of path algebra elements `components`, which defines the components in the resulting vector.

The returned vector is normalized, so the vector's components may not match the input components.

Chapter 7

Homomorphisms of Right Modules over Path Algebras

This chapter describes the categories, representations, attributes, and operations on homomorphisms between representations of quivers.

Given two homorphisms $f\colon L \to M$ and $g\colon M \to N$, then the composition is written $f * g$. The elements in the modules or the representations of a quiver are row vectors. Therefore the homomorphisms between two modules are acting on these row vectors, that is, if m_i is in $M[i]$ and $g_i\colon M[i] \to N[i]$ represents the linear map, then the value of g applied to m_i is the matrix product $m_i * g_i$.

The example used throughout this chapter is the following.

Example
```
gap> Q := Quiver(3,[[1,2,"a"],[1,2,"b"],[2,2,"c"],[2,3,"d"],[3,1,"e"]]);;
gap> KQ := PathAlgebra(Rationals, Q);;
gap> AssignGeneratorVariables(KQ);;
gap> rels := [d*e,c^2,a*c*d-b*d,e*a];;
gap> A := KQ/rels;;
gap> mat :=[["a",[[1,2],[0,3],[1,5]]],["b",[[2,0],[3,0],[5,0]]],
> ["c",[[0,0],[1,0]]],["d",[[1,2],[0,1]]],["e",[[0,0,0],[0,0,0]]]];;
gap> N := RightModuleOverPathAlgebra(A,mat);;
```

7.1 Categories and representation of homomorphisms

7.1.1 IsPathAlgebraModuleHomomorphism

▷ `IsPathAlgebraModuleHomomorphism(f)` (filter)

Arguments: `f` - any object in GAP.

Returns: true or false depending on if `f` belongs to the categories `IsPathAlgebraModuleHomomorphism`.

This defines the category `IsPathAlgebraModuleHomomorphism`.

7.1.2 RightModuleHomOverAlgebra

▷ `RightModuleHomOverAlgebra(M, N, mats)` (operation)

Arguments: *M*, *N* - two modules over the same (quotient of a) path algebra, *mats* - a list of matrices, one for each vertex in the quiver of the path algebra.

Returns: a homomorphism in the category `IsPathAlgebraModuleHomomorphism` from the module *M* to the module *N* given by the matrices *mats*.

The arguments *M* and *N* are two modules over the same algebra (this is checked), and *mats* is a list of matrices `mats[i]`, where `mats[i]` represents the linear map from `M[i]` to `N[i]` with `i` running through all the vertices in the same order as when the underlying quiver was created. If both `DimensionVector(M)[i]` and `DimensionVector(N)[i]` are non-zero, then `mats[i]` is a `DimensionVector(M)[i]` by `DimensionVector(N)[i]` matrix. If `DimensionVector(M)[i]` is zero and `DimensionVector(N)[i]` is non-zero, then `mats[i]` must be the zero `1` by `DimensionVector(N)[i]` matrix. Similarly for the other way around. If both `DimensionVector(M)[i]` and `DimensionVector(N)[i]` are zero, then `mats[i]` must be the `1` by `1` zero matrix. The function checks if *mats* is a homomorphism from the module *M* to the module *N* by checking that the matrices given in *mats* have the correct size and satisfy the appropriate commutativity conditions with the matrices in the modules given by *M* and *N*. The source (or domain) and the range (or codomain) of the homomorphism constructed can by obtained again by `Range` (7.2.22) and by `Source` (7.2.24), respectively.

Example
```
gap> L := RightModuleOverPathAlgebra(A,[["a",[0,1]],["b",[0,1]],
> ["c",[[0]]],["d",[[1]]],["e",[1,0]]]);
<[ 0, 1, 1 ]>
gap> DimensionVector(L);
[ 0, 1, 1 ]
gap> f := RightModuleHomOverAlgebra(L,N,[[[0,0,0]], [[1,0]],
> [[1,2]]]);
<<[ 0, 1, 1 ]> ---> <[ 3, 2, 2 ]>>

gap> IsPathAlgebraMatModuleHomomorphism(f);
true
```

7.2 Generalities of homomorphisms

7.2.1 \= (maps)

▷ `\= (maps)(f, g)` (operation)

Arguments: *f*, *g* - two homomorphisms between two modules.

Returns: true, if `Source(f) = Source(g)`, `Range(f) = Range(g)`, and the matrices defining the maps *f* and *g* coincide.

7.2.2 \+ (maps)

▷ `\+ (maps)(f, g)` (operation)

Arguments: *f*, *g* - two homomorphisms between two modules.

Returns: the sum *f+g* of the maps *f* and *g*.

The function checks if the maps have the same source and the same range, and returns an error message otherwise.

7.2.3 * (maps)

▷ `\* (maps)(f, g)` (operation)

Arguments: `f`, `g` - two homomorphisms between two modules, or one scalar and one homomorphism between modules.

Returns: the composition `fg` of the maps `f` and `g`, if the input are maps between representations of the same quivers. If `f` or `g` is a scalar, it returns the natural action of scalars on the maps between representations.

The function checks if the maps are composable, in the first case and in the second case it checks if the scalar is in the correct field, and returns an error message otherwise.

7.2.4 CoKernelOfWhat

▷ `CoKernelOfWhat(f)` (attribute)

Arguments: `f` - a homomorphism between two modules.

Returns: a homomorphism `g`, if `f` has been computed as the cokernel of the homomorphism `g`.

7.2.5 IdentityMapping

▷ `IdentityMapping(M)` (operation)

Arguments: `M` - a module.

Returns: the identity map between `M` and `M`.

7.2.6 ImageElm

▷ `ImageElm(f, elem)` (operation)

Arguments: `f` - a homomorphism between two modules, `elem` - an element in the source of `f`.

Returns: the image of the element `elem` in the source (or domain) of the homomorphism `f`.

The function checks if `elem` is an element in the source of `f`, and it returns an error message otherwise.

7.2.7 ImagesSet

▷ `ImagesSet(f, elts)` (operation)

Arguments: `f` - a homomorphism between two modules, `elts` - an element in the source of `f`, or the source of `f`.

Returns: the non-zero images of a set of elements `elts` in the source of the homomorphism `f`, or if `elts` is the source of `f`, it returns a basis of the image.

The function checks if the set of elements `elts` consists of elements in the source of `f`, and it returns an error message otherwise.

Example
```
gap> B := BasisVectors(Basis(N));
[ [ [ 1, 0, 0 ], [ 0, 0 ], [ 0, 0 ] ],
  [ [ 0, 1, 0 ], [ 0, 0 ], [ 0, 0 ] ],
  [ [ 0, 0, 1 ], [ 0, 0 ], [ 0, 0 ] ],
  [ [ 0, 0, 0 ], [ 1, 0 ], [ 0, 0 ] ],
  [ [ 0, 0, 0 ], [ 0, 1 ], [ 0, 0 ] ],
  [ [ 0, 0, 0 ], [ 0, 0 ], [ 1, 0 ] ],
  [ [ 0, 0, 0 ], [ 0, 0 ], [ 0, 1 ] ] ]
gap> PreImagesRepresentative(f,B[4]);
[ [ 0 ], [ 1 ], [ 0 ] ]
gap> PreImagesRepresentative(f,B[5]);
fail
gap> BL := BasisVectors(Basis(L));
[ [ [ 0 ], [ 1 ], [ 0 ] ], [ [ 0 ], [ 0 ], [ 1 ] ] ]
gap> ImageElm(f,BL[1]);
[ [ 0, 0, 0 ], [ 1, 0 ], [ 0, 0 ] ]
gap> ImagesSet(f,L);
[ [ [ 0, 0, 0 ], [ 1, 0 ], [ 0, 0 ] ],
  [ [ 0, 0, 0 ], [ 0, 0 ], [ 1, 2 ] ] ]
gap> ImagesSet(f,BL);
[ [ [ 0, 0, 0 ], [ 1, 0 ], [ 0, 0 ] ],
  [ [ 0, 0, 0 ], [ 0, 0 ], [ 1, 2 ] ] ]
gap> z := Zero(f);;
gap> f = z;
false
gap> Range(f) = Range(z);
true
gap> y := ZeroMapping(L,N);;
gap> y = z;
true
gap> id := IdentityMapping(N);;
gap> f*id;;
gap> #This causes an error!
gap> id*f;
Error, codomain of the first argument is not equal to the domain of th\
e second argument,  called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;;
gap> 2*f + z;
<<[ 0, 1, 1 ]> ---> <[ 3, 2, 2 ]>>
```

7.2.8 ImageOfWhat

▷ ImageOfWhat(*f*) (attribute)

Arguments: *f* - a homomorphism between two modules.

Returns: a homomorphism g, if *f* has been computed as the image projection or the image inclusion of the homomorphism g.

7.2.9 IsInjective

▷ `IsInjective(f)` (property)

Arguments: *f* - a homomorphism between two modules.
Returns: `true` if the homomorphism *f* is one-to-one.

7.2.10 IsIsomorphism

▷ `IsIsomorphism(f)` (operation)

Arguments: *f* - a homomorphism between two modules.
Returns: `true` if the homomorphism *f* is an isomorphism.

7.2.11 IsLeftMinimal

▷ `IsLeftMinimal(f)` (property)

Arguments: *f* - a homomorphism between two modules.
Returns: `true` if the homomorphism *f* is left minimal.

7.2.12 IsRightMinimal

▷ `IsRightMinimal(f)` (property)

Arguments: *f* - a homomorphism between two modules.
Returns: `true` if the homomorphism *f* is right minimal.

Example
```
gap> L := RightModuleOverPathAlgebra(A,[["a",[0,1]],["b",[0,1]],
> ["c",[[0]]],["d",[[1]]],["e",[1,0]]]);;
gap> f := RightModuleHomOverAlgebra(L,N,[[[0,0,0]], [[1,0]],
> [[1,2]]]);
<<[ 0, 1, 1 ]> ---> <[ 3, 2, 2 ]>>

gap> g := CoKernelProjection(f);
<<[ 3, 2, 2 ]> ---> <[ 3, 1, 1 ]>>

gap> CoKernelOfWhat(g) = f;
true
gap> h := ImageProjection(f);
<<[ 0, 1, 1 ]> ---> <[ 0, 1, 1 ]>>

gap> ImageOfWhat(h) = f;
true
gap> IsInjective(f); IsSurjective(f); IsIsomorphism(f);
true
false
false
gap> IsIsomorphism(h);
true
```

7.2.13 IsSplitEpimorphism

▷ `IsSplitEpimorphism(f)` (property)

Arguments: `f` - a homomorphism between two modules.
Returns: `true` if the homomorphism `f` is a splittable epimorphism, otherwise `false`.

7.2.14 IsSplitMonomorphism

▷ `IsSplitMonomorphism(f)` (property)

Arguments: `f` - a homomorphism between two modules.
Returns: `true` if the homomorphism `f` is a splittable monomorphism, otherwise `false`.

Example
```
gap> S := SimpleModules(A)[1];;
gap> H := HomOverAlgebra(N,S);;
gap> IsSplitMonomorphism(H[1]);
false
gap> IsSplitEpimorphism(H[1]);
true
```

7.2.15 IsSurjective

▷ `IsSurjective(f)` (property)

Arguments: `f` - a homomorphism between two modules.
Returns: `true` if the homomorphism `f` is onto.

7.2.16 IsZero

▷ `IsZero(f)` (property)

Arguments: `f` - a homomorphism between two modules.
Returns: `true` if the homomorphism `f` is a zero homomorphism.

7.2.17 KernelOfWhat

▷ `KernelOfWhat(f)` (attribute)

Arguments: `f` - a homomorphism between two modules.
Returns: a homomorphism `g`, if `f` has been computed as the kernel of the homomorphism `g`.

Example
```
gap> L := RightModuleOverPathAlgebra(A,[["a",[0,1]],["b",[0,1]],
> ["c",[[0]]],["d",[[1]]],["e",[1,0]]]);
<[ 0, 1, 1 ]>
gap> f := RightModuleHomOverAlgebra(L,N,[[[0,0,0]], [[1,0]],
> [[1,2]]]);;
gap> IsZero(0*f);
true
```

```
gap> g := KernelInclusion(f);
<<[ 0, 0, 0 ]> ---> <[ 0, 1, 1 ]>>

gap> KnownAttributesOfObject(g);
[ "Range", "Source", "PathAlgebraOfMatModuleMap", "KernelOfWhat" ]
gap> KernelOfWhat(g) = f;
true
```

7.2.18 LeftInverseOfHomomorphism

▷ `LeftInverseOfHomomorphism(f)` (attribute)

Arguments: *f* - a homomorphism between two modules.

Returns: `false` if the homomorphism *f* is not a splittable epimorphism, otherwise it returns a splitting of the split epimorphism *f*.

7.2.19 MatricesOfPathAlgebraMatModuleHomomorphism

▷ `MatricesOfPathAlgebraMatModuleHomomorphism(f)` (operation)

Arguments: *f* - a homomorphism between two modules.

Returns: the matrices defining the homomorphism *f*.

Example
```
gap> MatricesOfPathAlgebraMatModuleHomomorphism(f);
[ [ [ 0, 0, 0 ] ], [ [ 1, 0 ] ], [ [ 1, 2 ] ] ]
gap> Range(f);
<[ 3, 2, 2 ]>
gap> Source(f);
<[ 0, 1, 1 ]>
gap> Source(f) = L;
true
```

7.2.20 PathAlgebraOfMatModuleMap

▷ `PathAlgebraOfMatModuleMap(f)` (attribute)

Arguments: *f* – a homomorphism between two path algebra modules (`PathAlgebraMatModule`).

Returns: the algebra over which the range and the source of the homomorphism *f* is defined.

7.2.21 PreImagesRepresentative

▷ `PreImagesRepresentative(f, elem)` (operation)

Arguments: *f* - a homomorphism between two modules, *elem* - an element in the range of *f*.

Returns: a preimage of the element *elem* in the range (or codomain) the homomorphism *f* if a preimage exists, otherwise it returns `fail`.

The function checks if *elem* is an element in the range of *f* and returns an error message if not.

7.2.22 Range

▷ `Range(f)` (attribute)

Arguments: `f` - a homomorphism between two modules.
Returns: the range (or codomain) the homomorphism `f`.

7.2.23 RightInverseOfHomomorphism

▷ `RightInverseOfHomomorphism(f)` (attribute)

Arguments: `f` - a homomorphism between two modules.
Returns: `false` if the homomorphism `f` is not a splittable monomorphism, otherwise it returns a splitting of the split monomorphism `f`.

7.2.24 Source

▷ `Source(f)` (attribute)

Arguments: `f` - a homomorphism between two modules.
Returns: the source (or domain) the homomorphism `f`.

7.2.25 Zero

▷ `Zero(f)` (operation)

Arguments: `f` - a homomorphism between two modules.
Returns: the zero map between `Source(f)` and `Range(f)`.

7.2.26 ZeroMapping

▷ `ZeroMapping(M, N)` (operation)

Arguments: `M`, `N` - two modules.
Returns: the zero map between `M` and `N`.

7.2.27 HomomorphismFromImages

▷ `HomomorphismFromImages(M, N, genImages)` (operation)

Arguments: `M`, `N` – two modules, `genImages` – a list.
Returns: A map *f* between `M` and `N`, given by `genImages`.

Let `B` be the basis `BasisVectors( Basis( M ) )` of `M`. Then the number of elements of `genImages` should be equal to the number of elements of `B`, and `genImages[i]` is an element of `N` and the image of `B[i]` under `f`. The method fails if `f` is not a homomorphism, or if `B[i]` and `genImages[i]` are supported in different vertices.

7.3 Homomorphisms and modules constructed from homomorphisms and modules

7.3.1 CoKernel

▷ `CoKernel(f)` (attribute)

Arguments: `f` - a homomorphism between two modules.
Returns: the cokernel of a homomorphism `f` between two modules.
This function returns the cokernel of the homomorphism `f` as a module.

7.3.2 CoKernelProjection

▷ `CoKernelProjection(f)` (attribute)

Arguments: `f` - a homomorphism between two modules.
Returns: the cokernel of a homomorphism `f` between two modules.
This function returns the cokernel of the homomorphism `f` as the projection homomorphism from the range of the homomorphism `f` to the cokernel of the homomorphism `f`.

