HAPprime

Datatypes reference manual

Version 0.3

10 December 2008

Paul Smith

Paul Smith
— Email: paul.smith@nuigalway.ie
— Homepage: http://www.maths.nuigalway.ie/~pas
— Address: Department of Mathematics,
National University of Ireland, Galway
Galway,
Ireland.

Copyright

Acknowledgements

HAPprime is supported by a Marie Curie Transfer of Knowledge grant based at the Department of Mathematics, NUI Galway (MTKD-CT-2006-042685)

Contents

Chapter 1

Introduction

The HAPprime package is a GAP package which supplements the HAP package (http://hamilton.nuigalway.ie/Hap/www/), providing new and improved functions for doing homological algebra over small prime-power groups. A detailed overview of the HAPprime package, with examples and documentation of the high-level functions, is provided in the accompanying HAPprime user guide.

This document, the datatypes reference manual, supplements the HAPprime user guide. It describes the new GAP datatypes defined by the HAPprime package, and all of the associated functions for working with each of these datatypes. The datatypes are

FpGModuleGF (Chapter 5) a free $\mathbb{F}G$-module compactly represented in terms of generating elements, with operations that do as much manipulation as possible within this form, thus minimizing memory use.

FpGModuleHomomorphismGF (Chapter 6) a free linear homomorphism between two $\mathbb{F}G$-modules, each represented as a FpGModuleGF. this also uses the compact generator form to save memory in its operations.

HAPResolution (Chapter 2) this datatype, defined in the HAP package, represents a free $\mathbb{F}G$-resolution of a $\mathbb{F}G$-module. HAPprime extends the definition of this datatype to save memory, and provides additional functions to operate on resolutions.

HAPDerivation (Chapter 8) a derivation over a polynomial ring R. In particular, HAPprime provides functions to calculate the kernel and homology of derivations for polynomials over prime fields.

In addition, Chapter 10 provides documentation for some general functions defined in HAPprime which extend some of the basic GAP functionality in areas such as matrices and polynomials.

Each chapter of this reference manual begins with an overview of the datatype, and then implementation details of any interesting functions. The function reference of related functions then follows, subdivided into sections of related functions. Examples demonstrating the use of each function are given at the end of each section.

Chapter 2

Resolutions

A free $\mathbb{F}G$-resolution of an $\mathbb{F}G$-module M is a sequence of module homomorphisms

$$\ldots \to M_{n+1} \to M_n \to M_{n-1} \to \ldots \to M_1 \to M_0 \twoheadrightarrow M$$

Where each M_n is a free $\mathbb{F}G$-module and the image of $d_{n+1} : M_{n+1} \to M_n$ equals the kernel of $d_n : M_n \to M_{n-1}$ for all $n > 0$.

2.1 The `HAPResolution` datatype in HAPprime

Both HAP and HAPprime use the `HAPResolution` datatype to store resolutions, and you should refer to the HAP documentation for full details of this datatype. With resolutions computed by HAP, the boundary maps which define the module homomorphisms are stored as lists of $\mathbb{Z}G$-module words, each of which is an integer pair `[i,g]`. By contrast, when HAPprime computes resolutions it stores the boundary maps as lists of G-generating vectors (as used in `FpGModuleHomomorphismGF`, see Chapter 6). Over small finite fields (and in particular in GF(2)), these compressed vectors take far less memory, saving at least a factor of two for long resolutions. The different data storage method is entirely an internal change - as far as the used is concerned, both versions behave exactly the same.

2.2 Implementation: Constructing resolutions

Given the definition of a free $\mathbb{F}G$-resolution given above, a resolution of a module M can be calculated by construction. If there are k generators for M, we can set M_0 equal to the free $\mathbb{F}G$-module $(\mathbb{F}G)^k$, and the module homomorphism $d_0 : M_0 \to M$ to be the one that sends the ith standard generator of $(\mathbb{F}G)^k$ to the ith element of M. We can now recursively construct the other modules and module homomorphisms in a similar manner. Given a boundary homomorphism $d_n = M_n \to M_{n-1}$, the kernel of this can be calculated. Then given a set of generators (ideally a small set) for $ker(d_n)$, we can set $M_{n+1} = (\mathbb{F}G)^{|ker(d_n)|}$, and the new module homomorphism d_{n+1} to be the one mapping the standard generators of M_{n+1} onto the generators of $ker(d_n)$.

HAPprime implements the construction of resolutions using this method. The construction is divided into two stages. The creation of the first homomorphism in the resolution for M is performed by the function `LengthZeroResolutionPrimePowerGroup` (2.3.2), or for a resolution of the trivial $\mathbb{F}G$-module $\mathbb{F}$, the first two homomorphisms can be stated without calculation using `LengthOneResolutionPrimePowerGroup` (2.3.1). Once this initial sequence is created, a longer resolution can be created by repeated application of one

of ExtendResolutionPrimePowerGroupGF (**HAPprime: ExtendResolutionPrimePowerGroupGF**), ExtendResolutionPrimePowerGroupRadical (**HAPprime: ExtendResolutionPrimePowerGroupRadical**) or ExtendResolutionPrimePowerGroupGF2 (**HAPprime: ExtendResolutionPrimePowerGroupGF2**), each of which extends the resolution by one stage by constructing a new module and homomorphism mapping onto the minimal generators of the kernel of the last homomorphism of the input resolution. These extension functions differ in speed and the amount of memory that they use. The lowest-memory version, ExtendResolutionPrimePowerGroupGF (**HAPprime: ExtendResolutionPrimePowerGroupGF**), uses the block structure of module generating vectors (see Section 5.2.1) and calculates kernels of the boundary homomorphisms using KernelOfModuleHomomorphismSplit (6.6.3) and finds a minimal set of generators for this kernel using MinimalGeneratorsModuleGF (5.5.9). The much faster but memory-hungry ExtendResolutionPrimePowerGroupRadical (**HAPprime: ExtendResolutionPrimePowerGroupRadical**) uses KernelOfModuleHomomorphism (6.6.3) and MinimalGeneratorsModuleRadical (5.5.9) respectively. ExtendResolutionPrimePowerGroupGF2 (**HAPprime: ExtendResolutionPrimePowerGroupGF2**) uses KernelOfModuleHomomorphismGF (6.6.3) whic partitions the boundary homomorphism matrix using $\mathbb{F}G$-reduction. This gives a small memory saving over the Radical method, but can take as long as the GF scheme.

The construction of resolutions of length n is wrapped up in the functions ResolutionPrimePowerGroupGF, ResolutionPrimePowerGroupRadical and ResolutionPrimePowerGroupAutoMem, which (as well as the extension functions) are fully documented in Section (**HAPprime: ResolutionPrimePowerGroup**) of the HAPprime user manual.

2.3 Resolution construction functions

2.3.1 LengthOneResolutionPrimePowerGroup

◊ LengthOneResolutionPrimePowerGroup(*G*) (function)

Returns: HAPResolution

Returns a free $\mathbb{F}G$-resolution of length 1 for group *G* (which must be of a prime power), i.e. the resolution

$$\mathbb{F}G^{k_1} \to \mathbb{F}G \twoheadrightarrow \mathbb{F}$$

This function requires very little calculation: the first stage of the resolution can simply be stated given a set of minimal generators for the group.

2.3.2 LengthZeroResolutionPrimePowerGroup

◊ LengthZeroResolutionPrimePowerGroup(*M*) (function)

Returns: HAPResolution

Returns a minimal free $\mathbb{F}G$-resolution of length 0 for the FpGModuleGF module *M*, i.e. the resolution

$$\mathbb{F}G^{k_0} \twoheadrightarrow M$$

This function requires little calculation since the the first stage of the resolution can simply be stated if the module has minimal generators: each standard generator of the zeroth-degree module M_0 maps onto a generator of *M*. If *M* does not have minimal generators, they are calculated using MinimalGeneratorsModuleRadical (5.5.9).

2.4 Resolution data access functions

2.4.1 ResolutionLength

◊ `ResolutionLength(R)` (method)

Returns: Integer

Returns the length (i.e. the maximum index k) in the resolution `R`.

2.4.2 ResolutionGroup

◊ `ResolutionGroup(R)` (method)

Returns: Group

Returns the group of the resolution `R`.

2.4.3 ResolutionFpGModuleGF

◊ `ResolutionFpGModuleGF(R, k)` (method)

Returns: `FpGModuleGF`

Returns the module M_k in the resolution `R`, as a `FpGModuleGF` (see Chapter 5), assuming the canonical action.

2.4.4 ResolutionModuleRank

◊ `ResolutionModuleRank(R, k)` (method)

Returns: Integer

Returns the $\mathbb{F}G$ rank of the kth module M_k in the resolution.

2.4.5 ResolutionModuleRanks

◊ `ResolutionModuleRanks(R)` (method)

Returns: List of integers

Returns a list containg the $\mathbb{F}G$ rank of the each of the modules M_k in the resolution `R`.

2.4.6 BoundaryFpGModuleHomomorphismGF

◊ `BoundaryFpGModuleHomomorphismGF(R, k)` (method)

Returns: `FpGModuleHomomorphismGF`

Returns the kth boundary map in the resolution `R`, as a `FpGModuleHomomorphismGF`. This represents the linear homomorphism $d_k : M_k \longrightarrow M_{k-1}$.

2.4.7 ResolutionsAreEqual

◊ `ResolutionsAreEqual(R, S)` (operation)

Returns: Boolean

Returns `true` if the resolutions appear to be equal, `false` otherwise. This compares the torsion coefficients of the homology from the two resolutions.

2.5 Example: Computing and working with resolutions

In this example we construct a minimal free $\mathbb{F}G$-resolution of length four for the group $G = D_8 \times Q_8$ of order 64, which will be the sequence

$$(\mathbb{F}G)^{22} \to (\mathbb{F}G)^{15} \to (\mathbb{F}G)^{9} \to (\mathbb{F}G) \twoheadrightarrow \mathbb{F}$$

We first build each stage explicitly, starting with `LengthOneResolutionPrimePowerGroup` (2.3.1) followed by repeated applications of `ExtendResolutionPrimePowerGroupRadical` (**HAPprime: ExtendResolutionPrimePowerGroupRadical**). We extract various properties of this resolution. Finally, we construct equivalent resolutions for G using `ResolutionPrimePowerGroupGF` (**HAPprime: ResolutionPrimePowerGroupGF (for group)**) and `ResolutionPrimePowerGroupGF2` (**HAPprime: ResolutionPrimePowerGroupGF2 (for group)**) and check that the three are equivalent.

Example

```
gap> G := DirectProduct(DihedralGroup(8), SmallGroup(8, 4));
<pc group of size 64 with 6 generators>
gap> R := LengthOneResolutionPrimePowerGroup(G);
Resolution of length 1 in characteristic 2 for <pc group of size 64 with
6 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> R := ExtendResolutionPrimePowerGroupRadical(R);;
gap> R := ExtendResolutionPrimePowerGroupRadical(R);;
gap> R := ExtendResolutionPrimePowerGroupRadical(R);
Resolution of length 4 in characteristic 2 for <pc group of size 64 with
6 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> #
gap> ResolutionLength(R);
4
gap> ResolutionGroup(R);
<pc group of size 64 with 6 generators>
gap> ResolutionModuleRanks(R);
[ 4, 9, 15, 22 ]
gap> ResolutionModuleRank(R, 3);
15
gap> M2 := ResolutionFpGModuleGF(R, 2);
Full canonical module FG^9 over the group ring of <pc group of size 64 with
6 generators> in characteristic 2

gap> d3 := BoundaryFpGModuleHomomorphismGF(R, 3);
<Module homomorphism>

gap> ImageOfModuleHomomorphism(d3);
Module over the group ring of <pc group of size 64 with
6 generators> in characteristic 2 with 15 generators in FG^9.

gap> #
```

```
gap> S := ResolutionPrimePowerGroupGF(G, 4);
Resolution of length 4 in characteristic 2 for <pc group of size 64 with
6 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> ResolutionsAreEqual(R, S);
true
gap> T := ResolutionPrimePowerGroupGF2(G, 4);
Resolution of length 4 in characteristic 2 for <pc group of size 64 with
6 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> ResolutionsAreEqual(R, T);
true
```

Further example of constructing resolutions and extracting data from them are given in Sections 5.4.11, 5.5.11, 5.6.3, 6.5.7 and 6.6.4 in this reference manual, and also the chapter of (**HAPprime: Examples**) in the HAPprime user guide.

2.6 Miscellaneous resolution functions

2.6.1 BestCentralSubgroupForResolutionFiniteExtension

◊ `BestCentralSubgroupForResolutionFiniteExtension(G[, n])` (operation)

Returns: Group

Returns the central subgroup of G that is likely to give the smallest module ranks when using the HAP function `ResolutionFiniteExtension` (**HAP: ResolutionFiniteExtension**). That function computes a non-minimal resolution for G from the twisted tensor product of resolutions for a normal subgroup $N \lhd G$ and the quotient group G/N. The ranks of the modules in the resolution for G are the products of the module ranks of the resolutions for these smaller groups. This function tests n terms of the minimal resolutions for all the central subgroups of G and the corresponding quotients to find the subgroup/quotient pair with the smallest module ranks. If n is not provided, then $n = 5$ is used.

Chapter 3

Graded algebras

A graded algebra A is a ring that has a direct sum decomposition into Abelian additive groups (called degrees)

$$A = \bigoplus_{n\in\mathbb{N}} A_n = A_0 \oplus A_1 \oplus A_2 \oplus \dots$$

and where multiplication of elements follows a grading such that

$$A_s \times A_r \longrightarrow A_{s+r}$$

This means that for any elements $x \in A_s$ and $y \in A_r$, the product lies in degree $r+s$, i.e. $xy \in A_{s+r}$.

3.1 Graded algebras in HAP (and HAPprime)

Small finite algebras can be represented explicitly in GAP using a basis and a structure constant table (**Tutorial: Algebras**), which specifies all products of basis elements. HAP extends this definition to provide a finite approximation to a graded algebra, storing all basis elements and products up to a given degree, n. The graded algebras used by HAP add a new component to the algebra object `A`, the function `A!.degree(e)` which returns the degree of an algebra element `e`.

HAPprime uses this HAP algebra object and adds some attributes, providing information on the basis and generators of the algebra. A ring presentation for this algebra can also be computed and represented using the `GradedAlgebraPresentation` datatype (see Section 4).

3.2 Data access functions

3.2.1 ModPRingGeneratorDegrees

◊ `ModPRingGeneratorDegrees(A)` (attribute)

Returns: List

Returns a list containing the degree of each generator in a minimal generating set for the mod-p cohomology ring `A`. The ith degree in the list corresponds to the ith generator returned by `ModPRingGenerators` (**HAP: ModPRingGenerators**)

3.2.2 ModPRingNiceBasis

◊ ModPRingNiceBasis(*A*) (attribute)

Returns: List

Returns the information needed to convert the basis for *A* (see Basis (**Reference: Basis**)) into a nicer basis consisting of only products of ring generators. The function returns a pair of lists [Coeff, Bas]. The list Coeff is a change-of-basis matrix, where the *i*th row gives the standard basis element *i* in terms of the nice basis. The list Bas can be used to form the new basis and is a list of integers where the *i*th 'nice basis' element is given by Product(List(Bas[i], x->Basis(A)[x])).

This attribute returns exactly the same list as is provided by component A!.niceBasis (see ModPCohomologyRing (**HAP: ModPCohomologyRing**), but automatically constructs this list if it is not available.

3.2.3 ModPRingNiceBasisAsPolynomials

◊ ModPRingNiceBasisAsPolynomials(*A*) (attribute)

Returns: List

A list which gives the 'nice basis' of the algebra *A* (as returned by the second element of ModPRingNiceBasis (3.2.2)) in terms of products of the indeterminates in the ring presentation (as given by PresentationOfGradedStructureConstantAlgebra (3.3.1)). The *i*th entry in the list corresponds to the *i*th basis element returned by ModPRingNiceBasis (3.2.2).

3.2.4 ModPRingBasisAsPolynomials

◊ ModPRingBasisAsPolynomials(*A*) (attribute)

Returns: List

A list which gives the basis of the algebra *A* (as returned by Basis(A)) in terms of sums of products of the indeterminates in the ring presentation (as given by PresentationOfGradedStructureConstantAlgebra (3.3.1)). The *i*th entry in the list corresponds to the *i*th basis element returned by Basis (**Reference: Basis**).

