QPA

Quivers and Path Algebras

Version version 1.4

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

Quick Start

This chapter is intended for those who would like to get started with QPA right away by playing with a few examples. A simple example is presented first:

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:

Example

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

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

Example

```
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:

Example

```
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:

Example

```
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:

Example

```
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[, arrow1, arrow2, ...])` (function)
◊ `Quiver([vertex1, vertex2, ..., ][arrow1, arrow2, ...])` (function)
◊ `Quiver(adjacencymatrix)` (function)

Returns: a quiver, which satisfies the property `IsQuiver` (2.3.1).

The first construction takes the number *N* of vertices and a list of arrows to specify the graph Γ and hence the quiver. The second construction takes a list of vertex names and a list of arrows for the same purpose. The third construction takes an adjacency matrix for the graph Γ.

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)

is true when *object* is a quiver.

2.3.2 IsAcyclicQuiver

◊ IsAcyclicQuiver(*object*) (property)

is true when *object* is a quiver with no cycles.

2.3.3 IsFinite

◊ IsFinite(*object*) (property)

is true when *object* is a finite set. Synonymous with 'IsAcyclicQuiver'.

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)

The operation . operates on *Q*, a quiver, and an element, a vertex or an arrow, to allow access to generators of the quiver. If you have named your vertices and arrows the 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(*object*) (attribute)

An attribute. Returns a list of paths that are vertices in *object*.

2.5.3 ArrowsOfQuiver

◊ ArrowsOfQuiver(*object*) (attribute)

An attribute. Returns a list of paths that are arrows in *object*.

2.5.4 AdjacencyMatrixOfQuiver

◊ AdjacencyMatrixOfQuiver(*object*) (attribute)

An attribute. Returns the adjacency matrix of *object*.

2.5.5 GeneratorsOfQuiver

◊ GeneratorsOfQuiver(*object*) (attribute)

An attribute. Returns a list of the vertices and the arrows in *object*.

2.5.6 NumberOfVertices

◊ NumberOfVertices(*object*) (attribute)

An attribute. Returns the number of vertices in *object*.

2.5.7 NumberOfArrows

◊ NumberOfArrows(*object*) (attribute)

An attribute. Returns the number of arrows in *object*.

2.5.8 OrderingOfQuiver

◊ OrderingOfQuiver(*object*) (attribute)

An attribute. Returns the ordering used to order elements in *object*. See Section 2.4 for more information.

2.5.9 OppositeOfQuiver

◊ OppositeOfQuiver(*Q*) (operation)

This takes the quiver *Q* and produces the opposite 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(*object*) (attribute)

An attribute. Returns the source (first) vertex of *object*.

2.7.2 TargetOfPath

◊ TargetOfPath(*object*) (attribute)

An attribute. Returns the target (last) vertex of *object*.

2.7.3 LengthOfPath

◊ LengthOfPath(*object*) (attribute)

An attribute. Returns the length of *object*.

2.7.4 WalkOfPath

◊ WalkOfPath(*object*) (attribute)

An attribute. Returns a list of the arrows that constitue *object* in order.

2.7.5 *

◊ *(*p*, *q*) (operation)

The operation * operates on *p* and *q*, which are two paths in the same quiver. It 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)

The operation = operates on two paths *p* and *q*, and compares the paths for equality. Two paths are equal if they have the same source and the same target and if they have the same walks. The paths *p* and *q* must be in the same quiver.

2.7.7 <

◊ <(*p*, *q*) (operation)

The operation < operates on two paths *p* and *q*, and compares them with respect to the ordering of the quiver. This is meaningful only if *p* and *q* are in the same 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(*object*) (attribute)

An attribute. Returns a list of arrows having *object* as target. Only meaningful if *object* is in a quiver.

2.8.2 OutgoingArrowsOfVertex

◊ OutgoingArrowsOfVertex(*object*) (attribute)

An attribute. Returns a list of arrows having *object* as source.

2.8.3 InDegreeOfVertex

◊ InDegreeOfVertex(*object*) (attribute)

An attribute. Returns the number of arrows having *object* as target. Only meaningful if *object* is in a quiver.

2.8.4 OutDegreeOfVertex

◊ OutDegreeOfVertex(*object*) (attribute)

An attribute. Returns the number of arrows having *object* as source.