7.3.3 EndModuloProjOverAlgebra

▷ `EndModuloProjOverAlgebra(M)` (operation)

Arguments: `M` - a module.
Returns: the natural homomorphism from the endomorphism ring of `M` to the endomorphism ring of `M` modulo the ideal generated by those endomorphisms of `M` which factor through a projective module.
The operation returns an error message if the zero module is entered as an argument.

7.3.4 EndOfModuleAsQuiverAlgebra

▷ `EndOfModuleAsQuiverAlgebra(M)` (operation)

Arguments: `M` - a PathAlgebraMatModule.
Returns: a list of three elements, (i) the endomorphism ring of `M`, (ii) the adjacency matrix of the quiver of the endomorphism ring and (iii) the endomorphism ring as a quiver algebra.
Suppose `M` is a module over a quiver algebra over a field K. The function checks if the endomorphism ring of `M` is K-elementary (not necessary for it to be a quiver algebra, but this is a TODO improvement), and returns error message otherwise.

7.3.5 EndOverAlgebra

▷ `EndOverAlgebra(M)` (attribute)

Arguments: `M` - a module.
Returns: the endomorphism ring of `M` as a subalgebra of the direct sum of the full matrix rings

of `DimensionVector(M)[i] x DimensionVector(M)[i]`, where *i* runs over all vertices where `DimensionVector(M)[i]` is non-zero.

The endomorphism is an algebra with one, and one can apply for example `RadicalOfAlgebra` to find the radical of the endomorphism ring.

7.3.6 FromEndMToHomMM

▷ `FromEndMToHomMM(f)` (operation)

Arguments: *f* – an element in `EndOverAlgebra(M)`.

Returns: the homomorphism from *M* to *M* corresponding to the element *f* in the endomorphism ring `EndOverAlgebra(M)` of *M*.

7.3.7 FromHomMMToEndM

▷ `FromHomMMToEndM(f)` (operation)

Arguments: *f* – an element in `HomOverAlgebra(M,M)`.

Returns: the element *f* in the endomorphism ring `EndOverAlgebra(M)` of *M* corresponding to the the homomorphism from *M* to *M* given by *f*.

7.3.8 HomFactoringThroughProjOverAlgebra

▷ `HomFactoringThroughProjOverAlgebra(M, N)` (operation)

Arguments: *M*, *N* - two modules.

Returns: a basis for the vector space of homomorphisms from *M* to *N* which factors through a projective module.

The function checks if *M* and *N* are modules over the same algebra, and returns an error message otherwise.

7.3.9 HomFromProjective

▷ `HomFromProjective(m, M)` (operation)

Arguments: *m*, *M* - an element and a module.

Returns: the homomorphism from the indecomposable projective module defined by the support of the element *m* to the module *M*.

The function checks if *m* is an elememt in *M* and if the element *m* is supported in only one vertex. Otherwise it returns fail.

7.3.10 HomOverAlgebra

▷ `HomOverAlgebra(M, N)` (operation)

Arguments: *M*, *N* - two modules.

Returns: a basis for the vector space of homomorphisms from *M* to *N*.

The function checks if *M* and *N* are modules over the same algebra, and returns an error message and fail otherwise.

7.3.11 Image

▷ `Image(f)` (attribute)

Arguments: *f* - a homomorphism between two modules.
Returns: the image of a homomorphism `f` as a module.

7.3.12 ImageInclusion

▷ `ImageInclusion(f)` (attribute)

Arguments: *f* - a homomorphism between two modules.
Returns: the inclusion of the image of a homomorphism `f` into the range of `f`.

7.3.13 ImageProjection

▷ `ImageProjection(f)` (attribute)

Arguments: *f* - a homomorphism between two modules.
Returns: the projection from the source of `f` to the image of the homomorphism `f`.

7.3.14 ImageProjectionInclusion

▷ `ImageProjectionInclusion(f)` (attribute)

Arguments: *f* - a homomorphism between two modules.
Returns: both the projection from the source of `f` to the image of the homomorphism `f` and the inclusion of the image of a homomorphism `f` into the range of `f` as a list of two elements (first the projection and then the inclusion).

7.3.15 IsomorphismOfModules

▷ `IsomorphismOfModules(M, N)` (operation)

Arguments: *M, N* - two PathAlgebraMatModules.
Returns: false if *M* and *N* are two non-isomorphic modules, otherwise it returns an isomorphism from *M* to *N*.
The function checks if *M* and *N* are modules over the same algebra, and returns an error message otherwise.

7.3.16 Kernel

▷ `Kernel(f)` (attribute)
▷ `KernelInclusion(f)` (attribute)

Arguments: *f* - a homomorphism between two modules.

Returns: the kernel of a homomorphism *f* between two modules.

The first variant `Kernel` returns the kernel of the homomorphism *f* as a module, while the latter one returns the inclusion homomorphism of the kernel into the source of the homomorphism *f*.

Example
```
gap> hom := HomOverAlgebra(N,N);
[ <<[ 3, 2, 2 ]> ---> <[ 3, 2, 2 ]>>
    , <<[ 3, 2, 2 ]> ---> <[ 3, 2, 2 ]>>
    , <<[ 3, 2, 2 ]> ---> <[ 3, 2, 2 ]>>
    , <<[ 3, 2, 2 ]> ---> <[ 3, 2, 2 ]>>
    , <<[ 3, 2, 2 ]> ---> <[ 3, 2, 2 ]>>
     ]
gap> g := hom[1];
<<[ 3, 2, 2 ]> ---> <[ 3, 2, 2 ]>>

gap> M := CoKernel(g);
<[ 2, 2, 2 ]>
gap> f := CoKernelProjection(g);
<<[ 3, 2, 2 ]> ---> <[ 2, 2, 2 ]>>

gap> Range(f) = M;
true
gap> endo := EndOverAlgebra(N);
<algebra-with-one of dimension 5 over Rationals>
gap> RadicalOfAlgebra(endo);
<algebra of dimension 3 over Rationals>
gap> B := BasisVectors(Basis(N));
[ [ [ 1, 0, 0 ], [ 0, 0 ], [ 0, 0 ] ],
  [ [ 0, 1, 0 ], [ 0, 0 ], [ 0, 0 ] ],
  [ [ 0, 0, 1 ], [ 0, 0 ], [ 0, 0 ] ],
  [ [ 0, 0, 0 ], [ 1, 0 ], [ 0, 0 ] ],
  [ [ 0, 0, 0 ], [ 0, 1 ], [ 0, 0 ] ],
  [ [ 0, 0, 0 ], [ 0, 0 ], [ 1, 0 ] ],
  [ [ 0, 0, 0 ], [ 0, 0 ], [ 0, 1 ] ] ]
gap> p := HomFromProjective(B[1],N);
<<[ 1, 4, 3 ]> ---> <[ 3, 2, 2 ]>>

gap> U := Image(p);
<[ 1, 2, 2 ]>
gap> projinc := ImageProjectionInclusion(p);
[ <<[ 1, 4, 3 ]> ---> <[ 1, 2, 2 ]>>
    , <<[ 1, 2, 2 ]> ---> <[ 3, 2, 2 ]>>
     ]
gap> U = Range(projinc[1]);
true
gap> Kernel(p);
<[ 0, 2, 1 ]>
```

7.3.17 LeftMinimalVersion

▷ `LeftMinimalVersion(f)` (attribute)

Arguments: *f* - a homomorphism between two modules.

Returns: the left minimal version `f'` of the homomorphism `f` together with the a list `B` of modules such that the direct sum of the modules, `Range(f')` and the modules in the list `B` is isomorphic to `Range(f)`.

7.3.18 RightMinimalVersion

▷ `RightMinimalVersion(f)` (attribute)

Arguments: *f* - a homomorphism between two modules.

Returns: the right minimal version `f'` of the homomorphism `f` together with the a list `B` of modules such that the direct sum of the modules, `Source(f')` and the modules on the list `B` is isomorphic to `Source(f)`.

Example

```
gap> H:= HomOverAlgebra(N,N);;
gap> RightMinimalVersion(H[1]);
[ <<[ 1, 0, 0 ]> ---> <[ 3, 2, 2 ]>>
    , [ <[ 2, 2, 2 ]> ] ]
gap> LeftMinimalVersion(H[1]);
[ <<[ 3, 2, 2 ]> ---> <[ 1, 0, 0 ]>>
    , [ <[ 2, 2, 2 ]> ] ]
gap> S := SimpleModules(A)[1];;
gap> MinimalRightApproximation(N,S);
<<[ 1, 0, 0 ]> ---> <[ 1, 0, 0 ]>>

gap> S := SimpleModules(A)[3];;
gap> MinimalLeftApproximation(S,N);
<<[ 0, 0, 1 ]> ---> <[ 2, 2, 2 ]>>
```

7.3.19 RadicalOfModuleInclusion

▷ `RadicalOfModuleInclusion(M)` (attribute)

Arguments: *M* - a module.

Returns: the inclusion of the radical of the module *M* into *M*.

The radical of *M* can be accessed using `Source`, or it can be computed directly via the command `RadicalOfModule` (6.4.32).

7.3.20 SocleOfModuleInclusion

▷ `SocleOfModuleInclusion(M)` (operation)

Arguments: *M* - a module.

Returns: the inclusion of the socle of the module *M* into *M*.

The socle of *M* can be accessed using `Source`, or it can be computed directly via the command `SocleOfModule` (6.4.35).

7.3.21 SubRepresentationInclusion

▷ `SubRepresentationInclusion(`*`M, gens`*`)` (operation)

Arguments: *`M`* - a module, *`gens`* - a list of elements in *`M`*.

Returns: the inclusion of the submodule generated by the generators *`gens`* into the module *`M`*.

The function checks if *`gens`* consists of elements in *`M`*, and returns an error message otherwise. The module given by the submodule generated by the generators *`gens`* can be accessed using `Source`.

7.3.22 TopOfModuleProjection

▷ `TopOfModuleProjection(`*`M`*`)` (operation)

Arguments: *`M`* - a module.

Returns: the projection from the module *`M`* to the top of the module *`M`*.

The module given by the top of the module *`M`* can be accessed using `Range` of the homomorphism.

Example
```
gap> f := RadicalOfModuleInclusion(N);
<<[ 0, 2, 2 ]> ---> <[ 3, 2, 2 ]>>

gap> radN := Source(f);
<[ 0, 2, 2 ]>
gap> g := SocleOfModuleInclusion(N);
<<[ 1, 0, 2 ]> ---> <[ 3, 2, 2 ]>>

gap> U := SubRepresentationInclusion(N,[B[5]+B[6],B[7]]);
<<[ 0, 2, 2 ]> ---> <[ 3, 2, 2 ]>>

gap> h := TopOfModuleProjection(N);
<<[ 3, 2, 2 ]> ---> <[ 3, 0, 0 ]>>
```

7.3.23 TraceOfModule

▷ `TraceOfModule(`*`M, N`*`)` (operation)

Arguments: *`M`*, *`C`* – two path algebra modules (`PathAlgebraMatModule`).

Returns: the trace of the module *`M`* in the module *`N`* as an inclusion homomorhpism from the trace of *`M`* to *`N`*.

Chapter 8

Homological algebra

This chapter describes the homological algebra that is implemented in QPA.

The example used throughout this chapter is the following.

```
Example
gap> Q := Quiver(3,[[1,2,"a"],[1,2,"b"],[2,2,"c"],[2,3,"d"],
> [3,1,"e"]]);;
gap> KQ := PathAlgebra(Rationals, Q);;
gap> AssignGeneratorVariables(KQ);;
gap> rels := [d*e,c^2,a*c*d-b*d,e*a];;
gap> A := KQ/rels;;
gap> mat := [["a",[[1,2],[0,3],[1,5]]],["b",[[2,0],[3,0],[5,0]]],
> ["c",[[0,0],[1,0]]],["d",[[1,2],[0,1]]],["e",[[0,0,0],[0,0,0]]]];;
gap> N := RightModuleOverPathAlgebra(A,mat);;
```

8.1 Homological algebra

8.1.1 1stSyzygy

▷ `1stSyzygy(`*M*`)` (attribute)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`).
Returns: the first syzygy of the representation *M* as a representation.

8.1.2 AllComplementsOfAlmostCompleteTiltingModule

▷ `AllComplementsOfAlmostCompleteTiltingModule(`*M*`, `*X*`)` (operation)
▷ `AllComplementsOfAlmostCompleteCotiltingModule(`*M*`, `*X*`)` (operation)

Arguments: *M*, *X* - two PathAlgebraMatModule's.
Returns: all the complements of the almost complete (co-)tilting module *M* as two exact sequences, the first is all complements which are gotten as an *add M*-resolution of *X* and the second is all complements which are gotten as an *add M*-coresolution of *X*. If there are no complements to the left of *X*, then an empty list is returned. Similarly for to the right of *X*. In particular, if *X* has no other complements the list `[[],[]]` is returned.

8.1.3 CotiltingModule

▷ `CotiltingModule(M, n)` (operation)

Arguments: `M`, `n` - a PathAlgebraMatModule and a positive integer.

Returns: false if `M` is not a cotilting module of injective dimension at most `a`. Otherwise, it returns the injective dimension of `M` and the resolution of all indecomposable injective modules in *add M*.

8.1.4 DominantDimensionOfAlgebra

▷ `DominantDimensionOfAlgebra(A, n)` (operation)

Arguments: `A`, `n` - a quiver algebra, a positive integer.

Returns: the dominant dimension of the algebra `A` if the dominant dimension is less or equal to `n`. If the function can decide that the dominant dimension is infinite, it returns `infinity`. Otherwise, if the dominant dimension is larger than `n`, then it returns `false`.

8.1.5 DominantDimensionOfModule

▷ `DominantDimensionOfModule(M, n)` (operation)

Arguments: `M`, `n` - a PathAlgebraMatModule, a positive integer.

Returns: the dominant dimension of the module `M` if the dominant dimension is less or equal to `n`. If the function can decide that the dominant dimension is infinite, it returns `infinity`. Otherwise, if the dominant dimension is larger than `n`, then it returns `false`.

8.1.6 ExtAlgebraGenerators

▷ `ExtAlgebraGenerators(M, n)` (operation)

Arguments: `M` - a module, `n` - a positive integer.

Returns: a list of three elements, where the first element is the dimensions of Ext^[0..n](M,M), the second element is the number of minimal generators in the degrees [0..n], and the third element is the generators in these degrees.

This function computes the generators of the Ext-algebra $Ext^*(M,M)$ up to degree `n`.

8.1.7 ExtOverAlgebra

▷ `ExtOverAlgebra(M, N)` (operation)

Arguments: `M`, `N` - two modules.

Returns: a list of three elements `ExtOverAlgebra`, where the first element is the map from the first syzygy, $\Omega(M)$ to the projective cover, $P(M)$ of the module `M`, the second element is a basis of $\mathrm{Ext}^1(M,N)$ in terms of elements in $\mathrm{Hom}(\Omega(M),N)$ and the third element is a function that takes as an argument a homomorphism in `Hom(Omega(M),N)` and returns the coefficients of this element when written in terms of the basis of $\mathrm{Ext}^1(M,N)$.

The function checks if the arguments `M` and `N` are modules of the same algebra, and returns an error message otherwise. It $\mathrm{Ext}^1(M,N)$ is zero, an empty list is returned.

8.1.8 FaithfulDimension

▷ `FaithfulDimension(M)` (attribute)

Arguments: `M` - a PathAlgebraMatModule.
Returns: the faithful dimension of the module `M`.

8.1.9 GlobalDimensionOfAlgebra

▷ `GlobalDimensionOfAlgebra(A, n)` (operation)

Arguments: `A`, `n` - a quiver algebra, a positive integer.
Returns: the global dimension of `A` if the global dimension is less or equal to `n`. If the function can decide that the global dimension is infinite, it returns `infinity`. Otherwise, if the global dimension is larger than `n`, then it returns `false`.

8.1.10 GorensteinDimension

▷ `GorensteinDimension(A)` (attribute)

Arguments: `A` - a quiver algebra.
Returns: the Gorenstein dimension of `A`, if the Gorenstein dimension has been computed. Otherwise it returns an error message.

8.1.11 GorensteinDimensionOfAlgebra

▷ `GorensteinDimensionOfAlgebra(A, n)` (operation)

Arguments: `A`, `n` - a quiver algebra, a positive integer.
Returns: the Gorenstein dimension of `A` if the Gorenstein dimension is less or equal to `n`. Otherwise, if the Gorenstein dimension is larger than `n`, then it returns `false`.