3.3 Other functions

3.3.1 PresentationOfGradedStructureConstantAlgebra

◊ PresentationOfGradedStructureConstantAlgebra(*A*) (attribute)

Returns: GradedAlgebraPresentation

Returns a ring presentation for the graded algebra *A*. The ring *A* must be a structure constant algebra with embedded degrees, such as is returned by ModPCohomologyRing (**HAP: ModPCohomologyRing**). The generators of the GradedAlgebraPresentation (as returned by IndeterminatesOfGradedAlgebraPresentation (4.3.3) are in one-to-one correspondance with the generators of *A* as returned by ModPRingGenerators (**HAP: ModPRingGenerators**) (ignoring the first generator, which is in degree zero).

3.4 Example: Graded algebras and mod-p cohomology rings

The mod-p cohomology ring of a p-group is a graded algebra (see (**HAPprime: Computing mod-p cohomology rings and their Poincaré series**)). As an example, we use the HAP function `ModPCohomologyRing` (**HAP: ModPCohomologyRing**) to compute this algebra for the quaternion group Q_8 up to and including degree five. We can display its generators and basis (and degrees), and the information to construct a 'nice basis'

Example

```
gap> A := ModPCohomologyRing(SmallGroup(8, 4), 5);
<algebra of dimension 9 over GF(2)>
gap> Display(Basis(A));
CanonicalBasis( Algebra( GF(2), [ v.1, v.2, v.3, v.4, v.5, v.6, v.7, v.8, v.9
 ] ) )
gap> List(Basis(A), i->A!.degree(i));
[ 0, 1, 1, 2, 2, 3, 4, 5, 5 ]
gap> ModPRingGenerators(A);
[ v.1, v.2, v.3, v.7 ]
gap> ModPRingGeneratorDegrees(A);
[ 0, 1, 1, 4 ]
gap> ModPRingNiceBasis(A);
[ [ [ 1, 0, 0, 0, 0, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0, 0, 0, 0, 0 ],
      [ 0, 0, 1, 0, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 1, 0, 0, 0 ],
      [ 0, 0, 0, 0, 1, 1, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 0, 0, 0, 1 ],
      [ 0, 0, 0, 1, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 0, 0, 1, 0 ],
      [ 0, 0, 0, 0, 0, 0, 1, 1, 0 ] ],
  [ [ 1, 1, 1 ], [ 1, 1, 2 ], [ 1, 1, 3 ], [ 1, 1, 7 ], [ 1, 2, 2 ],
      [ 1, 2, 3 ], [ 1, 2, 7 ], [ 1, 3, 7 ], [ 2, 2, 3 ] ] ]
```

A presentation for this algebra can be constructed using `PresentationOfGradedStructureConstantAlgebra` (3.3.1), which (see below) returns the presentation to be

$$H^*(G,\mathbb{F}) = \frac{\mathbb{F}[x_1,x_2,x_3]}{\langle x_1^2+x_1x_2+x_2^2, x_2^3\rangle}$$

where x_1 and x_2 have degree one and x_3 is degree four. The 'nice basis' referred to above is monomials of degree up to five in this polynomial ring, and is not necessarily the same as the basis of the algebra given by `Basis` (**Reference: Basis**), as demonstrated below.

Example

```
gap> A := ModPCohomologyRing(SmallGroup(8, 4), 5);;
gap> PresentationOfGradedStructureConstantAlgebra(A);
Graded algebra GF(2)[ x_1, x_2, x_3 ] / [ x_1^2+x_1*x_2+x_2^2, x_2^3
 ] with indeterminate degrees [ 1, 1, 4 ]

gap> ModPRingNiceBasisAsPolynomials(A);
[ Z(2)^0, x_1, x_2, x_3, x_1^2, x_1*x_2, x_1*x_3, x_2*x_3, x_1^2*x_2 ]
gap> ModPRingBasisAsPolynomials(A);
[ Z(2)^0, x_1, x_2, x_1*x_2, x_1^2+x_1*x_2, x_1^2*x_2, x_3, x_2*x_3,
  x_1*x_3+x_2*x_3 ]
```

Chapter 4

Presentations of graded algebras

A graded algebra A is a an algebra that has additional structure, called a grading (see Section 3). Graded algebras of the type found in HAPprime have a presentation as a quotient of a polynomial ring

$$H^*(G,\mathbb{F}) = \frac{\mathbb{F}[x_1,x_2,\ldots,x_n]}{\langle I_1,I_2,\ldots,I_m\rangle}$$

where the polynomial ring indeterminates x_i each have an associated degree d_i and the I_j are relations which together generate an ideal in the ring.

4.1 The `GradedAlgebraPresentation` datatype

For algebras that have a presentation as a quotient of a polynomial ring, the `GradedAlgebraPresentation` datatype stores a quotient R/I where:

- R is a polynomial ring
- I is a set of relations in R that generate an ideal

and it also stores a grading in the form of

- the degree of each indeterminate of R

4.2 Construction function

4.2.1 GradedAlgebraPresentation construction functions

◊ `GradedAlgebraPresentation(R, I, degs)` (operation)
◊ `GradedAlgebraPresentationNC(R, I, degs)` (operation)

Returns: `GradedAlgebraPresentation`

Construct a `GradedAlgebraPresentation` object representing a presentation of a graded algebra as the quotient of a polynomial ring `R` by the ideal `I` (as a list of relations in `R`) where the indeterminates of `R` (as returned by `IndeterminatesOfGradedAlgebraPresentation` (4.3.3) have degrees `degs` respectively.

The function `GradedAlgebraPresentation` checks that the arguments are compatible, while the `NC` method performs no checks.

4.3 Data access functions

4.3.1 BaseRing

◊ BaseRing(A) (attribute)

Returns: Polynomial ring

Returns the base ring of the graded algebra presentation A.

4.3.2 CoefficientsRing

◊ CoefficientsRing(A) (attribute)

Returns: Ring

Returns the ring of coefficients of the graded algebra presentation A.

4.3.3 IndeterminatesOfGradedAlgebraPresentation

◊ IndeterminatesOfGradedAlgebraPresentation(A) (attribute)

Returns: List

Returns the indeterminates used in the graded algebra presentation A.

4.3.4 GeneratorsOfPresentationIdeal

◊ GeneratorsOfPresentationIdeal(A) (attribute)

Returns: List

Returns the relations in the ring presentation for the graded algebra A. The relations are returned sorted in order of increasing degree, and by indeterminate within each degree.

4.3.5 PresentationIdeal

◊ PresentationIdeal(A) (attribute)

Returns: Ideal

Returns the ideal in the graded algebra presentation A as a GAP ideal (**Reference: Ideal**).

4.3.6 IndeterminateDegrees

◊ IndeterminateDegrees(A) (attribute)

Returns: List

Returns the degrees of the polynomial ring indeterminates in the graded algebra presentation A. The ordering corresponds to the order of the ring indeterminates returned by IndeterminatesOfGradedAlgebraPresentation (4.3.3).

4.3.7 Example: Constructing and accessing data of a GradedAlgebraPresentation

We demonstrate creating a GradedAlgebraPresentation object and reading back its data by creating the graded algebra A with presentation $\mathbb{F}_2[x_1,x_2,x_3]/(x_1x_2,x_1^3+x_2^3)$ where x_1 and x_2 have degree 1 and x_3 has degree 4

Example

```
gap> R := PolynomialRing(GF(2), 3);;
gap> A := GradedAlgebraPresentation(R, [R.1*R.2, R.1^3+R.2^3], [1,1,4]);
Graded algebra GF(2)[ x_1, x_2, x_3 ] / [ x_1*x_2, x_1^3+x_2^3
 ] with indeterminate degrees [ 1, 1, 4 ]
gap> BaseRing(A);
GF(2)[x_1,x_2,x_3]
gap> CoefficientsRing(A);
GF(2)
gap> IndeterminatesOfGradedAlgebraPresentation(A);
[ x_1, x_2, x_3 ]
gap> PresentationIdeal(A);
<two-sided ideal in GF(2)[x_1,x_2,x_3], (2 generators)>
gap> GeneratorsOfPresentationIdeal(A);
[ x_1*x_2, x_1^3+x_2^3 ]
gap> IndeterminateDegrees(A);
[ 1, 1, 4 ]
```

4.4 Other functions

4.4.1 TensorProduct

◊ `TensorProduct(A, B)` (operation)
◊ `TensorProduct(coll)` (operation)

Returns: GradedAlgebraPresentation

Returns a presentation for the graded algebra that is the tensor product of two graded algebras presented by `A` and `B`, or of a list of graded algebras.

4.4.2 IsIsomorphicGradedAlgebra

◊ `IsIsomorphicGradedAlgebra(A, B)` (operation)

Returns `true` if the graded algebras `A` and `B` are isomorphic, or `false` otherwise. This function tries all possible ring isomorphisms, so may take a considerable length of time for graded algebras with a large number of dimensions in each degree.

4.4.3 IsAssociatedGradedRing

◊ `IsAssociatedGradedRing(A, B)` (operation)

Returns `true` if the algebra `A` is an associated graded ring of the algebra `B`. This is the case if the additive structure is the same (i.e. the Hilbert-Poincaré series is the same), and the generators for `A` (and their degrees) are included in the generators for `B`.

4.4.4 DegreeOfRepresentative

◊ `DegreeOfRepresentative(A, p)` (operation)

Returns: Integer

Returns the degree of a polynomial representative `p` from the graded ring presentation `A`.

4.4.5 MaximumDegreeForPresentation

◊ MaximumDegreeForPresentation(A) (attribute)

Returns: Integer

Returns the maximum degree in generators or relations that is needed to generate the graded algebra presentation A. This is not necessarily the same as the largest degree in any of the relations and generators - some relations may be redundant (for example due to being a Groebner basis), so this routine checks for the largest degree of a required generator, and returns the maximum of this and the generator degrees.

4.4.6 SubspaceDimensionDegree

◊ SubspaceDimensionDegree(A, d) (operation)
◊ SubspaceDimensionDegree(A, degs) (operation)

Returns: Integer or list

Returns the dimension of degree d of the graded algebra A, or a list of dimensions corresponding to the list of degrees degs.

4.4.7 SubspaceBasisRepsByDegree

◊ SubspaceBasisRepsByDegree(A, d) (operation)
◊ SubspaceBasisRepsByDegree(A, degs) (operation)

Returns: List or list of lists

Returns a basis for degree d of the graded algebra A, or a list of bases for the list of degrees degs. Each basis is returned as a list of representatives.

4.4.8 CoefficientsOfPoincareSeries

◊ CoefficientsOfPoincareSeries(A, n) (operation)

Returns: List

Returns the first n coefficients of the Poincaré series for the graded algebra with A. These are equal to the dimensions of degrees 0 to $n-1$ of the algebra (a fact that is used in the function SubspaceDimensionDegree (4.4.6)).

This function uses the singular *package.*

4.4.9 HilbertPoincareSeries

◊ HilbertPoincareSeries(A) (attribute)

Returns: Rational function

Returns the Poincaré series for the graded algebra A. This is a rational function $P(t)/Q(t)$ which is a is a polynomial whose coefficients are the dimensions of each degree of the algebra.

This function uses the singular *package.*

4.4.10 LHSSpectralSequence

◊ LHSSpectralSequence(G[, N], n) (operation)
◊ LHSSpectralSequenceLastSheet(G[, N]) (operation)

Returns: GradedAlgebraPresentation or list

Computes the Lyndon-Hoschild-Serre spectral sequence for the group extension $N \to G \to G/N$. If a normal suggroup `N` is not provided, then the largest central subgroup of G is used, or (if the order of the centre is larger than $\sqrt{|G|}$) then the central subgroup that leads to the smallest initial sheet size is chosen.

The function `LHSSpectralSequence` returns the first n sheets of the spectral sequence, or all of the sequence up to convergence, if that occurs before the $(n+1)$th sheet. The Lyndon-Hoschild-Serre spectral sequence starts at the E_2 sheet, so the first element in returned list will always be empty. If n is set to `infinity` then the length of the returned list equals the number of sheets for convergence, and the last sheet in the list is the limiting sheet.

The function `LHSSpectralSequenceLastSheet` returns only the limiting sheet of the spectral sequence. This ring is an associated graded algebra of the mod-p cohomology ring of G, with the same additive structure while not necessarily being isomorphic to it.

There are four options (**Reference: Options Stack**) which can be used to guide this algorithm:

- `InitialLHSBicomplexSize` can be used to specify the initial size of the bicomplex (the default is 5). If, in the process of computing the spectral sequence, this is found to be too small then the algorithm restarts with a larger value. Specifying a larger initial value in these cases can save time.

- `LargerLHSBicomplexBreak` if set to `true` will force the calculation to enter a break loop before restarting with a larger bicomplex, should the bicomplex be found to be too small. The user user is prompted to type `return;` before continuing. The default behaviour is `false`, i.e. no prompt.

- `LargerLHSBicomplexFail` if set to `true` will return `fail` should the bicomplex be found to be too small. The default behaviour is `false`, i.e. to either restart or prompt, depending on the setting of the previous option.

- `NoInductiveProof` if set to `true` will not check that the cohomology rings for N and G/N are correct. Instead, it will compute the cohomology rings only up to the degree needed for the bicomplex size (5 by default, or specified by the `InitialLHSBicomplexSize` option).

4.5 Example: Computing the Lyndon-Hoschild-Serre spectral sequence and mod-p cohomology ring for a small p-group

The Lyndon-Hoschild-Serre spectral sequence is relates the cohomologies of a normal subgroup N and a quotient group G/N to the cohomology of the total group G: the limiting sheet of the sequence is an associated graded ring of the cohomology of G.

In this example we calculate the Lyndon-Hoschild-Serre spectral sequence for a group of order 16 using the centre of G as our normal subgroup. By asking for an infinite number of terms, this function calculates enough terms to be sure that the sequence has converged. We compare the dimensions in the first (E_2) and last (E_∞) sheet, we demonstrate that the limiting sheet (the last in the list) is a graded algebra by multiplying some elements, and we calculate the Poincaré series of the last sheet.

Example

```
gap> G := SmallGroup(16, 4);;
gap> SS := LHSSpectralSequence(G, Centre(G), infinity);
[ , Graded algebra GF(2)[ x_1, x_2, x_3, x_4 ] /
    [  ] with indeterminate degrees [ 1, 1, 1, 1 ],
  Graded algebra GF(2)[ x_1, x_2, x_3, x_4 ] / [ x_2^2, x_1^2+x_1*x_2
```

```
     ] with indeterminate degrees [ 1, 1, 2, 2 ],
  Graded algebra GF(2)[ x_1, x_2, x_3, x_4 ] / [ x_2^2, x_1^2+x_1*x_2
     ] with indeterminate degrees [ 1, 1, 2, 2 ] ]
gap> # i.e. we identify convergence after 3 terms
gap> #
gap> # Compare the dimensions of the first and last sheet
gap> SubspaceDimensionDegree(SS[2], [1..10]);
[ 4, 10, 20, 35, 56, 84, 120, 165, 220, 286 ]
gap> SubspaceDimensionDegree(SS[3], [1..10]);
[ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ]
gap> #
gap> # Take the two basis elements from degree 1 and check that the
gap> # product is in degree two
gap> B := SubspaceBasisRepsByDegree(SS[3], 1);
[ x_1, x_2 ]
gap> DegreeOfRepresentative(SS[3], B[1]*B[2]);
2
gap> #
gap> # And find the Poincare series
gap> HilbertPoincareSeries(SS[3]);
(1)/(x_1^2-2*x_1+1)
```

The largest degree in the presentation for the limiting sheet in the Lyndon-Hoschild-Serre spectral sequence for G is the same as the largest degree in the presentation for the mod-p cohomology ring of G. We continue this example by calculating this maximum degree, n, for our group G and then computing the mod-p cohomology ring. We confirm that the cohomology ring is an associated graded ring of the limiting sheet of the spectral sequence, and check whether in this case it is in fact also isomorphic.

