QPA

Quivers and Path Algebras

Version version 1.07

September 2010

The QPA-team
http://sourceforge.net/projects/quiverspathalg/
(See also http://www.math.ntnu.no/~oyvinso/QPA/**)**

We can add a comment here.

Address: Virginia Tech, Blacksburg, USA
NTNU, Trondheim, Norway

Abstract

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:

Gerard Brunick	Quivers, path algebras
Randall Cone	Code modernization and cleanup, GBNP interface (for Groebner bases), projective resolutions, user documentation
George Yuhasz	User documentation, matrix representations of path algebras

Colophon

This is the Colophon page.

Contents

Chapter 1

Introduction/Quick Start

This chapter is intended for those who would like to get started with QPA right away by playing with a few examples. An important feature to be aware of concerning using the software is how to display the results one is obtaining. Here there are three basic ways of doing this, through `View`, `Print` or `Display`. The command `View(M)` prints a short information about the object `M`, the command `Print(M)` prints a longer information about the object `M`, and finally The command `Display(M)` prints an even fuller information about the object `M`, in general. However, for some objects these may all coincide. First we present a simple example:

1.1 Example 1

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

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

We can request the list of vertices and the list of arrows for *q*:

Example
```
gap> VerticesOfQuiver(q);
[ u ]
gap> ArrowsOfQuiver(q);
[ a, b ]
```

Next we create the path algebra *pa* from *q* over the rational numbers:

Example
```
gap> pa := PathAlgebra(Rationals,q);
<algebra-with-one over Rationals, with 3 generators>
```

In this case it is interesting to note that we've created an algebra isomorphic to the free algebra on two generators. We now retrieve and label the generators and multiplicative identity for *pa*:

Example
```
gap> gens := GeneratorsOfAlgebra(pa);
[ (1)*u, (1)*a, (1)*b ]
gap> u := gens[1];
(1)*u
gap> a := gens[2];
```

```
(1)*a
gap> b := gens[3];
(1)*b
gap> id := One(pa);
(1)*u
```

As we expect, in this case, the multiplicative identity for *pa* and the single vertex *u* are one in the same:

```
gap> u = id;
true
```

We now create a list of generators for an ideal and ask for its Groebner basis:

```
gap> polys := [a*b*a-b,b*a*b-b];
[ (-1)*b+(1)*a*b*a, (-1)*b+(1)*b*a*b ]
gap> gb := GBNPGroebnerBasis(polys,pa);
[ (-1)*a*b+(1)*b*a, (-1)*a*b+(1)*b^2, (-1)*b+(1)*a^2*b ]
```

Next, we create an ideal *I* in {\GAP} using the Groebner basis *gb* found above, and then the quotient pa/I:

```
gap> I := Ideal(pa,gb);
<two-sided ideal in <algebra-with-one over Rationals, with 3
generators>,
  (3 generators)>
```

Once we have the generators for a Groebner basis, we set the appropriate property for the ideal *I*:

```
gap> grb := GroebnerBasis(I,gb);
<partial two-sided Groebner basis containing 3 elements>
```

1.2 Example 2

In this next example we create another path algebra that is essentially the free algebra on six generators. We then find the Groebner basis for a commutative example from (create bibliographic reference here) the book ”Some Tapas of Computer Algebra” by A.M. Cohen, H. Cuypers, H. Sterk. We create the underlying quiver, and from it the path algebra over the rational numbers:

```
gap> q := Quiver(["u"],[["u","u","a"],["u","u","b"], ["u","u","c"],
>                     ["u","u","d"],["u","u","e"],["u","u","f"]]);
<quiver with 1 vertices and 6 arrows>
gap> fq := PathAlgebra(Rationals,q);
<algebra-with-one over Rationals, with 7 generators>
```

Next, the generators are labeled and the list of polynomials is entered:

Example
```
gap> gens := GeneratorsOfAlgebra(fq);
[ (1)*u, (1)*a, (1)*b, (1)*c, (1)*d, (1)*e, (1)*f ]
gap> u := gens[1];; a := gens[2];; b := gens[3];; c := gens[4];;
gap> d := gens[5];; e := gens[6];; f := gens[7];;
gap> polys := [ e*a,
>             a^3 + f*a,
>             a^9 + c*a^3,
>             a^81 + c*a^9 + d*a^3,
>             a^27 + d*a^81 + e*a^9 + f*a^3,
>             b + c*a^27 + e*a^81 + f*a^9,
>             c*b + d*a^27 + f*a^81,
>             a + d*b + e*a^27,
>             c*a + e*b + f*a^27,
>             d*a + f*b,
>             b^3 - b,
>             a*b - b*a, a*c - c*a,
>             a*d - d*a, a*e - e*a,
>             a*f - f*a, b*c - c*b,
>             b*d - d*b, b*e - e*b,
>             b*f - f*b, c*d - d*c,
>             c*e - e*c, c*f - f*c,
>             d*e - e*d, d*f - f*d,
>             e*f - f*e
> ];;
```

Finally, the Groebner basis is found:

Example
```
gap> gb := GBNPGroebnerBasis(polys,fq);
[ (1)*a, (1)*b, (-1)*c*d+(1)*d*c, (-1)*c*e+(1)*e*c, (-1)*d*e+(1)*e*d,
  (-1)*c*f+(1)*f*c, (-1)*d*f+(1)*f*d, (-1)*e*f+(1)*f*e ]
```

1.3 Example 3

The next example is from B. Keller's PhD thesis, p. 26:

Example
```
gap> q := Quiver(["u","v"],[["u","v","c"],["u","u","b"],["u","u","a"]]);
<quiver with 2 vertices and 3 arrows>
gap> pa := PathAlgebra(Rationals,q);
<algebra-with-one over Rationals, with 5 generators>
gap>
gap> # Get generators of path algebra:
gap> gens := GeneratorsOfAlgebra(pa);
[ (1)*u, (1)*v, (1)*c, (1)*b, (1)*a ]
gap> u := gens[1];; v := gens[2];; c := gens[3];;
gap> b := gens[4];; a := gens[5];; id := One(pa);;
gap>
gap> polys := [a*b*c+b*a*b+a+c];
[ (1)*c+(1)*a+(1)*b*a*b+(1)*a*b*c ]
gap> gb := GBNPGroebnerBasis(polys,pa);
[ (-1)*b*c+(1)*a*c, (1)*a+(1)*b*a*b, (1)*c+(1)*a*b*c, (-1)*b*a^2+(1)*a^2*b ]
```

1.4 Example 4

Here's an example that doesn't meet our necessary criteria that all elements in a generating set have monomials in the arrow ideal. Since the given path algebra is isomorphic to a free algebra, the single vertex is sent to the identity and there are no complications. First, we set up the algebra and generating set:

Example
```
gap> q := Quiver(["u"],[["u","u","x"],["u","u","y"]]);
<quiver with 1 vertices and 2 arrows>
gap> f := Rationals;
Rationals
gap> fq := PathAlgebra(f,q);
<algebra-with-one over Rationals, with 3 generators>
gap>
gap> # Get generators of path algebra:
gap> gens := GeneratorsOfAlgebra(fq);
[ (1)*u, (1)*x, (1)*y ]
gap> u := gens[1];; x := gens[2];; y := gens[3];; id := One(fq);;
gap> polys := [x*y-y*x,x^2*y-id,x*y^2-id];
[ (1)*x*y+(-1)*y*x, (-1)*u+(1)*x^2*y, (-1)*u+(1)*x*y^2 ]
```

Then we ask GBNP for its Groebner basis:

Example
```
gap> gb := GBNPGroebnerBasisNC(polys,fq);
The given path algebra is isomorphic to a free algebra.
[ (-1)*x+(1)*y, (-1)*u+(1)*x^3 ]
```

NOTE: It is important to realize that we've used the routine 'GBNPGroebnerBasisNC' which doesn't check that all elements in a given list have non-vertex monomials. So, if we run the standard QPA Groebner basis routine on this example, we get the following:

Example
```
gap> GBNPGroebnerBasis(polys,pa);
Please make sure all elements are in the given path algebra,
and each summand of each element is not (only) a vertex.
false
```

Chapter 2

Quivers

2.1 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 3.2 are normally not useful in isolation; typically, they are invoked by the functions for constructing quivers in Section 2.2.

2.1.1 InfoQuiver

◊ `InfoQuiver` (info class)

is the info class for functions dealing with quivers.

2.2 Constructing Quivers

2.2.1 Quiver

◊ `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 satisfies the property `IsQuiver` (2.3.1).

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
```

2.2.2 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 2.4 for more information.

2.3 Categories and Properties of Quivers

2.3.1 IsQuiver

◊ `IsQuiver(`*`object`*`)` (property)

Returns: true when `object` is a quiver.

2.3.2 IsAcyclicQuiver

◊ `IsAcyclicQuiver(`*`quiver`*`)` (property)

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

```
Example
gap> quiver1 := Quiver(2,[[1,2]]);
<quiver with 2 vertices and 1 arrows>
gap> IsQuiver("v1");
```

```
false
gap> IsQuiver(quiver1);
true
gap> IsAcyclicQuiver(quiver1);
true
gap> quiver2 := Quiver(["u","v"],[["u","v"],["v","u"]]);
<quiver with 2 vertices and 2 arrows>
gap> IsAcyclicQuiver(quiver2);
false
gap> IsFinite(quiver1);
true
gap> IsFinite(quiver2);
false
```

2.4 Orderings of paths in a quiver

To be written.

2.5 Attributes and Operations for Quivers

2.5.1 .

◊ `.(`*`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.

2.5.2 VerticesOfQuiver

◊ `VerticesOfQuiver(`*`quiver`*`)` (attribute)

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

2.5.3 ArrowsOfQuiver

◊ `ArrowsOfQuiver(`*`quiver`*`)` (attribute)

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

2.5.4 AdjacencyMatrixOfQuiver

◊ `AdjacencyMatrixOfQuiver(`*`quiver`*`)` (attribute)

Returns: the adjacency matrix of *`quiver`*.

2.5.5 GeneratorsOfQuiver

◊ `GeneratorsOfQuiver(`*`quiver`*`)` (attribute)

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

2.5.6 NumberOfVertices

◊ `NumberOfVertices(quiver)` (attribute)

Returns: the number of vertices in *quiver*.

2.5.7 NumberOfArrows

◊ `NumberOfArrows(quiver)` (attribute)

Returns: the number of arrows in *quiver*.

2.5.8 OrderingOfQuiver

◊ `OrderingOfQuiver(quiver)` (attribute)

Returns: the ordering used to order elements in *quiver*. See Section 2.4 for more information.

2.5.9 OppositeOfQuiver

◊ `OppositeOfQuiver(quiver)` (operation)

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”.

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 := OppositeOfQuiver(q1);
<quiver with 2 vertices and 4 arrows>
gap> VerticesOfQuiver(q1);
[ u_op, v_op ]
gap> ArrowsOfQuiver(q1);
[ a_op, b_op, c_op, d_op ]
```

2.6 Categories and Properties of Paths

2.6.1 IsPath

◊ `IsPath(object)` (category)

All path objects are in this category.

2.6.2 IsVertex

◊ `IsVertex(object)` (category)

All vertices are in this category.

2.6.3 IsArrow

◊ `IsArrow(object)` (category)

All arrows are in this category.

2.6.4 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
```

2.7 Attributes and Operations of Paths

2.7.1 SourceOfPath

◊ `SourceOfPath(path)` (attribute)

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

2.7.2 TargetOfPath

◊ `TargetOfPath(path)` (attribute)

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

2.7.3 LengthOfPath

◊ `LengthOfPath(path)` (attribute)

Returns: the length of `path`.

2.7.4 WalkOfPath

◊ `WalkOfPath(path)` (attribute)

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

2.7.5 *

◊ `*(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
```

2.7.6 =

◊ `=(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.

2.7.7 <

◊ `<(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.

Example
```
gap> q1.a=q1.b;
false
gap> q1.a < q1.v;
false
gap> q1.a < q1.c;
true
```

2.8 Attributes of Vertices

2.8.1 IncomingArrowsOfVertex

◊ `IncomingArrowsOfVertex(vertex)` (attribute)
Returns: a list of arrows having *vertex* as target. Only meaningful if *vertex* is in a quiver.

2.8.2 OutgoingArrowsOfVertex

◊ `OutgoingArrowsOfVertex(vertex)` (attribute)
Returns: a list of arrows having *vertex* as source.

2.8.3 InDegreeOfVertex

◊ `InDegreeOfVertex(vertex)` (attribute)
Returns: the number of arrows having *vertex* as target. Only meaningful if *vertex* is in a quiver.

2.8.4 OutDegreeOfVertex

◊ `OutDegreeOfVertex(vertex)` (attribute)
Returns: the number of arrows having *vertex* as source.

2.8.5 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 3

Path Algebras

3.1 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.

3.2 Constructing Path Algebras

3.2.1 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);
<algebra-with-one over Rationals, with 5 generators>
```

3.3 Categories and Properties of Path Algebras

3.3.1 IsPathAlgebra

◊ `IsPathAlgebra(object)` (property)

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

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

3.4 Attributes and Operations for Path Algebras

3.4.1 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>
```

3.4.2 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.

3.4.3 .

◊ `.(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
```

3.5 Operations on Path Algebra Elements

3.5.1 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.

3.5.2 <

◊ `<(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.

3.5.3 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.

3.5.4 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.

3.5.5 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
```

3.5.6 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.

3.5.7 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.

3.5.8 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
```

3.5.9 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*.

3.5.10 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`*.

3.5.11 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.

3.6 Constructing Quotients of Path Algebras

See Chapter 60: Algebras in the GAP manual on how to construct an ideal and a quotient of an algebra. When the quotient is constructed, it is still a path algebra [?!] and thus the commands introduced for path algebras also works with quotients.

3.7 Ideals and operations on ideals

3.7.1 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` (3.8.1).

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> I := Ideal(FQ, [FQ.a - FQ.b*FQ.c, FQ.d*FQ.d]);
<two-sided ideal in <algebra-with-one over Rationals, with 6
  generators>, (2 generators)>
gap> GeneratorsOfIdeal(I);
[ (1)*a+(-1)*b*c, (1)*d^2 ]
gap> IsIdealInPathAlgebra(I);
true
```

3.7.2 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*.