2.8.5 NeighborsOfVertex

◊ NeighborsOfVertex(*object*) (attribute)

An attribute. Returns a list of neighbors of *object*, that is, vertices that are targets of arrows having *object* 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 and a quiver Q. The path algebra FQ contains all finite linear combinations of elements 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 "ref:algebras" for functionality such as generators, basis functions, and mappings.

3.1.1 InfoPathAlgebra

◊ InfoPathAlgebra (info class)

is the info class for functions dealing with path algebras.

3.2 Constructing Path Algebras

3.2.1 PathAlgebra

◊ PathAlgebra(*F, Q*) (function)

Returns: the path algebra FG of Q over the field F.

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 := FiniteField(23);
GF(23)
gap> fq := PathAlgebra(f,q);
<algebra-with-one over GF(23), with 6 generators>
```

* NO: the FOLLOWING is not true:

3.2.2 OrderedBy

◊ OrderedBy(*path, algebra, ordering*) (function)

Returns: a copy of *path algebra* whose elements are ordered by *ordering*. See Section 2.4 for more information on orderings.

3.3 Categories and Properties of Path Algebras

3.3.1 IsPathAlgebra

◊ IsPathAlgebra(*object*) (property)

is true when *object* is a path algebra.

Example

```
gap> IsPathAlgebra(fq);
true
gap> IsPathAlgebra(q);
false
```

3.4 Attributes and Operations for Path Algebras

3.4.1 QuiverOfPathAlgebra

◊ QuiverOfPathAlgebra(*object*) (attribute)

An attribute. Returns the quiver from which *object* was constructed.

Example

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

3.4.2 .

◊ .(*FQ, element*) (operation)

The operation . operates on a path algebra *FQ* and an element *element*, which is a vertex or an arrow in the quiver Q. It returns the generator as an element of the path algebra.

Example

```
gap> fq.v;
(Z(23)^0)*v
gap> fq.b;
(Z(23)^0)*b
```

>OrderingOfAlgebra(*path algebra*) returns the ordering of the quiver of the path algebra.

3.5 Operations on Path Algebra Elements

3.5.1 <

◊ <(*a, b*) (operation)

The operation < operates on elements *a* and *b* of a path algebra FQ, and they are compared using the ordering for the path algebra. See Section 2.4 for more information on orderings.

3.5.2 IsLeftUniform

◊ IsLeftUniform(*element*) (operation)

The operation IsLeftUniform operates on an *element* in a path algebra, and it returns true if each monomial in *element* has the same source vertex.

3.5.3 IsRightUniform

◊ IsRightUniform(*element*) (operation)

The operation IsRightUniform operates on an *element* in a path algebra, and it returns true if each monomial in *element* has the same target vertex.

3.5.4 IsUniform

◊ IsUniform(*element*) (operation)

The operation IsUniform operates on an *element* in a path algebra, and it returns true if each monomial in *element* has both the same source vertex and the same target vertex.

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;
(Z(23)^0)*a*b+(Z(23)^0)*b*d*c*b*d
gap> IsLeftUniform(another);
true
gap> IsRightUniform(another);
true
gap> IsUniform(another);
true
```

3.5.5 LeadingTerm

◊ LeadingTerm(*element*) (operation)
◊ Tip(*element*) (operation)

The operation LeadingTerm or equivalently Tip operates on an *x* in a path algebra, and it returns is the term in *element* whose monomial is largest among those monomials that have nonzero coefficients; this term is known as the *tip* of *element*.

3.5.6 LeadingCoefficient

◊ LeadingCoefficient(*element*) (operation)
◊ TipCoefficient(*element*) (operation)

The operation LeadingCoefficient or equivalently TipCoefficient operates on an *element* in a path algebra, and it returns the coefficient of the tip of *element*. This is an element of the field.

3.5.7 LeadingMonomial

◊ LeadingMonomial(*element*) (operation)
◊ TipMonomial(*element*) (operation)

The operation LeadingMonomial or equivalently TipMonomial operates on an *element* in a path algebra, and it returns the monomial of the tip of *element*; it is the largest monomial occurring in *element* with a nonzero coefficient. This is an element of the underlying quiver, not of the path algebra.

Example

```
gap> elem := fq.a*fq.b*fq.c + fq.b*fq.d*fq.c+fq.d*fq.d;
(Z(23)^0)*d^2+(Z(23)^0)*a*b*c+(Z(23)^0)*b*d*c
gap> LeadingTerm(elem);
(Z(23)^0)*b*d*c
gap> LeadingCoefficient(elem);
Z(23)^0
gap> LeadingMonomial(elem);
b*d*c
```

3.5.8 MakeUniformOnRight

◊ MakeUniformOnRight(*elems*) (operation)

The operation MakeUniformOnRight operates on a list *elems* of elements in a path algebra, and it returns a list of right uniform elements generated by each element of *elems*.