Example

```
gap> G := SmallGroup(16, 4);;
gap> Einf := LHSSpectralSequenceLastSheet(G, Centre(G));
Graded algebra GF(2)[ x_1, x_2, x_3, x_4 ] / [ x_2^2, x_1^2+x_1*x_2
 ] with indeterminate degrees [ 1, 1, 2, 2 ]
gap> #
gap> # Find the maximum degree
gap> n := MaximumDegreeForPresentation(Einf);
2
gap> #
gap> # And calculate the cohomology ring
gap> H := Mod2CohomologyRingPresentation(G, n);
Graded algebra GF(2)[ x_1, x_2, x_3, x_4 ] / [ x_1*x_2+x_2^2, x_1^2
 ] with indeterminate degrees [ 1, 1, 2, 2 ]
gap> #
gap> # Check for an associated graded ring, and isomorphism
gap> IsAssociatedGradedRing(H, Einf);
true
gap> IsIsomorphicGradedAlgebra(H, Einf);
true
```

Chapter 5

$\mathbb{F}G$-modules

Let $\mathbb{F}G$ be the group ring of the group G over the field $\mathbb{F}$. In this package we only consider the case where G is a finite p-groups and $\mathbb{F} = \mathbb{F}_p$ is the field of p elements. In addition, we only consider free $\mathbb{F}G$-modules.

5.1 The `FpGModuleGF` datatype

Modules and submodules of free $\mathbb{F}G$-modules are represented in HAPprime using the `FpGModuleGF` datatype, where the '`GF`' stands for 'Generator Form'. This defines a module using a group G and a set of G-generating vectors for the module's vector space, together with a group action which operates on those vectors. A free $\mathbb{F}G$-module $\mathbb{F}G$ can be considered as a vector space $\mathbb{F}^{|G|}$ whose basis is the elements of G. An element of $(\mathbb{F}G)^n$ is the direct sum of n copies of $\mathbb{F}G$ and, as an element of `FpGModuleGF`, is represented as a vector of length $n|G|$ with coefficients in $\mathbb{F}$. Representing our $\mathbb{F}G$-module elements as vectors is ideal for our purposes since GAP's representation and manipulation of vectors and matrices over small prime fields is very efficient in both memory and computation time.

The HAP package defines a `FpGModule` object, which is similar but stores a vector space basis rather than a G-generating set for the module's vector space. Storing a G-generating set saves memory, both in passive storage and in allowing more efficient versions of some computation algorithms.

There are a number of construction functions for `FpGModuleGF`s: see 5.3.1 for details. A $\mathbb{F}G$-module is defined by the following:

- `gens`, a list of G-generating vectors for the underlying vector space. These do not need to be minimal - they could even be a vector space basis. The `MinimalGeneratorsModule` functions (5.5.9) can be used to convert a module to one with a minimal set of generators.
- `group`, the group G for the module
- `action`, a function `action(g, u)` that represents the module's group action on vectors. It takes a group element $g \in G$ and a vector `u` of length `actionBlockSize` and returns another vector `v` of the same length that is the product $v = gu$. If `action` is not provided, the canonical group permutation action is used. If the vector `u` is an integer multiple of `actionBlockSize` in length, the function `action` acts block-wise on the vector.
- `actionBlockSize`, the length of vectors upon which `action` operates. This is usually the order of the group, $|G|$ (for example for the canonical action), but it is possible to specify this to

support other possible group actions that might act on larger vectors. `actionBlockSize` will always be equal to the ambient dimension of the module $\mathbb{F}G^1$.

The group, action and block size are internally wrapped up into a record `groupAndAction`, with entries `group`, `action` and `actionBlockSize`. This is used to simplify the passing of parameters to some functions.

Some additional information is sometimes needed to construct particular classes of `FpGModuleGF`:

- `ambientDimension`, the length of vectors in the generating set: for a module $(\mathbb{F}G)^n$, this is equal to $n \times$`actionBlockSize`. This is needed in the case when the list of generating vectors is empty.
- `form`, a string detailing whether the generators are known to be minimal or not, and if so in which format. It can be one of `"unknown"`, `"fullcanonical"`, `"minimal"`, `"echelon"` or `"semiechelon"`. Some algorithms require a particular form, and algorithms such as `EchelonModuleGenerators` (5.6.1) that manipulate a module's generators to create these forms set this entry.

5.2 Implementation details: Block echelon form

5.2.1 Generating vectors and their block structure

Consider the vector representation of an element in the $\mathbb{F}G$-module $(\mathbb{F}G)^2$, where G is a group of order four:

$$v \in (\mathbb{F}G)^2 = (g_1 + g_3, g_1 + g_2 + g_4) = [1010|1101]$$

The first block of four entries in the vector correspond to the first $\mathbb{F}G$ summand and the second block to the second summand (and the group elements are numbered in the order provided by the GAP function `Elements` (**Reference: Elements**)).

Given a G-generating set for a $\mathbb{F}G$-module, the usual group action permutes the group elements, and thus the effect on the vector is to permute the equivalent vector elements. Each summand is independent, and so elements are permuted within the blocks (normally of size $|G|$) seen in the example above. A consequence of this is that if any block (i.e. summand) in a generator is entirely zero, then it remains zero under group (or $\mathbb{F}$) multiplication and so that generator contributes nothing to that part of the vector space. This fact enables some of the structure of the module's vector space to be inferred from the G-generators, without needing a full vector space basis . A desirable set of G-generators is one that has many zero blocks, and what we call the 'block echelon' form is one that has this property.

5.2.2 Matrix echelon reduction and head elements

The block echelon form of a $\mathbb{F}G$-module generating set is analagous to the echelon form of matrices, used as a first stage in many matrix algorithms, and we first briefly review matrix echelon form. In a (row) echelon-form matrix, the head element in each row (the first non-zero entry) is the identity, and is to the right of the head in the previous row. A consequence of this is that the values below each head are all zero. All zero rows are at the bottom of the matrix (or are removed). GAP also defines a semi-echelon form, in which it is guaranteed that all values below each head is zero, but not that each head is to the right of those above it.

Matrices can be converted into (semi-)echelon form by using Gaussian elimination to perform row reduction (for example the GAP function `SemiEchelonMat` (**Reference: SemiEchelonMat**)). A

typical algorithm gradually builds up a list of matrix rows with unique heads, which will eventually be an echelon-form set of basis elements for the row space of the matrix. This set is initialised with the first row of the matrix, and the algorithm is then applied to each subsequent row in turn. The basis rows in the current set are used to reduce the next row of the matrix: if, after reduction, it is non-zero then it will have a unique head and is added to the list of basis rows; if it is zero then it may be removed. The final set of vectors will be a semi-echelon basis for the row space of the original matrix, which can then be permuted to give an echelon basis if required.

5.2.3 Echelon block structure and minimal generators

In the same way that the echelon form is useful for vector space generators, we can convert a set of $\mathbb{F}G$-module generators into echelon form. However, unlike multiplication by a field element, the group action on generating vectors also permutes the vector elements, so a strict echelon form is less useful. Instead, we define a 'block echelon' form, treating the blocks in the vector (see example above) as the $\mathbb{F}G$-elements to be converted into echelon form. In block-echelon form, the first non-zero block in each row is as far to the right as possible. Often, the first non-zero block in a row will be to the right of the first non-zero block in the row above, but when several generating vectors are needed in a block, this may not be the case. The following example creates a random submodule of $(\mathbb{F}G)^n$ by picking five generating vectors at random. This module is first displayed with the original generators, and then they are converted to block echelon form using the the HAPprime function `EchelonModuleGenerators` (5.6.1). The two generating sets both span the same vector space (i.e. the same $\mathbb{F}G$ module), but the latter representation is much more useful.

Example

```
gap> M := FpGModuleGF(RandomMat(5, 32, GF(2)), DihedralGroup(8));;
gap> Display(M);
Module over the group ring of Group( [ f1, f2, f3 ] )
 in characteristic 2 with 5 generators in FGˆ4.
[.1..1.1.|.1....1.|1111.11.|11.1111.]
[11.1..1.|1....11.|1...111.|1...11..]
[11..1.1.|1.1.1...|11...1..|.11.11..]
[11111111|111...1.|.11...1.|.1..1111]
[.1111111|1.1.111.|..1..1..|1.111...]
gap> echM := EchelonModuleGenerators(M);
rec( module := Module over the group ring of <pc group of size 8 with
    3 generators> in characteristic 2 with 4 generators in FGˆ
    4. Generators are in minimal echelon form., headblocks := [ 1, 2, 3, 4 ] )
gap> Display(echM.module);
Module over the group ring of Group( [ f1, f2, f3 ] )
 in characteristic 2 with 4 generators in FGˆ4.
[.1..1.1.|.1....1.|1111.11.|11.1111.]
[........|.1111..1|111.1...|.11.11.1]
[........|........|.1..1.1.|.1.1.111]
[........|........|........|..1111.1]
Generators are in minimal echelon form.gap>
gap> M = echM.module;
true
```

The main algorithm for converting a set of generators into echelon form is `SemiEchelonModuleGeneratorsDestructive` (5.6.1). The generators for the $\mathbb{F}G$ module are represented as rows of a matrix, and (with the canonical action) the first $|G|$ columns of this matrix

correspond to the first block, the next $|G|$ columns to the second block, and so on. The first block of the matrix is taken and the vector space V_b spanned by the rows of that block is found (which will be a a subspace of $\mathbb{F}^{|G|}$). Taking the rows in the first block, find (by gradually leaving out rows) a minimal subset that generates V_b. The rows of the full matrix that correspond to this minimal subset form the first rows of the block-echelon form generators. Taking those rows, and all G-multiples of them, now calculate a semi-echelon basis for the vector space that they generate (using `SemiEchelonMatDestructive` (**Reference: SemiEchelonMatDestructive**)). This is used to reduce all of the other generators. Since the rows we have chosen span the space of the first block, the first block in all the other rows will be reduced to zero. We can now move on to the second block.

We now look at the rows of the matrix that start (have their first non-zero entry) in the second block. In addition, some of the generators used for the first block might additionally give rise to vector space basis vectors with head elements in the second blocks. The rows need to be stored during the first stage and reused here. We find a minimal set of the matrix rows with second-block heads that, when taken with any second-block heads from the first stage, generate the entire space spanned by the second block. The vector-space basis from this new minimal set is then used to reduce the rest of the generating rows as before, reducing all of the other rows' second blocks to zero. The process is then repeated for each other block. Any generators that are completely zero are then removed. The algorithm is summarised in the following pseudocode:

```
Let X be the list of generators still to convert (initially all the generators)
Let Xe = [] be the list of generators already in block-echelon form
Define X{b} to represent the $b$th block from generators X
Define V(X) to represent the vector space generated by generators X
-------------------------------------------------------------------------------
for b in [1..blocks]
  1. Find a minimal set of generators Xm from X such that
     V(Xm{b} + Xe{b}) = V(X{b} + Xe{b})
  2. Remove Xm from X and add them to Xe
  3. Find a semi-echelon basis for V(Xe) and use this to reduce the elements
     of block b in remaining vectors of X to zero
end for
```

The result of this algorithm is a new generating set for the module that is minimal in the sense that no vector can be removed from the set and leave it still spanning the same vector space. In the case of a $\mathbb{F}G$-module with $\mathbb{F}$=GF(2), this is a globally minimal set: there is no possible alternative set with fewer generators.

5.2.4 Intersection of two modules

Knowing the block structure of the modules enables the intersection of two modules to be calculated more efficiently. Consider two modules U and V with the block structure as given in this example:

Example

```
gap> DisplayBlocks(U);
[*..]
[**.]
[.*.]
gap> DisplayBlocks(V);
[.**]
```

```
[.**]
[..*]
```

To calculate the intersection of the two modules, it is normal to expand out the two modules to find the vector space $U_{\mathbb{F}}$ and $V_{\mathbb{F}}$ of the two modules and calculate the intersection as a standard vector-space intersection. However, in this case, since U has no elements in the last block, and V has no elements in the first block, the intersection must only have elements in the middle block. This means that the first generator of U and the last generator of V can not be in the intersection and can be ignored for the purposes of the intersection calculation. In general, rather than expanding the entirety of U and V into an $\mathbb{F}$-basis to calculate their intersection, one can expand U and V more intelligently into $\mathbb{F}$-bases $U'_{\mathbb{F}}$ and $V'_{\mathbb{F}}$ which are smaller than $U_{\mathbb{F}}$ and $V_{\mathbb{F}}$ but have the same intersection.

Having determined (by comparing the block structure of U and V) that only the middle block in our example contributes to the intersection, we only need to expand out the rows of U and V that have heads in that block. The first generator of U generates no elements in the middle block, and trivially be ignored. The second row of U may or may not contribute to the intersection: this will need expanding out and echelon reduced. The expanded rows that don't have heads in the central block can then be discarded, with the other rows forming part of the basis of $U'_{\mathbb{F}}$. Likewise, the third generator of U is expanded and echelon reduced to give the rest of the basis for $U'_{\mathbb{F}}$. To find $V'_{\mathbb{F}}$, the first two generators are expanded, semi-echelon reduced and the rows with heads in the middle block kept. The third generator can be ignored. Finally, the intersection of $U'_{\mathbb{F}}$ and $V'_{\mathbb{F}}$ can found using, for example, `SumIntersectionMatDestructive` (**HAPprime Datatypes: SumIntersectionMatDestructive**).

This algorithm, implemented in the function `IntersectionModulesGF` (5.7.3), will (for modules whose generators have zero columns) use less memory than a full vector-space expansion, and in the case where U and V have no intersection, may need to perform no expansion at all. In the worst case, both U and V will need a full expansion, using no more memory than the naive implementation. Since any full expansion is done row-by-row, with echelon reduction each time, it will in general still require less memory (but will be slower).

5.3 Construction functions

5.3.1 FpGModuleGF construction functions

◊ `FpGModuleGF(gens, G[, action, actionBlockSize])` (operation)
◊ `FpGModuleGF(gens, groupAndAction)` (operation)
◊ `FpGModuleGF(ambientDimension, G[, action, actionBlockSize])` (operation)
◊ `FpGModuleGF(ambientDimension, groupAndAction)` (operation)
◊ `FpGModuleGF(G, n)` (operation)
◊ `FpGModuleGF(groupAndAction, n)` (operation)
◊ `FpGModuleGF(HAPmod)` (operation)
◊ `FpGModuleGFNC(gens, G, form, action, actionBlockSize)` (operation)
◊ `FpGModuleGFNC(ambientDimension, G, action, actionBlockSize)` (operation)
◊ `FpGModuleGFNC(gens, groupAndAction[, form])` (operation)

Returns: FpGModuleGF

Creates and returns a `FpGModuleGF` module object. The most commonly-used constructor requires a list of generators *gens* and a group *G*. The group action and block size can be specified using the *action* and *actionBlockSize* parameters, or if these are omitted then the canonical action is assumed. These parameters can also be wrapped up in a `groupAndAction` record (see 5.1).

An empty FpGModuleGF can be constructed by specifying a group and an *ambientDimension* instead of a set of generators. A module spanning $(\mathbb{F}G)^n$ with canonical generators and action can be constructed by giving a group *G* and a rank *n*. A FpGModuleGF can also be constructed from a HAP FpGModule *HAPmod*.

The generators in a FpGModuleGF do not need to be a minimal set. If you wish to create a module with minimal generators, construct the module from a non-minimal set *gens* and then use one of the MinimalGeneratorsModule functions (5.5.9). When constructing a FpGModuleGF from a FpGModule, the HAP function GeneratorsOfFpGModule (**HAP: GeneratorsOfFpGModule**) is used to provide a set of generators, so in this case the generators will be minimal.

Most of the forms of this function perform a few (cheap) tests to make sure that the arguments are self-consistent. The NC versions of the constructors are provided for internal use, or when you know what you are doing and wish to skip the tests. See Section 5.3.5 below for an example of usage.

5.3.2 FpGModuleFromFpGModuleGF

◊ FpGModuleFromFpGModuleGF(*M*) (operation)

Returns: FpGModule

Returns a HAP FpGModule that represents the same module as the FpGModuleGF *M*. This uses ModuleVectorSpaceBasis (5.5.7) to find the vector space basis for *M* and constructs a FpGModule with this vector space and the same group and action as *M* See Section 5.3.5 below for an example of usage.

TODO: This currently constructs an FpGModule object explicitly. It should use a constructor once one is provided

5.3.3 MutableCopyModule

◊ MutableCopyModule(*M*) (operation)

Returns: FpGModuleGF

Returns a copy of the module *M* where the generating vectors are fully mutable. The group and action in the new module is identical to that in *M* - only the list of generators is copied and made mutable. (The assumption is that this function used in situations where you just want a new generating set.)

5.3.4 CanonicalAction

◊ CanonicalAction(*G*) (attribute)
◊ CanonicalActionOnRight(*G*) (attribute)
◊ CanonicalGroupAndAction(*G*) (attribute)

Returns: Function action(g,v) or a record with elements (group, action, actionOnRight, actionBlockSize)

Returns a function of the form action(g,v) that performs the canonical group action of an element g of the group *G* on a vector v (acting on the left by default, or on the right with the OnRight version). The GroupAndAction version of this function returns the actions in a record together with the group and the action block size. Under the canonical action, a free module $\mathbb{F}G$ is represented as a vector of length $|G|$ over the field $\mathbb{F}$, and the action is a permutation of the vector elements.