3.7.3 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*).

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

◊ `\in (elt. in path alg. and ideal)(elt, I)` (operation)

Arguments: *elt* - an element in a path algebra, *I* - an ideal in the same path algebra (i.e. with `IsIdealInPathAlgebra` (3.8.1) 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.

3.8 Categories and properties of ideals

3.8.1 IsIdealInPathAlgebra

◊ `IsIdealInPathAlgebra(I)` (property)

Arguments: *I* – an IsFLMLOR object.
Returns: true whenever *I* is an ideal in a path algebra.

3.8.2 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 Rˆ2 and *I* contains Rˆn for some n, where R is the arrow ideal.
Technical note: The second condition is checked by verifying if respective quotient algebra is finite dimensional (and this uses Groebner bases machinery).

3.8.3 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.

3.9 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 <algebra-with-one over Rationals, with 6
  generators>, (2 generators)>
gap> GroebnerBasis(I, gb);
<complete two-sided Groebner basis containing 2 elements>
gap> quot := FQ/I;
<algebra-with-one over Rationals, with 6 generators>
```

3.10 Categories and Properties of Quotients of Path Algebras

3.10.1 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;
<algebra-with-one over Rationals, with 6 generators>
gap> IsQuotientOfPathAlgebra(quot);
true
gap> IsQuotientOfPathAlgebra(FQ);
false
```

3.10.2 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).

3.11 Attributes and Operations for Quotients of Path Algebras

3.11.1 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.
Technical note: For a path algebra it uses a standard GAP method (it breaks for infinite dimensional case!). 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).

3.11.2 IsSelfinjectiveAlgebra

◊ `IsSelfinjectiveAlgebra(A)` (operation)

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.

3.11.3 IsSymmetricAlgebra

◊ `IsSymmetricAlgebra(A)` (operation)

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.

3.11.4 IsWeaklySymmetricAlgebra

◊ `IsWeaklySymmetricAlgebra(A)` (operation)

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.

3.11.5 LoewyLength

◊ `LoewyLength(A)` (operation)

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`.

3.11.6 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.

3.11.7 CoxeterMatrix

◊ `CoxeterMatrix(A)` (operation)

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.

3.11.8 CoxeterPolynomial

◊ `CoxeterPolynomial(A)` (operation)

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.

3.11.9 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`, after having checked that `A` is a finite dimensional quotient of a path algebra (the check is not implemented and also not implemented for path algebras).

3.12 Attributes and Operations on Elements of Quotients of Path Algebra

3.12.1 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
```

3.12.2 Coefficients

◊ `Coefficients(`*element*`)` (operation)

The operation `Coefficients` operates on an *element* of a quotient of a path algebra, and it returns the coefficients of the *element* in terms of its canonical basis.
Note: Not in QPA, takes two arguments in GAP.

3.12.3 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
```

3.12.4 <

◊ `<(`*a*, *b*`)` (operation)

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

3.12.5 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.

3.12.6 OriginalPathAlgebra

◊ `OriginalPathAlgebra(`*`algebra`*`)` (operation)

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 a quotient of a path algebra.

3.13 Predefined classes of (quotients of) path algebras

3.13.1 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.

3.13.2 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.

3.13.3 NakayamaAlgebra

◊ `NakayamaAlgebra(`*`admiss-seq, field`*`)` (function)

Arguments: *`admiss-seq`* – a list of positive integers, *`field`* – a field.
Returns: The Nakayama algebra corresponding to *`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);
<algebra-with-one over Rationals, with 3 generators>
gap> QuiverOfPathAlgebra(alg);
<quiver with 2 vertices and 1 arrows>
```

3.13.4 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*.

3.13.5 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` (3.13.6) and also an Example below).

3.13.6 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` (3.13.5), I is an admissible ideal (`IsAdmissibleIdeal` (3.8.2)) 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 and 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 (cf. `IsSpecialBiserialQuiver` (3.13.5)).

3.13.7 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` (3.13.6) and I is generated by monomials (= "zero-relations") (cf. `IsMonomialIdeal` (3.8.3)).

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> I1 := Ideal(A, rel1);;  quo1 := A/I1;;
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> I2 := Ideal(A, rel2);; quo2 := A/I2;;
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> I3 := Ideal(A, rel3);; quo3 := A/I3;;
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> I4 := Ideal(A, rel4);; quo4 := A/I4;;
gap> IsSpecialBiserialAlgebra(quo4); IsStringAlgebra(quo4);
false
false
```

3.14 Opposite algebras

3.14.1 OppositeQuiver

◊ `OppositeQuiver(`*Q*`)` (attribute)

Arguments: *Q* – a quiver.
Returns: the opposite quiver Q^{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` (3.14.2) 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`.

3.14.2 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` (3.14.1) and `OppositePath` (3.14.2).

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
```

3.14.3 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` (3.14.1)). 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` (3.14.4) 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`.

3.14.4 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` (3.14.3) and `OppositePathAlgebraElement` (3.14.4).

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
```

3.15 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` (3.15.1) of the quivers of the original algebras.

The operation `TensorProductOfAlgebras` (3.15.6) computes the tensor products of two quotients of path algebras as a quotient of a path algebra.

3.15.1 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).

3.15.2 QuiverProductDecomposition

◊ `QuiverProductDecomposition(Q)` (attribute)

Arguments: *Q* – a quiver.

Returns: the original quivers *Q* is a product of, if *Q* was created by the `QuiverProduct` (3.15.1) operation.

The value of this attribute is an object in the category `IsQuiverProductDecomposition` (3.15.3).

3.15.3 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` (3.15.4) and `ProjectFromProductQuiver` (3.15.5).

3.15.4 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.

3.15.5 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.

```
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
```

3.15.6 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` (3.15.1) of the quivers of the operands.

3.15.7 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` (3.15.6)).

3.15.8 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` (3.15.6)).