3.5.9 MappedExpression

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

The operation MappedExpression operates on *expr* from a path algebra and and two equal-length lists of generators *gens1* and *gens2* for subalgebras. The *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*.

3.5.10 VertexPosition

◊ VertexPosition(*elm*) (function)

Returns: the position of the vertex *v* in the list of vertices of which the element *elm* is a multiplum of.

The function assumes that a multiplum of a trivial path is entered.

3.6 Constructing Quotients of Path Algebras

See Chapter ”ref:algebras” 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 same commands may be used with quotients. Also since a quotient is still an algebra, refer to ”ref:algebras”.

Example

```
gap> I := Ideal(fq,[fq.a * fq.b, fq.d * fq.d - fq.b * fq.c]);
<two-sided ideal in <algebra-with-one over GF(23), with 6 generators>,
  (2 generators)>
gap> GeneratorsOfIdeal(I);
[ (Z(23)^0)*a*b, (Z(23)^11)*b*c+(Z(23)^0)*d^2 ]
gap> quot := fq/I;
<algebra-with-one over GF(23), with 6 generators>
```

3.7 Ideals

3.7.1 Ideals and operations on ideals

3.7.2 Ideal

◊ Ideal(*KQ*, *elems*) (operation)

The operation Ideal defines the ideal generated by *elems* in the path algebra *KQ* (See Ideals in the reference manual of GAP).

3.7.3 NthPowerOfArrowIdeal

◊ NthPowerOfArrowIdeal(*KQ*, *n*) (operation)

The operation NthPowerOfArrowIdeal defines the ideal generated all the paths of length *n* in the path algebra *KQ*.

3.7.4 AddNthPowerToRelations

◊ AddNthPowerToRelations(*KQ*, *rels*, *n*) (operation)

The operation AddNthPowerToRelations append the list *rels* the paths of length *n* in the path algebra *KQ*. The object *rels* must be a list of elements in the path algebra *KQ*, and *n* must be a positive integer.

3.7.5 Attributes of ideals

Groebner Basis Of an Ideal: For many of the functions related to quotients, you will need to compute a Groebner basis of the ideal. Refer to the chapters ”qpa:groebner basis” and ”qpa:using opal with gap” to learn more.

Example

```
gap> J := Ideal(fq,[fq.a*fq.b]);
<two-sided ideal in <algebra-with-one over GF(23), with 6 generators>,
  (1 generators)>
```

```
gap> anotherquot := fq/J;
<algebra-with-one over GF(23), with 6 generators>
gap> gb := GroebnerBasis(J,[fq.a*fq.b]);
<complete two-sided Groebner basis containing 1 elements>
gap> SetIsCompleteGroebnerBasis(gb, true);
gap> IsCompleteGroebnerBasis(gb);
true
gap> gb = GroebnerBasisOfIdeal(J);
true
```

3.8 Categories and Properties of Quotients of Path Algebras

3.8.1 IsQuotientOfPathAlgebra

◊ IsQuotientOfPathAlgebra(*object*) (property)

is true when *object* is a quotient of a path algebra.

Example

```
gap> IsQuotientOfPathAlgebra(quot);
true
gap> IsQuotientOfPathAlgebra(fq);
false
```

3.9 Attributes and Operations for Quotients of Path Algebras

3.9.1 NormalFormFunction

◊ NormalFormFunction(*object*) (attribute)

is a function that can compute normal forms for elements of *object*. It may be supplied by the user.

3.9.2 IsElementOfQuotientOfPathAlgebra

◊ IsElementOfQuotientOfPathAlgebra(*object*) (property)

is true if *object* is an element of some quotient of a path algebra.

Example

```
gap> this := anotherquot.a*anotherquot.b;
[(Z(23)^0)*a*b]
gap> IsElementOfQuotientOfPathAlgebra(this);
true
gap> IsElementOfQuotientOfPathAlgebra(fq);
false
```

3.9.3 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. Question: Does this only take one argument?

3.9.4 IsSelfinjectiveAlgebra

◊ IsSelfinjectiveAlgebra(*A*) (operation)

This function takes a path algebra or a quotient of a path algebra as an argument *A*, and returns fail if *A* is not finite dimensional. Otherwise it returns true or false according to whether *A* is selfinjective or not.