Note that these functions are attributes of a group, so that the canonical action for a particular group object will always be an identical function (which is desirable for comparing and combining

modules and submodules).

5.3.5 Example: Constructing a `FpGModuleGF`

The example below constructs four different $\mathbb{F}G$-modules, where G is the quaternion group of order eight, and the action is the canonical action in each case:

1. `M` is the module $(\mathbb{F}G)^3$
2. `S` is the submodule of $(\mathbb{F}G)^3$ with elements only in the first summand
3. `T` is a random submodule `M` generated by five elements
4. `U` is the trivial (zero) submodule of $(\mathbb{F}G)^4$

We check whether `S`, `T` and `U` are submodules of `M`, examine a random element from `M`, and convert `S` into a HAP `FpGModule`. For the other functions used in this example, see Section 5.8.

Example

```
gap> G := SmallGroup(8, 4);;
gap> M := FpGModuleGF(G, 3);
Full canonical module FGˆ3 over the group ring of <pc group of size 8 with
3 generators> in characteristic 2
gap> gen := ListWithIdenticalEntries(24, Zero(GF(2)));;
gap> gen[1] := One(GF(2));;
gap> S := FpGModuleGF([gen], G);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 1 generator in FGˆ
3. Generators are in minimal echelon form.
gap> T := RandomSubmodule(M, 5);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 5 generators in FGˆ3.
gap> U := FpGModuleGF(32, CanonicalGroupAndAction(G));
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 0 generators in FGˆ
4. Generators are in minimal echelon form.
gap>
gap> IsSubModule(M, S);
true
gap> IsSubModule(M, T);
true
gap> IsSubModule(M, U);
false
gap>
gap> e := RandomElement(M);
<a GF2 vector of length 24>
gap> Display([e]);
 . 1 1 . . 1 . . . . . 1 . . 1 1 . . 1 . 1 . . 1
gap> IsModuleElement(S, e);
false
gap> IsModuleElement(T, e);
true
gap>
gap> FpGModuleFromFpGModuleGF(S);
```

```
Module of dimension 8 over the group ring of <pc group of size 8 with
3 generators> in characteristic 2
```

5.4 Data access functions

5.4.1 ModuleGroup

◊ `ModuleGroup(M)` (operation)

Returns: Group

Returns the group of the module `M`. See Section 5.4.11 below for an example of usage.

5.4.2 ModuleGroupOrder

◊ `ModuleGroupOrder(M)` (operation)

Returns: Integer

Returns the order of the group of the module `M`. This function is identical to `Order(ModuleGroup(M))`, and is provided for convenience. See Section 5.4.11 below for an example of usage.

5.4.3 ModuleAction

◊ `ModuleAction(M)` (operation)

Returns: Function

Returns the group action function of the module `M`. This is a function `action(g, v)` that takes a group element `g` and a vector `v` and returns a vector `w` that is the result of $w = gv$. See Section 5.4.11 below for an example of usage.

5.4.4 ModuleActionBlockSize

◊ `ModuleActionBlockSize(M)` (operation)

Returns: Integer

Returns the block size of the group action of the module `M`. This is the length of vectors (or the factor of the length) upon which the group action function acts. See Section 5.4.11 below for an example of usage.

5.4.5 ModuleGroupAndAction

◊ `ModuleGroupAndAction(M)` (operation)

Returns: Record with elements `(group, action, actionOnRight, actionBlockSize)`

Returns details of the module's group and action in a record with the following elements:

- `group` The module's group
- `action` The module's group action, as a function of the form `action(g, v)` that takes a vector `v` and returns the vector $w = gv$
- `actionOnRight` The module's group action when acting on the right, as a function of the form `action(g, v)` that takes a vector `v` and returns the vector $w = vg$

- `actionBlockSize` The module's group action block size. This is the ambient dimension of vectors in the module $\mathbb{F}G$

This function is provided for convenience, and is used by a number of internal functions. The canonical groups and action can be constructed using the function `CanonicalGroupAndAction` (5.3.4). See Section 5.4.11 below for an example of usage.

5.4.6 ModuleCharacteristic

◊ `ModuleCharacteristic(M)` (operation)

Returns: Integer

Returns the characteristic of the field $\mathbb{F}$ of the $\mathbb{F}G$-module `M`. See Section 5.4.11 below for an example of usage.

5.4.7 ModuleField

◊ `ModuleField(M)` (operation)

Returns: Field

Returns the field $\mathbb{F}$ of the $\mathbb{F}G$-module `M`. See Section 5.4.11 below for an example of usage.

5.4.8 ModuleAmbientDimension

◊ `ModuleAmbientDimension(M)` (operation)

Returns: Integer

Returns the ambient dimension of the module `M`. The module `M` is represented as a submodule of $\mathbb{F}G^n$ using generating vectors for a vector space. This function returns the dimension of this underlying vector space. This is equal to the length of each generating vector, and also $n \times$`actionBlockSize`. See Section 5.4.11 below for an example of usage.

5.4.9 AmbientModuleDimension

◊ `AmbientModuleDimension(M)` (operation)

Returns: Integer

The module `M` is represented a submodule embedded within the free module $\mathbb{F}G^n$. This function returns n, the dimension of the ambient module. This is the same as the number of blocks. See Section 5.4.11 below for an example of usage.

5.4.10 DisplayBlocks (for FpGModuleGF)

◊ `DisplayBlocks(M)` (operation)

Returns: nothing

Displays the structure of the module generators `gens` in a compact human-readable form. Since the group action permutes generating vectors in blocks of length `actionBlockSize`, any block that contains non-zero elements will still contain non-zero elements after the group action, but a block that is all zero will remain all zero. This operation displays the module generators in a per-block form, with a `*` where the block is non-zero and `.` where the block is all zero.

The standard GAP methods `View` (**Reference: View**), `Print` (**Reference: Print**) and `Display` (**Reference: Display**) are also available.) See Section 5.6.3 below for an example of usage.

5.4.11 Example: Accessing data about a FpGModuleGF

In the following example, we construct three terms of a (minimal) resolution of the dihedral group of order eight, which is a chain complex of $\mathbb{F}G$-modules.

$$(\mathbb{F}G)^3 \to (\mathbb{F}G)^3 \to \mathbb{F}G \to \mathbb{F} \to 0$$

We obtain the last homomorphism in this chain complex and calculate its kernel, returning this as a FpGModuleGF. We can use the data access functions described above to extract information about this module.

See Chapters 6 and 2 respectively for more information about $\mathbb{F}G$-module homomorphisms and resolutions in HAPprime

Example

```
gap> R := ResolutionPrimePowerGroupRadical(DihedralGroup(8), 2);
Resolution of length 2 in characteristic 2 for <pc group of size 8 with
3 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> phi := BoundaryFpGModuleHomomorphismGF(R, 2);
<Module homomorphism>

gap> M := KernelOfModuleHomomorphism(phi);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 15 generators in FG^3.

gap> # Now find out things about this module M
gap> ModuleGroup(M);
<pc group of size 8 with 3 generators>
gap> ModuleGroupOrder(M);
8
gap> ModuleAction(M);
function( g, v ) ... end
gap> ModuleActionBlockSize(M);
8
gap> ModuleGroupAndAction(M);
rec( group := <pc group of size 8 with 3 generators>,
  action := function( g, v ) ... end,
  actionOnRight := function( g, v ) ... end, actionBlockSize := 8 )
gap> ModuleCharacteristic(M);
2
gap> ModuleField(M);
GF(2)
gap> ModuleAmbientDimension(M);
24
gap> AmbientModuleDimension(M);
3
```

5.5 Generator and vector space functions

5.5.1 ModuleGenerators

◊ ModuleGenerators(*M*) (operation)

Returns: List of vectors

Returns, as the rows of a matrix, a list of the set of currently-stored generating vectors for the vector space of the module *M*. Note that this set is not necessarily minimal. The function ModuleGeneratorsAreMinimal (5.5.2) will return true if the set is known to be minimal, and the MinimalGeneratorsModule functions (5.5.9) can be used to ensure a minimal set, if necessary. See Section 5.5.11 below for an example of usage.

5.5.2 ModuleGeneratorsAreMinimal

◊ ModuleGeneratorsAreMinimal(*M*) (operation)

Returns: Boolean

Returns true if the module generators are known to be minimal, or false otherwise. Generators are known to be minimal if the one of the MinimalGeneratorsModule functions (5.5.9) have been previously used on this module, or if the module was created from a HAP FpGModule. See Section 5.5.11 below for an example of usage.

5.5.3 ModuleGeneratorsAreEchelonForm

◊ ModuleGeneratorsAreEchelonForm(*M*) (operation)

Returns: Boolean

Return true if the module generators are known to be in echelon form, or (i.e. EchelonModuleGenerators (5.6.1) has been called for this module), or false otherwise. Some algorithms work more efficiently if (or require that) the generators of the module are in block-echelon form, i.e. each generator in the module's list of generators has its first non-zero block in the same location or later than the generator before it in the list. See Section 5.5.11 below for an example of usage.

5.5.4 ModuleIsFullCanonical

◊ ModuleIsFullCanonical(*M*) (operation)

Returns: Boolean

Returns true if the module is known to represent the full module $\mathbb{F}G^n$, with canonical generators and group action, or false otherwise. A module is only known to be canonical if it was constructed using the canonical module FpGModuleGF constructor (FpGModuleGF (5.3.1)). If this is true, the module is displayed in a concise form, and some functions have a trivial implementation. See Section 5.5.11 below for an example of usage.

5.5.5 ModuleGeneratorsForm

◊ ModuleGeneratorsForm(*M*) (operation)

Returns: String

Returns a string giving the form of the module generators. This may be one of the following:

- "unknown" The form is not known

- "minimal" The generators are known to be minimal, but not in any particular form
- "fullcanonical" The generators are the canonical (and minimal) generators for $\mathbb{F}G^n$
- "semiechelon" The generators are minimal and in semi-echelon form.
- "echelon" The generators are minimal and in echelon form

See Section 5.5.11 below for an example of usage.

5.5.6 ModuleRank

◊ ModuleRank(*M*) (operation)
◊ ModuleRankDestructive(*M*) (operation)

Returns: Integer

Returns the rank of the module *M*, i.e. the number of minimal generators. If the module is already in minimal form (according to ModuleGeneratorsAreMinimal (5.5.2)) then the number of generators is returned with no calculation. Otherwise, MinimalGeneratorsModuleGF (5.5.9) or MinimalGeneratorsModuleGFDestructive (5.5.9) respectively are used to find a set of minimal generators. See Section 5.5.11 below for an example of usage.

5.5.7 ModuleVectorSpaceBasis

◊ ModuleVectorSpaceBasis(*M*) (operation)

Returns: List of vectors

Returns a matrix whose rows are a basis for the vector space of the FpGModuleGF module *M*. Since FpGModuleGF stores modules as a minimal *G*-generating set, this function has to calculate all *G*-multiples of this generating set and row-reduce this to find a basis. See Section 5.5.11 below for an example of usage.

TODO: A GF version of this one

5.5.8 ModuleVectorSpaceDimension

◊ ModuleVectorSpaceDimension(*M*) (operation)

Returns: Integer

Returns the dimension of the vector space of the module *M*. Since FpGModuleGF stores modules as a minimal *G*-generating set, this function has to calculate all *G*-multiples of this generating set and row-reduce this to find the size of the basis. See Section 5.5.11 below for an example of usage.

TODO: A GF version of this function

5.5.9 MinimalGeneratorsModule

◊ MinimalGeneratorsModuleGF(*M*) (operation)
◊ MinimalGeneratorsModuleGFDestructive(*M*) (operation)
◊ MinimalGeneratorsModuleRadical(*M*) (operation)

Returns: FpGModuleGF

Returns a module equal to the FpGModuleGF *M*, but which has a minimal set of generators. Two algorithms are provided:

- The two GF versions use EchelonModuleGenerators (5.6.1) and EchelonModuleGeneratorsDestructive (5.6.1) respectively. In characteristic two, these return a set of minimal generators, and use less memory than the Radical version, but take longer. If the characteristic is not two, these functions revert to MinimalGeneratorsModuleRadical.
- The Radical version uses the radical of the module in a manner similar to the function **HAP: GeneratorsOfFpGModule**. This is much faster, but requires a considerable amount of temporary storage space.

See Section 5.5.11 below for an example of usage.

5.5.10 RadicalOfModule

◊ RadicalOfModule(*M*) (operation)

Returns: FpGModuleGF

Returns radical of the FpGModuleGF module *M* as another FpGModuleGF. The radical is the module generated by the vectors $v - gv$ for all v in the set of generating vectors for M and all g in a set of generators for the module's group.

The generators for the returned module will not be in minimal form: the MinimalGeneratorsModule functions (5.5.9) can be used to convert the module to a minimal form if necessary. See Section 5.5.11 below for an example of usage.

5.5.11 Example: Generators and basis vectors of a FpGModuleGF

Starting with the same module as in the earlier example (Section 5.4.11), we now investigate the generators of the module M. The generating vectors (of which there are 15) returned by the function KernelOfModuleHomomorphism (6.6.3) are not a minimal set, but the function MinimalGeneratorsModuleGF (5.5.9) creates a new object N representing the same module, but now with only four generators. The vector space spanned by these generators has 15 basis vectors, so representing the module by a G-generating set instead is much more efficient. (The original generating set in M was in fact an $\mathbb{F}$-basis, so the dimension of the vector space should come as no surprise.)

We can also find the radical of the module, and this is used internally for the faster, but more memory-hungry, MinimalGeneratorsModuleRadical (5.5.9).

Example

```
gap> R := ResolutionPrimePowerGroupRadical(DihedralGroup(8), 2);;
gap> phi := BoundaryFpGModuleHomomorphismGF(R, 2);;
gap> M := KernelOfModuleHomomorphism(phi);;
gap> #
gap> ModuleGenerators(M);
[ <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24> ]
gap> ModuleGeneratorsAreMinimal(M);
false
gap> ModuleGeneratorsForm(M);
```

```
"unknown"
gap> #
gap> N := MinimalGeneratorsModuleGF(M);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 4 generators in FG^
3. Generators are in minimal echelon form.

gap> M = N;    # Check that the new module spans the same space
true
gap> ModuleGeneratorsAreEchelonForm(N);
true
gap> ModuleIsFullCanonical(N);
false
gap> M = N;
true
gap> ModuleVectorSpaceBasis(N);
[ <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24> ]
gap> ModuleVectorSpaceDimension(N);
15
gap> #
gap> N2 := MinimalGeneratorsModuleRadical(M);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 4 generators in FG^
3. Generators are minimal.

gap> #
gap> R := RadicalOfModule(M);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 120 generators in FG^3.

gap> N2 = N;
true
```

5.6 Block echelon functions

5.6.1 EchelonModuleGenerators

◊ EchelonModuleGenerators(*M*) (operation)
◊ EchelonModuleGeneratorsDestructive(*M*) (operation)
◊ SemiEchelonModuleGenerators(*M*) (operation)
◊ SemiEchelonModuleGeneratorsDestructive(*M*) (operation)
◊ EchelonModuleGeneratorsMinMem(*M*) (operation)
◊ EchelonModuleGeneratorsMinMemDestructive(*M*) (operation)
◊ SemiEchelonModuleGeneratorsMinMem(*M*) (operation)

◊ SemiEchelonModuleGeneratorsMinMemDestructive(*M*) (operation)

Returns: Record (module, headblocks)

Returns a record with two components:

- module A module whose generators span the same vector space as that of the input module *M*, but whose generators are in a block echelon (or semi-echelon) form
- headblocks A list giving, for each generating vector in *M*, the block in which the head for that generating row lies

In block-echelon form. each generator is row-reduced using *G*g-multiples of the other other generators to produce a new, equivalent generating set where the first non-zero block in each generator is as far to the right as possible. The resulting form, with many zero blocks, can allow more memory-efficient operations on the module. See Section 5.2 for details. In addition, the standard (non-MinMem) form guarantees that the set of generators are minimal in the sense that no generator can be removed from the set while leaving the spanned vector space the same. In the GF(2) case, this is the global minimum.