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 );
<algebra-with-one over Rationals, with 3 generators>
gap> fq2 := PathAlgebra( Rationals, q2 );
<algebra-with-one over Rationals, with 8 generators>
gap> I := Ideal( fq2, [ fq2.b * fq2.d - fq2.c * fq2.e ] );
<two-sided ideal in <algebra-with-one over Rationals, with 8 generators>,
 (1 generators)>
gap> quot := fq2 / I;
<algebra-with-one over Rationals, with 8 generators>
gap> t := TensorProductOfAlgebras( fq1, quot );
<algebra-with-one over Rationals, with 20 generators>
gap> SimpleTensor( [ fq1.a, quot.b ], t );
[(1)*a_v1*u2_b]
gap> t_dec := TensorProductDecomposition( t );
[ <algebra-with-one over Rationals, with 3 generators>,
  <algebra-with-one over Rationals, with 8 generators> ]
gap> t_dec[ 1 ] = fq1;
true
```

3.15.9 EnvelopingAlgebra

◊ `EnvelopingAlgebra(FQ)` (operation)

Arguments: *FQ* – a (quotient of) a path algebra.

Returns: the enveloping algebra $FQ^{\mathrm{e}} = FQ \otimes FQ^{\mathrm{op}}$ of *FQ*

3.15.10 IsEnvelopingAlgebra

◊ `IsEnvelopingAlgebra(A)` (attribute)

Arguments: *A* – an algebra.
Returns: true if and only if *A* is the result of a call to `EnvelopingAlgebra` (3.15.9).

Chapter 4

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.

4.1 Constructing a Groebner Basis

4.1.1 InfoGroebnerBasis

◊ `InfoGroebnerBasis` (info class)

is the info class for functions dealing with Groebner basis.

4.1.2 GroebnerBasis

◊ `GroebnerBasis(`*A*`, `*rels*`)` (function)

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

Returns: an object *GB* in the 'IsGroebnerBasis' category with 'IsCompleteGroebnerBasis' property set on true.

Sets also *GB* as a value of the attribute GroebnerBasisOfIdeal 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.

4.2 Categories and Properties of Groebner Basis

4.2.1 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.

```
Example
gap> Q := Quiver( 3, [ [1,2,"a"], [2,3,"b"] ] );
<quiver with 3 vertices and 2 arrows>
gap> PA := PathAlgebra( Rationals, Q );
<algebra-with-one over Rationals, with 5 generators>
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 <algebra-with-one over Rationals, with 5 generators>,
  (1 generators)>
gap> grb := GroebnerBasis( I, gb );
<complete two-sided Groebner basis containing 1 elements>
gap> alg := PA/I;
<algebra-with-one over Rationals, with 5 generators>
gap> IsGroebnerBasis(gb);
false
gap> IsGroebnerBasis(grb);
true
```

4.2.2 IsTipReducedGroebnerBasis

◊ `IsTipReducedGroebnerBasis(`*gb*`)` (property)

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

4.2.3 IsCompletelyReducedGroebnerBasis

◊ `IsCompletelyReducedGroebnerBasis(`*gb*`)` (property)

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

4.2.4 IsHomogeneousGroebnerBasis

◊ `IsHomogeneousGroebnerBasis(`*gb*`)` (property)

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

4.2.5 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.

4.3 Attributes and Operations for Groebner Basis

4.3.1 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.

4.3.2 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*.

4.3.3 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.

4.3.4 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.

4.3.5 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.

4.3.6 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.

4.3.7 Nontips

◊ `Nontips(GB)` (operation)

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’.

4.3.8 AdmitsFinitelyManyNontips

◊ `AdmitsFinitelyManyNontips(GB)` (operation)

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

4.3.9 NontipSize

◊ `NontipSize(GB)` (operation)

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

4.3.10 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.

4.4 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.

4.4.1 IsRightGroebnerBasis

◊ `IsRightGroebnerBasis(object)` (property)

Arguments: `object` – any object in GAP.
Returns: true when `object` a right Groebner basis.

4.4.2 RightGroebnerBasisOfIdeal

◊ `RightGroebnerBasisOfIdeal(I)` (attribute)

Arguments: `I` – a right ideal.
Returns: a right Groebner basis of a right ideal, `I`, is one has been computed.

4.4.3 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.

Chapter 5

Using GBNP with Gap

5.1 GBNP

GBNP is a non-commutative Groebner Basis package which assumes that all algebras involved will be free algebras over a finite number of non-commuting generators. It also assumes that the ordering on the monomials is left length-lexicographic.

5.2 Setting up GBNP

in progress...

5.3 Relevant GBNP internals

in progress...

5.4 Communicating with GBNP

in progress...

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 are presentations defined by vertex projective modules.

6.1 Matrix Modules

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. In practice when creating the modules all we need to know is the transformations and 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

◊ `RightModuleOverPathAlgebra(`*`A, mats`*`)` (operation)
◊ `RightModuleOverPathAlgebra(`*`A, dim_vector, gens`*`)` (operation)

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

Returns: a module over a path algebra or over a qoutient of a path algebra in the second variant.

In the first function call, the list of matrices *`mats`* can take on three different forms.

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.

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, each arrow needs to be assigned a matrix, otherwise an error will be returned. The function verifies that the dimensions of the matrices and vector spaces are correct and match, and that each arrow has only one matrix assigned to it.

In the second 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 if the algebra *A* is a (quotient of a) path algebra and if the matrices of the action of the arrows have the correct size according to the dimension vector entered and also whether or not the relations of the algebra are satisfied.

Example
```
gap&gt; Q := Quiver(2, [[1, 2, "a"], [2, 1, "b"],[1, 1, "c"]]);
&lt;quiver with 2 vertices and 3 arrows&gt;
gap&gt; P := PathAlgebra(Rationals, Q);
&lt;algebra-with-one over Rationals, with 5 generators&gt;
gap&gt; matrices := [["a", [[1,0,0],[0,1,0]]], ["b", [[0,1],[1,0],[0,1]]],
&gt; ["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&gt; M := RightModuleOverPathAlgebra(P,matrices);
&lt;right-module over &lt;algebra-with-one over Rationals, with 5
generators&gt;&gt;
gap&gt; mats := [ [[1,0,0], [0,1,0]], [[0,1],[1,0],[0,1]], [[0,0],[1,0]] ];;
gap&gt; N := RightModuleOverPathAlgebra(P,mats);
&lt;right-module over &lt;algebra-with-one over Rationals, with 5
generators&gt;&gt;
gap&gt; arrows := ArrowsOfQuiver(Q);
[ a, b, c ]
gap&gt; mats := [[arrows[1], [[1,0,0],[0,1,0]]],
&gt; [arrows[2], [[0,1],[1,0],[0,1]]], [arrows[3], [[0,0],[1,0]]]];;
gap&gt; N := RightModuleOverPathAlgebra(P,mats);
&lt;right-module over &lt;algebra-with-one over Rationals, with 5
generators&gt;&gt;
gap&gt; # Next we give the vertex simple associate to vertex 1.
gap&gt; M :=
RightModuleOverPathAlgebra(P,[["a",[1,0]],["b",[0,1]],["c",[[0]]]]);
&lt;right-module over &lt;algebra-with-one over Rationals, with 5
                                        generators&gt;&gt;
gap&gt; # Finally, the next defines the zero representation of the quiver.
```