3.9.5 LoewyLength

◊ LoewyLength(*A*) (operation)

This function takes a pathalgebra or a quotient of a path algebra as an argument *A*, and returns fail if *A* is not finite dimensional. Otherwise it returns the Loewy length of the algebra *A*.

3.9.6 CartanMatrix

◊ CartanMatrix(*A*) (operation)

This function returns the Cartan matrix of the algebra *A*, after having checked that *A* is a finite dimensional quotient of a path algebra.

3.9.7 CoxeterMatrix

◊ CoxeterMatrix(*A*) (operation)

This function returns the Coxeter matrix of the algebra *A*, after having checked that *A* is a finite dimensional quotient of a path algebra.

3.9.8 CoxeterPolynomial

◊ CoxeterPolynomial(*A*) (operation)

This function returns the Coxeter polynomial of the algebra *A*, after having checked that *A* is a finite dimensional quotient of a path algebra.

3.9.9 Centre/Center

◊ Centre/Center(*A*) (operation)

This function 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).

3.10 Attributes and Operations on Elements of Quotients of Path Algebra

3.10.1 IsNormalForm

◊ IsNormalForm(*element*) (operation)

The operation IsNormalForm operates on an *element* from a path algebra, and it is true if *element* is known to be in normal form.

Example

```
gap> IsNormalForm(this);
false
```

3.10.2 <

◊ <(*a*, *b*) (operation)

The operation < operates on two elements *a* and *b* from a path algebra FQ, and it compares them using the ordering for the path algebra.

3.10.3 ElementOfQuotientOfPathAlgebra

◊ ElementOfQuotientOfPathAlgebra(*family*, *element*, *computenormal*) (operation)

The operation ElementOfQuotientOfPathAlgebra operates on an *element* in a path algebra, and it projects it into the quotient given by *family*. If *computenormal* is true, then the normal form of the projection of *element* is returned.

3.10.4 OriginalPathAlgebra

◊ OriginalPathAlgebra(*algebra*) (operation)

The operation OriginalPathAlgebra operates on an *algebra*. If it is a quotient of a path algebra or just a path algebra itself, it returns 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.

3.11 Predefined classes of quotient of path algebras

3.11.1 NakayamaAlgebra

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

Returns: the Nakayama algebra corresponding to the addmissible sequence *admiss-seq* over the field *field*, or the admissible sequence entered if entered sequence is not an admissible sequence.

This function creates a Nakayama algebra from an admissible sequence over a field.

3.11.2 TruncatedPathAlgebra

◊ TruncatedPathAlgebra(K, Q, n) (operation)

Returns: the truncated path algebra KQ/I, where I is the ideal generated by all paths of length n in KQ. The object K must be a field, Q a quiver and n a positive integer.

3.12 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 (3.12.1) of the quivers of the original algebras.

The operation TensorProductOfAlgebras (3.12.6) computes the tensor products of two quotients of path algebras as a quotient of a path algebra.

3.12.1 QuiverProduct

◊ QuiverProduct(Q1, Q2) (operation)

Creates the product quiver Q1 × Q2. A vertex in Q1 × 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).

3.12.2 QuiverProductDecomposition

◊ QuiverProductDecomposition(Q) (attribute)

Contains the original quivers Q is a product of, if Q was created by the QuiverProduct (3.12.1) operation. The value of this attribute is an object in the category IsQuiverProductDecomposition (3.12.3).

3.12.3 IsQuiverProductDecomposition

◊ IsQuiverProductDecomposition (filter)

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 (3.12.4) and ProjectFromProductQuiver (3.12.5).

3.12.4 IncludeInProductQuiver

◊ IncludeInProductQuiver(L, Q) (operation)

Includes paths q_1 and q_2 from two quivers into the product of these quivers. 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.

The argument `L` is a list containing the paths q_1 and q_2 to be included; Q is the product quiver to include them in.

3.12.5 ProjectFromProductQuiver

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

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

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

Example

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

3.12.6 TensorProductOfAlgebras

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

The operation `TensorProductOfAlgebras` produces the tensor product of two (quotients of) path algebras `FQ1` and `FQ2`. The result is a quotient of a path algebra, whose quiver is the `QuiverProduct` (3.12.1) of the quivers of the operands.