8.1.12 HaveFiniteCoresolutionInAddM

▷ `HaveFiniteCoresolutionInAddM(N, M, n)` (operation)

Arguments: `N`, `M`, `n` - two PathAlgebraMatModule's and an integer.
Returns: false if `N` does not have a coresolution of length at most `n` in `add M`, otherwise it returns the coresolution of `N` of length at most `n`.

8.1.13 HaveFiniteResolutionInAddM

▷ `HaveFiniteResolutionInAddM(N, M, n)` (operation)

Arguments: *N*, *M*, *n* - two PathAlgebraMatModule's and an integer.
Returns: false if *N* does not have a resolution of length at most *n* in *add M*, otherwise it returns the resolution of *N* of length at most *n*.

8.1.14 InjDimension

▷ `InjDimension(M)` (attribute)

Arguments: *M* - a PathAlgebraMatModule.

If the injetive dimension of the module *M* has been computed, then the projective dimension is returned.

8.1.15 InjDimensionOfModule

▷ `InjDimensionOfModule(M, n)` (operation)

Arguments: *M, n* - a PathAlgebraMatModule, a positive integer.
Returns: Returns the injective dimension of the module *M* if it is less or equal to *n*. Otherwise it returns false.

8.1.16 IsCotiltingModule

▷ `IsCotiltingModule(M)` (attribute)

Arguments: *M* - a PathAlgebraMatModule.
Returns: true if the module *M* has been checked to be a cotilting mdoule, otherwise it returns an error message.

8.1.17 IsOmegaPeriodic

▷ `IsOmegaPeriodic(M, n)` (operation)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`), *n* – be a positive integer.
Returns: `i`, where `i` is the smallest positive integer less or equal `n` such that the representation *M* is isomorphic to the `i`-th syzygy of *M*, and false otherwise.

8.1.18 IsTtiltingModule

▷ `IsTtiltingModule(M)` (attribute)

Arguments: *M* - a PathAlgebraMatModule.
Returns: true if the module *M* has been checked to be a tilting mdoule, otherwise it returns an error message.

8.1.19 IyamaGenerator

▷ `IyamaGenerator(M)` (operation)

Arguments: `M` – a path algebra module (`PathAlgebraMatModule`).
Returns: a module N such that `M` is a direct summand of N and such that the global dimension of the endomorphism ring of N is finite using the algorithm provided by Osamu Iyama (add reference here).

8.1.20 LeftFacMApproximation

▷ `LeftFacMApproximation(C, M)` (operation)
▷ `MinimalLeftFacMApproximation(C, M)` (operation)

Arguments: `C`, `M` – two path algebra modules (`PathAlgebraMatModule`).
Returns: a left $\mathrm{Fac}M$-approximation of the module C, where the first version returns a not necessarily minimal left $\mathrm{Fac}M$-approximation and the second returns a minimal approximation.

8.1.21 LeftMutationOfTiltingModuleComplement

▷ `LeftMutationOfTiltingModuleComplement(M, N)` (operation)
▷ `LeftMutationOfCotiltingModuleComplement(M, N)` (operation)

Arguments: `M`, `N` – two path algebra modules (`PathAlgebraMatModule`).
Returns: a left mutation of the complement `N` of the almost complete tilting/cotilting module `M`, if such a complement exists. Otherwise it returns false.

8.1.22 LeftSubMApproximation

▷ `LeftSubMApproximation(C, M)` (operation)
▷ `MinimalLeftSubMApproximation(C, M)` (operation)

Arguments: `C`, `M` – two path algebra modules (`PathAlgebraMatModule`).
Returns: a minimal left $\mathrm{Sub}M$-approximation of the module C.

8.1.23 LiftingInclusionMorphisms

▷ `LiftingInclusionMorphisms(f, g)` (operation)

Arguments: `f`, `g` - two homomorphisms with common range.
Returns: a factorization of g in terms of `f`, whenever possible and `fail` otherwise.

Given two inclusions $f\colon B \to C$ and $g\colon A \to C$, this function constructs a morphism from A to B, whenever the image of g is contained in the image of `f`. Otherwise the function returns fail. The function checks if `f` and g are one-to-one, if they have the same range and if the image of g is contained in the image of `f`.

8.1.24 LiftingMorphismFromProjective

▷ `LiftingMorphismFromProjective(f, g)` (operation)

Arguments: `f`, g - two homomorphisms with common range.

Returns: a factorization of g in terms of `f`, whenever possible and `fail` otherwise.

Given two morphisms $f\colon B \to C$ and $g\colon P \to C$, where P is a direct sum of indecomposable projective modules constructed via `DirectSumOfModules` and `f` an epimorphism, this function finds a lifting of g to B. The function checks if P is a direct sum of indecomposable projective modules, if `f` is onto and if `f` and g have the same range.

Example

```
gap> B := BasisVectors(Basis(N));
[ [ [ 1, 0, 0 ], [ 0, 0 ], [ 0, 0 ] ],
  [ [ 0, 1, 0 ], [ 0, 0 ], [ 0, 0 ] ],
  [ [ 0, 0, 1 ], [ 0, 0 ], [ 0, 0 ] ],
  [ [ 0, 0, 0 ], [ 1, 0 ], [ 0, 0 ] ],
  [ [ 0, 0, 0 ], [ 0, 1 ], [ 0, 0 ] ],
  [ [ 0, 0, 0 ], [ 0, 0 ], [ 1, 0 ] ],
  [ [ 0, 0, 0 ], [ 0, 0 ], [ 0, 1 ] ] ]
gap> g := SubRepresentationInclusion(N,[B[1],B[4]]);
<<[ 1, 2, 2 ]> ---> <[ 3, 2, 2 ]>>

gap> f := SubRepresentationInclusion(N,[B[1],B[2]]);
<<[ 2, 2, 2 ]> ---> <[ 3, 2, 2 ]>>

gap> LiftingInclusionMorphisms(f,g);
<<[ 1, 2, 2 ]> ---> <[ 2, 2, 2 ]>>

gap> S := SimpleModules(A);
[ <[ 1, 0, 0 ]>, <[ 0, 1, 0 ]>, <[ 0, 0, 1 ]> ]
gap> homNS := HomOverAlgebra(N,S[1]);
[ <<[ 3, 2, 2 ]> ---> <[ 1, 0, 0 ]>>
    , <<[ 3, 2, 2 ]> ---> <[ 1, 0, 0 ]>>
    , <<[ 3, 2, 2 ]> ---> <[ 1, 0, 0 ]>>
     ]
gap> f := homNS[1];
<<[ 3, 2, 2 ]> ---> <[ 1, 0, 0 ]>>

gap> p := ProjectiveCover(S[1]);
<<[ 1, 4, 3 ]> ---> <[ 1, 0, 0 ]>>

gap> LiftingMorphismFromProjective(f,p);
<<[ 1, 4, 3 ]> ---> <[ 3, 2, 2 ]>>
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
```

8.1.25 MinimalLeftAddMApproximation

▷ `MinimalLeftAddMApproximation(C, M)` (attribute)

▷ `MinimalLeftApproximation(C, M)` (attribute)

Arguments: `C`, `M` - two modules.

Returns: the minimal left add$M$-approximation of the module `C`. Note: The order of the arguments is opposite of the order for minimal right approximations.

8.1.26 MinimalRightApproximation

▷ `MinimalRightApproximation(M, C)` (attribute)
▷ `MinimalRightAddMApproximation(M, C)` (attribute)

Arguments: `M`, `C` - two modules.

Returns: the minimal right add$M$-approximation of the module `C`. Note: The order of the arguments is opposite of the order for minimal left approximations.

8.1.27 MorphismOnKernel

▷ `MorphismOnKernel(f, g, alpha, beta)` (operation)
▷ `MorphismOnImage(f, g, alpha, beta)` (operation)
▷ `MorphismOnCoKernel(f, g, alpha, beta)` (operation)

Arguments: `f`, `g`, `alpha`, `beta` - four homomorphisms of modules.

Returns: the morphism induced on the kernels, the images or the cokernels of the morphisms `f` and `g`, respectively, whenever $f\colon A \to B$, $\beta\colon B \to B'$, $\alpha\colon A \to A'$ and $g\colon A' \to B'$ forms a commutative diagram.

It is checked if `f`, `g`, `alpha`, `beta` forms a commutative diagram, that is, if $f\beta - \alpha g = 0$.

Example
```
gap> hom := HomOverAlgebra(N,N);
[ <<[ 3, 2, 2 ]> ---> <[ 3, 2, 2 ]>>
    , <<[ 3, 2, 2 ]> ---> <[ 3, 2, 2 ]>>
    , <<[ 3, 2, 2 ]> ---> <[ 3, 2, 2 ]>>
    , <<[ 3, 2, 2 ]> ---> <[ 3, 2, 2 ]>>
    , <<[ 3, 2, 2 ]> ---> <[ 3, 2, 2 ]>>
     ]
gap> g := MorphismOnKernel(hom[1],hom[2],hom[1],hom[2]);
<<[ 2, 2, 2 ]> ---> <[ 2, 2, 2 ]>>

gap> IsomorphicModules(Source(g),Range(g));
true
gap> p := ProjectiveCover(N);
<<[ 3, 12, 9 ]> ---> <[ 3, 2, 2 ]>>

gap> N1 := Kernel(p);
<[ 0, 10, 7 ]>
gap> pullback := PullBack(p,hom[1]);
[ <<[ 3, 12, 9 ]> ---> <[ 3, 2, 2 ]>>
    , <<[ 3, 12, 9 ]> ---> <[ 3, 12, 9 ]>>
     ]
gap> Kernel(pullback[1]);
<[ 0, 10, 7 ]>
gap> IsomorphicModules(N1,Kernel(pullback[1]));
true
```

```
gap> t := LiftingMorphismFromProjective(p,p*hom[1]);
<<[ 3, 12, 9 ]> ---> <[ 3, 12, 9 ]>>

gap> s := MorphismOnKernel(p,p,t,hom[1]);
<<[ 0, 10, 7 ]> ---> <[ 0, 10, 7 ]>>

gap> Source(s) = N1;
true
gap> q := KernelInclusion(p);
<<[ 0, 10, 7 ]> ---> <[ 3, 12, 9 ]>>

gap> pushout := PushOut(q,s);
[ <<[ 0, 10, 7 ]> ---> <[ 3, 12, 9 ]>>
    , <<[ 3, 12, 9 ]> ---> <[ 3, 12, 9 ]>>
     ]
gap> U := CoKernel(pushout[1]);
<[ 3, 2, 2 ]>
gap> IsomorphicModules(U,N);
true
```

8.1.28 NthSyzygy

▷ `NthSyzygy(M, n)` (operation)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`), *n* – a positive integer.

Returns: the *n*-th syzygy of *M* if it has projective dimension at least *n*, and if the projective dimension is less than *n*, then it returns the last non-zero projective syzygy.

This functions computes the *n*-th syzygy of the module *M* by successively computing first, second, third, ... syzygy of *M* the operation `1stSyzygy` (8.1.1) and at each stage checking if the is a projective module.

8.1.29 NthSyzygyNC

▷ `NthSyzygyNC(M, n)` (operation)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`), *n* – a positive integer.

Returns: the *n*-th syzygy of the module *M*, unless the projective dimension of *M* is less or equal to `n-1`, in which case it returns the zero module. It does not check if the `n`-th syzygy is projective or not.

8.1.30 NumberOfComplementsOfAlmostCompleteTiltingModule

▷ `NumberOfComplementsOfAlmostCompleteTiltingModule(M)` (operation)
▷ `NumberOfComplementsOfAlmostCompleteCotiltingModule(M)` (operation)

Arguments: *M* – a PathAlgebraMatModule.

Returns: the number complements of an almost complete tilting/cotilting module *M*, assuming that *M* is an almost complete tilting module.

8.1.31 ProjDimension

▷ `ProjDimension(M)` (attribute)

Arguments: `M` - a PathAlgebraMatModule.
Returns: the projective dimension of the module `M`, if it has been computed.

8.1.32 ProjDimensionOfModule

▷ `ProjDimensionOfModule(M)` (operation)

Arguments: `M, n` - a PathAlgebraMatModule, a positive integer.
Returns: Returns the projective dimension of the module `M` if it is less or equal to `n`. Otherwise it returns false.

8.1.33 ProjectiveCover

▷ `ProjectiveCover(M)` (attribute)

Arguments: `M` - a module.
Returns: the projective cover of `M`, that is, returns the map $P(M) \to M$.
If the module `M` is zero, then the zero map to `M` is returned.

8.1.34 ProjectiveResolutionOfPathAlgebraModule

▷ `ProjectiveResolutionOfPathAlgebraModule(M, n)` (operation)

Arguments: `M` - a path algebra module (`PathAlgebraMatModule`), `n` - a positive integer.
Returns: in terms of attributes `RProjectives`, `ProjectivesFList` and `Maps` a projective resolution of `M` out to stage `n`, where `RProjectives` are the projectives in the resolution lifted up to projectives over the path algebra, `ProjectivesFList` are the generators of the projective modules given in `RProjectives` in terms of elements in the first projective in the resolution and `Maps` contains the information about the maps in the resolution.

The algorithm for computing this projective resolution is based on the paper [GSZ01]. In addition, the algebra over which the modules are defined is available via the attribute `ParentAlgebra`.

8.1.35 PullBack

▷ `PullBack(f, g)` (operation)

Arguments: `f`, `g` - two homomorphisms with a common range.
Returns: the pullback of the maps `f` and `g`.
It is checked if `f` and `g` have the same range. Given the input $f\colon A \to B$ (horizontal map) and $g\colon C \to B$ (vertical map), the pullback E is returned as the two homomorphisms $[f', g']$, where $f'\colon E \to C$ (horizontal map) and $g'\colon E \to A$ (vertical map).

8.1.36 PushOut

▷ `PushOut(f, g)` (operation)

Arguments: `f`, g - two homomorphisms between modules with a common source.
Returns: the pushout of the maps `f` and g.

It is checked if `f` and g have the same source. Given the input $f: A \to B$ (horizontal map) and $g: A \to C$ (vertical map), the pushout E is returned as the two homomorphisms $[f', g']$, where $f': C \to E$ (horizontal map) and $g': B \to E$ (vertical map).

Example
```
gap> S := SimpleModules(A);
[ <[ 1, 0, 0 ]>, <[ 0, 1, 0 ]>, <[ 0, 0, 1 ]> ]
gap> Ext := ExtOverAlgebra(S[2],S[2]);
[ <<[ 0, 1, 2 ]> ---> <[ 0, 2, 2 ]>>
    , [ <<[ 0, 1, 2 ]> ---> <[ 0, 1, 0 ]>>
         ], function( map ) ... end ]
gap> Length(Ext[2]);
1
gap> # i.e. Ext^1(S[2],S[2]) is 1-dimensional
gap> pushout := PushOut(Ext[2][1],Ext[1]);
[ <<[ 0, 2, 2 ]> ---> <[ 0, 2, 0 ]>>
    , <<[ 0, 1, 0 ]> ---> <[ 0, 2, 0 ]>>
     ]
gap> f := CoKernelProjection(pushout[1]);
<<[ 0, 2, 0 ]> ---> <[ 0, 0, 0 ]>>

gap> U := Range(pushout[1]);
<[ 0, 2, 0 ]>
```

8.1.37 RightFacMApproximation

▷ `RightFacMApproximation(M, C)` (operation)
▷ `MinimalRightFacMApproximation(M, C)` (operation)

Arguments: `M`, `C` – two path algebra modules (`PathAlgebraMatModule`).
Returns: a minimal right Fac$M$-approximation of the module C.

8.1.38 RightMutationOfTiltingModuleComplement

▷ `RightMutationOfTiltingModuleComplement(M, N)` (operation)
▷ `RightMutationOfCotiltingModuleComplement(M, N)` (operation)

Arguments: `M`, `N` – two path algebra modules (`PathAlgebraMatModule`).
Returns: a right mutation of the complement `N` of the almost complete tilting/cotilting module `M`, if such a complement exists. Otherwise it returns false.

8.1.39 RightSubMApproximation

▷ `RightSubMApproximation(M, C)` (operation)
▷ `MinimalRightSubMApproximation(M, C)` (operation)

Arguments: `M`, `C` – two path algebra modules (`PathAlgebraMatModule`).

Returns: a right Sub$M$-approximation of the module C, where the first version returns a not necessarily minimal right Sub$M$-approximation and the second returns a minimal approximation.

8.1.40 N_RigidModule

▷ `N_RigidModule(M, n)` (operation)

Arguments: `M`, `n` - a PathAlgebraMatModule, an integer.