```
gap&gt; M :=
RightModuleOverPathAlgebra(P,[["a",[0,0]],["b",[0,0]],["c",[0,0]]]);
&lt;right-module over &lt;algebra-with-one over Rationals, with 5
                                        generators&gt;&gt;
gap&gt; Dimension(M);
0
gap&gt; Basis(M);
Basis( &lt;
0-dimensional right-module over &lt;algebra-with-one over Rationals, with
5 generators&gt;&gt;, [  ] )
gap> # Using 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,[2,3],matrices);
<right-module over <algebra-with-one over Rationals, with 5 generators>>
gap> M := RightModuleOverPathAlgebra(P,[2,3],[]);
<right-module over <algebra-with-one over Rationals, with 5 generators>>
```

6.2 Categories Of Matrix Modules

6.2.1 IsPathAlgebraModule

◊ `IsPathAlgebraModule(`*`object`*`)` (filter)

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

These matrix modules fall under the category 'IsAlgebraModule' with the added filter of 'IsPathAlgebraModule'. Operations available for algebra modules can be applied to path algebra modules. See "ref:representations of algebras" for more details. These modules are also vector spaces over the field of the path algebra. So refer to "ref:vector spaces" for descriptions of the basis and elementwise operations available.

6.3 Acting on Module Elements

6.3.1 ^

◊ `^(`*`m, p`*`)` (operation)

Arguments: *m* – an element in a module, *p* – a path in a path 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);
<right-module over <algebra-with-one over Rationals, with 5
generators>>
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);
<algebra-with-one over Rationals, with 8 generators>
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> I:= Ideal(KQ,rels);;
gap> gb:= GBNPGroebnerBasis(rels,KQ);;
gap> gbb:= GroebnerBasis(I,gb);;
gap> A:= KQ/I;
<algebra-with-one over Rationals, with 8 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);
<right-module over <algebra-with-one over Rationals, with 8 generators>>
```

6.4.1 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 `PathAlgebraMatModule`s over the same (quotient of a) path algebra.

6.4.2 DimensionVector

◊ `DimensionVector(M)` (operation)

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

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

A shortcoming of this that it is not defined for modules of quotients of path algebras.

6.4.3 Dimension

◊ `Dimension(M)` (operation)

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

Returns: the dimension of the representation *M*.

6.4.4 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.

6.4.5 DirectSumOfModules

◊ `DirectSumOfModules(L)` (operation)

Arguments: *L* – a list of `PathAlgebraMatModule`s 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`, `DirectSumProjections` and `DirectSumInclusions`.

6.4.6 IsDirectSumOfModules

◊ `IsDirectSumOfModules(M)` (attribute)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`).
Returns: true if *M* is constructed via the command `DirectSumOfModules`.

6.4.7 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`.

6.4.8 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`.
Using the example above.

Example
```
gap> N2:=DirectSumOfModules([N,N]);
<14-dimensional right-module over <algebra-with-one of dimension
17 over Rationals>>
gap> proj:=DirectSumProjections(N2);
[ <mapping: <14-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> >,
  <mapping: <14-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> > ]
gap> inc:=DirectSumInclusions(N2);
[ <mapping: <7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    14-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> >,
  <mapping: <7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    14-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> > ]
```

6.4.9 1stSyzygy

◊ `1stSyzygy(`*M*`)` (operation)

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

6.4.10 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.

6.4.11 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.

6.4.12 IsInjectiveModule

◊ `IsInjectiveModule(`*M*`)` (operation)

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

6.4.13 IsProjectiveModule

◊ `IsProjectiveModule(`*M*`)` (operation)

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

6.4.14 IsSemisimpleModule

◊ `IsSemisimpleModule(`*M*`)` (operation)

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

6.4.15 IsSimpleModule

◊ `IsSimpleModule(M)` (operation)

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

6.4.16 LoewyLength

◊ `LoewyLength(M)` (operation)

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).

6.4.17 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.

6.4.18 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.

6.4.19 NthSyzygy

◊ `NthSyzygy(M, n)` (operation)

Arguments: *M* – a path algebra module (`PathAlgebraMatModule`), *n* – a positive integer.
Returns: the top of the syzygies until a syzygy is projective or the *n*-th syzygy has been computed.

6.4.20 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 projective dimension of *M*. It does not check if the `n`-th syzygy is projective or not.

6.4.21 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` (7.3.17) computes the inclusion of the radical of *M* into *M*.

6.4.22 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.

6.4.23 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.

6.4.24 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` (7.3.18).

6.4.25 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` (7.3.19).

6.4.26 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.

6.4.27 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` (7.3.20) computes the projection of the module *M* onto the top of the module *M*.

6.4.28 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.

6.4.29 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. A shortcoming of this that it is not defined for modules of quotients of path algebras.

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)` (operation)

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 IndecProjectiveModules

◊ `IndecProjectiveModules(A[, list])` (operation)

Arguments: `A` – a finite dimensional (quotient of a) path algebra, (optional) `list` – a list of integers.

Returns: a list of all the indecomposable projective representations over `A`, when only one argument is supplied. The second argument should be a list of integers, for example [1, 3, 4], which will return the indecomposable projective corresponding to vertex 1, 3 and 4, in this order.

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 IndecInjectiveModules

◊ `IndecInjectiveModules(A[, list])` (operation)

Arguments: `A` – a finite dimensional (quotient of a) path algebra, (optional) `list` – a list of integers.

Returns: a list of all the indecomposable injective representations over `A`, when only one argument is supplied. The second argument should be a list of integers, for example [1, 3, 4], which will return the indecomposable injective corresponding to vertex 1, 3 and 4, in this order.

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)` (operation)

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)` (operation)

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)` (operation)

Arguments: *M* – a representation of a path algebra KQ.
Returns: the dual of *M* over the opposite path algebra *KQ_op*.

6.6.2 DualOfModuleHomomorphism

◊ `DualOfModuleHomomorphism(f)` (operation)

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 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`.

6.6.5 TransposeOfModule

◊ `TransposeOfModule(M)` (operation)

Arguments: `M` – a path algebra module.
Returns: the transpose of the module `M`.

6.7 Vertex Projective Presentations

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`.

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 ”ref:Representations of Algebras”) and all operations for algebra modules are applicable to right projective modules. In particular, one can construct submodules using ‘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);
<algebra-with-one over GF(11), with 6 generators>
gap> P:=RightProjectiveModule(A,[A.v,A.v,A.w]);
<right-module over <algebra-with-one over GF(11), with 6 generators>>
gap> Dimension(P);
12
```

6.7.2 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.

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 <algebra-with-one of dimension 8 over GF(11)>>
gap> Dimension(S);
3
```

6.7.3 ^

◊ `^(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 ]
```

6.7.4 <

◊ `<(m1, m2)` (operation)

Arguments: `m1, m2` – two elements of a module over a path algebra (?).

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
```

6.7.5 /

◊ `/(M, N)` (operation)

Arguments: `M, N` – two finite dimensional modules over a path algebra (?).

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.

This

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

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> gen:= GeneratorsOfAlgebra(KQ);;
gap> a:= gen[4];;
gap> b:= gen[5];;
gap> c:= gen[6];;
gap> d:= gen[7];;
gap> e:= gen[8];;
gap> rels:= [d*e,c^2,a*c*d-b*d,e*a];;
gap> I:= Ideal(KQ,rels);;
gap> gb:= GBNPGroebnerBasis(rels,KQ);;
gap> gbb:= GroebnerBasis(I,gb);;
gap> A:= KQ/I;;
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

IsAdditiveElementWithZero, IsAdditiveElementWithInverse, IsGeneralMapping, RespectsAddition, RespectsZero, RespectsScalarMultiplication, IsTotal and IsSingleValued or not.