3.12.7 SimpleTensor

◊ `SimpleTensor(L, T)` (operation)

The operation `SimpleTensor` produces a simple tensor $x \otimes y$ in the tensor product of two path algebras. The argument `L` is a list containing the elements x and y. These should be elements of two (quotients of) path algebras, and `T` the tensor product of these algebras (produced by `TensorProductOfAlgebras` (3.12.6)).

3.12.8 TensorProductDecomposition

◊ TensorProductDecomposition(*T*) (attribute)

For a tensor product of quotients of path algebras (produced by TensorProductOfAlgebras (3.12.6)), this attribute contains a list of the factors in the tensor product.

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
```

3.12.9 EnvelopingAlgebra

◊ EnvelopingAlgebra(*FQ*) (operation)

Produces the enveloping algebra $FQ^{\mathrm{e}} = FQ \otimes FQ^{\mathrm{op}}$ of FQ, which should be (a quotient of) a path algebra.

3.12.10 IsEnvelopingAlgebra

◊ IsEnvelopingAlgebra(*A*) (attribute)

True if and only if A is the result of a call to EnvelopingAlgebra (3.12.9).

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 expected to be computed by external packages such as 'Opal' and 'Groebner'.

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)

Returns: an object in the 'IsGroebnerBasis' category.

The 'GroebnerBasis' global function takes an algebra A and a list of relations $rels \in A$ and creates an object in the 'IsGroebnerBasis' category. There are absolutely 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.

4.2 Categories and Properties of Groebner Basis

4.2.1 IsGroebnerBasis

◊ IsGroebnerBasis(*object*) (property)

is true when *object* is a Groebner basis and 'false' otherwise.

Example

```
gap>
```

4.2.2 IsTipReducedGroebnerBasis

◊ IsTipReducedGroebnerBasis(*object*) (property)

is true when *object* is a Groebner basis which is tip reduced.

Example

```
gap>
```

4.2.3 IsCompletelyReducedGroebnerBasis

◊ IsCompletelyReducedGroebnerBasis(*object*) (property)

is true when *object* is a Groebner basis which is completely reduced.

Example

```
gap>
```

4.2.4 IsHomogeneousGroebnerBasis

◊ IsHomogeneousGroebnerBasis(*object*) (property)

is true when *object* is a Groebnerbasis which is homogenous.

Example

```
gap>
```

4.2.5 IsCompleteGroebnerBasis

◊ IsCompleteGroebnerBasis(*object*) (property)

is true when *object* 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 basis is 'Opal'.

Example

```
gap>
```

4.3 Attributes and Operations for Groebner Basis

4.3.1 CompletelyReduce

◊ CompletelyReduce(*GB*, *a*) (operation)

The operation CompletelyReduce operates on an element *a* in a path algebra, and it reduces *a* by the Groebner basis *GB*. The reduced element is returned. If *a* is already completely reduced, the original element *a* is returned.

Example

```
gap>
```

4.3.2 CompletelyReduceGroebnerBasis

◊ CompletelyReduceGroebnerBasis(*GB*) (operation)

The operation CompletelyReduceGroebnerBasis operates on a Groebner basis *GB*, and it 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*.

Example

```
gap>
```

4.3.3 TipReduce

◊ TipReduce(*GB*, *a*) (operation)

The operation TipReduce operates on an element *a* in a path algebra, and it tip reduced by the Groebner basis *GB*. If *a* is already tip reduced, then the original *a* is returned.

Example

```
gap>
```

4.3.4 TipReduceGroebnerBasis

◊ TipReduceGroebnerBasis(*GB*) (operation)

The operation TipReduceGroebnerBasis operates on Groebner basis *GB*, and it returns an equivalent Groebner basis to *GB* such that each relation generating *GB* is tip reduced. If *GB* is already tip reduced, this function returns the original object *GB*, possibly with the addition of the 'IsTipReduced'' property set.

Example

```
gap>
```

4.3.5 Iterator

◊ Iterator(*GB*) (operation)

The operation Iterator operates on a Groebner basis *GB*, and it 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.

Example

```
gap>
```

4.3.6 Enumerator

◊ Enumerator(*GB*) (operation)

The operation Enumerator operates on a Groebner basis *GB*, and it creates an enumerator that enumerates the relations making up the Groebner basis. These relations should are enumerated in ascending order with respect to the ordering for the family the elements are contained in.

Example

```
gap>
```

4.3.7 Nontips

◊ Nontips(*GB*) (operation)

The operation Nontips operates on a Groebner basis *GB*, and it returns a list of nontip elements for a Groebner basis. 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'.

Example