Several versions of this algorithm are provided. The SemiEchelon versions of these functions do not guarantee a particular generator ordering, while the Echelon versions sort the generators into order of increasing initial zero blocks. The non-Destructive versions of this function return a new module and do not modify the input module; the Destructive versions change the generators of the input module in-place, and return this module. All versions are memory-efficient, and do not need a full vector-space basis. The MinMem versions are guaranteed to expand at most two generators at any one time, while the standard version may, in the (unlikely) worst-case, need to expand half of the generating set. As a result of this difference in the algorithm, the MinMem version is likely to return a greater number of generators (which will not be minimal), but those generators typically have a greater number of zero blocks after the first non-zero block in the generator. The MinMem operations are currently only implemented for GF(2) modules. See Section 5.6.3 below for an example of usage.

5.6.2 ReverseEchelonModuleGenerators

◊ ReverseEchelonModuleGenerators(*M*) (operation)

◊ ReverseEchelonModuleGeneratorsDestructive(*M*) (operation)

Returns: FpGModuleGF

Returns an FpGModuleGF module whose vector space spans the same space as the input module *M*, but whose generating vectors are modified to try to get as many zero blocks as possible at the end of each vector. This function performs echelon reduction of generating rows in a manner similar to EchelonModuleGenerators (5.6.1), but working from the bottom upwards. It is guaranteed that this function will not change the head block (the location of the first non-zero block) in each generating row, and hence if the generators are already in an upper-triangular form (e.g. following a call to EchelonModuleGenerators (5.6.1)) then it will not disturb that form and the resulting generators will be closer to a diagonal form.

The Destructive version of this function modifies the input module's generators in-place and then returns that module; the non-Destructive version works on a copy of the input module and so will not modify the original module.

This operation is currently only implemented for GF(2) modules. See Section 5.5.11 below for an example of usage.

5.6.3 Example: Converting a FpGModuleGF to block echelon form

We can construct a larger module than in the earlier examples (Sections 5.4.11 and 5.5.11) by taking the kernel of the third boundary homomorphism of a minimal resolution of a group of order 32, which as returned by the function KernelOfModuleHomomorphism (6.6.3) has a generating set with many redundant generators. We display the block structure of the generators of this module after applying various block echelon reduction functions.

Example

```
gap> R := ResolutionPrimePowerGroupRadical(SmallGroup(32, 10), 3);;
gap> phi := BoundaryFpGModuleHomomorphismGF(R, 3);;
gap> #
gap> M := KernelOfModuleHomomorphism(phi);
Module over the group ring of <pc group of size 32 with
5 generators> in characteristic 2 with 65 generators in FG^4.

gap> #
gap> N := SemiEchelonModuleGenerators(M);
rec( module := Module over the group ring of <pc group of size 32 with
    5 generators> in characteristic 2 with 5 generators in FG^
    4. Generators are in minimal semi-echelon form.
    , headblocks := [ 2, 3, 1, 1, 3 ] )
gap> DisplayBlocks(N.module);
Module over the group ring of Group( [ f1, f2, f3, f4, f5 ] )
 in characteristic 2 with 5 generators in FG^4.
[.*.*]
[..**]
[***.]
[****]
[..**]
Generators are in minimal semi-echelon form.
gap> N2 := SemiEchelonModuleGeneratorsMinMem(M);
rec( module := Module over the group ring of <pc group of size 32 with
    5 generators> in characteristic 2 with 9 generators in FG^4.
    , headblocks := [ 2, 1, 3, 1, 1, 4, 1, 3, 4 ] )
gap> DisplayBlocks(N2.module);
Module over the group ring of Group( [ f1, f2, f3, f4, f5 ] )
 in characteristic 2 with 9 generators in FG^4.
[.*..]
[**..]
[..**]
[****]
[****]
[...*]
[****]
[..**]
[...*]

gap> #
gap> EchelonModuleGeneratorsDestructive(M);;
gap> DisplayBlocks(M);
Module over the group ring of Group( [ f1, f2, f3, f4, f5 ] )
 in characteristic 2 with 5 generators in FG^4.
[***.]
```

```
[****]
[.*.*]
[..**]
[..**]
Generators are in minimal echelon form.
gap> ReverseEchelonModuleGeneratorsDestructive(M);
Module over the group ring of <pc group of size 32 with
5 generators> in characteristic 2 with 5 generators in FG^
4. Generators are in minimal echelon form.

gap> DisplayBlocks(M);
Module over the group ring of Group( [ f1, f2, f3, f4, f5 ] )
 in characteristic 2 with 5 generators in FG^4.
[***.]
[****]
[.*..]
[..*.]
[..**]
Generators are in minimal echelon form.
```

5.7 Sum and intersection functions

5.7.1 DirectSumOfModules

◊ DirectSumOfModules(*M*, *N*) (operation)
◊ DirectSumOfModules(*coll*) (operation)
◊ DirectSumOfModules(*M*, *n*) (operation)

Returns: FpGModule

Returns the FpGModuleGF module that is the direct sum of the specified modules (which must have a common group and action). The input can be either two modules *M* and *N*, a list of modules *coll*, or one module *M* and an exponent *n* specifying the number of copies of *M* to sum. See Section 5.7.5 below for an example of usage.

If the input modules all have minimal generators and/or echelon form, the construction of the direct sum guarantees that the output module will share the same form.

5.7.2 DirectDecompositionOfModule

◊ DirectDecompositionOfModule(*M*) (operation)
◊ DirectDecompositionOfModuleDestructive(*M*) (operation)

Returns: List of FpGModuleGFs

Returns a list of FpGModuleGFs whose direct sum is equal to the input FpGModuleGF module *M*. The list may consist of one element: the original module.

This function relies on the block structure of a set of generators that have been converted to both echelon and reverse-echelon form (see EchelonModuleGenerators (5.6.1) and ReverseEchelonModuleGenerators (5.6.2)), and calls these functions if the module is not already in echelon form. In this form, it can be possible to trivially identify direct summands. There is no guarantee either that this function will return a decomposition if one is available, or that the modules

returned in a decomposition are themselves indecomposable. See Section 5.7.5 below for an example of usage.

The Destructive version of this function uses the Destructive versions of the echelon functions, and so modifies the input module and returns modules who share generating rows with the modified *M*. The non-Destructive version operates on a copy of *M*, and returns modules with unique rows.

5.7.3 IntersectionModules

◊ IntersectionModules(*M*, *N*) (operation)
◊ IntersectionModulesGF(*M*, *N*) (operation)
◊ IntersectionModulesGFDestructive(*M*, *N*) (operation)
◊ IntersectionModulesGF2(*M*, *N*) (operation)

Returns: FpGModuleGF

Returns the FpGModuleGF module that is the intersection of the modules *M* and *N*. This function calculates the intersection using vector space methods (i.e. SumIntersectionMatDestructive (**HAPprime Datatypes: SumIntersectionMatDestructive**)). The standard version works on the complete vector space bases of the input modules. The GF version considers the block structure of the generators of *M* and *N* and only expands the necessary rows and blocks. This can lead to a large saving and memory if *M* and *N* are in echelon form and have a small intersection. See Section 5.2.4 for details. See Section 5.7.5 below for an example of usage. The GF2 version computes the intersection by a *G*-version of the standard vector space algorithm, using EchelonModuleGenerators (5.6.1) to perform echelon reduction on an augmented set of generators. This is much slower than the GF version, but may use less memory.

The vector spaces in FpGModuleGFs are assumed to all be with respect to the same canonical basis, so it is assumed that modules are compatible if they have the same group and the same ambient dimension.

The Destructive version of the GF function corrupts or permutes the generating vectors of *M* and *N*, leaving it invalid; the non-destructive version operates on copies of them, leaving the original modules unmodified. The generating vectors in the module returned by this function are in fact also a *vector space* basis for the module, so will not be minimal. The returned module can be converted to a set of minimal generators using one of the MinimalGeneratorsModule functions (5.5.9).

This operation is currently only defined for GF(2).

5.7.4 SumModules

◊ SumModules(*M*, *N*) (operation)

Returns: FpGModuleGF

Returns the FpGModuleGF module that is the sum of the input modules *M* and *N*. This function simply concatenates the generating vectors of the two modules and returns the result. If a set of minimal generators are needed then use one of the MinimalGeneratorsModule functions (5.5.9) on the result. See Section 5.7.5 below for an example of usage.

The vector spaces in FpGModuleGF are assumed to all be with respect to the same canonical basis, so it is assumed that modules are compatible if they have the same group and the same ambient dimension.

5.7.5 Example: Sum and intersection of FpGModuleGFs

We can construct the direct sum of $\mathbb{F}G$-modules, and (attempt to) calculate a direct decomposition of a module. For example, we can show that

$$(\mathbb{F}G)^2 \oplus \mathbb{F}G = \mathbb{F}G \oplus \mathbb{F}G \oplus \mathbb{F}G$$

Example

```
gap> G := CyclicGroup(64);;
gap> FG := FpGModuleGF(G, 1);
Full canonical module FG^1 over the group ring of <pc group of size 64 with
6 generators> in characteristic 2

gap> FG2 := FpGModuleGF(G, 2);
Full canonical module FG^2 over the group ring of <pc group of size 64 with
6 generators> in characteristic 2

gap> M := DirectSumOfModules(FG2, FG);
Full canonical module FG^3 over the group ring of <pc group of size 64 with
6 generators> in characteristic 2

gap> DirectDecompositionOfModule(M);
[ Module over the group ring of <pc group of size 64 with
    6 generators> in characteristic 2 with 1 generator in FG^
    1. Generators are in minimal echelon form.
    , Module over the group ring of <pc group of size 64 with
    6 generators> in characteristic 2 with 1 generator in FG^
    1. Generators are in minimal echelon form.
    , Module over the group ring of <pc group of size 64 with
    6 generators> in characteristic 2 with 1 generator in FG^
    1. Generators are in minimal echelon form.
     ]
```

We can also compute the sum and intersection of $\mathbb{F}G$-modules. In the example below we construct two submodules of $\mathbb{F}G$, where G is the dihedral group of order four: `M` is the submodule generated by g_1+g_2, and `N` is the submodule generated by $g_1+g_2+g_3+g_4$. We calculate their sum and intersection. Since `N` is in this case a submodule of `M` it is easy to check that the correct results are obtained.

Example

```
gap> G := DihedralGroup(4);;
gap> M := FpGModuleGF([[1,1,0,0]]*One(GF(2)), G);
Module over the group ring of <pc group of size 4 with
2 generators> in characteristic 2 with 1 generator in FG^
1. Generators are in minimal echelon form.

gap> N := FpGModuleGF([[1,1,1,1]]*One(GF(2)), G);
Module over the group ring of <pc group of size 4 with
2 generators> in characteristic 2 with 1 generator in FG^
1. Generators are in minimal echelon form.

gap> #
gap> S := SumModules(M,N);
Module over the group ring of <pc group of size 4 with
2 generators> in characteristic 2 with 2 generators in FG^1.
```

```
gap> I := IntersectionModules(M,N);
Module over the group ring of <pc group of size 4 with
2 generators> in characteristic 2 with 1 generator in FG^1.

gap> #
gap> S = M and I = N;
true
```

5.8 Miscellaneous functions

5.8.1 = (for FpGModuleGF)

◊ =(*M*, *N*) (operation)

Returns: Boolean

Returns `true` if the modules are equal, `false` otherwise. This checks that the groups and actions in each module are equal (i.e. identical), and that the vector space spanned by the two modules are the same. (All vector spaces in `FpGModuleGF`s of the same ambient dimension are assumed to be embedded in the same canonical basis.) See Section 5.5.11 above for an example of usage.

5.8.2 IsModuleElement

◊ `IsModuleElement`(*M*, *elm*) (operation)

◊ `IsModuleElement`(*M*, *coll*) (operation)

Returns: Boolean

Returns `true` if the vector *elm* can be interpreted as an element of the module *M*, or `false` otherwise. If a collection of elements is given as the second argument then a list of responses is returned, one for each element in the collection. See Section 5.3.5 above for an example of usage.

5.8.3 IsSubModule

◊ `IsSubModule`(*M*, *N*) (operation)

Returns: Boolean

Returns `true` if the module *N* is a submodule of *M*. This checks that the modules have the same group and action, and that the generators for module *N* are elements of the module *M*. (All vector spaces in `FpGModuleGF`s of the same ambient dimension are assumed to be embedded in the same canonical basis.) See Section 5.3.5 above for an example of usage.

5.8.4 RandomElement

◊ `RandomElement`(*M*[, *n*]) (operation)

Returns: Vector

Returns a vector which is a random element from the module *M*. If a second argument, *n* is give, then a list of *n* random elements is returned. See Section 5.3.5 above for an example of usage.

5.8.5 Random Submodule

◊ RandomSubmodule(*M, ngens*) (operation)

Returns: FpGModuleGF

Returns a FpGModuleGF module that is a submodule of *M*, with *ngens* generators selected at random from elements of *M*. These generators are not guaranteed to be minimal, so the rank of the submodule will not necessarily be equal to *ngens*. If a module with minimal generators is required, the MinimalGeneratorsModule functions (5.5.9) can be used to convert the module to a minimal form See Section 5.3.5 above for an example of usage.

Chapter 6

$\mathbb{F}G$-module homomorphisms

6.1 The `FpGModuleHomomorphismGF` datatype

Linear homomorphisms between free $\mathbb{F}G$-modules (as `FpGModuleGF` objects - see Chapter 5) are represented in HAPprime using the `FpGModuleHomomorphismGF` datatype. This represents module homomorphisms in a similar manner to $\mathbb{F}G$-modules, using a set of generating vectors, in this case vectors that generate the images in the target module of the generators of the source module.

Three items need to be passed to the constructor function `FpGModuleHomomorphismGF` (6.4.1):

- `source` the source `FpGModuleGF` module for the homomorphism
- `target` the target `FpGModuleGF` module for the homomorphism
- `gens` a list of vectors that are the images (in `target`) under the homomorphisms of each of the generators stored in `source`

6.2 Calculating the kernel of a $\mathbb{F}G$-module homorphism by splitting into two homomorphisms

HAPprime represents a homomorphism between two $\mathbb{F}G$-modules as a list of generators for the image of the homomorphism. Each generator is given as an element in the target module, represented as a vector in the same manner as used in the `FpGModuleGF` datatype (see Chapter 5). Given a set of such generating vectors, an $\mathbb{F}$-generating set for the image of the homomorphism (as elements of the target module's vector space) is given by taking all G-multiples of the generators. Writing the vectors in this expanded set as a matrix, the kernel of the boundary homomorphism is the (left) null-space of this matrix. As with `FpGModuleGF`s, the block structure of the generating vectors (see Section 5.2.1) allows this null-space to be calculated without necessarily expanding the whole matrix.

This basic algorithm is implemented in the HAPprime function `KernelOfModuleHomomorphismSplit` (6.6.3). The generating vectors for a module homomorphism H are divided in half, with the homomorphism generated by the first half of the generating vectors being called U and that by the second half being called V. Given this partition the kernel of H can be defined as

$$ker(H) = preim_U(I) \cap [-preim_V(I)]$$

where

- $I = im(U) \cap im(V)$ is the intersection of the images of the two homomorphisms U and V
- $preim_U(I)$ the set of all preimages of I under U
- $preim_V(I)$ the set of all preimages of I under V

Rather than computing the complete set of preimages, instead the implementation takes a preimage representative of each generator for I and adds the kernel of the homomorphisms U and V. The means that instead of calculating the null-space of the full expanded matrix, we can compute the answer by calculating the kernels of two homomorphisms with fewer generators, as well as the intersection of two modules, and some preimage representatives. Each of these operations takes less memory than the naive null-space calculation. The intersection of two $\mathbb{F}G$-modules can be compactly calculated using the generators' block structure (see Section 5.2.4), while the kernels of U and V can be computed recursively using these same algorithm. The block structure can also help in calculating the preimage, but at a considerable cost in time, so this is not done. However, since U and V have fewer generators than the original homomorphism H, a space saving is still made.

In the case where the problem is seperable, i.e. a U and V can be found for which there is no intersection, this approach can give a large saving. The separable components of the homomorphism can be readily identified from the block structure of the generators (they are the rows which share no blocks or heads with other rows), and the kernels of these can be calculated independently, with no intersection to worry about. This is implemented in the alternative algorithm `KernelOfModuleHomomorphismIndependentSplit` (6.6.3).