Returns: true if `M` is a `n`-rigid module. Otherwise it returns false.

8.1.41 TiltingModule

▷ `TiltingModule(M, n)` (operation)

Arguments: `M`, `n` - a PathAlgebraMatModule and a positive integer.

Returns: false if `M` is not a tilting module of projective dimension at most `n`. Otherwise, it returns the projective dimension of `M` and the coresolution of all indecomposable projective modules in add$M$.

Chapter 9

Auslander-Reiten theory

This chapter describes the functions implemented for almost split sequences and Auslander-Reiten theory in QPA.

9.1 Almost split sequences and AR-quivers

9.1.1 AlmostSplitSequence

▷ `AlmostSplitSequence(M)` (attribute)

Arguments: *M* - an indecomposable non-projective module.

Returns: the almost split sequence ending in the module *M* if it is indecomposable and not projective. It returns the almost split sequence in terms of two maps, a left minimal almost split map and a right minimal almost split map.

The range of the right minimal almost split map is not necessarily equal to the module *M* one started with, but isomorphic. The function assumes that the module *M* is indecomposable.

9.1.2 PredecessorOfModule

▷ `PredecessorOfModule(M, n)` (operation)

Arguments: *M* - an indecomposable non-projective module and *n* - a positive integer.

Returns: the predecessors of the module *M* in the AR-quiver of the algebra *M* is given over of distance less or equal to *n*.

It returns two lists, the first is the indecomposable modules in the different layers and the second is the valuations for the arrows in the AR-quiver. The different entries in the first list are the modules at distance zero, one, two, three, and so on, until layer *n*. The `m`-th entry in the second list is the valuations of the irreducible morphism from indecomposable module number `i` in layer `m+1` to indecomposable module number `j` in layer `m` for the values of `i` and `j` there is an irreducible morphism. Whenever `false` occur in the output, it means that this valuation has not been computed. The function assumes that the module *M* is indecomposable and that the quotient of the path algebra is given over a finite field.

Example
```
gap> A := KroneckerAlgebra(GF(4),2);
<GF(2^2)[<quiver with 2 vertices and 2 arrows>]>
```

```
gap> S := SimpleModules(A)[1];
<[ 1, 0 ]>
gap> ass := AlmostSplitSequence(S);
[ <<[ 3, 2 ]> ---> <[ 4, 2 ]>>
    , <<[ 4, 2 ]> ---> <[ 1, 0 ]>>
     ]
gap> DecomposeModule(Range(ass[1]));
[ <[ 2, 1 ]>, <[ 2, 1 ]> ]
gap> PredecessorsOfModule(S,5);
[ [ [ <[ 1, 0 ]> ], [ <[ 2, 1 ]> ], [ <[ 3, 2 ]> ], [ <[ 4, 3 ]> ],
      [ <[ 5, 4 ]> ], [ <[ 6, 5 ]> ] ],
  [ [ [ 1, 1, [ 2, false ] ] ], [ [ 1, 1, [ 2, 2 ] ] ],
      [ [ 1, 1, [ 2, 2 ] ] ], [ [ 1, 1, [ 2, 2 ] ] ],
      [ [ 1, 1, [ false, 2 ] ] ] ] ]
gap> A:=NakayamaAlgebra([5,4,3,2,1],GF(4));
<GF(2^2)[<quiver with 5 vertices and 4 arrows>]>
gap> S := SimpleModules(A)[1];
<[ 1, 0, 0, 0, 0 ]>
gap> PredecessorsOfModule(S,5);
[ [ [ <[ 1, 0, 0, 0, 0 ]> ], [ <[ 1, 1, 0, 0, 0 ]> ],
      [ <[ 0, 1, 0, 0, 0 ]>, <[ 1, 1, 1, 0, 0 ]> ],
      [ <[ 0, 1, 1, 0, 0 ]>, <[ 1, 1, 1, 1, 0 ]> ],
      [ <[ 0, 0, 1, 0, 0 ]>, <[ 0, 1, 1, 1, 0 ]>, <[ 1, 1, 1, 1, 1 ]>
         ], [ <[ 0, 0, 1, 1, 0 ]>, <[ 0, 1, 1, 1, 1 ]> ] ],
  [ [ [ 1, 1, [ 1, false ] ] ],
      [ [ 1, 1, [ 1, 1 ] ], [ 2, 1, [ 1, false ] ] ],
      [ [ 1, 1, [ 1, 1 ] ], [ 1, 2, [ 1, 1 ] ],
          [ 2, 2, [ 1, false ] ] ],
      [ [ 1, 1, [ 1, 1 ] ], [ 2, 1, [ 1, 1 ] ], [ 2, 2, [ 1, 1 ] ],
          [ 3, 2, [ 1, false ] ] ],
      [ [ 1, 1, [ false, 1 ] ], [ 1, 2, [ false, 1 ] ],
          [ 2, 2, [ false, 1 ] ], [ 2, 3, [ false, 1 ] ] ] ] ]
```

Chapter 10

Chain complexes

10.1 Introduction

If $\mathscr{A}$ is an abelian category, then a chain complex of objects of $\mathscr{A}$ is a sequence

$$\cdots \longrightarrow C_{i+1} \xrightarrow{d_{i+1}} C_i \xrightarrow{d_i} C_{i-1} \xrightarrow{d_{i-1}} \cdots$$

where C_i is an object of $\mathscr{A}$ for all i, and d_i is a morphism of $\mathscr{A}$ for all i such that the composition of two consecutive maps of the complex is zero. The maps are called the differentials of the complex. A complex is called *bounded above* (resp. below) if there is a bound b such that $C_i = 0$ for all $i > b$ (resp. $i < b$). A complex is *bounded* if it is both bounded below and bounded above.

The challenge when representing chain complexes in software is to handle their infinite nature. If a complex is not bounded, or not known to be bounded, how can we represent it in an immutable way? Our solution is to use a category called `InfList` (for "infinite list") to store the differentials of the complex. The properties of the `IsInfList` category is described in 10.2. An `IsComplex` object consists of one `IsInfList` for the differentials, and it also has an `IsCat` object as an attribute. The `IsCat` category is a representation of an abelian category, see 10.3.

To work with bounded complexes one does not need to know much about the `IsInfList` category. A bounded complex can be created by simply giving a list of the differentials and the degree of the first differential as input (see `FiniteComplex` (10.4.5)), and to create a stalk complex the stalk object and its degree suffice as input (see `StalkComplex` (10.4.6)). In both cases an `IsCat` object is also needed.

Example
```
gap> C := FiniteComplex(cat, 1, [g,f]);
0 -> 2:(1,0) -> 1:(2,2) -> 0:(1,1) -> 0
gap> Ms := StalkComplex(cat, M, 3);
0 -> 3:(2,2) -> 0
```

10.2 Infinite lists

In this section we give documentation for the `IsInfList` category. We start by giving a representation of $\pm\infty$. Then we quickly describe the `IsInfList` category, before we turn to the underlying structure of the infinite lists – the half infinite lists (`IsHalfInfList`). Most of the functionality of the infinite lists come from this category. Finally, we give the constructors for infinite lists, and some methods for manipulating such objects.

10.2.1 IsInfiniteNumber

▷ `IsInfiniteNumber` (Category)

A category for infinite numbers.

10.2.2 PositiveInfinity

▷ `PositiveInfinity` (Var)

A global variable representing the number ∞. It is greater than any integer, but it can not be compared to numbers which are not integers. It belongs to the `IsInfiniteNumber` category.

10.2.3 NegativeInfinity

▷ `NegativeInfinity` (Var)

A global variable representing the number $-\infty$. It is smaller than any integer, but it can not be compared to numbers which are not integers. It belongs to the `IsInfiniteNumber` category.

10.2.4 IsInfList

▷ `IsInfList` (Category)

An infinite list is an immutable representation of a list with possibly infinite range of indeces. It consists of three parts: The "middle part" is finite and covers some range $[a,b]$ of indices, the "positive part" covers the range $[b+1,\infty)$ of indices, and the "negative part" convers the range $(-\infty,a-1]$ of indices. Note that none of the three parts are mandatory: The middle part may be an empty list, and the positive part may be set to `fail` to achieve index range ending at $b<\infty$. Similary, if the index range has lower bound $a<\infty$, put the negative part to be `fail`.

Each of the two infinite parts are described in one of the following ways: (1) A finite list which is repeated indefinitely; (2) A function which takes an index in the list as argument and returns the corresponding list item; (3) A function which takes an item from the list as argument and returns the next item.

The two infinite parts are represented as "half infinite lists", see 10.2.5. An infinite list can be constructed in the following ways:

- From two half infinite lists and a middle part, `MakeInfListFromHalfInfLists` (10.2.22).
- Directly, by giving the same input as when constructing the above, `MakeInfList` (10.2.23).
- If all values of the infinite list are the image of the index under a function f, one can use `FunctionInfList` (10.2.24).
- If all values of the infinite list are the same, one can use `ConstantInfList` (10.2.25).
- If the infinite list has a finite range, one can use `FiniteInfList` (10.2.26).

In addition, new infinite lists can be constructed from others by shift, splice, concatenation, extracting parts or applying a function to the elements.

10.2.5 IsHalfInfList

▷ `IsHalfInfList` (Category)

A half infinite list is a representation of a list with indeces in the range $[a,\infty)$ or $(-\infty,b]$. An infinite list is typically made from two half infinite lists, and half infinite lists can be extracted from an infinite list. Hence, the half infinite list stores much of the information about an infinite list. One main difference between an infinite list and a half infinite list is that the half infinite list does not have any finite part, as the "middle" part of an infinite list.

10.2.6 \^

▷ `\^(list, pos)` (operation)

Arguments: *list* – either an infinite list or a half infinite list, *pos* – a valid index for *list*.
Returns: The value at position *pos* of *list*.

10.2.7 MakeHalfInfList

▷ `MakeHalfInfList(start, direction, typeWithArgs, callback)` (function)

Arguments: *start* – an integer, *direction* – either 1 or -1, *typeWithArgs* – a list which may have different formats, *callback* – a function.

Returns: A newly created half infinite list with index range from *start* to ∞, or from $-\infty$ to *start*.

If the range should be $[\mathtt{start},\infty)$ then the value of *direction* is 1. if the range should be $(-\infty,\mathtt{start}]$, then the value of *direction* is -1. The argument *typeWithArgs* can take one of the following forms:

- `[ "repeat", repeatList ]`
- `[ "next", nextFunction, initialValue ]`
- `[ "pos", posFunction ]`
- `[ "pos", posFunction, storeValues ]`

`repeatList` is a list of values that should be repeated in the half infinite list. `nextFunction` returns the value at position *i*, given the value at the previous position as argument. Here `initialValue` is the value at position `start`. Similarly, `posFunction` returns the value at any position *i*, and it may or may not store the values between the previous computed indeces and the newly computed index. The default value of `storeValues` is `true` for `"next"` and `"pos"`, and `false` for `"repeat"`. The argument `callback` is a function that is called whenever a new value of the list is computed. It takes three arguments: The current position, the direction and the type (that is, `typeWithArgs[1]`). If no callback function is needed, use `false`.

All the information given to create the list is stored, and can be retrieved later by the operations listed in 10.2.8–10.2.18.

Example
```
gap> # make a HalfInfList from 0 to inf which repeats the list [ 2, 4, 6 ]
gap> list1 := MakeHalfInfList( 0, 1, [ "repeat", [ 2, 4, 6 ] ], false );
```

```
<object>
gap> list1^0;
2
gap> list1^5;
6
gap> # make a HalfInfList from 0 to inf with x^2 in position x
gap> f := function(x) return x^2; end;;
gap> list2 := MakeHalfInfList( 0, 1, [ "pos", f, false ], false );
<object>
gap> list2^0;
0
gap> list2^10;
100
gap> # make a HalfInfList from 0 to -inf where each new value adds 3
gap> # to the previous and the value in position 0 is 10
gap> g := function(x) return x+3; end;;
gap> list3 := MakeHalfInfList( 0, -1, [ "next", g, 7 ], false );
<object>
gap> list3^0;
10
gap> list3^-10;
40
```

10.2.8 StartPosition

▷ `StartPosition(list)` (operation)

list – a half infinite list.
Returns: The start position of *list*.

10.2.9 Direction

▷ `Direction(list)` (operation)

list – a half infinite list.
Returns: The direction of *list* (either 1 or -1).

10.2.10 InfListType

▷ `InfListType(list)` (operation)

list – a half infinite list.
Returns: The type of *list* (either `"pos"`, `"repeat"` or `"next"`).

10.2.11 RepeatingList

▷ `RepeatingList(list)` (operation)

list – a half infinite list.
Returns: The repeating list of *list* if *list* is of type `"repeat"`, and `fail` otherwise.

10.2.12 ElementFunction

▷ `ElementFunction(list)` (operation)

`list` – a half infinite list.
Returns: The element function of `list` if `list` is of type `"next"` or `"pos"`, and `fail` otherwise.

10.2.13 IsStoringValues

▷ `IsStoringValues(list)` (operation)

`list` – a half infinite list.
Returns: `true` if all elements of the list are stored, `false` otherwise.

10.2.14 NewValueCallback

▷ `NewValueCallback(list)` (operation)

`list` – a half infinite list.
Returns: The callback function of the list.

10.2.15 IsRepeating

▷ `IsRepeating(list)` (operation)

`list` – a half infinite list.
Returns: `true` if the type of the list is `"repeat"`.

10.2.16 InitialValue

▷ `InitialValue(list)` (operation)

`list` – a half infinite list.
Returns: If the list is of type `"next"` then the initial value is returned, otherwise it fails.

10.2.17 LowestKnownPosition

▷ `LowestKnownPosition(list)` (operation)

`list` – a half infinite list.
Returns: The lowest index i such that the value at position i is known without computation (that is, it is either stored, or the list has type `"repeat"`).

10.2.18 HighestKnownValue

▷ `HighestKnownValue(list)` (operation)

list – a half infinite list.

Returns: The highest index i such that the value at position i is known without computation (that is, it is either stored, or the list has type `"repeat"`).

```
Example
gap> # we reuse the IsHalfInfLists from the previous example
gap> HighestKnownPosition(list1);
+inf
gap> HighestKnownPosition(list2);
"none"
gap> HighestKnownPosition(list3);
0
```

10.2.19 Shift

▷ `Shift(list, shift)` (operation)

Arguments: *list* – a half infinite list, *shift* – an integer.

Returns: A new half infinite list which is *list* with all values shifted *shift* positions to the right if *shift* is positive, and to the left if *shift* is negative.

10.2.20 Cut

▷ `Cut(list, pos)` (operation)

Arguments: *list* – a half infinite list, *pos* – an integer within the range of *list*.

Returns: A new half infinite list which is *list* with some part cut off.

If the direction of *list* is positive, then the new list has range from `cut` to ∞. If the direction of *list* is negative, then the new list has range from $-\infty$ to `cut`. The values at position i of the new half infinite list is the same as the value at position i of *list*.

10.2.21 HalfInfList

▷ `HalfInfList(list, func)` (operation)

Arguments: *list* – a half infinite list, *func* – a function which takes an element of the list as argument.

Returns: A half infinite list with the same range as *list*, where the value at position i is the image of the value at position i of *list* under *func*.

10.2.22 MakeInfListFromHalfInfLists

▷ `MakeInfListFromHalfInfLists(basePosition, middle, positive, negative)` (function)

Arguments: *basePosition* – an integer, *middle* – a list, *positive* – a half infinite list, *negative* – a half infinite list.

Returns: An infinite list with *middle* as is middle part, *positive* as its positive part and *negative* as its negative part.

The starting position of *positive* must be `basePosition + Length( middle )`, and the starting position of *negative* must be `basePosition - 1`. The returned list has `middle[1]` in position *basePosition*, `middle[2]` in position *basePosition + 1* and so on. Note that one probably wants the *positive* half infinite list to have direction 1, and the *negative* half infinite list to have direction -1.

Example
```
gap> # we want to construct an infinite list with 0 in position
gap> # 0 to 5, and x^2 in position x where x goes from 6 to inf,
gap> # and alternatingly 1 and -1 in position -1 to -inf.
gap> #
gap> basePosition := 0;;
gap> middle := [0,0,0,0,0,0];;
gap> f := function(x) return x^2; end;;
gap> positive := MakeHalfInfList( 6, 1, [ "pos", f, false ], false );
<object>
gap> altList := [ 1, -1 ];;
gap> negative := MakeHalfInfList( -1, -1, [ "repeat", altList ], false );
<object>
gap> inflist := MakeInfListFromHalfInfLists( basePosition, middle,
>                                           positive, negative );
<object>
gap> inflist^0; inflist^5; inflist^6; inflist^-1; inflist^-4;
0
0
36
1
-1
```

10.2.23 MakeInfList

▷ `MakeInfList(`*basePosition, middle, positive, negative, callback*`)` (function)

Argments: *basePosition* – an integer, *middle* – a list, *positive* – a list describing the positive part, *negative* – a list describing the negative part.