```
gap>
```

4.3.8 AdmitsFinitelyManyNontips

◊ AdmitsFinitelyManyNontips(*GB*) (operation)

The operation AdmitsFinitelyManyNontips operates on a Groebner basis *GB*, and it returns 'true' if the Groebner basis admits only finitely many nontips and 'false' otherwise. This operation only applies to complete Groebner basis.

Example

```
gap>
```

4.3.9 NontipSize

◊ NontipSize(*GB*) (operation)

The operation NontipSize operates on a Groebner basis *GB*, and it returns the number of nontips admitted by the Groebner basis *GB*. This operation is available only to complete basis.

Example

```
gap>
```

4.3.10 IsPrefixOfTipInTipIdeal

◊ IsPrefixOfTipInTipIdeal(*GB*, *R*) (operation)

The operation IsPrefixOfTipInTipIdeal operates on a Groebner basis *GB* and a relation *R*, and it checks the tip of a relation *R* to see if it 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.

Example

```
gap>
```

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)

is true when *object* a right Groebner basis.

Example

```
gap>
```

4.4.2 RightGroebnerBasisOfIdeal

◊ RightGroebnerBasisOfIdeal(*object*) (attribute)

An attribute. Stores a right Groebner basis of a right ideal, *object*, is one has been computed.

4.4.3 RightGroebnerBasis

◊ RightGroebnerBasis(*R*) (operation)

The operation RightGroebnerBasis operates on a right ideal *R* in a path algebra, and it constructs a right Groebner basis for the right ideal, *R*, which must support a right Groebner basis theory. Right now, this requires that R have 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)

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

In the first function call, `A` is a (quotient of a) path algebra. 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 first argument *A* is a (quotient of a) path algebra, 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)

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)

The operation ^ operates on an element m in a module and a path p in a path algebra, and it 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)

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

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

6.4.2 DimensionVector

◊ DimensionVector(*M*) (operation)

Returns: the dimension vector of the representation M. The argument M must be a PathAlgebraMatModule.

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

6.4.3 Dimension

◊ Dimension(*M*) (operation)

Returns: the dimension of the representation M. The argument M must be a PathAlgebraMatModule.

6.4.4 IsDirectSummand

◊ IsDirectSummand(*M, N*) (operation)

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)

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

The argument L must be a list of PathAlgebraMatModule's over the same (quotient of a) path algebra. In addition three attributes are attached to the result, IsDirectSumOfModules, DirectSumProjections and DirectSumInclusions.

6.4.6 IsDirectSumOfModules

◊ IsDirectSumOfModules(*M*) (attribute)

An attribute, returns true is M is constructed via the command DirectSumOfModules.

6.4.7 DirectSumInclusions

◊ DirectSumInclusions(*M*) (attribute)

An attribute, 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)

An attribute, 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)

Returns: the first syzygy of the representation *M* as a representation. The argument *M* must be a PathAlgebraMatModule.

6.4.10 IsInAdditiveClosure

◊ IsInAdditiveClosure(*M*, *N*) (operation)

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)

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. The argument *M* must be a PathAlgebraMatModule, and *n* must be a positive integer.

6.4.12 IsInjectiveModule

◊ IsInjectiveModule(*M*) (operation)

Returns: true if the representation *M* is injective. The argument *M* must be a PathAlgebraMatModule.

6.4.13 IsProjectiveModule

◊ IsProjectiveModule(*M*) (operation)

Returns: true if the representation *M* is projective. The argument *M* must be a PathAlgebraMatModule.

6.4.14 IsSemisimpleModule

◊ IsSemisimpleModule(*M*) (operation)

Returns: true if the representation *M* is semisimple. The argument *M* must be a PathAlgebraMatModule.

6.4.15 IsSimpleModule

◊ IsSimpleModule(*M*) (operation)

Returns: true if the representation *M* is simple. The argument *M* must be a PathAlgebraMatModule.

6.4.16 LoewyLength

◊ LoewyLength(*M*) (operation)

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)

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)

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)

Returns: the top of the syzygies until a syzygy is projective or the *n*-th syzygy has been computed. The argument *M* must be a PathAlgebraMatModule, and the argument *n* must be a positive integer.

6.4.20 NthSyzygyNC

◊ NthSyzygyNC(*M*, *n*) (operation)

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. The argument *M* must be a PathAlgebraMatModule, and the argument *n* must be a positive integer.

6.4.21 RadicalOfModule

◊ RadicalOfModule(*M*) (operation)

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)

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)

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)

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)

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)

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)

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)

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)

Returns: a list of the matrices that defines the representation *M* as a right module of the acting path algebra. The argument *M* must be a PathAlgebraMatModule.

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)

Returns: a list of bases for all the indecomposable projective representations over a finite dimensional (quotient of a) path algebra `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)

Returns: a list of all the indecomposable projective representations over a finite dimensional (quotient of a) path algebra `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)

Returns: a list of all the indecomposable injective representations over a finite dimensional (quotient of a) path algebra `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)

Returns: a list of all the simple representations over a finite dimensional (quotient of a) path algebra `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)

Returns: the zero representation over a finite dimensional (quotient of a) path algebra `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)

Takes the a representation *M* of a path algebra KQ and produces the dual of this representation over the opposite path algebra *KQ_op*.

6.6.2 DualOfModuleHomomorphism

◊ DualOfModuleHomomorphism(*f*) (operation)

Takes the a map *f* between two representations *M* and *N* over a path algebra *A* and produces the dual of this map over the opposite path algebra *A^op*.

6.6.3 DTr

◊ DTr(*M*[, *n*]) (operation)
◊ DualOfTranspose(*M*[, *n*]) (operation)

Returns: the dual of the transpose of the module *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.

The argument *M* must be a PathAlgebraMatModule and *n* must be an integer.

6.6.4 TrD

◊ TrD(*M*[, *n*]) (operation)
◊ TransposeOfDual(*M*[, *n*]) (operation)

Returns: the transpose of the dual of the module *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.

The argument *M* must be a PathAlgebraMatModule and *n* must be an integer.

6.6.5 TransposeOfModule

◊ TransposeOfModule(*M*) (operation)

Returns: the transpose of the module *M*.

The argument *M* must be a PathAlgebraMatModule.

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)

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 the list of vertices *verts*.

In this implemenation 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)

Returns: a vector in the module *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)

The operation ^ operates on an element *m* in a module and an element *a* in a path algebra, and it 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)

The operation < operates on elements *m1* and *m2* in ????, and it compares them. The result is '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)

The operation / operates on two finite dimensional modules *M* and *N* over a path algebra?, and it constructs 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 $f: L \to M$ and $g: M \to N$, 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 $g_i: M[i] \to N[i]$ 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 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 if 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 (7.2.1) and by Source (7.2.2), 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 Range

◇ Range(*f*) (operation)

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

Returns: the range (or codomain) the homomorphism *f*.

7.2.2 Source

◊ Source(*f*) (operation)

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

Returns: the source (or domain) the homomorphism *f*.

Example