6.3 Calculating the kernel of a $\mathbb{F}G$-module homorphism by column reduction and partitioning

The list of generators of the image of a $\mathbb{F}G$-module homomorphism can be interpreted as the rows of a matrix A with elements in $\mathbb{F}G$, and it is the kernel of this matrix which must be found (i.e. the solutions to $xA = 0$. If column reduction is performed on this matrix (by adding $\mathbb{F}G$-multiples of other columns to a column), the kernel is left unchanged, and this process can be performed to enable the kernel to be found by a recursive algorithm similar to standard back substitution methods.

Given the matrix $A = (a_{ij})$, take the $\mathbb{F}G$-module generated by the first row (a_{1j}) and find a minimal (or small) subset of elements $\{a_{1j}\}_{j\in J}$ that generate this module. Without altering the kernel, we can permute the columns of A such that $J = \{1 \dots t\}$. Taking $\mathbb{F}$ and G-multiples of these columns from the remaining columns, the first row of these columns can be reduced to zero, giving a new matrix A'. This matrix can be partitioned as follows:

$$\begin{pmatrix} B & 0 \\ C & D \end{pmatrix}$$

where B is $1 \times t$, C is $(m-1) \times t$ and D is $(m-1) \times (n-t)$. It is assumed that B and C are 'small' and operations on these can can be easily handled in memory using standard linear algebra, while D may still be large.

Taking the $\mathbb{F}G$-module generated by the t columns which form the BC partition of the matrix, we compute E, a set of minimal generators for the submodule of this which is zero in the first row. These are added as columns at the end of A', giving a matrix

$$\begin{pmatrix} B & 0 & 0 \\ C & D & E \end{pmatrix}$$

The kernel of this matrix can be shown to be

$$\begin{pmatrix} \ker B & 0 \\ L & \ker(DE) \end{pmatrix}$$

where

$$L = preim_B((\ker(DE))C)$$

The augmentation of D with E guarantees that this preimage always exists. Since B and C are small, both $\ker B$ and L are easy to compute using linear algebra, while $\ker(DE)$ can be computed by recursion.

Unfortunately, E can be large, and the accumulated increase of size of the matrix over many recursions negates the time and memory saving that this algorithm might be expected to give. Testing indicates that it is currently no faster than the `KernelOfModuleHomomorphismSplit` (6.6.3) method, and does not save much memory over the full expansion using linear algebra. An improved version of this algorithm would reduce E by D before augmentation, thus adding a smaller set of generators and restricting the explosion in size. If D were already in echelon form, this would also be time-efficient.

6.4 Construction functions

6.4.1 FpGModuleHomomorphismGF construction functions

◊ `FpGModuleHomomorphismGF(S, T, gens)` (operation)
◊ `FpGModuleHomomorphismGFNC(S, T, gens)` (operation)

Returns: `FpGModuleHomomorphismGF`

Creates and returns an `FpGModuleHomomorphismGF` module homomorphism object. This represents the homomorphism from the module `S` to the module `T` with a list of vectors `gens` whose rows are the images in `T` of the generators of `S`. The modules must (currently) be over the same group.

The standard constructor checks that the homomorphism is compatible with the modules, i.e. that the vectors in `gens` have the correct dimension and that they lie within the target module `T`. It also checks whether the generators of `S` are minimal. If they are not, then the homomorphism is created with a copy of `S` that has minimal generators (using `MinimalGeneratorsModuleRadical` (5.5.9)), and `gens` is also copied and converted to agree with the new form of `S`. If you wish to skip these checks then use the `NC` version of this function.

IMPORTANT: The generators of the module `S` and the generator matrix `gens` must be remain consistent for the lifetime of this homomorphism. If the homomorphism is constructed with a mutable source module or generator matrix, then you must be careful not to modify them while the homomorphism is needed.

6.4.2 Example: Constructing a `FpGModuleHomomorphismGF`

In this example we construct the module homomorphism $\phi : (\mathbb{F}G)^2 \to \mathbb{F}G$ which maps both generators of $(\mathbb{F}G)^2$ to the generator of $\mathbb{F}G$

Example

```
gap> G := SmallGroup(8, 4);;
gap> im := [1,0,0,0,0,0,0,0]*One(GF(2));
[ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ]
gap> phi := FpGModuleHomomorphismGF(
>              FpGModuleGF(G, 2),
```

```
>              FpGModuleGF(G, 1),
>              [im, im]);
<Module homomorphism>
```

6.5 Data access functions

6.5.1 SourceModule

◊ SourceModule(*phi*) (operation)

Returns: FpGModuleGF

Returns the source module for the homomorphism *phi*, as an FpGModuleGF.

6.5.2 TargetModule

◊ TargetModule(*phi*) (operation)

Returns: FpGModuleGF

Returns the targetmodule for the homomorphism *phi*, as an FpGModuleGF.

6.5.3 ModuleHomomorphismGeneratorMatrix

◊ ModuleHomomorphismGeneratorMatrix(*phi*) (operation)

Returns: List of vectors

Returns the generating vectors gens of the representation of the homomorphism *phi*. These vectors are the images in the target module of the generators of the source module.

6.5.4 DisplayBlocks (for FpGModuleHomomorphismGF)

◊ DisplayBlocks(*phi*) (method)

Returns: nothing

Prints a detailed description of the module in human-readable form, with the module generators and generator matrix shown in block form. The standard GAP methods View (**Reference: View**), Print (**Reference: Print**) and Display (**Reference: Display**) are also available.)

6.5.5 DisplayModuleHomomorphismGeneratorMatrix

◊ DisplayModuleHomomorphismGeneratorMatrix(*phi*) (method)

Returns: nothing

Prints a detailed description of the module homomorphism generating vectors gens in human-readable form. This is the display method used in the Display (**Reference: Display**) method for this datatype.

6.5.6 DisplayModuleHomomorphismGeneratorMatrixBlocks

◊ DisplayModuleHomomorphismGeneratorMatrixBlocks(*phi*) (method)

Returns: nothing

Prints a detailed description of the module homomorphism generating vectors gens in human-readable form. This is the function used in the DisplayBlocks (6.5.4) method.

6.5.7 Example: Accessing data about a FpGModuleHomomorphismGF

A free $\mathbb{F}G$ resolution is a chain complex of $\mathbb{F}G$-modules and homomorphisms, and the homomorphisms in a HAPResolution (see Chapter 2) can be extracted as a FpGModuleHomomorphismGF using the function BoundaryFpGModuleHomomorphismGF (2.4.6). We construct a resolution R and then examine the third resolution in the chain complex, which is a $\mathbb{F}G$-module homomorphism $d_3 : (\mathbb{F}G)^7 \to (\mathbb{F}G)^5$.

Example

```
gap> R := ResolutionPrimePowerGroupRadical(SmallGroup(64, 141), 3);;
#I  Dimension 2: rank 5
#I  Dimension 3: rank 7
gap> d3 := BoundaryFpGModuleHomomorphismGF(R, 3);;
gap> SourceModule(d3);
Full canonical module FGˆ7 over the group ring of <pc group of size 64 with
6 generators> in characteristic 2

gap> TargetModule(d3);
Full canonical module FGˆ5 over the group ring of <pc group of size 64 with
6 generators> in characteristic 2

gap> ModuleHomomorphismGeneratorMatrix(d3);
<an immutable 7x320 matrix over GF2>
gap> DisplayBlocks(d3);
Module homomorphism with source:
Full canonical module FGˆ7 over the group ring of Group(
[ f1, f2, f3, f4, f5, f6 ] )
 in characteristic 2

and target:
Full canonical module FGˆ5 over the group ring of Group(
[ f1, f2, f3, f4, f5, f6 ] )
 in characteristic 2

and generator matrix:
[*.*.*]
[*****]
[.**..]
[.**..]
[..**.]
[...**]
[...*.]
```

Note that the module homomorphism generating vectors in a resolution calculated using HAPprime are in block-echelon form (see Section 5.2). This makes it efficient to compute the kernel of this homomorphism using KernelOfModuleHomomorphismSplit (6.6.3), as described in Section 6.2, since there is only a small intersection between the images generated by the top and bottom halves of the generating vectors.

6.6 Image and kernel functions

6.6.1 ImageOfModuleHomomorphism

◊ ImageOfModuleHomomorphism(*phi*) (operation)
◊ ImageOfModuleHomomorphism(*phi, M*) (operation)
◊ ImageOfModuleHomomorphism(*phi, elm*) (operation)
◊ ImageOfModuleHomomorphism(*phi, coll*) (operation)
◊ ImageOfModuleHomomorphismDestructive(*phi, elm*) (operation)
◊ ImageOfModuleHomomorphismDestructive(*phi, coll*) (operation)

Returns: FpGModuleGF, vector or list of vectors depending on argument

For a module homomorphism *phi*, the one-argument function returns the module that is the image of the homomorphism, while the two-argument versions return the result of mapping of an FpGModuleGF *M*, a module element *elm* (given as a vector), or a collection of module elements *coll* through the homomorphism. This uses standard linear algebra to find the image of elements from the source module.

The Destructive versions of the function will corrupt the second parameter, which must be mutable as a result. The version of this operation that returns a module does not guarantee that the module will be in minimal form, and one of the MinimalGeneratorsModule functions (5.5.9) should be used on the result if a minimal set of generators is needed.

6.6.2 PreImageRepresentativeOfModuleHomomorphism

◊ PreImageRepresentativeOfModuleHomomorphism(*phi, elm*) (operation)
◊ PreImageRepresentativeOfModuleHomomorphism(*phi, coll*) (operation)
◊ PreImageRepresentativeOfModuleHomomorphism(*phi, M*) (operation)
◊ PreImageRepresentativeOfModuleHomomorphismGF(*phi, elm*) (operation)
◊ PreImageRepresentativeOfModuleHomomorphismGF(*phi, coll*) (operation)

For an element *elm* in the image of *phi*, this returns a representative of the set of preimages of *elm* under *phi*, otherwise it returns fail. If a list of vectors *coll* is provided then the function returns a list of preimage representatives, one for each element in the list (the returned list can contain fail entries if there are vectors with no solution). For an FpGModuleGF module *M*, this returns a module whose image under *phi* is *M* (or fail). The module returned will not necessarily have minimal generators, and one of the MinimalGeneratorsModule functions (5.5.9) should be used on the result if a minimal set of generators is needed.

The standard functions use linear algebra, expanding the generator matrix into a full matrix and using SolutionMat (**Reference: SolutionMat**) to calculate a preimage of *elm*. In the case where a list of vectors is provided, the matrix decomposition is only performed once, which can save significant time.

The GF versions of the functions can give a large memory saving when the generators of the homomorphism *phi* are in echelon form, and operate by doing back-substitution using the generator form of the matrices.

6.6.3 KernelOfModuleHomomorphism

◊ KernelOfModuleHomomorphism(*phi*) (operation)
◊ KernelOfModuleHomomorphismSplit(*phi*) (operation)

◊ KernelOfModuleHomomorphismIndependentSplit(*phi*) (operation)
◊ KernelOfModuleHomomorphismGF(*phi*) (operation)

Returns: FpGModuleGF

Returns the kernel of the module homomorphism *phi*, as an FpGModuleGF module. There are three independent algorithms for calculating the kernel, represented by different versions of this function:

- The standard version calculates the kernel by the obvious vector-space method. The homomorphism's generators are expanded into a full vector-space basis and the kernel of that vector space homomorphism is found. The generators of the returned module are in fact a vector space basis for the kernel module.
- The Split version divides the homomorphism into two (using the first half and the second half of the generating vectors), and uses the preimage of the intersection of the images of the two halves to calculate the kernel (see Section 6.2). If the generating vectors for *phi* are in block echelon form (see Section 5.2), then this approach provides a considerable memory saving over the standard approach.
- The IndependentSplit version splits the generating vectors into sets that generate vector spaces which have no intersection, and calculates the kernel as the sum of the kernels of those independent rows. If the generating vectors can be decomposed in this manner (i.e. the the generator matrix is in a diagonal form), this will provide a very large memory saving over the standard approach.
- The GF version performs column reduction and partitioning of the generator matrix to enable a recursive approach to computing the kernel (see Section 6.3). The level of partitioning is governed by the option MaxFGExpansionSize, which defaults to 10^9, allowing about 128Mb of memory to be used for standard linear algebra before partitioning starts. See (**Reference: Options Stack**) for details of using options in GAP

None of these basis versions of the functions guarantee to return a minimal set of generators, and one of the MinimalGeneratorsModule functions (5.5.9) should be used on the result if a minimal set of generators is needed. All of the functions leave the input homomorphism *phi* unchanged.

6.6.4 Example: Kernel and Image of a FpGModuleHomomorphismGF

A free $\mathbb{F}G$-resolution of a module is an exact sequence of module homomorphisms. In this example we use the functions ImageOfModuleHomomorphism (6.6.1) and KernelOfModuleHomomorphism (6.6.3) to check that one of the sequences in a resolution is exact, i.e. that in the sequence

$$M_3 \to M_2 \to M_1$$

the image of the first homomorphism $d_3 : M_3 \to M_2$ is the kernel of the second homomorphism $d_2 : M_2 \to M_1$

We also demonstrate that we can find the image and preimage of module elements under our module homomorphisms. We take an element e of M_2, in this case by taking the first generating element of the kernel of d_2, and map it up to M_3 and back.

Finally, we compute the kernel using the other available methods, and check that the results are the same.

Example

```
gap> R := ResolutionPrimePowerGroupRadical(SmallGroup(8, 3), 3);;
gap> d2 := BoundaryFpGModuleHomomorphismGF(R, 2);;
gap> d3 := BoundaryFpGModuleHomomorphismGF(R, 3);;
gap> #
gap> I := ImageOfModuleHomomorphism(d3);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 4 generators in FGˆ3.

gap> K := KernelOfModuleHomomorphism(d2);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 15 generators in FGˆ3.

gap> I = K;
true
gap> #
gap> e := ModuleGenerators(K)[1];;
gap> PreImageRepresentativeOfModuleHomomorphism(d3, e);
<a GF2 vector of length 32>
gap> f := PreImageRepresentativeOfModuleHomomorphism(d3, e);
<a GF2 vector of length 32>
gap> ImageOfModuleHomomorphism(d3, f);
<a GF2 vector of length 24>
gap> last = e;
true
gap> #
gap> L := KernelOfModuleHomomorphismSplit(d2);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 5 generators in FGˆ3.

gap> K = L;
true
gap> M := KernelOfModuleHomomorphismGF(d2);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 4 generators in FGˆ
3. Generators are minimal.

gap> K = M;
true
```

Chapter 7

Ring homomorphisms

A ring homomorphism is a linear function $f : R \to S$ that maps between two rings R and S and which preserves the operations of multiplication and addition:

$$f(a+b) = f(a) + f(b)$$
$$f(ab) = f(a)f(b)$$
$$f(1) = 1$$

7.1 The `HAPRingHomomorphism` datatype

The `HAPRingHomomorphism` datatype represents a particular class of ring homomorphisms (in fact usually ring isomorphisms), namely those between rings presented as quotient rings of polynomial rings, where the source and target rings have the same coefficient ring. They represent the mapping

$$\frac{R}{I} \to \frac{S}{J}$$

where $R = k[x_1, \ldots, x_n]$ and $S = k[y_1, \ldots, x_m]$ and I and J are ideals in R and S respectively.

Such a ring homomorphism may be specified by the following information

- a polynomial ring R
- a list of polynomials in R that generate the ideal I
- a list $Q = [q_1, ..., q_n]$ where each $q_i \in S/J$ is the image of the indeterminate x_i in R under the homomorphism.

The target ideal J is assumed to be the image of I under the homomorphism.

7.1.1 Implementation details

Various different internal representations are used for ring homomorphisms in HAPprime, depending on the source and target ring. The user need not be concerned with the different representations, which correspond to the five constructors `HAPRingToSubringHomomorphism` (7.2.1), `HAPSubringToRingHomomorphism` (7.2.2), `HAPRingHomomorphismByIndeterminateMap` (7.2.3), `HAPRingReductionHomomorphism` (7.2.4) and `HAPZeroRingHomomorphism` (7.2.5), and which are hopefully largely self-explanatory.

Most of the provided representations of ring homomorphisms use Gröbner bases and polynomial reduction to perform the mapping. This uses the singular package (**singular: singular: the GAP interface to Singular**) which gives good speed and access to improved monomial orderings. This datatype cannot be used without the singular package.