Returns: An infinite list with *middle* as is middle part, *positive* as its positive part and *negative* as its negative part.

The major difference between this construction and the previous is that here the half infinite lists that will make the positive and negative parts are not entered directly as arguments. Instead, one enters “description lists”, which are of the same format as the argument *typeWithArgs* of `MakeHalfInfList` (10.2.7).

Example
```
gap> # we construct the same infinite list as in the previous example
gap> basePosition := 0;;
gap> middle := [0,0,0,0,0,0];;
gap> f := function(x) return x^2; end;;
gap> altList := [ 1, -1 ];;
gap> inflist2 := MakeInfList( 0, middle, [ "pos", f, false ], [ "repeat",
>                             altList ], false );
<object>
gap> inflist2^0; inflist2^5; inflist2^6; inflist2^-1; inflist2^-4;
```

```
0
0
36
1
-1
```

10.2.24 FunctionInfList

▷ `FunctionInfList(func)` (function)

Arguments: `func` – a function that takes an integer as argument.
Returns: An infinite list where the value at position i is the function `func` applied to i.

10.2.25 ConstantInfList

▷ `ConstantInfList(value)` (function)

Arguments: `value` – an object.
Returns: An infinite list which has the object `value` in every position.

10.2.26 FiniteInfList

▷ `FiniteInfList(basePosition, list)` (function)

Arguments: `basePosition` – an integer, `list` – a list of length n.
Returns: An infinite list with $\texttt{list}[1],\ldots,\texttt{list}[n]$ in positions $\texttt{basePosition},\ldots,$ $\texttt{basePosition}+\texttt{n}$.
The range of this list is $[\texttt{basePosition},\texttt{basePosition}+\texttt{n}]$.

10.2.27 MiddleStart

▷ `MiddleStart(list)` (operation)

Arguments: `list` – an infinite list.
Returns: The starting position of the "middle" part of `list`.

10.2.28 MiddleEnd

▷ `MiddleEnd(list)` (operation)

Arguments: `list` – an infinite list.
Returns: The ending position of the middle part of `list`.

10.2.29 MiddlePart

▷ `MiddlePart(list)` (operation)

Arguments: `list` – an infinite list.
Returns: The middle part (as a list) of `list`.

10.2.30 PositivePart

▷ `PositivePart(list)` (operation)

Arguments: `list` – an infinite list.
Returns: The positive part (as a half infinite list) of `list`.

10.2.31 NegativePart

▷ `NegativePart(list)` (operation)

Arguments: `list` – an infinite list.
Returns: The negative part (as a halft infinite list) of `list`.

10.2.32 HighestKnownPosition

▷ `HighestKnownPosition(list)` (operation)

Arguments: `list` – an infinite list.
Returns: The highest index i such that the value at position i is known withouth computation.

10.2.33 LowestKnownPosition

▷ `LowestKnownPosition(list)` (operation)

Arguments: `list` – an infinite list.
Returns: The lowest index i such that the value at position i is known withouth computation.

10.2.34 UpperBound

▷ `UpperBound(list)` (operation)

Arguments: `list` – an infinite list.
Returns: The highest index in the range of the list.

10.2.35 LowerBound

▷ `LowerBound(list)` (operation)

Arguments: `list` – an infinite list.
Returns: The lowest index in the range of the list.

10.2.36 FinitePartAsList

▷ `FinitePartAsList(`*`list, startPos, endPos`*`)` (operation)

Arguments: *`list`* – an infinite list, *`startPos`* – an integer, *`endPos`* – an integer.
Returns: A list containing the values of *`list`* in positions `endPos`, ..., `startPos`.
Note that both integers in the input must be within the index range of *`list`*.

10.2.37 PositivePartFrom

▷ `PositivePartFrom(`*`list, pos`*`)` (operation)

Arguments: *`list`* – an infinite list, *`pos`* – an integer.
Returns: An infinite list (*not* a half infinite list) with index range from `pos` to `UpperBound(list)`.
The value at position *i* of the new infinite list is the same as the value at position *i* of *`list`*.

10.2.38 NegativePartFrom

▷ `NegativePartFrom(`*`list, pos`*`)` (operation)

Arguments: *`list`* – an infinite list, *`pos`* – an integer.
Returns: An infinite list (*not* a half infinite list) with index range from `LowerBound(list)` to `pos`.
The value at position *i* of the new infinite list is the same as the value at position *i* of *`list`*.

10.2.39 Shift

▷ `Shift(`*`list, shift`*`)` (operation)

Arguments: *`list`* – an infinite list, *`shift`* – an integer.
Returns: A new infinite list which is *`list`* with all values shifted *`shift`* positions to the right if *`shift`* is positive, and to the left if *`shift`* is negative.

10.2.40 Splice

▷ `Splice(`*`positiveList, negativeList, joinPosition`*`)` (operation)

Arguments: *`positiveList`* – an infinite list, *`negativeList`* – an infinite list, *`joinPosition`* – an integer.
Returns: A new infinite list which is identical to *`positiveList`* for indeces greater than *`joinPosition`* and identical to *`negativeList`* for indeces smaller than or equal to *`joinPosition`*.

10.2.41 InfConcatenation

▷ `InfConcatenation(`*`arg`*`)` (function)

Arguments: *arg* – a number of infinite lists.

Returns: A new infinite list.

If the length of *arg* is greater than or equal to 2, then the new infinite list consists of the following parts: It has the positive part of `arg[1]`, and the middle part is the concatenation of the middle parts of all lists in *arg*, such that `MiddleEnd` of the new list is the same as `MiddleEnd( arg[1] )`. The negative part of the new list is the negative part of `arg[Length(arg)]`, although shiftet so that it starts in the correct position.

Example
```
gap> # we do an InfConcatenation of three lists.
gap> f := function(x) return x; end;;
gap> g := function(x) return x+1; end;;
gap> h := function(x) return x^2; end;;
gap> InfList1 := MakeInfList( 0, [ 10 ], [ "pos", f, false ],
>                              [ "repeat", [ 10, 15 ] ], false );
<object>
gap> InfList2 := MakeInfList( 0, [ 20 ], [ "pos", g, false ],
>                              [ "repeat", [ 20, 25 ] ], false );
<object>
gap> InfList3 := MakeInfList( 0, [ 30 ], [ "pos", h, false ],
>                              [ "repeat", [ 30, 35 ] ], false );
<object>
gap> concList := InfConcatenation( InfList1, InfList2, InfList3 );
<object>
gap> MiddlePart(concList);
[ 30, 20, 10 ]
```

The newly created `concList` looks as follows around the middle part:

position	$\cdots$	3	2	1	0	-1	-2	-3	-4	-5	$\cdots$
value	$\cdots$	3	2	1	10	20	30	30	35	30	$\cdots$

10.2.42 InfList

▷ `InfList(`*list, func*`)` (operation)

Arguments: *list* – an infinite list, *func* – a function which takes an element of the list as argument.

Returns: An infinite list with the same range as *list*, where the value at position *i* is the image of the value at position *i* of *list* under *func*.

10.2.43 IntegersList

▷ `IntegersList` (global variable)

An infinite list with range $(-\infty,\infty)$ where the value at position *i* is the number *i* (that is, a representation of the integers).

10.3 Representation of categories

A chain complex consists of objects and morphisms from some category. In QPA, this category will usually be the category of right modules over some quotient of a path algebra.

10.3.1 IsCat

▷ `IsCat` (Category)

The category for categories. A category is a record, storing a number of properties that is specified within each category. Two categories can be compared using `=`. Currently, the only implemented category is the one of right modules over a (quotient of a) path algebra.

10.3.2 CatOfRightAlgebraModules

▷ `CatOfRightAlgebraModules(A)` (operation)

Arguments: *A* – a (quotient of a) path algebra.

Returns: The category mod *A*.

mod *A* has several properties, which can be accessed using the `.` mark. Some of the properties store functions. All properties are demonstrated in the following example.

- `zeroObj` – returns the zero module of mod *A*.
- `isZeroObj` – returns true if the given module is zero.
- `zeroMap` – returns the ZeroMapping function.
- `isZeroMapping` – returns the IsZero test.
- `composeMaps` – returns the composition of the two given maps.
- `ker` – returns the Kernel function.
- `im` – returns the Image function.
- `isExact` – returns true if two consecutive maps are exact.

Example

```
gap> alg;
<algebra-with-one over Rationals, with 7 generators>
gap> # L, M, and N are alg-modules
gap> # f: L --> M and g: M --> N are non-zero morphisms
gap> cat := CatOfRightAlgebraModules(alg);
<cat: right modules over algebra>
gap> cat.zeroObj;
<right-module over <algebra-with-one over Rationals, with 7 generators>>
gap> cat.isZeroObj(M);
false
gap> cat.zeroMap(M,N);
<mapping: <3-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*v4], [(1)*a], [(1)*b], [(1)*c] ])> ->
  <1-dimensional right-module over AlgebraWithOne( Rationals,
  [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*v4], [(1)*a], [(1)*b], [(1)*c] ] )> >
gap> cat.composeMaps(g,f);
<mapping: <1-dimensional right-module over AlgebraWithOne( Rationals,
  [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*v4], [(1)*a], [(1)*b], [(1)*c]]
  -> <1-dimensional right-module over AlgebraWithOne( Rationals,
  [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*v4], [(1)*a], [(1)*b], [(1)*c] ] )> >
gap> cat.ker(g);
<2-dimensional right-module over <algebra-with-one over Rationals,
  with 7 generators>>
gap> cat.isExact(g,f);
false
```

10.4 Making a complex

The most general constructor for complexes is the function `Complex` (10.4.3). In addition to this, there are constructors for common special cases:

- `ZeroComplex` (10.4.4)
- `StalkComplex` (10.4.6)
- `FiniteComplex` (10.4.5)
- `ShortExactSequence` (10.4.7)

10.4.1 IsComplex

▷ `IsComplex` (Category)

The category for chain complexes.

10.4.2 IsZeroComplex

▷ `IsZeroComplex` (Category)

Category for zero complexes, subcategory of `IsComplex` (10.4.1).

10.4.3 Complex

▷ `Complex(`*`cat, baseDegree, middle, positive, negative`*`)` (function)

Returns: A newly created chain complex

The first argument, *`cat`* is an `IsCat` (10.3.1) object describing the category to create a chain complex over.

The rest of the arguments describe the differentials of the complex. These are divided into three parts: one finite ("middle") and two infinite ("positive" and "negative"). The positive part contains all differentials in degrees higher than those in the middle part, and the negative part contains all differentials in degrees lower than those in the middle part. (The middle part may be placed anywhere, so the positive part can – despite its name – contain some differentials of negative degree. Conversely, the negative part can contain some differentials of positive degree.)

The argument *middle* is a list containing the differentials for the middle part. The argument *baseDegree* gives the degree of the first differential in this list. The second differential is placed in degree *baseDegree* $+1$, and so on. Thus, the middle part consists of the degrees

$$\mathit{baseDegree}, \quad \mathit{baseDegree}+1, \quad \ldots \quad \mathit{baseDegree}+\mathrm{Length}(\mathit{middle}).$$

Each of the arguments *positive* and *negative* can be one of the following:

- The string `"zero"`, meaning that the part contains only zero objects and zero morphisms.
- A list of the form `[ "repeat", L ]`, where `L` is a list of morphisms. The part will contain the differentials in `L` repeated infinitely many times. The convention for the order of elements in `L` is that `L[1]` is the differential which is closest to the middle part, and `L[Length(L)]` is farthest away from the middle part.
- A list of the form `[ "pos", f ]` or `[ "pos", f, store ]`, where `f` is a function of two arguments, and `store` (if included) is a boolean. The function `f` is used to compute the differentials in this part. The function `f` is not called immediately by the `Complex` constructor, but will be called later as the differentials in this part are needed. The function call `f(C,i)` (where `C` is the complex and `i` an integer) should produce the differential in degree `i`. The function may use `C` to look up other differentials in the complex, as long as this does not cause an infinite loop. If `store` is `true` (or not specified), each computed differential is stored, and they are computed in order from the one closest to the middle part, regardless of which order they are requested in.
- A list of the form `[ "next", f, init ]`, where `f` is a function of one argument, and `init` is a morphism. The function `f` is used to compute the differentials in this part. For the first differential in the part (that is, the one closest to the middle part), `f` is called with `init` as argument. For the next differential, `f` is called with the first differential as argument, and so on. Thus, the differentials are

  $$f(\mathrm{init}), \quad f^2(\mathrm{init}), \quad f^3(\mathrm{init}), \quad \ldots$$

  Each differential is stored when it has been computed.

10.4.4 ZeroComplex

▷ `ZeroComplex(cat)` (function)

Returns: A newly created zero complex

This function creates a zero complex (a complex consisting of only zero objects and zero morphisms) over the category described by the `IsCat` (10.3.1) object *cat*.

10.4.5 FiniteComplex

▷ `FiniteComplex(cat, baseDegree, differentials)` (function)

Returns: A newly created complex

This function creates a complex where all but finitely many objects are the zero object.

The argument *cat* is an `IsCat` (10.3.1) object describing the category to create a chain complex over.

The argument *differentials* is a list of morphisms. The argument *baseDegree* gives the degree for the first differential in this list. The subsequent differentials are placed in degrees $\mathit{baseDegree}+1$, and so on.

This means that the *differentials* argument specifies the differentials in degrees

$$\mathit{baseDegree},\quad \mathit{baseDegree}+1,\quad \ldots\quad \mathit{baseDegree}+\mathrm{Length}(\mathit{differentials});$$

and thus implicitly the objects in degrees

$$\mathit{baseDegree}-1,\quad \mathit{baseDegree},\quad \ldots\quad \mathit{baseDegree}+\mathrm{Length}(\mathit{differentials}).$$

All other objects in the complex are zero.

Example
```
gap> # L, M and N are modules over the same algebra A
gap> # cat is the category mod A
gap> # f: L --> M and g: M --> N maps
gap> C := FiniteComplex(cat, 1, [g,f]);
0 -> 2:(1,0) -> 1:(2,2) -> 0:(1,1) -> 0
```

10.4.6 StalkComplex

▷ StalkComplex(*cat, obj, degree*) (function)

Arguments: *cat* – a category, *obj* – an object in *cat*, *degree* – the degree *obj* should be placed in.

Returns: a newly created complex.

The new complex is a stalk complex with *obj* in position *degree*, and zero elsewhere.

Example
```
gap> Ms := StalkComplex(cat, M, 3);
0 -> 3:(2,2) -> 0
```

10.4.7 ShortExactSequence

▷ ShortExactSequence(*cat, f, g*) (function)

Arguments: *cat* – a category, *f* and *g* – maps in *cat*, where $f\colon A \to B$ and $g\colon B \to C$.

Returns: a newly created complex.

If the sequence $0 \to A \to B \to C \to 0$ is exact, this complex (with B in degree 0) is returned.

Example
```
gap> ses := ShortExactSequence(cat, f, g);
0 -> 1:(0,0,1,0) -> 0:(0,1,1,1) -> -1:(0,1,0,1) -> 0
```

10.5 Information about a complex

10.5.1 CatOfComplex

▷ CatOfComplex(*C*) (attribute)

Returns: The category the objects of the complex *C* live in.

10.5.2 ObjectOfComplex

▷ `ObjectOfComplex(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: The object at position `i` in the complex.

10.5.3 DifferentialOfComplex

▷ `DifferentialOfComplex(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: The map in `C` between objects at positions i and $i-1$.

10.5.4 DifferentialsOfComplex

▷ `DifferentialsOfComplex(C)` (attribute)

Arguments: `C` – a complex
Returns: The differentials of the complex, stored as an `IsInfList` object.

10.5.5 CyclesOfComplex

▷ `CyclesOfComplex(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: The i-cycle of the complex, that is the subobject $Ker(d_i)$ of `ObjectOfComplex(C,i)`.

10.5.6 BoundariesOfComplex

▷ `BoundariesOfComplex(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: The i-boundary of the complex, that is the subobject $Im(d_{i+1})$ of `ObjectOfComplex(C,i)`.