```
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
```

7.2.3 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 functions checks if *elem* is an element in the range of *f*.

7.2.4 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 functions checks if *elem* is an element in the source of *f*, and it returns an error message otherwise.

7.2.5 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 functions 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 ] ] ]
```

7.2.6 Zero

◊ `Zero(f)` (operation)

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

7.2.7 ZeroMapping

◊ `ZeroMapping(M, N)` (operation)

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

7.2.8 IdentityMapping

◊ `IdentityMapping(M)` (operation)

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

7.2.9 \= (maps)

◊ `\= (maps)(f, g)` (operation)

Arguments: *f*, *g* - two homomorphisms between two modules.
Returns: true, if `Source(f)=Source(g)`, `Range(f)=Range(g)`, the matrices defining the maps *f* and *g* coincide.

7.2.10 \+ (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 functions checks if the maps have the same source and the same range, and returns an error message otherwise.

7.2.11 * (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 functions 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]
 ] )> >
```

7.2.12 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*.

7.2.13 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*.

7.2.14 IsInjective

◊ IsInjective(*f*) (attribute)

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

7.2.15 IsSurjective

◊ IsSurjective(*f*) (attribute)

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

7.2.16 IsIsomorphism

◊ IsIsomorphism(*f*) (attribute)

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
```

7.2.17 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*.