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 (7.2.2) and by Source (7.2.3), respectively.

Example
```
gap> L := RightModuleOverPathAlgebra(A,[["a",[0,1]],["b",[0,1]],
["c",[[0]]],["d",[[1]]],["e",[1,0]]]);
<right-module over <algebra-with-one over Rationals, with 8 generators>>
gap> DimensionVector(L);
[ 0, 1, 1 ]
gap> f := RightModuleHomOverAlgebra(L,N,[[[0,0,0]], [[1,0]], [[1,2]]]);
<mapping: <2-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
[(1)*e] ] )> -> <7-dimensional right-module over AlgebraWithOne(Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e] ] )> >
gap> IsPathAlgebraModuleHomomorphism(f);
true
```

7.2 Generalities of homomorphisms

7.2.1 MatricesOfPathAlgebraMatModuleHomomorphism

◊ MatricesOfPathAlgebraMatModuleHomomorphism(*f*) (operation)

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

Returns: the matrices defining the homomorphism *f*.

7.2.2 Range

◊ `Range(f)` (attribute)

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

7.2.3 Source

◊ `Source(f)` (attribute)

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

Example
```
gap> MatricesOfPathAlgebraMatModuleHomomorphism(f);
[ [ [ 0, 0, 0 ] ], [ [ 1, 0 ] ], [ [ 1, 2 ] ] ]
gap> Range(f);
<7-dimensional right-module over <algebra-with-one over Rationals, with
8 generators>>
gap> Source(f);
<2-dimensional right-module over <algebra-with-one over Rationals, with
8 generators>>
gap> Source(f) = L;
true
```

7.2.4 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.

7.2.5 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.

7.2.6 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

```
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 ] ] ]
```

7.2.7 Zero

◊ `Zero(f)` (operation)

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

7.2.8 ZeroMapping

◊ `ZeroMapping(M, N)` (operation)

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

7.2.9 IdentityMapping

◊ `IdentityMapping(M)` (operation)

Arguments: *M* - a module.
Returns: the identity map between *M* and *M*.

7.2.10 \= (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.

7.2.11 \+ (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.

7.2.12 * (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.

Example
```
gap> z:=Zero(f);
<mapping: <2-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> f = z;
false
gap> Range(f) = Range(z);
true
gap> y := ZeroMapping(L,N);
<mapping: <2-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> y = z;
true
gap> id := IdentityMapping(N);
<mapping: <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> f*id;
```

```
<mapping: <2-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> id*f;
Error, codomain of the first argument is not equal to the domain of the 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>
gap> 2*f + z;
<mapping: <2-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
```

7.2.13 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*.

7.2.14 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*.

7.2.15 IsInjective

◊ `IsInjective(f)` (property)

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

7.2.16 IsSurjective

◊ `IsSurjective(f)` (property)

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

7.2.17 IsIsomorphism

◊ `IsIsomorphism(f)` (property)

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

```
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]]]);;
gap> g := CoKernelProjection(f);
<mapping: <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <5-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> CoKernelOfWhat(g) = f;
true
gap> h := ImageProjection(f);
<mapping: <2-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <2-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> ImageOfWhat(h) = f;
true
gap> IsInjective(f); IsSurjective(f); IsIsomorphism(f); IsIsomorphism(h);
true
false
false
true
```

7.2.18 IsSplitEpimorphism

◊ `IsSplitEpimorphism(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 homomorphism *f*.

7.2.19 IsSplitMonomorphism

◊ `IsSplitMonomorphism(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 homomorphism *f*.

```
Example
gap> S := SimpleModules(A)[1];;
gap> H := HomOverAlgebra(N,S);;
gap> IsSplitMonomorphism(H[1]);
false
gap> f := IsSplitEpimorphism(H[1]);
<mapping: <1-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> IsSplitMonomorphism(f);
<mapping: <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <1-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
```

7.2.20 IsZero

◇ `IsZero(f)` (property)

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

7.2.21 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]]]);
<right-module over <algebra-with-one over Rationals, with 8 generators>>
gap> f := RightModuleHomOverAlgebra(L,N,[[[0,0,0]], [[1,0]], [[1,2]]]);;
gap> IsZero(0*f);
true
gap> KnownAttributesOfObject(g);
[ "Range", "Source", "PathAlgebraOfMatModuleMap", "KernelOfWhat" ]
gap> KernelOfWhat(g) = f;
true
```

7.3 Homomorphisms and modules constructed from homomorphisms and modules

7.3.1 CoKernel

◇ `CoKernel(f)` (attribute)
◇ `CoKernelProjection(f)` (attribute)

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

Returns: the cokernel of a homomorphism `f` between two modules.

The first variant `CoKernel` returns the cokernel of the homomorphism `f` as a module, while the latter one returns the projection homomorphism from the range of the homomorphism `f` to the cokernel of the homomorphism `f`.

7.3.2 EndOverAlgebra

◊ `EndOverAlgebra(M)` (operation)

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.

7.3.3 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.

7.3.4 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.

7.3.5 Image

◊ `Image(f)` (attribute)
◊ `ImageProjection(f)` (attribute)
◊ `ImageInclusion(f)` (attribute)
◊ `ImageProjectionInclusion(f)` (attribute)

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

Returns: the image of a homomorphism `f` between two modules.

The first variant `Image` returns the image of the homomorphism `f` as a module. The second returns the projection from the source of `f` to the image of the homomorphism `f`. The third returns the inclusion of the image into the range of the homomorphism *f*. The last one returns both the projection and the inclusion as a list of two elements (first the projection and then the inclusion).

7.3.6 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);
[ <mapping: <7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> >,
  <mapping: <7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> >,
  <mapping: <7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> >,
  <mapping: <7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> >,
  <mapping: <7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> > ]
gap> g := hom[1];
<mapping: <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> M := CoKernel(g);
<6-dimensional right-module over <algebra-with-one over Rationals, with
```

```
8 generators>>
gap> f := CoKernelProjection(g);
<mapping: <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <6-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
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);
<mapping: <8-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> U := Image(p);
<5-dimensional right-module over <algebra-with-one over Rationals, with
8 generators>>
gap> projinc := ImageProjectionInclusion(p);
[ <mapping: <8-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    5-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> >,
  <mapping: <5-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> > ]
gap> U = Range(projinc[1]);
true
gap> Kernel(p);
<3-dimensional right-module over <algebra-with-one over Rationals, with
8 generators>>
```

7.3.7 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`.