10.5.7 HomologyOfComplex

▷ `HomologyOfComplex(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: The ith homology of the complex, that is, $Ker(d_i)/Im(d_{i+1})$.
Note: this operation is currently not available. When working in the category of right kQ/I-modules, it is possible to "cheat" and use the following procedure to compute the homology of a complex:

Example
```
gap> C;
0 -> 4:(0,1) -> 3:(1,0) -> 2:(2,2) -> 1:(1,1) -> 0:(2,2) -> 0
gap> # Want to compute the homology in degree 2
gap> f := DifferentialOfComplex(C,3);
<mapping: <1-dimensional right-module over AlgebraWithOne( Rationals,
  [ [(1)*v1], [(1)*v2], [(1)*a], [(1)*b] ] )> ->
  < 4-dimensional right-module over AlgebraWithOne( Rationals,
  [ [(1)*v1], [(1)*v2], [(1)*a], [(1)*b] ] )> >
gap> g := KernelInclusion(DifferentialOfComplex(C,2));
  <mapping: <2-dimensional right-module over AlgebraWithOne( Rationals,
  [ [(1)*v1], [(1)*v2], [(1)*a], [(1)*b] ] )> ->
  < 4-dimensional right-module over AlgebraWithOne( Rationals,
  [ [(1)*v1], [(1)*v2], [(1)*a], [(1)*b] ] )> >
gap> # We know that Im f is included in Ker g, so can find the
gap> # lifting morphism h from C_3 to Ker g.
gap> h := LiftingInclusionMorphisms(g,f);
  <mapping: <1-dimensional right-module over AlgebraWithOne( Rationals,
  [ [(1)*v1], [(1)*v2], [(1)*a], [(1)*b] ] )> ->
  < 2-dimensional right-module over AlgebraWithOne( Rationals,
  [ [(1)*v1], [(1)*v2], [(1)*a], [(1)*b] ] )> >
gap> # The cokernel of h is Ker g / Im f
gap> Homology := CoKernel(h);
<1-dimensional right-module over <algebra-with-one over Rationals, with
  4 generators>>
```

10.5.8 IsFiniteComplex

▷ `IsFiniteComplex(C)` (operation)

Arguments: C – a complex.
Returns: true if C is a finite complex, false otherwise.

10.5.9 UpperBound

▷ `UpperBound(C)` (operation)

Arguments: C – a complex.
Returns: If it exists: The smallest integer i such that the object at position i is non-zero, but for all $j > i$ the object at position j is zero.

If C is not a finite complex, the operation will return fail or infinity, depending on how C was defined.

10.5.10 LowerBound

▷ `LowerBound(C)` (operation)

Arguments: C – a complex.
Returns: If it exists: The greatest integer i such that the object at position i is non-zero, but for all $j < i$ the object at position j is zero.

If C is not a finite complex, the operation will return fail or negative infinity, depending on how C was defined.

10.5.11 LengthOfComplex

▷ `LengthOfComplex(C)` (operation)

Arguments: *C* – a complex.
Returns: the length of the complex.

The length is defined as follows: If *C* is a zero complex, the length is zero. If *C* is a finite complex, the lenght is the upper bound – the lower bound + 1. If *C* is an inifinite complex, the lenght is infinity.

10.5.12 HighestKnownDegree

▷ `HighestKnownDegree(C)` (operation)

Arguments: *C* – a complex.
Returns: The greatest integer *i* such that the object at position *i* is known (or computed).
For a finite complex, this will be infinity.

10.5.13 LowestKnownDegree

▷ `LowestKnownDegree(C)` (operation)

Arguments: *C* – a complex.
Returns: The smallest integer *i* such that the object at position *i* is known (or computed).
For a finite complex, this will be negative infinity.

Example
```
gap> C;
0 -> 4:(0,1) -> 3:(1,0) -> 2:(2,2) -> 1:(1,1) -> 0:(2,2) -> 0
gap> IsFiniteComplex(C);
true
gap> UpperBound(C);
4
gap> LowerBound(C);
0
gap> LengthOfComplex(C);
5
gap> HighestKnownDegree(C);
+inf
gap> LowestKnownDegree(C);
-inf
```

10.5.14 IsExactSequence

▷ `IsExactSequence(C)` (property)

Arguments: *C* – a complex.
Returns: true if *C* is exact at every position.
If the complex is not finite and not repeating, the function fails.

10.5.15 IsExactInDegree

▷ `IsExactInDegree(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: true if `C` is exact at position `i`.

10.5.16 IsShortExactSequence

▷ `IsShortExactSequence(C)` (property)

Arguments: `C` – a complex.
Returns: true if `C` is exact and of the form

$$\ldots \to 0 \to A \to B \to C \to 0 \to \ldots$$

This could be positioned in any degree (as opposed to the construction of a short exact sequence, where B will be put in degree zero).

Example
```
gap> C;
0 -> 4:(0,1) -> 3:(1,0) -> 2:(2,2) -> 1:(1,1) -> 0:(2,2) -> 0
gap> IsExactSequence(C);
false
gap> IsExactInDegree(C,1);
true
gap> IsExactInDegree(C,2);
false
```

10.5.17 ForEveryDegree

▷ `ForEveryDegree(C, func)` (operation)

Arguments: `C` – a complex, `func` – a function operating on two consecutive maps.
Returns: true if `func` returns true for any two consecutive differentials, fail if this can not be decided, false otherwise.

10.6 Transforming and combining complexes

10.6.1 Shift

▷ `Shift(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: A new complex, which is a shift of `C`.
If `i` > 0, the complex is shifted to the left. If `i` < 0, the complex is shifted to the right. Note that shifting might change the differentials: In the shifted complex, d_{new} is defined to be $(-1)^i d_{old}$.

Example
```
gap> C;
0 -> 4:(0,1) -> 3:(1,0) -> 2:(2,2) -> 1:(1,1) -> 0:(2,2) -> 0
```

```
gap> Shift(C,1);
0 -> 3:(0,1) -> 2:(1,0) -> 1:(2,2) -> 0:(1,1) -> -1:(2,2) -> 0
gap> D := Shift(C,-1);
0 -> 5:(0,1) -> 4:(1,0) -> 3:(2,2) -> 2:(1,1) -> 1:(2,2) -> 0
gap> dc := DifferentialOfComplex(C,3)!.maps;
[ [ [ 1, 0 ] ], [ [ 0, 0 ] ] ]
gap> dd := DifferentialOfComplex(D,4)!.maps;
[ [ [ -1, 0 ] ], [ [ 0, 0 ] ] ]
gap> MatricesOfPathAlgebraMatModuleHomomorphism(dc);
[ [ [ 1, 0 ] ], [ [ 0, 0 ] ] ]
gap> MatricesOfPathAlgebraMatModuleHomomorphism(dd);
[ [ [ -1, 0 ] ], [ [ 0, 0 ] ] ]
```

10.6.2 ShiftUnsigned

▷ `ShiftUnsigned(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: A new complex, which is a shift of `C`.
Does the same as `Shift`, except it does not change the sign of the differential. Although this is a non-mathematical definition of shift, it is still useful for technical purposes, when manipulating and creating complexes.

10.6.3 YonedaProduct

▷ `YonedaProduct(C, D)` (operation)

Arguments: `C`, `D` – complexes.
Returns: The Yoneda product of the two complexes, which is a complex.
To compute the Yoneda product, `C` and `D` must be such that the object in degree `LowerBound(C)` equals the object in degree `UpperBound(D)`, that is

$$\ldots \to C_{i+1} \to C_i \to A \to 0 \to \ldots$$

$$\ldots \to 0 \to A \to D_j \to D_{j-1} \to \ldots$$

The product is of this form:

$$\ldots \to C_{i+1} \to C_i \to D_j \to D_{j-1} \to \ldots$$

where the map $C_i \to D_j$ is the composition of the maps $C_i \to A$ and $A \to D_j$. Also, the object D_j is in degree j.

Example
```
gap> C2;
0 -> 4:(0,1) -> 3:(1,0) -> 2:(2,2) -> 1:(1,1) -> 0:(0,0) -> 0
gap> C3;
0 -> -1:(1,1) -> -2:(2,2) -> -3:(1,1) -> 0
gap> YonedaProduct(C2,C3);
0 -> 1:(0,1) -> 0:(1,0) -> -1:(2,2) -> -2:(2,2) -> -3:(1,1) -> 0
```

10.6.4 BrutalTruncationBelow

▷ `BrutalTruncationBelow(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: A newly created complex.
Replace all objects with degree $j < i$ with zero. The differentials affected will also become zero.

10.6.5 BrutalTruncationAbove

▷ `BrutalTruncationAbove(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: A newly created complex.
Replace all objects with degree $j > i$ with zero. The differentials affected will also become zero.

10.6.6 BrutalTruncation

▷ `BrutalTruncation(C, i, j)` (operation)

Arguments: `C` – a complex, `i, j` – integers.
Returns: A newly created complex.
Brutally truncates in both ends. The integer arguments must be ordered such that `i` > `j`.

10.6.7 SyzygyTruncation

▷ `SyzygyTruncation(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: A newly created complex.
Replace the object in degree i with the kernel of d_i, and d_{i+1} with the natural inclusion. All objects in degree $j > i+1$ are replaced with zero.

10.6.8 CosyzygyTruncation

▷ `CosyzygyTruncation(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: A newly created complex.
Replace the object in degree $i-2$ with the cokernel of d_i, and d_{i-1} with the natural projection. All objects in degree $j < i-2$ are replaced with zero.

10.6.9 SyzygyCosyzygyTruncation

▷ `SyzygyCosyzygyTruncation(C, i, j)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: A newly created complex.
Performs both the above truncations. The integer arguments must be ordered such that `i` > `j`.

10.7 Chain maps

An `IsChainMap` (10.7.1) object represents a chain map between two complexes over the same category.

10.7.1 IsChainMap

▷ `IsChainMap` (Category)

The category for chain maps.

10.7.2 ChainMap

▷ `ChainMap(`*`source, range, basePosition, middle, positive, negative`*`)` (function)

Arguments: *`source`*, *`range`* – complexes, *`basePosition`* – an integer, *`middle`* – a list of morphisms, *`positive`* – a list or the string `"zero"`, *`negative`* – a list or the string `"zero"`.

Returns: A newly created chain map

The arguments *`source`* and *`range`* are the complexes which the new chain map should map between.

The rest of the arguments describe the individual morphisms which constitute the chain map, in a similar way to the last four arguments to the `Complex` (10.4.3) function.

The morphisms of the chain map are divided into three parts: one finite ("middle") and two infinite ("positive" and "negative"). The positive part contains all morphisms in degrees higher than those in the middle part, and the negative part contains all morphisms in degrees lower than those in the middle part. (The middle part may be placed anywhere, so the positive part can – despite its name – contain some morphisms of negative degree. Conversely, the negative part can contain some morphisms of positive degree.)

The argument *`middle`* is a list containing the morphisms for the middle part. The argument *`baseDegree`* gives the degree of the first morphism in this list. The second morphism is placed in degree *`baseDegree`* $+1$, and so on. Thus, the middle part consists of the degrees

$$\mathit{baseDegree}, \quad \mathit{baseDegree}+1, \quad \ldots \quad \mathit{baseDegree}+\mathrm{Length}(\mathit{middle})-1.$$

Each of the arguments *`positive`* and *`negative`* can be one of the following:

- The string `"zero"`, meaning that the part contains only zero morphisms.
- A list of the form `[ "repeat", L ]`, where `L` is a list of morphisms. The part will contain the morphisms in `L` repeated infinitely many times. The convention for the order of elements in `L` is that `L[1]` is the morphism which is closest to the middle part, and `L[Length(L)]` is farthest away from the middle part. (Using this only makes sense if the objects of both the source and range complex repeat in a compatible way.)
- A list of the form `[ "pos", f ]` or `[ "pos", f, store ]`, where `f` is a function of two arguments, and `store` (if included) is a boolean. The function `f` is used to compute the morphisms in this part. The function `f` is not called immediately by the `ChainMap` constructor, but will be called later as the morphisms in this part are needed. The function call `f(M,i)` (where `M` is the chain map and `i` an integer) should produce the morphism in degree `i`. The function

may use `M` to look up other morphisms in the chain map (and to access the source and range complexes), as long as this does not cause an infinite loop. If `store` is `true` (or not specified), each computed morphism is stored, and they are computed in order from the one closest to the middle part, regardless of which order they are requested in.

- A list of the form `[ "next", f, init ]`, where `f` is a function of one argument, and `init` is a morphism. The function `f` is used to compute the morphisms in this part. For the first morphism in the part (that is, the one closest to the middle part), `f` is called with `init` as argument. For the next morphism, `f` is called with the first morphism as argument, and so on. Thus, the morphisms are

  $$f(\text{init}), \quad f^2(\text{init}), \quad f^3(\text{init}), \quad \ldots$$

  Each morphism is stored when it has been computed.

10.7.3 ZeroChainMap

▷ `ZeroChainMap(`*`source, range`*`)` (function)

Returns: A newly created zero chain map

This function creates a zero chain map (a chain map in which every morphism is zero) from the complex *`source`* to the complex *`range`*.

10.7.4 FiniteChainMap

▷ `FiniteChainMap(`*`source, range, baseDegree, morphisms`*`)` (function)

Returns: A newly created chain map

This function creates a complex where all but finitely many morphisms are zero.

The arguments *`source`* and *`range`* are the complexes which the new chain map should map between.

The argument *`morphisms`* is a list of morphisms. The argument *`baseDegree`* gives the degree for the first morphism in this list. The subsequent morphisms are placed in degrees *`baseDegree`* + 1, and so on.

This means that the *`morphisms`* argument specifies the morphisms in degrees

$$\mathtt{baseDegree}, \quad \mathtt{baseDegree}+1, \quad \ldots \quad \mathtt{baseDegree}+\text{Length}(\mathtt{morphisms})-1.$$

All other morphisms in the chain map are zero.

10.7.5 ComplexAndChainMaps

▷ `ComplexAndChainMaps(`*`sourceComplexes, rangeComplexes, basePosition, middle, positive, negative`*`)` (function)

Arguments: *`sourceComplexes`* – a list of complexes, *`rangeComplexes`* – a list of complexes, *`basePosition`* – an integer, *`middle`* – a list of morphisms, *`positive`* – a list or the string `"zero"`, *`negative`* – a list or the string `"zero"`.

Returns: A list consisting of a newly created complex, and one or more newly created chain maps.

This is a combined constructor to make one complex and a set of chain maps at the same time. All the chain maps will have the new complex as either source or range.

The argument *sourceComplexes* is a list of the complexes to be sources of the chain maps which have the new complex as range. The argument *rangeComplexes* is a list of the complexes to be ranges of the chain maps which have the new complex as source.

Let S and R stand for the lengths of the lists *sourceComplexes* and *rangeComplexes*, respectively. Then the number of new chain maps which are created is $S+R$.

The last four arguments describe the individual differentials of the new complex, as well as the inidividual morphisms which constitute each of the new chain maps. These arguments are treated in a similar way to the last four arguments to the `Complex` (10.4.3) and `ChainMap` (10.7.2) constructors. In those constructors, the last four arguments describe, for each degree, how to get the differential or morphism for that degree. Here, we for each degree need both a differential for the complex, and one morphism for each chain map. So for each degree i, we will have a list

$$L_i = [d_i, m_i^1, \ldots, m_i^S, n_i^1, \ldots, n_i^R],$$

where d_i is the differential for the new complex in degree i, m_i^j is the morphism in degree i of the chain map from `sourceComplexes[j]` to the new complex, and n_i^j is the morphism in degree i of the chain map from the new complex to `rangeComplexes[j]`.

The degrees of the new complex and chain maps are divided into three parts: one finite ("middle") and two infinite ("positive" and "negative"). The positive part contains all degrees higher than those in the middle part, and the negative part contains all degrees lower than those in the middle part.

The argument *middle* is a list containing the lists L_i for the middle part. The argument *baseDegree* gives the degree of the first morphism in this list. The second morphism is placed in degree *baseDegree* $+1$, and so on. Thus, the middle part consists of the degrees

$$\mathit{baseDegree}, \quad \mathit{baseDegree}+1, \quad \ldots \quad \mathit{baseDegree}+\mathrm{Length}(\mathit{middle})-1.$$

Each of the arguments *positive* and *negative* can be one of the following:

- The string `"zero"`, meaning that the part contains only zero morphisms.