7.1.2 Elimination orderings

Using Gröbner bases to map from one polynomial ring R to another polynomial ring S relies on applying a global ordering to the joint ring $R \cup S$. For all polynomials $p \in R$ and $q \in S$, this ordering must give $p > q$, so that terms involving elements of R will be replaced by those in S. A straightforward solution is to use a lexicographic term ordering which orders the indeterminates of R before those of S. However, computing Gröbner bases using a lexicographic ordering can be much more expensive that with other orderings, or sometimes even a change of the order of indeterminates is enough change the order of the computation. A range of different ordering can be requested by using the GAP options stack (**Reference: Options Stack**), and setting the options `EliminationIndexOrder` and `EliminationBlockOrdering` as described below.

The option `EliminationIndexOrder` determines the indeterminate ordering to use. Possible values are the following strings:

- `"Forward"` (default) Indeterminates are ordered as in the definition of R and S: $p_1 > p_2 > \ldots > p_n > q_1 > q_2 > \ldots > q_m$.
- `"Reverse"` Lexicographic ordering in reverse order to that in the definition of R and S: $p_n > p_{n-1} > \ldots > p_1 > q_m > q_{m-1} > \ldots > q_1$.
- `"Shuffle"` Lexicographic ordering using a random shuffle of the indeterminates in R and S.
- `"ShuffleNN"` where `NN` is some non-negative integer. This is the same as `Shuffle`, but using the value of `NN` as the random seed (and hence is deterministic).

When comparing two monomials, the elimination ordering ensures that the parts of the two monomials involving the indeterminates from R are compared first, and then in the event of a tie, the part involving those from S are compared. The option `EliminationBlockOrdering` determines the monomial ordering to use for each of these two comparisons. This is given as a string which is the concatenation of the two required orderings. For example, the default ordering is `"LexGrlex"` which means lexicographic ordering over the indeterminates of R, and graded lexicographic over the indeterminates of S.

- `"Lex"` Lexicographic ordering, the equivalent of the GAP ordering `MonomialLexOrdering` (**Reference: MonomialLexOrdering**)).
- `"Grlex"` Graded lexicographic ordering, the equivalent of `MonomialGrlexOrdering` (**Reference: MonomialGrlexOrdering**).
- `"Grevlex"` Graded reverse lexicographic ordering, the equivalent of `MonomialGrevlexOrdering` (**Reference: MonomialGrevlexOrdering**).

7.2 Construction functions

7.2.1 HAPRingToSubringHomomorphism

◊ HAPRingToSubringHomomorphism(*Rring, Rrels, Simages*) (operation)

Returns: HAPRingHomomorphism

Creates a HAPRingHomomorphism which represents the mapping $R/I \to S/J$. In this form, *Rring* a polynomial ring R and *Rrels* an ideal I in that ring. The image of the indeterminates of *R* under this mapping are given in *Simages* and generate the ring S, while the relations *Rrels* are mapped to generate J. The ring S may be a subring of the full polynomial ring in its indeterminates.

7.2.2 HAPSubringToRingHomomorphism

◊ HAPSubringToRingHomomorphism(*Rgens, Rrels, Sring*) (operation)
◊ HAPSubringToRingHomomorphism(*Rgens, Sring, Srels*) (operation)

Returns: HAPRingHomomorphism

Creates a HAPRingHomomorphism which represents the mapping $R/I \to S/J$. The ring R is generated by a set of polynomials *Rgens* (so R may be a subring of the full polynomial ring in its indeterminates). The images of *Rgens* under the mapping are the indeterminates of the polynomial ring given in *Sring*. The ideals can be specified either as a set of relations *Srels* in the target ring S, or as a set of relations *Rrels* in the source ring. In this second case, *Rrels* can be polynomials in the full polynomial ring, in which case the ideal I is the intersection of the ideal they generate in the full ring with the subring generated by *Rgens*. In both cases, the specified ideal is mapped with the homomorphism (or its inverse) to find the corresponding ideal in the other ring.

This ring homomorphism uses Gröbner bases to perform the mapping, and the time taken to calculate the basis in this function can be influenced by the choice of monomial ordering. See 7.1.2 for more details.

7.2.3 HAPRingHomomorphismByIndeterminateMap

◊ HAPRingHomomorphismByIndeterminateMap(*R, Rrels, S*) (operation)

Returns: HAPRingHomomorphism

Creates a HAPRingHomomorphism which represents the map $R/I \to S/J$ which is a simple relabelling of indeterminates: the image of the ith indeterminate of *R* under the mapping is taken to be the ith indeterminate of *S*. The ideal I is generated by *Rrels* and are mapped using the homomorphism to generate J.

7.2.4 HAPRingReductionHomomorphism (for ring presentation)

◊ HAPRingReductionHomomorphism(*R, Rrels[, avoid]*) (operation)
◊ HAPRingReductionHomomorphism(*phi[, avoid]*) (operation)

Returns: HAPRingHomomorphism

For a polynomial ring *R* and ideal I generated by *Rrels*, this function finds an isomorphic ring in fewer indeterminates (or the same number, if this is not possible). This new ring will avoid the indeterminates of *R* and any further indeterminates listed in *avoid*. The function returns the map between R/I and the new ring.

In the second form, this function reduces the target ring of the ring homomorphism *phi* and returns the map between this and the reduced ring. This map will also avoid the indeterminates in the source ring of *phi*.

7.2.5 HAPZeroRingHomomorphism

◊ HAPZeroRingHomomorphism(*R, Rrels*) (operation)

Returns: HAPRingHomomorphism

Creates a HAPRingHomomorphism which maps from the ring *R*, with an ideal generated by *Rrels*, into the trival ring generated by zero.

7.2.6 InverseRingHomomorphism

◊ InverseRingHomomorphism(*phi*) (attribute)

Returns: HAPRingHomomorphism

Returns (as a ring homomorphism) the inverse of the ring homomorphism *phi*.

If the inverse homomorphism requires an elimination Gröbner basis to perform the mapping (for example when computing the inverse of a HAPRingHomomorphism constructed with HAPRingToSubringHomomorphism (7.2.1)) then the ordering can be specified using the options stack. See 7.1.2 for more details.

7.2.7 CompositionRingHomomorphism

◊ CompositionRingHomomorphism(*phiA, phiB*) (operation)

Returns: HAPRingHomomorphism

Returns the ring homomorphism that is the composition of the ring homomorphisms *phiA* and *phiB*. The source ring of *phiB* must be in the image ring of *phiA*.

7.3 Data access functions

7.3.1 SourceGenerators

◊ SourceGenerators(*phi*) (attribute)

Returns: List

A list of generators for the source ring R/I of the ring homomorphism. *phi*.

7.3.2 SourceRelations

◊ SourceRelations(*phi*) (attribute)

Returns: List

A list of the relations that generate the ideal I of in the source ring of the ring homomorphism *phi*.

7.3.3 SourcePolynomialRing

◊ SourcePolynomialRing(*phi*) (attribute)

Returns: PolynomialRing

Returns the polynomial ring which contains the source ring of the ring homomorphism *phi*. Polynomials to be mapped by *phi* must be in this ring.

7.3.4 ImageGenerators

◊ ImageGenerators(*phi*) (attribute)

Returns: List

A list of generators for the image ring S/J of the ring homomorphism *phi*.

7.3.5 ImageRelations

◊ ImageRelations(*phi*) (attribute)

Returns: List

A list of the relations that generate the ideal J of in the image ring of the ring homomorphism *phi*.

7.3.6 ImagePolynomialRing

◊ ImagePolynomialRing(*phi*) (attribute)

Returns: PolynomialRing

Returns the polynomial ring which contains the image of the ring homomorphism *phi*>. All polynomials mapped by *phi* will be in this ring.

7.4 General functions

7.4.1 ImageOfRingHomomorphism

◊ ImageOfRingHomomorphism(*phi, poly*) (operation)
◊ ImageOfRingHomomorphism(*phi, coll*) (operation)

Returns: Polynomial or list

Returns the image of the polynomial *poly* under the ring homomorphism *phi*. The input must be an element(s) of the source ring of *phi* (see SourcePolynomialRing (7.3.3)).

7.4.2 PreimageOfRingHomomorphism

◊ PreimageOfRingHomomorphism(*phi, poly*) (operation)
◊ PreimageOfRingHomomorphism(*phi, coll*) (operation)

Returns: Polynomial or list

Returns the preimage of the polynomial *poly* under the ring homomorphism *phi*. The input must be an element(s) of the image ring of *phi* (see ImagePolynomialRing (7.3.6)). This function is a synonym for ImageOfRingHomomorphism(InverseRingHomomorphism(phi), poly).

7.5 Example: Constructing and using a HAPRingHomomorphism

As an initial example, we shall construct a ring homomorphism $\phi : k[x_1,x_2] \to k[x_3,x_4]$ (i.e. ideal) with the mapping $x_1 \mapsto x_3$ and $x_2 \mapsto x_3 + x_4$. In all these examples, we shall take k to be the field of

two elements, GF(2). We demonstrate that this is an isomorphism by mapping a polynomial from the source ring to the target and back again.

Example

```
gap> R := PolynomialRing(GF(2), 2);;
gap> S := PolynomialRing(GF(2), 2, IndeterminatesOfPolynomialRing(R));;
gap> phi := HAPRingToSubringHomomorphism(R, [], [S.1, S.1+S.2]);
<Ring homomorphism>
gap> p := ImageOfRingHomomorphism(phi, R.1^3+R.1*R.2+R.2);
x_3^3+x_3^2+x_3*x_4+x_3+x_4
gap> PreimageOfRingHomomorphism(phi, p);
x_1^3+x_1*x_2+x_2
```

Some ring presentations are not in minimal form: there is an isomorphic ring in fewer indeterminates. The `HAPRingReductionHomomorphism` (7.2.4) function can find and return an isomorphism that maps to this reduced ring. For example, the ring $k[x_1,x_2,x_3]/(x_1^2+x_2^3,x_2^2+x_3)$ has an isomorphic presentation in only two indeterminates, as this computation shows:

Example

```
gap> R := PolynomialRing(GF(2), 3);;
gap> I := [R.1^2+R.2^3, R.2^2+R.3];
[ x_2^3+x_1^2, x_2^2+x_3 ]
gap> phi := HAPRingReductionHomomorphism(R, I);
<Ring homomorphism>
gap> ImagePolynomialRing(phi);
GF(2)[x_4,x_5]
gap> ImageRelations(phi);
[ x_5^3+x_4^2 ]
```

The source and target of `HAPRingHomomorphisms` can be quotient rings, and any relations can be specified at the source of target. When mapping from a subring to a full ring, the source relations do not necessarily need to be specified in the source ring, but could instead be given in its containing ring. For example, consider the ring R/I where $R = k[x_1,x_2,x_3]$ and $I = x_1^3+x_3$ and the subring generated by (x_1,x_2,x_3^2). If we wish to specify the homomorphism $\phi : R/I \to S/J$ where $S = k[x_4,x_5,x_6]$ with the natural map from the generators of R to those of S then we only need specify the original ideal I, even though it is not in the subring. The intersection between I and the subring is found to be $x_1^4+x_3^2$, and it is this ideal that is used in the homomorphism, as this example shows:

Example

```
gap> R := PolynomialRing(GF(2), 3);;
gap> I := [R.1^2+R.3];
[ x_1^2+x_3 ]
gap> S := PolynomialRing(GF(2), 3, IndeterminatesOfPolynomialRing(R));;
gap> phi := HAPSubringToRingHomomorphism([R.1, R.2, R.3^2], I, S);
<Ring homomorphism>
gap> Display(phi);
Ring homomorphism
  x_1 -> x_4
  x_2 -> x_5
  x_3^2 -> x_6
with relations
  [ x_1^2+x_3 ]
and
  [ x_4^4+x_6 ]
```

```
gap> Display(InverseRingHomomorphism(phi));
Ring homomorphism
  x_4 -> x_1
  x_5 -> x_2
  x_6 -> x_3^2
with relations
  [ x_4^4+x_6 ]
and
  [ x_1^4+x_3^2 ]
```

Ring homomorphisms can also be composed with each other. For example, we can take the `HAPSubringToRingHomomorphism` (7.2.2) above and change the target indeterminates by composing this with a `HAPRingHomomorphismByIndeterminateMap` (7.2.3):

Example

```
gap> R := PolynomialRing(GF(2), 3);;
gap> I := [R.1^2+R.3];
[ x_1^2+x_3 ]
gap> S := PolynomialRing(GF(2), 3, IndeterminatesOfPolynomialRing(R));;
gap> phi := HAPSubringToRingHomomorphism([R.1, R.2, R.3^2], I, S);
<Ring homomorphism>
gap> #
gap> T := PolynomialRing(GF(2), 3, [R.1, R.2, R.3, S.1, S.2, S.3]);;
gap> phi2 := HAPRingHomomorphismByIndeterminateMap(
>   ImagePolynomialRing(phi), ImageRelations(phi), T);
<Ring homomorphism>
gap> Display(CompositionRingHomomorphism(phi, phi2));
Ring homomorphism
  x_1 -> x_7
  x_2 -> x_8
  x_3^2 -> x_9
with relations
  [ x_1^2+x_3 ]
and
  [ x_7^4+x_9 ]
```

Chapter 8

Derivations

A derivation d of a ring R is a map $R \longrightarrow R$ that obeys the product rule

$$d(rs) = d(r)s + rd(s)$$

for elements r, s in the field, and which is also distributive over addition in the ring

$$d(r+s) = d(r) + d(s)$$

8.1 The `HAPDerivation` datatype

The `HAPDerivation` datatype represents a derivation of a polynomial ring R. Given the product and addition rules above, a derivation can be completely specified in terms of the images of the ring indeterminates. Consequently, a `HAPDerivation` is specified by providing:

- a polynomial ring R
- a list of images, under the derivation, of each of the ring's indeterminates.

Optionally, you can also provide

- a set of relations I (which is assumed to be empty if none is provided).

The support for derivations over quotient rings using `HAPDerivation` is limited. The ring is always assumed to simply be the polynomial ring R, but when calculating the kernel and homology (see `KernelOfDerivation` (8.5.2) and `HomologyOfDerivation` (8.5.3)), any relations in I are also included in the result's relations, which is the equivalent of calculating the kernel of homology of the derivation $R/I \longrightarrow R/I$.

8.2 Computing the kernel and homology of a derivation

Computing the kernel of a derivation is difficult in the general case, but for polynomial rings over $\mathbb{F}_p$ we have a novel solution which allows derivations to be considered as module homomorphisms.

Consider a ring R over $GF(2)$. For any $r \in R$,

$$d(r^2) = d(rr) = d(r)r + rd(r) = 2d(r) = 0$$

In general, for a ring $R = \mathbb{F}_p[x_1,\ldots,x_n]$, the subring $S = \mathbb{F}_p[x_1^p,\ldots,x_n^p]$ is in the kernel of d. For this choice of S, R can always be written as a free S-module with 2^n generators.

Consider now taking the derivation of sr where $s \in S$ and $r \in R$,

$$d(sr) = sd(r) + d(s)r = sd(r)$$

since s is in the kernel of d. In other words, when R is written as an S-module, the derivation d behaves exactly as as an S-module homomorphism.

The Singular computer algebra system can compute the kernels of module homomorphisms over polynomial rings, and the singular package provides an interface between Singular and GAP. The function `KernelOfDerivation` (8.5.2) constructs S, expresses d as an S-module homomorphism (as well as converting any ideal), and asks Singular for the kernel. Singular returns the result as a set of module generators, which are then reduced to a set of ring generators and a presentation found. This makes further use of Singular, and the `HAPRingHomomorphism` datatype (see Chapter 7).

The homology of a derivation is defined to be

$$Hom(d) = \ker(d)/im(d)$$

Given the kernel of a derivation and the image (expressed as the images of the module generators), the homology can be readily computed, as is done by `HomologyOfDerivation` (8.5.3). The natural presentation obtained by concatenating the kernel relations and the image typically contains redundancies and the `HAPRingReductionHomomorphism` (7.2.4) is used to generate a nicer presentation involving fewer indeterminates and relations.

8.3 Construction function

8.3.1 HAPDerivation construction functions

◊ `HAPDerivation(R[, I], images)` (operation)
◊ `HAPDerivationNC(R, I, images)` (operation)

Returns: `HAPDerivation`

Construct a `HAPDerivation` object representing the derivation d where `R` is a polynomial ring and `images` is the list of the images of the ring indeterminates under the derivation, $\{d(x_1), d(x_2), \ldots, d(x_n)\}$. An optional set of relations `I` can also be provided, which are passed to `KernelOfDerivation` (8.5.2) when calculating the kernel or homology of this derivation. The function `HAPDerivation` checks that the arguments are compatible, while the `NC` method performs no checks.