7.3.8 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]]);
<mapping: <5-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> f := SubRepresentationInclusion(N,[B[1],B[2]]);
<mapping: <6-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> LiftingInclusionMorphisms(f,g);
<mapping: <5-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <6-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> S := SimpleModules(A);
[ <right-module over <algebra-with-one over Rationals, with 8 generators>>,
  <right-module over <algebra-with-one over Rationals, with 8 generators>>,
  <right-module over <algebra-with-one over Rationals, with 8 generators>> ]
gap> homNS := HomOverAlgebra(N,S[1]);
[ <mapping: <
    7-dimensional right-module over AlgebraWithOne( Rationals, ... )> -> <
    1-dimensional right-module over AlgebraWithOne( Rationals, ... )> >,
  <mapping: <
    7-dimensional right-module over AlgebraWithOne( Rationals, ... )> -> <
    1-dimensional right-module over AlgebraWithOne( Rationals, ... )> >,
  <mapping: <
    7-dimensional right-module over AlgebraWithOne( Rationals, ... )> -> <
    1-dimensional right-module over AlgebraWithOne( Rationals, ... )> > ]
gap> f := homNS[1];
<mapping: <
7-dimensional right-module over AlgebraWithOne( Rationals, ... )> -> <
1-dimensional right-module over AlgebraWithOne( Rationals, ... )> >
gap> p := ProjectiveCover(S[1]);
<mapping: <8-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <1-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> LiftingMorphismFromProjective(f,p);
<mapping: <8-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
```

7.3.9 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)`.

7.3.10 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)`.

7.3.11 MinimalLeftApproximation

◊ `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.

7.3.12 MinimalRightApproximation

◊ `MinimalRightApproximation(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.

Example
```
gap> H:= HomOverAlgebra(N,N);;
gap> RightMinimalVersion(H[1]);
[ <mapping: <1-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> >,
  [ <6-dimensional right-module over <algebra-with-one of dimension
        17 over Rationals>> ] ]
gap> LeftMinimalVersion(H[1]);
[ <mapping: <7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    1-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> >,
  [ <6-dimensional right-module over <algebra-with-one of dimension
        17 over Rationals>> ] ]
gap> S:=SimpleModules(A)[1];;
gap> MinimalRightApproximation(N,S);
<mapping: <1-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <1-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> S:=SimpleModules(A)[3];;
gap> MinimalLeftApproximation(S,N);
<mapping: <1-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <6-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
```

7.3.13 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 $f\beta - \alpha g = 0$.

7.3.14 ProjectiveCover

◊ `ProjectiveCover(M)` (operation)

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.

7.3.15 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).

7.3.16 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 $f\colon A \to B$ (horizontal map) and $g\colon A \to C$ (vertical map), the pushout E is returned as the two homomorphisms $[f', g']$, where $f'\colon C \to E$ (horizontal map) and $g'\colon B \to E$ (vertical map).

Example

```
gap> g := MorphismOnKernel(hom[1],hom[2],hom[1],hom[2]);
<mapping: <6-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <6-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> IsomorphicModules(Source(g),Range(g));
true
gap> p := ProjectiveCover(N);
<mapping: <24-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> N1 := Kernel(p);
<17-dimensional right-module over <algebra-with-one over Rationals, with
8 generators>>
gap> pullback := PullBack(p,hom[1]);
```

```
[ <mapping: <24-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    7-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> >,
  <mapping: <24-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    24-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> > ]
gap> Kernel(pullback[1]);
<17-dimensional right-module over <algebra-with-one over Rationals, with
8 generators>>
gap> IsomorphicModules(N1,Kernel(pullback[1]));
true
gap> t := LiftingMorphismFromProjective(p,p*hom[1]);
<mapping: <24-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <24-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> s := MorphismOnKernel(p,p,t,hom[1]);
<mapping: <17-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <17-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> Source(s)=N1;
true
gap> q := KernelInclusion(p);
<mapping: <17-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <24-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> pushout := PushOut(q,s);
[ <mapping: <17-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    24-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> >,
  <mapping: <24-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    24-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> > ]
gap> U := CoKernel(pushout[1]);
<7-dimensional right-module over <algebra-with-one over Rationals, with
8 generators>>
```

```
gap> IsomorphicModules(U,N);
true
```

7.3.17 RadicalOfModuleInclusion

◊ `RadicalOfModuleInclusion(M)` (operation)

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` (6.4.21).

7.3.18 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` (6.4.24).

7.3.19 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`.

7.3.20 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);
<mapping: <4-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> radN := Source(f);
<4-dimensional right-module over <algebra-with-one over Rationals, with
8 generators>>
```

```
gap> g := SocleOfModuleInclusion(N);
<mapping: <3-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> U := SubRepresentationInclusion(N,[B[5]+B[6],B[7]]);
<mapping: <4-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
gap> h := TopOfModuleProjection(N);
<mapping: <7-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <3-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
```

7.4 Homological algebra

7.4.1 ExtOverAlgebra

◊ `ExtOverAlgebra(M, N)` (operation)
◊ `ExtOverAlgebraAdd(M, N)` (operation)

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

Returns: a list of two elements `ExtOverAlgebra` or three elements `ExtOverAlgebraAdd`, 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 which preforms the addition in $\mathrm{Ext}^1(M,N)$.

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.

7.4.2 ExtAlgebraGenerators

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

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

Returns: 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*.

7.5 Auslander-Reiten theory

7.5.1 AlmostSplitSequence

◊ AlmostSplitSequence(*M*) (operation)

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.

Example
```
gap> S := SimpleModules(A);
[ <right-module over <algebra-with-one over Rationals, with 8 generators>>,
  <right-module over <algebra-with-one over Rationals, with 8 generators>>,
  <right-module over <algebra-with-one over Rationals, with 8 generators>> ]
gap> Ext:=ExtOverAlgebra(S[2],S[2]);
[ <mapping: <3-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    4-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> >,
  [ <mapping: <3-dimensional right-module over AlgebraWithOne( Rationals,
        [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
          [(1)*e] ] )> -> <
        1-dimensional right-module over AlgebraWithOne( Rationals,
        [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
          [(1)*e] ] )> > ] ]
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]);
[ <mapping: <1-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    2-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> >,
  <mapping: <4-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    2-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> > ]
gap> f:= CoKernelProjection(pushout[1]);
<mapping: <2-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> -> <1-dimensional right-module over AlgebraWithOne( Rationals,
[ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d], [(1)*e]
 ] )> >
```

```
gap> U := Range(pushout[1]);
<2-dimensional right-module over <algebra-with-one over Rationals, with
8 generators>>
gap> assU := AlmostSplitSequence(U);
[ <mapping: <6-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    8-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> >,
  <mapping: <8-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> -> <
    2-dimensional right-module over AlgebraWithOne( Rationals,
    [ [(1)*v1], [(1)*v2], [(1)*v3], [(1)*a], [(1)*b], [(1)*c], [(1)*d],
      [(1)*e] ] )> > ]
gap> assU[1]*assU[2] = Zero(assU[1]*assU[2]);
true
```

Chapter 8

Chain complexes

(Not completely documentet yet.)

8.1 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.

8.1.1 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.