- A list of the form `[ "repeat", L ]`, where `L` is a list of morphisms. The part will contain the morphisms in `L` repeated infinitely many times. The convention for the order of elements in `L` is that `L[1]` is the morphism which is closest to the middle part, and `L[Length(L)]` is farthest away from the middle part. (Using this only makes sense if the objects of both the source and range complex repeat in a compatible way.)

- A list of the form `[ "pos", f ]` or `[ "pos", f, store ]`, where `f` is a function of two arguments, and `store` (if included) is a boolean. The function `f` is used to compute the morphisms in this part. The function `f` is not called immediately by the `ChainMap` constructor, but will be called later as the morphisms in this part are needed. The function call `f(M,i)` (where `M` is the chain map and `i` an integer) should produce the morphism in degree `i`. The function may use `M` to look up other morphisms in the chain map (and to access the source and range complexes), as long as this does not cause an infinite loop. If `store` is `true` (or not specified), each computed morphism is stored, and they are computed in order from the one closest to the middle part, regardless of which order they are requested in.

- A list of the form `[ "next", f, init ]`, where `f` is a function of one argument, and `init` is a morphism. The function `f` is used to compute the morphisms in this part. For the first morphism in the part (that is, the one closest to the middle part), `f` is called with `init` as argument. For the next morphism, f is called with the first morphism as argument, and so on. Thus, the morphisms are

$$f(\text{init}), \quad f^2(\text{init}), \quad f^3(\text{init}), \quad \dots$$

Each morphism is stored when it has been computed.

The return value of the ComplexAndChainMaps constructor is a list

$$[C, M_1, \dots, M_S, N_1, \dots, N_R],$$

where C is the new complex, $M_1, \dots, M_S$ are the new chain maps with C as range, and $N_1, \dots, N_R$ are the new chain maps with C as source.

10.7.6 MorphismOfChainMap

▷ MorphismOfChainMap(*M, i*) (operation)

Arguments: *M* – a chain map, *i* – an integer.
Returns: The morphism at position *i* in the chain map.

10.7.7 MorphismsOfChainMap

▷ MorphismsOfChainMap(*M*) (attribute)

Arguments: *M* – a chain map.
Returns: The morphisms of the chain map, stored as an IsInfList (10.2.4) object.

10.7.8 ComparisonLifting

▷ ComparisonLifting(*f, PC, EC*) (operation)

Arguments: *f* – a map between modules M and N, *PC* – a chain complex, *EC* – a chain complex.
Returns: The map *f* lifted to a chain map from *PC* to *EC*.

The complex *PC* must have M in some fixed degree i, it should be bounded with only zero objects in degrees smaller than i, and it should have only projective objects in degrees greater than i (or projective objects in degrees $[i+1, j]$ and zero in degrees greater than j). The complex *EC* should also have zero in degrees smaller than i, it should have N in degree i and it should be exact for all degrees. The returned chain map has *f* in degree i.

10.7.9 ComparisonLiftingToProjectiveResolution

▷ ComparisonLiftingToProjectiveResolution(*f*) (operation)

Arguments: *f* – a map between modules M and N.
Returns: The map *f* lifted to a chain map from the projective resolution of M to the projective resolution of N.

The returned chain map has *f* in degree -1 (the projective resolution of a module includes the module itself in degree -1).

10.7.10 MappingCone

▷ `MappingCone(f)` (operation)

Arguments: `f` – a chain map between chain complexes A and B.

Returns: A list with the mapping cone of `f` and the inclusion of B into the cone, and the projection of the cone onto $A[-1]$.

Example

```
gap> # Constructs a quiver and a quotient of a path algebra
gap> Q := Quiver( 4, [ [1,2,"a"], [2,3,"b"], [3,4,"c"] ] );;
gap> PA := PathAlgebra( Rationals, Q );;
gap> rels := [ PA.a*PA.b ];;
gap> gb := GBNPGroebnerBasis( rels, PA );;
gap> I := Ideal( PA, gb );;
gap> grb := GroebnerBasis( I, gb );;
gap> alg := PA/I;;
gap>
gap> # Two modules M and N, and a map between them
gap> M := RightModuleOverPathAlgebra( alg, [0,1,1,0], [["b", [[1]] ]] );;
gap> N := RightModuleOverPathAlgebra( alg, [0,1,0,0], [] );;
gap> f := RightModuleHomOverAlgebra(M, N, [ [[0]],[[1]],[[0]],[[0]] ]);;
gap>
gap> # Lifts f to a map between the projective resolutions of M and N
gap> lf := ComparisonLiftingToProjectiveResolution(f);
<chain map>
gap>
gap> # Computes the mapping cone of the chain map
gap> H := MappingCone(lf);
[ --- -> -1:(0,1,0,0) -> ---, <chain map>, <chain map> ]
gap> cone := H[1];
--- -> -1:(0,1,0,0) -> ---
gap> ObjectOfComplex(Source(lf),0);
<[ 0, 1, 1, 1 ]>
gap> ObjectOfComplex(Range(lf),1);
<[ 0, 0, 1, 1 ]>
gap> ObjectOfComplex(cone,1);
<[ 0, 1, 2, 2 ]>
gap> Source(H[2]) = Range(lf);
true
```

Chapter 11

Projective resolutions and the bounded derived category

What is implemented so far for working with the bounded derived category $\mathscr{D}^b(\text{mod}A)$. We use the isomorphism $\mathscr{D}^b(\text{mod}A) \cong \mathscr{K}^{-,b}(\text{proj}A)$, and will hence need a way to describe complexes where all objectives are projective (or, dually, injective).

11.1 Projective and injective complexes

11.1.1 IsProjectiveComplex

▷ `IsProjectiveComplex(C)` (property)

Arguments: `C` – a complex.

Returns: true if `C` is either a finite complex of projectives or an infinite complex of projectives constructed as a projective resolution (`ProjectiveResolutionOfComplex` (11.2.1)), false otherwise.

A complex for which this property is true, will be printed in a different manner than ordinary complexes. Instead of writing the dimension vector of the objects in each degree, the indecomposable direct summands are listed (for instance `P1`, `P2` ... , where P_i is the indecomposable projective module corresponding to vertex i of the quiver). Note that if a complex is both projective and injective, it is printed as a projective complex.

11.1.2 IsInjectiveComplex

▷ `IsInjectiveComplex(C)` (property)

Arguments: `C` – a complex.

Returns: true if `C` is either a finite complex of injectives or an infinite complex of injectives constructed as $D\text{Hom}_A(-,A)$ of a projective complex (`ProjectiveToInjectiveComplex` (11.2.2)), false otherwise.

A complex for which this property is true, will be printed in a different manner than ordinary complexes. Instead of writing the dimension vector of the objects in each degree, the indecomposable direct summands are listed (for instance `I1`, `I2` ... , where I_i is the indecomposable injective module corresponding to vertex i of the quiver). Note that if a complex is both projective and injective, it is printed as a projective complex.

11.1.3 ProjectiveResolution

▷ `ProjectiveResolution(M)` (operation)

Arguments: `M` – a module.
Returns: The projective resolution of `M` with `M` in degree -1.

11.2 The bounded derived category

Let $\mathscr{D}^b(\mathrm{mod}A)$ denote the bounded derived category. If C is an element of $\mathscr{D}^b(\mathrm{mod}A)$, that is, a bounded complex of A-modules, there exists a projective resolution P of C which is a complex of projective A-modules quasi-isomorphic to C. Moreover, there exists such a P with the following properties:

- P is minimal (in the homotopy category).
- C is bounded, so $C_i = 0$ for $i < k$ for a lower bound k and $C_i = 0$ for $i > j$ for an upper bound j. Then $P_i = 0$ for $i < k$, and P is exact in degree i for $i > j$.

The function `ProjectiveResolutionOfComplex` computes such a projective resolution of any bounded complex. If A has finite global dimension, then $\mathscr{D}^b(\mathrm{mod}A)$ has AR-triangles, and there exists an algorithm for computing the AR-translation of a complex $C \in \mathscr{D}^b(\mathrm{mod}A)$:

- Compute a projective resolution P' of C.
- Shift P' one degree to the right.
- Compute $I = D\mathrm{Hom}_A(P', A)$ to get a complex of injectives.
- Compute a projective resolution P of I.

Then P is the AR-translation of C, sometimes written $\tau(C)$. The following documents the QPA functions for working with complexes in the derived category.

11.2.1 ProjectiveResolutionOfComplex

▷ `ProjectiveResolutionOfComplex(C)` (operation)

Arguments: `C` – a finite complex.
Returns: A projective complex P which is the projective resolution of C, as described in the introduction to this section.

If the algebra has infinite global dimension, the projective resolution of C could possibly be infinite.

11.2.2 ProjectiveToInjectiveComplex

▷ `ProjectiveToInjectiveComplex(P)` (operation)
▷ `ProjectiveToInjectiveFiniteComplex(P)` (operation)

Arguments: `P` – a bounded below projective complex.
Returns: An injective complex $I = D\mathrm{Hom}_A(P, A)$.

P and I will always have the same length. Especially, if P is unbounded above, then so is I. If P is a finite complex (that is; `LengthOfComplex(P)` is an integer) then the simpler method `ProjectiveToInjectiveFiniteComplex` is used.

11.2.3 TauOfComplex

▷ `TauOfComplex(C)` (operation)

Arguments: `C` – a finite complex over an algebra of finite global dimension.
Returns: A projective complex P which is the AR-translation of `C`.

This function only works when the algebra has finite global dimension. It will always assume that both the projective resolutions computed are finite.

11.2.4 Example

The following example illustrates the above mentioned functions and properties. Note that both `ProjectiveResolutionOfComplex` and `ProjectiveToInjectiveComplex` return complexes with a nonzero *positive* part, whereas `TauOfComplex` always returns a complex for which `IsFiniteComplex` returns true. Also note that after the complex `C` in the example is found to have the `IsInjectiveComplex` property, the printing of the complex changes.

The algebra in the example is kQ/I, where Q is the quiver $1 \longrightarrow 2 \longrightarrow 3$ and I is generated by the composition of the arrows. We construct C as the stalk complex with the injective I_1 in degree 0.

Example
```
gap> alg;
<Rationals[<quiver with 3 vertices and 2 arrows>]/
<two-sided ideal in <Rationals[<quiver with 3 vertices and 2 arrows>]>,
  (1 generators)>>
gap> cat := CatOfRightAlgebraModules(alg);
<cat: right modules over algebra>
gap> C := StalkComplex(cat, IndecInjectiveModules(alg)[1], 0);
0 -> 0:(1,0,0) -> 0
gap> ProjC := ProjectiveResolutionOfComplex(C);
--- -> 0: P1 -> 0
gap> InjC := ProjectiveToInjectiveComplex(ProjC);
--- -> 1: I2 -> 0: I1 -> 0
gap> TauC := TauOfComplex(C);
0 -> 1: P3 -> 0
gap> IsProjectiveComplex(C);
false
gap> IsInjectiveComplex(C);
true
gap> C;
0 -> 0: I1 -> 0
```

11.2.5 StarOfMapBetweenProjectives

▷ `StarOfMapBetweenProjectives(f, list_i, list_j)` (operation)
▷ `StarOfMapBetweenIndecProjectives(f, i, list_j)` (operation)
▷ `StarOfMapBetweenDecompProjectives(f, list_i, list_j)` (operation)

Arguments: `f` – a map between to projective modules $P = \bigoplus P_i$ and $Q = \bigoplus Q_j$, each of which were constructed as direct sums of indecomposable projective modules; `list_i` – describes the summands of P; `list_j` – describes the summands of Q. If $P = P_1 \oplus P_3 \oplus P_3$ (where P_i is the indecomposable projective representation in vertex i), then `list_i` is [1,3,3].

Returns: The map $f^* = \mathrm{Hom}_A(f,A) : \mathrm{Hom}_A(Q,A) \to \mathrm{Hom}_A(P,A)$ in A^{op} (where A is the original algebra).

The function `StarOfMapBetweenProjectives` is supposed to be called from within the `ProjectiveToInjectiveComplex` method, and might not do as expected when called from somewhere else.

The other similarly named functions are called from within the first.

Chapter 12

Combinatorial representation theory

12.1 Introduction

Here we introduce the implementation of the software package CREP initially designed for MAPLE.

12.2 Different unit forms

12.2.1 IsUnitForm

▷ `IsUnitForm` (Category)

The category for unit forms, which we identify with symmetric integral matrices with 2 along the diagonal.

12.2.2 BilinearFormOfUnitForm

▷ `BilinearFormOfUnitForm(B)` (attribute)

Arguments: `B` – a unit form.

Returns: the bilinear form associated to a unit form `B`.

The bilinear form associated to the unitform `B` given by a matrix `B` is defined for two vectors `x` and `y` as: $x * B * y^T$.

12.2.3 IsWeaklyNonnegativeUnitForm

▷ `IsWeaklyNonnegativeUnitForm(B)` (property)

Arguments: `B` – a unit form.

Returns: true is the unitform `B` is weakly non-negative, otherwise false.

The unit form `B` is weakly non-negative is $B(x,y) \geq 0$ for all $x \neq 0$ in $\mathbb{Z}^n$, where n is the dimension of the square matrix associated to `B`.

12.2.4 IsWeaklyPositiveUnitForm

▷ `IsWeaklyPositiveUnitForm(B)` (property)

Arguments: `B` – a unit form.

Returns: true is the unitform `B` is weakly positive, otherwise false.

The unit form `B` is weakly positive if $B(x,y) > 0$ for all $x \neq 0$ in $\mathbb{Z}^n$, where n is the dimension of the square matrix associated to `B`.

12.2.5 PositiveRootsOfUnitForm

▷ `PositiveRootsOfUnitForm(B)` (attribute)

Arguments: `B` – a unit form.

Returns: the positive roots of a unit form, if the unit form is weakly positive. If they have not been computed, an error message will be returned saying "no method found!".

This attribute will be attached to `B` when `IsWeaklyPositiveUnitForm` is applied to `B` and it is weakly positive.

12.2.6 QuadraticFormOfUnitForm

▷ `QuadraticFormOfUnitForm(B)` (attribute)

Arguments: `B` – a unit form.

Returns: the quadratic form associated to a unit form `B`.

The quadratic form associated to the unitform `B` given by a matrix `B` is defined for a vector `x` as: $\frac{1}{2}x * B * x^T$.

12.2.7 SymmetricMatrixOfUnitForm

▷ `SymmetricMatrixOfUnitForm(B)` (attribute)

Arguments: `B` – a unit form.

Returns: the symmetric integral matrix which defines the unit form `B`.

12.2.8 TitsUnitFormOfAlgebra

▷ `TitsUnitFormOfAlgebra(A)` (operation)

Arguments: `A` – a finite dimensional (quotient of a) path algebra (by an admissible ideal).

Returns: the Tits unit form associated to the algebra `A`.

This function returns the Tits unitform associated to a finite dimensional quotient of a path algebra by an admissible ideal or path algebra, given that the underlying quiver has no loops or minimal relations that starts and ends in the same vertex. That is, then it returns a symmetric matrix B such that for $x = (x_1, ..., x_n)(1/2) * (x_1, ..., x_n)B(x_1, ..., x_n)^T = \sum_{i=1}^n x_i^2 - \sum_{i,j} \dim_k \mathrm{Ext}^1(S_i, S_j)x_i x_j + \sum_{i,j} \dim_k \mathrm{Ext}^2(S_i, S_j)x_i x_j$, where n is the number of vertices in Q.

12.2.9 EulerBilinearFormOfAlgebra

▷ `EulerBilinearFormOfAlgebra(A)` (operation)

Arguments: `A` – a finite dimensional (quotient of a) path algebra (by an admissible ideal).

Returns: the Euler (non-symmetric) bilinear form associated to the algebra `A`.

This function returns the Euler (non-symmetric) bilinear form associated to a finite dimensional (basic) quotient of a path algebra `A`. That is, it returns a bilinear form (function) defined by
$f(x,y) = x * \mathrm{CartanMatrix}(A)^{(-1)} * y$
It makes sense only in case `A` is of finite global dimension.

12.2.10 UnitForm

▷ `UnitForm(B)` (operation)

Arguments: `B` – an integral matrix.

Returns: the unit form in the category `IsUnitForm` (12.2.1) associated to the matrix `B`.

The function checks if `B` is a symmetric integral matrix with 2 along the diagonal, and returns an error message otherwise. In addition it sets the attributes, `BilinearFormOfUnitForm` (12.2.2), `QuadraticFormOfUnitForm` (12.2.6) and `SymmetricMatrixOfUnitForm` (12.2.7).