7.2.18 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]
  ] )> >
```

7.2.19 IsZero

◊ IsZero(*f*) (property)

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

Returns: true if the homomorphism *f* is a zero homomorphism.

7.2.20 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*.

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 representations of a quiver.

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: - a module.

Returns: the endomorphism ring of *M* as a subalgebra of the direct sum of the full matrix rings of *dim_M[i] x dim_M[i]*, where *i* runs over all vertices.

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 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 representation of the quiver. 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.

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 representation of the quiver, while the latter one returns the inclusion homomorphism of the kernel into the source of the homomorphism *f*.

7.3.7 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, Source(f') and the modules in the list B is isomorphic to Source(f).

7.3.8 LiftingInclusionMorphisms

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

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

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

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

7.3.9 LiftingMorphismFromProjective

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

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

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

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

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, beta, alpha*) (operation)
◊ MorphismOnImage(*f, g, beta, alpha*) (operation)
◊ MorphismOnCoKernel(*f, g, beta, alpha*) (operation)

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

Returns: the morphism induced on the kernels, the images or the cokernels of the morphisms *f* and *g*, respectively.

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

7.3.14 ProjectiveCover

◊ ProjectiveCover(*M*) (operation)

Arguments: *M* - a module.

Returns: the projective cover of `M`, that is, returns the map `P(M) --> 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 between modules with a common range.

Returns: the pullback of the maps `f` and `g`.

It is checked if `f` and `g` has the same range. The pullback is returned as two homomorphisms from the pullback to the source of `f` and to the source of `g`.

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` has the same source. The pushout is returned as two homomorphisms from the range of `f` and from the range of `g` to the pushout.

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` - elements in `M`, .

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

The function checks if `gens` are 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.

7.4 Homological algebra

7.4.1 ExtOverAlgebra

◊ ExtOverAlgebra(*M, N*) (operation)

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

Returns: the map from the first syzygy, *Omega(M)* to the projective cover, *P(M)* of the module *M*, and in addition a basis of $Ext^1(M,N)$ in terms of elements in $Hom(Omega(M),N)$.

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

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 equal to the module *M* one started with, but isomorphic.

Chapter 8

Chain complexes

TODO

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. (TODO)

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)
- `SingleObjectComplex` (**??**)
- `FiniteComplex` (8.2.5)
- `ShortExactSequence` (**??**)

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 (**??**) object describing the category to create a chain complex over.

The rest of the arguments describe the differentials of the complex. These are divided into three parts: one finite ("middle") and two infinite ("positive" and "negative"). The positive part contains all differentials in degrees higher than those in the middle part, and the negative part contains all differentials in degrees lower than those in the middle part. (The middle part may be placed anywhere, so the positive part can – despite its name – contain some differentials of negative degree. Conversely, the negative part can contain some differentials of positive degree.)

The argument *middle* is a list containing the differentials for the middle part. The argument *baseDegree* gives the degree of the first differential in this list. The second differential is placed in degree *baseDegree* + 1, and so on. Thus, the middle part consists of the degrees

$$\mathit{baseDegree},\quad \mathit{baseDegree}+1,\quad \ldots\quad \mathit{baseDegree}+\text{Length}(\mathit{middle}).$$

Each of the arguments *positive* and *negative* can be one of the following:

- The string "zero", meaning that the part contains only zero objects and zero morphisms.
- A list of the form ["repeat", L], where L is a list of morphisms. The part will contain the differentials in L repeated infinitely many times. The convention for the order of elements in L is that L[1] is the differential which is closest to the middle part, and L[Length(L)] is farthest away from the middle part.
- A list of the form ["pos", f] or ["pos", f, store], where f is a function of two arguments, and store (if included) is a boolean. The function f is used to compute the differentials in this part. The function f is not called immediately by the Complex constructor, but will be called later as the differentials in this part are needed. The function call f(C,i) (where C is the complex and i an integer) should produce the differential in degree i. The function may use C to look up other differentials in the complex, as long as this does not cause an infinite loop. If store is true (or not specified), each computed differential is stored, and they are computed in order from the one closest to the middle part, regardless of which order they are requested in.
- A list of the form ["next", f, init], where f is a function of one argument, and init is a morphism. The function f is used to compute the differentials in this part. For the first differential in the part (that is, the one closest to the middle part), f is called with init as argument. For the next differential, f is called with the first differential as argument, and so on. Thus, the differentials are

  $$f(\text{init}),\quad f^2(\text{init}),\quad f^3(\text{init}),\quad \ldots$$

  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 (**??**) 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` (**??**) 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

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

and thus implicitly the objects in degrees

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

All other objects in the complex are zero.

8.3 Information about a complex

8.4 Transforming and combinig complexes

8.5 Chain maps

Appendix A

An Appendix

This is an appendix.

References

Index