8.1.2 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
```

8.2 Making a complex

The most general constructor for complexes is the function `Complex` (8.2.3). In addition to this, there are constructors for common special cases:

- `ZeroComplex` (8.2.4)
- `StalkComplex` (8.2.6)
- `FiniteComplex` (8.2.5)
- `ShortExactSequence` (8.2.7)

8.2.1 IsComplex

◊ `IsComplex` (Category)

The category for chain complexes.

8.2.2 IsZeroComplex

◊ `IsZeroComplex` (Category)

Category for zero complexes, subcategory of `IsComplex` (8.2.1).

8.2.3 Complex

◊ `Complex(`*`cat, baseDegree, middle, positive, negative`*`)` (function)

Returns: A newly created chain complex

The first argument, *cat* is an `IsCat` (8.1.1) 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 $\mathtt{baseDegree} + 1$, and so on. Thus, the middle part consists of the degrees

$$\mathtt{baseDegree}, \quad \mathtt{baseDegree} + 1, \quad \ldots \quad \mathtt{baseDegree} + \mathrm{Length}(\mathtt{middle}).$$

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.

8.2.4 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` (8.1.1) object *cat*.

8.2.5 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` (8.1.1) 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 $baseDegree + 1$, and so on.

This means that the *differentials* argument specifies the differentials in degrees

$$baseDegree, \quad baseDegree+1, \quad \ldots \quad baseDegree+\mathrm{Length}(differentials);$$

and thus implicitly the objects in degrees

$$baseDegree-1, \quad baseDegree, \quad \ldots \quad baseDegree+\mathrm{Length}(differentials).$$

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
```

8.2.6 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
```

8.2.7 ShortExactSequence

◊ `ShortExactSequence(cat, f, g)` (function)

Arguments: `cat` – a category, `f` and `g` – maps in `cat`, where `f`: $A \to B$ and `g`: $B \to C$.
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
```

8.3 Information about a complex

8.3.1 CatOfComplex

◊ `CatOfComplex(C)` (attribute)

Returns: The category the objects of the complex `C` live in.

8.3.2 ObjectOfComplex

◊ `ObjectOfComplex(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: The object at position `i` in the complex.

8.3.3 DifferentialOfComplex

◊ `DifferentialOfComplex(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: The map in `C` between objects at positions i and $i-1$.

8.3.4 DifferentialsOfComplex

◊ `DifferentialsOfComplex(C)` (attribute)

Arguments: `C` – a complex
Returns: The differentials of the complex, stored as an `IsInfList` object.

8.3.5 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)`.

8.3.6 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)`.

8.3.7 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>>
```

8.3.8 IsFiniteComplex

◊ `IsFiniteComplex(C)` (operation)

Arguments: `C` – a complex.
Returns: true if `C` is a finite complex, false otherwise.

8.3.9 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.

8.3.10 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.

8.3.11 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.

8.3.12 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.

8.3.13 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
```

8.3.14 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.

8.3.15 IsExactInDegree

◊ `IsExactInDegree(C, i)` (operation)

Arguments: `C` – a complex, `i` – an integer.
Returns: true if `C` is exact at position `i`.

8.3.16 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
```

8.4 Transforming and combining complexes

8.4.1 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 ] ] ]
```

8.4.2 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
```

8.4.3 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.

8.4.4 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.

8.4.5 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`.

8.4.6 CutComplexAbove

◊ `CutComplexAbove(C)` (operation)

Arguments: `C` – a bounded complex with possibly nonzero *positive* part, but zero *negative* part.
Returns: A newly created complex.
If `C` is known to be a bounded complex (e.g. a projective resolution in a module category of finite global dimension), one can use this method to get a new representation of `C` with `UpperBound` $< \infty$.

Example
```
gap> C;
--- -> 0: (1,1,0) -> 0
gap> D := CutComplexAbove(C);
0 -> 2:(0,0,1) -> 1:(0,1,1) -> 0:(1,1,0) -> 0
gap> UpperBound(C);
fail
gap> UpperBound(D);
2 gap> ObjectOfComplex(C, 2);
<[ 0, 0, 1 ]>
```

8.4.7 CutComplexBelow

◊ `CutComplexBelow(C)` (operation)

Arguments: `C` – a bounded complex with possibly nonzero *negative* part, but zero *positive* part.
Returns: A newly created complex.

If C is known to be a bounded complex (e.g. a injective resolution in a module category of finite global dimension), one can use this method to get a new representation of C with `LowerBound` $< \infty$.

8.5 Chain maps

An `IsChainMap` (8.5.1) object represents a chain map between two complexes over the same category.

8.5.1 IsChainMap

◊ `IsChainMap` (Category)

The category for chain maps.

8.5.2 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` (8.2.3) 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 $\mathit{baseDegree}+1$, 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.

8.5.3 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`*.

(TODO: this function is not implemented.)

8.5.4 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.

(TODO: this function is not implemented.)

8.5.5 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` (8.2.3) and `ChainMap` (8.5.2) 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

$$L_i = [d_i, m_i^1, \dots, m_i^S, n_i^1, \dots, n_i^R],$$

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

$$\mathit{baseDegree}, \quad \mathit{baseDegree}+1, \quad \dots \quad \mathit{baseDegree}+\mathrm{Length}(\mathit{middle})-1.$$

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.

The return value of the `ComplexAndChainMaps` constructor is a list

$$[C, M_1, \ldots, M_S, N_1, \ldots, N_R],$$

where C is the new complex, $M_1, \ldots, M_S$ are the new chain maps with C as range, and $N_1, \ldots, N_R$ are the new chain maps with C as source.

8.5.6 MorphismOfChainMap

◊ `MorphismOfChainMap(M, i)` (operation)

Arguments: *M* – a chain map, *i* – an integer.
Returns: The morphism at position *i* in the chain map.

8.5.7 MorphismsOfChainMap

◊ `MorphismsOfChainMap(M)` (attribute)

Arguments: *M* – a chain map
Returns: The morphisms of the chain map, stored as an `IsInfList` (**??**) object.

Chapter 9

Projective resolutions and the bounded derived category

9.1 Projective and injective complexes

Not finished yet! Some bugs and stuff to be fixed, both in the documentation and in the code ...

9.1.1 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` (9.2.1)), 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.

9.1.2 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 $D\mathrm{Hom}_A(-,A)$ of a projective complex (`ProjectiveToInjectiveComplex` (9.2.2)), 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.

9.1.3 ProjectiveResolution

◊ `ProjectiveResolution(M)` (operation)

Arguments: `M` – a module.
Returns: The projective resolution of `M` with `M` in degree -1.

9.2 The bounded derived category

Let $\mathcal{D}^b(\text{mod}A)$ denote the bounded derived category. If C is an element of $\mathcal{D}^b(\text{mod}A)$, 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 $\mathcal{D}^b(\text{mod}A)$ has AR-triangles, and there exists an algorithm for computing the AR-translation of a complex $C \in \mathcal{D}^b(\text{mod}A)$:

- Compute a projective resolution P' of C.
- Shift P' one degree to the right.
- Compute $I = D\text{Hom}_A(P', A)$ 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.

9.2.1 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.

9.2.2 ProjectiveToInjectiveComplex

◊ `ProjectiveToInjectiveComplex(P)` (operation)

Arguments: `P` – a bounded below projective complex.
Returns: An injective complex $I = D\text{Hom}_A(P, A)$.
P and I will always have the same length. Especially, if P is unbounded above, then so is I.

9.2.3 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.

9.2.4 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
```

Appendix A

An Appendix

This is an appendix.

References

Index