Chapter 13

Degeneration order for modules in finite type

13.1 Introduction

This is an implementation of several tools for computing degeneration order for modules over algebras of finite type. It can be treated as a "subpackage" of QPA and used separately since the functions do not use any of QPA routines so far.
This subpackage has a little bit different philosophy than QPA in general. Namely, the "starting point" is not an algebra A defined by a Gabriel quiver with relations but an Auslander-Reiten (A-R) quiver of the category mod A, defined by numerical data (see `ARQuiverNumerical` (13.3.1)). All the indecomposables (actually their isoclasses) have unique natural numbers established at the beginning, by invoking `ARQuiverNumerical` (13.3.1). This function should be used before all further computations. An arbitrary module M is identified by its multiplicity vector (the sequence of multiplicities of all the indecomposables appearing in a direct sum decomposition of M).
Here we always assume that A is an algebra of finite representation type. Note that in this case deg-order coincide with Hom-order, and this fact is used in the algorithms of this subpackage. The main goal of this subpackage is to give tools for testing a deg-order relation between two A-modules and determining (direct) deg-order predecessors and successors (see 13.2 for basic definitions from this theory). As a side effect one can also obtain the dimensions of Hom-spaces between arbitrary modules (and in particular the dimension vectors of indecomposable modules).

13.2 Basic definitions

Here we briefly recall the basic notions we use in all the functions from this chapter.

Let A be an algebra. We say that for two A-modules M and N of the same dimension vector d, M degenerates to N (N is a degeneration of M) iff N belongs to a Zariski closure of the orbit of M in a variety $mod_A(d)$ of A-modules of dimension vector d. If it is the case, we write $M <= N$. It is well known that

(1) The relation $<=$ is a partial order on the set of isomorphism classes of A-modules of dimension vector d.

(2) If A is an algebra of finite representation type, $<=$ coincides with so-called Hom-order

$<=_{Hom}$, defined as follows: $M <=_{Hom} N$ iff $[X,M] <= [X,N]$ for all indecomposable A-modules X, where by $[Y,Z]$ we denote always the dimension of a Hom-space between Y and Z.

Further, if $M < N$ (i.e. $M <= N$ and M is not isomorphic to N), we say that M is a deg-order predecessor of N (resp. N is a deg-order successor of M). Moreover, we say that M is a direct deg-order predecessor of N if $M < N$ and there is no M' such that $M < M' < N$ (similarly for successors).

13.3 Defining Auslander-Reiten quiver in finite type

13.3.1 ARQuiverNumerical

▷ `ARQuiverNumerical(`*`ind, proj, list`*`)` (function)
▷ `ARQuiverNumerical(`*`name`*`)` (function)
▷ `ARQuiverNumerical(`*`name, param1`*`)` (function)
▷ `ARQuiverNumerical(`*`name, param1, param2`*`)` (function)

Arguments: *`ind`* - number of indecomposable modules in our category;
`proj` - number of indecomposable projective modules in our category;
`list` - list of lists containing description of meshes in A-R quiver defined as follows:
`list`[i] = description of mesh ending in vertex (indec. mod.) number i having the shape [a1,...,an,t] where
a1,...,an = numbers of direct predecessors of i in A-R quiver;
t = number of tau(i), or 0 if tau i does not exist (iff i is projective).
In particular if i is projective *`list`*[i]=[a1,...,an,0] where a1,...,an are indec. summands of rad(i).

OR:
`list` second version - if the first element of *`list`* is a string "orbits" then the remaining elements should provide an alternative (shorter than above) description of A-R quiver as follows.
`list`[2] is a list of descriptions of orbits identified by chosen representatives. We assume that in case an orbit is non-periodic, then a projective module is its representative. Each element of list *`list`*[2] is a description of i-th orbit and has the shape:
`[l, [i1,t1], ... , [is,ts]]` where
l = length of orbit - 1
`[i1,t1], ... , [is,ts]` all the direct predecessors of a representative of this orbit, of the shape tau^{-t1}(i1), and i1 denotes the representative of orbit no. i1, and so on.
We assume first p elements of *`list`*[2] are the orbits of projectives.

REMARK: we ALWAYS assume that indecomposables with numbers 1..*`proj`* are projectives and the only projectives (further dimension vectors are interpreted according to this order of projectives!).

Alternative arguments:
`name` = string with the name of predefined A-R quiver;
`param1` = (optional) parameter for *`name`*;
`param2` = (optional) second parameter for *`name`*.

Call ARQuiverNumerical("what") to get a description of all the names and parameters for currently available predefined A-R quivers. **Returns:** an object from the category `IsARQuiverNumerical` (13.3.2).

This function "initializes" Auslander-Reiten quiver and performs all necessary preliminary computations concerning mainly determining the matrix of dimensions of all Hom-spaces between indecomposables.

Examples.
Below we define an A-R quiver of a path algebra of the Dynkin quiver D4 with subspace orientation of arrows.

Example
```
gap> a := ARQuiverNumerical(12, 4, [ [0],[1,0],[1,0],[1,0],[2,3,4,1],[5,2],[5,3],[5,4],[6,7,8,5]
<ARQuiverNumerical with 12 indecomposables and 4 projectives>
```

The same A-R quiver (with possibly little bit different enumeration of indecomposables) can be obtained by invoking:

Example
```
gap> b := ARQuiverNumerical(12, 4, ["orbits", [ [2], [2,[1,0]], [2,[1,0]], [2,[1,0]] ] ]);
<ARQuiverNumerical with 12 indecomposables and 4 projectives>
```

This A-R quiver can be also obtained by:

Example
```
gap> a := ARQuiverNumerical("D4 subspace");
<ARQuiverNumerical with 12 indecomposables and 4 projectives>
```

since this is one of the predefined A-R quivers.
Another example of predefined A-R quiver: for an algebra from Bongartz-Gabriel list of maximal finite type algebras with two simple modules. This is an algebra with number 5 on this list.

Example
```
gap> a := ARQuiverNumerical("BG", 5);
<ARQuiverNumerical with 72 indecomposables and 2 projectives>
```

13.3.2 IsARQuiverNumerical

▷ `IsARQuiverNumerical` (Category)

Objects from this category represent Auslander-Reiten (finite) quivers and additionally contain all data necessary for further computations (as components accessed as usual by !.name-of-component):
ARdesc = numerical description of AR quiver (as *`list`* in `ARQuiverNumerical` (13.3.1)),
DimHomMat = matrix [dim Hom (i,j)] (=> rows 1..p contain dim. vectors of all indecomposables),
Simples = list of numbers of simple modules.

13.3.3 NumberOfIndecomposables

▷ `NumberOfIndecomposables(`*`AR`*`)` (attribute)

Argument: *`AR`* - an object from the category `IsARQuiverNumerical` (13.3.2).
Returns: the number of indecomposable modules in *`AR`*.

13.3.4 NumberOfProjectives

▷ `NumberOfProjectives(AR)` (attribute)

Argument: *AR* - an object from the category `IsARQuiverNumerical` (13.3.2).
Returns: the number of indecomposable projective modules in *AR*.

13.4 Elementary operations

13.4.1 DimensionVector (DimVectFT)

▷ `DimensionVector(AR, M)` (operation)

Arguments: *AR* - an object from the category `IsARQuiverNumerical` (13.3.2);
M - a number of an indecomposable module in *AR* or a multiplicity vector (cf. 13.1). **Returns:** a dimension vector of a module *M* in the form of a list. The order of dimensions in this list corresponds to an order of projectives defined in *AR* (cf. `ARQuiverNumerical` (13.3.1)).

Example
```
gap> a := ARQuiverNumerical("D4 subspace");
<ARQuiverNumerical with 12 indecomposables and 4 projectives>
gap> DimensionVector(a, 7);
[ 1, 1, 0, 1 ]
gap> DimensionVector(a, [0,1,0,0,0,0,2,0,0,0,0,0]);
[ 3, 3, 0, 2 ]
```

13.4.2 DimHom

▷ `DimHom(AR, M, N)` (operation)

Arguments: *AR* - an object from the category `IsARQuiverNumerical` (13.3.2);
M - a number of indecomposable module in *AR* or a multiplicity vector;
N - a number of indecomposable module in *AR* or a multiplicity vector (cf. 13.1).
Returns: the dimension of the homomorphism space between modules *M* and *N*.

13.4.3 DimEnd

▷ `DimEnd(AR, M)` (operation)

Arguments: *AR* - an object from the category `IsARQuiverNumerical` (13.3.2);
M - a number of indecomposable module in *AR* or a multiplicity vector (cf. 13.1).
Returns: the dimension of the endomorphism algebra of a module *M*.

13.4.4 OrbitDim

▷ `OrbitDim(AR, M)` (operation)

Arguments: *AR* - an object from the category `IsARQuiverNumerical` (13.3.2);
M - a number of indecomposable module in *AR* or a multiplicity vector (cf. 13.1).

Returns: the dimension of the orbit of module *M* (in the variety of representations of quiver with relations).

OrbitDim(*M*) = d_1^2+...+d_p^2 - dim End(*M*), where (d_i)_i = DimensionVector(*M*).

13.4.5 OrbitCodim

▷ `OrbitCodim(AR, M, N)` (operation)

Arguments: *AR* - an object from the category `IsARQuiverNumerical` (13.3.2);
M - a number of indecomposable module in *AR* or a multiplicity vector;
N - a number of indecomposable module in *AR* or a multiplicity vector (cf. 13.1).

Returns: the codimension of orbits of modules *M* and *N* (= dim End(*N*) - dim End(*M*)). [explain more???]

NOTE: The function does not check if it makes sense, i.e. if *M* and *N* are in the same variety (= dimension vectors coincide)!

13.4.6 DegOrderLEQ

▷ `DegOrderLEQ(AR, M, N)` (operation)

Arguments: *AR* - an object from the category `IsARQuiverNumerical` (13.3.2);
M - a number of indecomposable module in *AR* or a multiplicity vector;
N - a number of indecomposable module in *AR* or a multiplicity vector (cf. 13.1).

Returns: true if *M*<=*N* in a degeneration order i.e. if *N* is a degeneration of *M* (see 13.2), and false otherwise.

NOTE: Function checks if it makes sense, i.e. if *M* and *N* are in the same variety (= dimension vectors coincide). If not, it returns false and additionally prints warning.

Example
```
gap> a := ARQuiverNumerical("R nilp");
<ARQuiverNumerical with 7 indecomposables and 2 projectives>
gap> DimensionVector(a, 2);  DimensionVector(a, 3);
[ 2, 1 ]
[ 2, 1 ]
gap> DegOrderLEQ(a, 2, 3);
true
gap> DegOrderLEQ(a, 3, 2);
false
```

13.4.7 DegOrderLEQNC

▷ `DegOrderLEQNC(AR, M, N)` (operation)

Arguments: *AR* - an object from the category `IsARQuiverNumerical` (13.3.2);
M - a number of indecomposable module in *AR* or a multiplicity vector;
N - a number of indecomposable module in *AR* or a multiplicity vector (cf. 13.1).

Returns: true if `M<=N` in a degeneration order i.e. if `N` is a degeneration of `M` (see 13.2), and false otherwise.

NOTE: Function does Not Check ("NC") if it makes sense, i.e. if `M` and `N` are in the same variety (= dimension vectors coincide). If not, the result doesn't make sense!
It is useful when one wants to speed up computations (does not need to check the dimension vectors).

13.4.8 PrintMultiplicityVector

▷ `PrintMultiplicityVector(M)` (operation)

`M` - a list = multiplicity vector (cf. 13.1).

This function prints the multiplicity vector `M` in a more "readable" way (especially useful if `M` is long and sparse). It prints a "sum" of non-zero multiplicities in the form "multiplicity * (number-of-indecomposable)".

13.4.9 PrintMultiplicityVectors

▷ `PrintMultiplicityVectors(list)` (operation)

`list` - a list of multiplicity vectors (cf. 13.1).

This function prints all the multiplicity vectors from the `list` in a more "readable" way, as `PrintMultiplicityVector` (13.4.8).

13.5 Operations returning families of modules

The functions from this section use quite advanced algorithms on (potentially) big amount of data, so their runtimes can be long for "big" A-R quivers!

13.5.1 ModulesOfDimVect

▷ `ModulesOfDimVect(AR, which)` (operation)

Arguments: `AR` - an object from the category `IsARQuiverNumerical` (13.3.2);
`which` - a number of an indecomposable module in `AR` or a dimension vector (see `DimensionVector` (13.4.1)). **Returns:** a list of all modules (= multiplicity vectors, see 13.1) with dimension vector equal to `which`.

13.5.2 DegOrderPredecessors

▷ `DegOrderPredecessors(AR, M)` (operation)

Arguments: `AR` - an object from the category `IsARQuiverNumerical` (13.3.2);
`M` - a number of indecomposable module in `AR` or a multiplicity vector (cf. 13.1).

Returns: a list of all modules (= multiplicity vectors) which are the predecessors of module `M` in a degeneration order (see 13.2).

Example
```
gap> a := ARQuiverNumerical("BG", 5);
<ARQuiverNumerical with 72 indecomposables and 2 projectives>
gap> preds := DegOrderPredecessors(a, 60);; Length(preds);
18
gap> DegOrderLEQ(a, preds[7], 60);
true
gap> dpreds := DegOrderDirectPredecessors(a, 60);; Length(dpreds);
5
gap> PrintMultiplicityVectors(dpreds);
1*(14) + 1*(64)
1*(10) + 1*(71)
1*(9) + 1*(67)
1*(5) + 1*(17) + 1*(72)
1*(1) + 1*(5) + 1*(20)
```

13.5.3 DegOrderDirectPredecessors

▷ `DegOrderDirectPredecessors(AR, M)` (operation)

Arguments: *AR* - an object from the category `IsARQuiverNumerical` (13.3.2);
M - a number of indecomposable module in *AR* or a multiplicity vector (cf. 13.1).

Returns: a list of all modules (= multiplicity vectors) which are the direct predecessors of module *M* in a degeneration order (see 13.2).

13.5.4 DegOrderPredecessorsWithDirect

▷ `DegOrderPredecessorsWithDirect(AR, M)` (operation)

Arguments: *AR* - an object from the category `IsARQuiverNumerical` (13.3.2);
M - a number of indecomposable module in *AR* or a multiplicity vector (cf. 13.1).

Returns: a pair (2-element list) [*p*, *dp*] where
p = the same as a result of `DegOrderPredecessors` (13.5.2);
dp = the same as a result of `DegOrderDirectPredecessors` (13.5.3);

The function generates predecessors only once, so the runtime is exactly the same as DegOrderDirectPredecessors.

13.5.5 DegOrderSuccessors

▷ `DegOrderSuccessors(AR, M)` (operation)

Arguments: *AR* - an object from the category `IsARQuiverNumerical` (13.3.2);
M - a number of indecomposable module in *AR* or a multiplicity vector (cf. 13.1).

Returns: a list of all modules (= multiplicity vectors) which are the successors of module *M* in a degeneration order (see 13.2).

13.5.6 DegOrderDirectSuccessors

▷ `DegOrderDirectSuccessors(AR, M)` (operation)

Arguments: *AR* - an object from the category `IsARQuiverNumerical` (13.3.2);
M - a number of indecomposable module in *AR* or a multiplicity vector (cf. 13.1).

Returns: a list of all modules (= multiplicity vectors) which are the direct successors of module *M* in a degeneration order (see 13.2).

13.5.7 DegOrderSuccessorsWithDirect

▷ `DegOrderSuccessorsWithDirect(AR, M)` (operation)

Arguments: *AR* - an object from the category `IsARQuiverNumerical` (13.3.2);
M - a number of indecomposable module in *AR* or a multiplicity vector (cf. 13.1).

Returns: a pair (2-element list) [*s*, *ds*] where
s = the same as a result of `DegOrderSuccessors` (13.5.5);
ds = the same as a result of `DegOrderDirectSuccessors` (13.5.6);

The function generates successors only once, so the runtime is exactly the same as DegOrderDirectSuccessors.

References

[Gre00] E. L. Green. Multiplicative bases, gröbner bases, and right gröbner bases. *J. Symbolic Comput.*, 29:601–623, 2000. 27

[GSZ01] E. L. Green, Ø. Solberg, and D. Zacharia. Minimal projective resolutions. *Trans. Amer. Math. Soc.*, 353:2915–2939, 2001. 96

Index