8.4 Data access function

8.4.1 DerivationRing

◊ `DerivationRing(d)` (attribute)

Returns: Polynomial ring

Returns the ring over which the derivation `d` is defined.

8.4.2 DerivationImages

◊ DerivationImages(*d*) (attribute)

Returns: List of polynomials

A derivation *d* over a (quotient) polynomial ring is defined by a set of images. This function returns this list of images. The *i*th element of the list is the image of indeterminate number *i* in that ring family. (Note that indeterminate number *i* is not necessarily the *i*th indeterminate in that particular ring. See (**Reference: Indeterminates**) for more details.)

8.4.3 DerivationRelations

◊ DerivationRelations(*d*) (attribute)

Returns: List of polynomials

Returns the relations of the quotient ring over which the derivation *d* is defined.

8.4.4 Example: Constructing and accessing data of a HAPDerivation

We demonstrate creating a HAPDerivation object and reading back its data by creating a derivation d of the polynomial ring over the integers with two indeterminates, $\mathbb{Z}[x_1,x_2,x_3]$, which has the mapping $d(x_1) = d(x_2) = 1, d(x_3) = 0$.

Example

```
gap> ring := PolynomialRing(Integers, 3);;
gap> d := HAPDerivation(ring, [One(ring), One(ring), Zero(ring)]);
<Derivation>
gap>
gap> DerivationRing(d);
Integers[x_1,x_2,x_3]
gap> DerivationImages(d);
[ 1, 1, 0 ]
gap> DerivationRelations(d);
[  ]
```

8.5 Image, kernel and homology functions

8.5.1 ImageOfDerivation

◊ ImageOfDerivation(*d, poly*) (operation)

Returns: Polynomial

Returns the image of the polynomial *poly* under the derivation *d*. (*poly* must be a polynomial in the derivation's ring.)

8.5.2 KernelOfDerivation

◊ KernelOfDerivation(*d[, avoid]*) (operation)

Returns: List

Returns a ring presentation S/J for the kernel of the derivation *d*, where S is a polynomial ring and J are a set of relations. This operation returns a list with the following elements:

1. the new polynomial ring S

2. a basis for the ideal, as a set of relations J

3. the ring isomorphism between the kernel (i.e. a subring of the derivation's ring) and the new ring. This is given using the `HAPRingHomomorphism` type, for details of which see 7

An optional parameter, `avoid`, can be provided which lists indeterminates to avoid when creating the the new polynomial ring.

This function is only available if the package singular *is available.*

8.5.3 HomologyOfDerivation

◊ `HomologyOfDerivation(d[, avoid])` (operation)

Returns: List

Returns a ring presentation S/J for the homology of the derivation `d`, where S is a polynomial ring and J are a set of relations. Returns a polynomial ring presentation for the homology $ker(d)/im(d)$ of the derivation `d`. This operation returns a list with the following elements:

1. the new polynomial ring S

2. a basis for the ideal, as a set of relations J

3. the ring isomorphism between the kernel (i.e. a subring of the derivation's ring) and the new ring. This is given using the `HAPRingHomomorphism` type, for details of which see 7

An optional parameter, `avoid`, can be provided which lists indeterminates to avoid when creating the the new polynomial ring.

8.5.4 Example: Homology of a `HAPDerivation`

In this example, we wish to calculate the homology of a derivation d on the ring $\mathbb{F}_2[x,y,z]/(yx^2+xy^2)$ where $d(x)=d(y)=0$ and $d(z)=yx^2+xy^2$. To demonstrate some of the other functions, we also calculate the kernel of this derivation and the image of the element xz.

Example

```
gap> ring := PolynomialRing(GF(2), 3);;
gap> x := ring.1;; y := ring.2;; z := ring.3;;
gap> d := HAPDerivation(ring, [x^2 + x*y + y^2],
>                [Zero(ring), Zero(ring), x^2*y + x*y^2]);
<Derivation>
gap> #
gap> ImageOfDerivation(d, x*z);
x_1^3*x_2+x_1^2*x_2^2
gap> KernelOfDerivation(d);
[ GF(2)[x_7,x_8,x_9], [ x_7^2+x_7*x_8+x_8^2 ], <Ring homomorphism> ]
gap> HomologyOfDerivation(d);
[ GF(2)[x_4,x_5,x_6], [ x_4^2+x_4*x_5+x_5^2, x_5^3 ], <Ring homomorphism> ]
```

The result we were after, the homology of d, can be read from the final result as follows. The first two entries in the list give the homology group as a quotient ring: the quotient of the polynomial ring $\mathbb{F}_2[x_6,x_7,x_8]$ with the two relations $(x_7^2+x_7x_8+x_8^2, x_8^3)$. The last entry in the list gives the new indeterminates in terms of the original indeterminates, and from this we know that it would be friendlier to write the result as

$$\frac{\mathbb{F}_2[x,y,e^2]}{(x^2+xy+y^2, y^3)}$$

The kernel of the derivation can read from the result in a similar manner.

Chapter 9

Poincaré series

The Poincaré series for the mod-p cohomology ring $H^*(G,\mathbb{F})$ is the infinite series

$$a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots$$

where a_k is the dimension of the vector space $H^k(G,\mathbb{F})$. The Poincaré function is a rational function $P(x)/Q(x)$ which is equal to the Poincaré series.

9.1 Computing the Poincaré series using spectral sequences

HAPprime can calculate a provably-correct Poincaré series for the mod-p cohomology ring of a small p-group using spectral sequences, without having to compute the actual cohomology ring. The limiting sheet of a Lyndon-Hochschild-Serre spectral sequence for a group G will be a ring with the same additive structure as the cohomology ring for G, and thus the same Poincaré series. This is implemented in the HAPprime function `PoincareSeriesLHS` (**HAPprime: PoincareSeriesLHS**). See the documentation for that function in the user guide for more details.

9.2 Computing the Poincaré series using a minimal resolution

Given a resolution R of length n for a group G, the HAP function `PoincareSeries` (**HAP: PoincareSeries**) calculates a quotient of polynomials such that the coefficient of x^k equals the dimension of $H^k(G,\mathbb{F})$ for $k = 1$ to $k = n$. Given a sufficiently long resolution R, this quotient should equal the true Poincaré series. The function can also automatically find a suitable value for n by trying resolutions of increasing length until a consistent estimate is found for the Poincaré series. This is likely to be correct, but we have no proof that this will always be the case.

The function `ExtendResolutionPrimePowerGroupAutoMem` (**HAPprime: ExtendResolutionPrimePowerGroupAutoMem**) allows HAPprime to provide a simplified implementation for calculating Poincaré series in the case where n is not specified (since extending existing resolutions is difficult in HAP). The HAPprime function `PoincareSeriesAutoMem` (**HAPprime Datatypes: PoincareSeriesAutoMem**) is a replacement for HAP's `PoincareSeries` (**HAP: PoincareSeries**) in the case where G is a p-group and the optimal n is not known.

By using the HAPprime resolution-calculation functions, `PoincareSeriesAutoMem` (**HAPprime Datatypes: PoincareSeriesAutoMem**) saves memory when storing resolution, and switches to the `GF` algorithm when memory is low. As a result, it can calculate the Poincaré series for groups that would be impossible using HAP without having about five times the memory available to the machine.

9.2.1 PoincareSeriesAutoMem

◊ PoincareSeriesAutoMem(*G[, n]*) (operation)

Returns: Rational function

For a finite p-group *G*, this function calculates and returns a quotient of polynomials $f(x) = P(x)/Q(x)$ (i.e. the Poincaré series) whose coefficient of x^k equals the rank of the vector space $H_k(G,\mathbb{F}_p)$ for all k in the range $k = 1$ to $k = n$. If no value is given for n then increasing values of n are tried to find the minimum value which gives a consistent Poincaré series, defined as a the minimum value $n > 10$ such that `PoincareSeries(G, n) = PoincareSeries(G, n-1) = PoincareSeries(G, n-2)`.

This function uses the HAP function `PoincareSeries` (**HAP: PoincareSeries**) to calculate the Poincaré series, but the HAPprime function `ExtendResolutionPrimePowerGroupAutoMem` (**HAPprime: ExtendResolutionPrimePowerGroupAutoMem**) to calculate and gradually extend the resolution, so should be both faster and more memory-efficient than using `PoincareSeries` by itself. See Section 9.3 for an example.

The Poincaré series calculated using this function is likely to be correct, but we have no proof that this will be the case. If a correct Poincaré series is required, use `PoincareSeriesLHS` (**HAPprime: PoincareSeriesLHS**)

9.3 Example Poincaré series computations

This example compares the time taken by `PoincareSeries` (**HAP: PoincareSeries**) and `PoincareSeriesAutoMem` (**HAPprime Datatypes: PoincareSeriesAutoMem**), and shows that the times are roughly comparable:

Example

```
gap> G := SmallGroup(64, 210);;
gap> # Compute the Poincare series using HAP
gap> P1 := PoincareSeries(G);time;
(x_1^4+x_1^2+x_1+1)/(-x_1^7+3*x_1^6-5*x_1^5+7*x_1^4-7*x_1^3+5*x_1^2-3*x_1+1)
51355
gap> # Compute the Poincare series using HAPprime
gap> P2 := PoincareSeriesAutoMem(G);time;
(x_1^4+x_1^2+x_1+1)/(-x_1^7+3*x_1^6-5*x_1^5+7*x_1^4-7*x_1^3+5*x_1^2-3*x_1+1)
39774
gap> P1 = P2;
true
```

The HAPprime function `PoincareSeriesLHS` (**HAPprime: PoincareSeriesLHS**) uses an alternative approach to compute Poincaré series which are guaranteed to be correct. In many cases it is also faster, as we see if we compute the Poincaré series for the same group using this function:

Example

```
gap> G := SmallGroup(64, 210);;
gap> P3 := PoincareSeriesLHS(G);time;
(x_1^4+x_1^2+x_1+1)/(-x_1^7+3*x_1^6-5*x_1^5+7*x_1^4-7*x_1^3+5*x_1^2-3*x_1+1)
3564
```

9.4 The Poincaré series of groups of order 64 and 128

Using HAPprime, on a dual-processor AMD Opteron 250 machine, we have calculated the Poincaré series for all of the groups of order 64 using the resolution-based technique. Most computed within a few seconds using only a few Mb of memory. With a maximum of 1Gb of memory available to GAP, four groups (numbers 4, 60, 242 and 266 from the GAP `SmallGroup` library) needed to switch to using `ExtendResolutionPrimePowerGroupGF` (**HAPprime: ExtendResolutionPrimePowerGroupGF**). Calculating the Poincaré series of the most difficult group, `SmallGroup(64, 60)`, took 24 days, computing a resolution whose last term was $M_{16} = (\mathbb{F}G)^{2445}$. The complete list of the Poincaré series for all groups of order 64 is available on the HAPprime website http://www.maths.nuigalway.ie/~pas/CHA/HAPprime/HAPprimeindex.html

There is an on-going programme of calculating the Poincaré series for the groups of order 128. To date, using the same constraints as for the groups of order 64 above, we have computed the Poincaré series for about half of them. For latest details, again please see the HAPprime website.

Chapter 10

General Functions

Some of the functions provided by HAPprime are not specifically aimed at homological algebra or extending the HAP package. The functions in this chapter, which are used internally by HAPprime extend some of the standard GAP functions and datatypes.

10.1 Matrices

For details of the standard GAP vector and matrix functions, see (**Tutorial: matrices**) and (**Reference: Matrices**) in the GAP tutorial and reference manuals. HAPprime provides improved versions of a couple of standard matrix operations, and two small helper functions.

10.1.1 SumIntersectionMatDestructive

◊ `SumIntersectionMatDestructive(`*U, V*`)` (operation)
◊ `SumIntersectionMatDestructiveSE(`*Ubasis, Uheads, Vbasis, Vheads*`)` (operation)

Returns a list of length 2 with, at the first position, the sum of the vector spaces generated by the rows of *U* and *V*, and, at the second position, the intersection of the spaces.

Like the GAP core function `SumIntersectionMat` (**Reference: SumIntersectionMat**), this performs Zassenhaus' algorithm to compute bases for the sum and the intersection. However, this version operates directly on the input matrices (thus corrupting them), and is rewritten to require only approximately 1.5 times the space of the original input matrices. By contrast, the original GAP version uses three times the memory of the original matrices to perform the calculation, and since it doesn't modify the input matrices will require a total of four times the space of the original matrices.

The function `SumIntersectionMatDestructiveSE` takes as arguments not a pair of generating matrices, but a pair of semi-echelon basis matrices and the corresponding head locations, such as is returned by a call to `SemiEchelonMatDestructive` (**Reference: SemiEchelonMatDestructive**) (these arguments must all be mutable, so `SemiEchelonMat` (**Reference: SemiEchelonMat**) cannot be used). This function is used internally by `SumIntersectionMatDestructive`, and is provided for the occasions when the user might already have the semi-echelon versions available, in which case a small amount of time will be saved.

10.1.2 SolutionMat (for multiple vectors)

◊ SolutionMat(*M*, *V*) (operation)
◊ SolutionMatDestructive(*M*, *V*) (operation)

Calculates, for each row vector v_i in the matrix *V*, a solution to $x_i \times M = v_i$, and returns these solutions in a matrix X, whose rows are the vectors x_i. If there is not a solution for a v_i, then fail is returned for that row.

These functions are identical to the kernel functions SolutionMat (**Reference: SolutionMat**) and SolutionMatDestructive (**Reference: SolutionMatDestructive**), but are provided for cases where multiple solutions using the same matrix *M* are required. In these cases, using this function is far faster, since the matrix is only decomposed once.

The Destructive version corrupts both the input matrices, while the non-Destructive version operates on copies of these.

10.1.3 IsSameSubspace

◊ IsSameSubspace(*U*, *V*) (operation)

Returns true if the subspaces spanned by the rows of *U* and *V* are the same, false otherwise.

This function treats the rows of the two matrices as vectors from the same vector space (with the same basis), and tests whether the subspace spanned by the two sets of vectors is the same.

10.1.4 PrintDimensionsMat

◊ PrintDimensionsMat(*M*) (operation)

Returns a string containing the dimensions of the matrix *M* in the form "mxn", where m is the number of rows and n the number of columns. If the matrix is empty, the returned string is "empty".

10.1.5 Example: matrices and vector spaces

GAP uses rows of a matrix to represent basis vectors for a vector space. In this example we have two matrics U and V that we suspect represent the same subspace. Using SolutionMat (10.1.2) we can see that V lies in U, but IsSameSubspace (10.1.3) shows that they are the same subspace, as is confirmed by having identical sums and intersections.

Example

```
gap> U := [[1,2,3],[4,5,6]];;
gap> V := [[3,3,3],[5,7,9]];;
gap> SolutionMat(U, V);
[ [ -1, 1 ], [ 1, 1 ] ]
gap> IsSameSubspace(U, V);
true
gap> SumIntersectionMatDestructive(U, V);
[ [ [ 1, 2, 3 ], [ 0, 1, 2 ] ], [ [ 0, 1, 2 ], [ 1, 0, -1 ] ] ]
gap> IsSameSubspace(last[1], last[2]);
true
gap> PrintDimensionsMat(V);
"2x3"
```

10.2 Polynomials

GAP provides some functions for manipulating polynomials and polynomial ideals - see (**Reference: Polynomials and Rational Functions**). The HAPprime packages adds further functions to decompose polynomials into terms and monomials, and some functions for tidying up polynomial ideals.

10.2.1 TermsOfPolynomial

◊ `TermsOfPolynomial(poly)` (attribute)

Returns: List of pairs

Returns a list of the terms in the polynomial. This list is a list of pairs of the form `[mon, coeff]` where `mon` is a monomial and `coeff` is the coefficient of that monomial in the polynomial. The monomials are sorted according to the total degree/lexicographic order (the same as the in `MonomialGrLexOrdering` (**Reference: MonomialGrlexOrdering**)).

10.2.2 IsMonomial (for polynomial)

◊ `IsMonomial(poly)` (attribute)

Returns: Boolean

Returns `true` if *poly* is a monomial, i.e. the polynomial contains only one term.

10.2.3 UnivariateMonomialsOfMonomial

◊ `UnivariateMonomialsOfMonomial(mon)` (attribute)

Returns: List

Returns a list of the univariate monomials of the largest order whose product equals *mon*. The univariate monomials are sorted according to their indeterminate number.

10.2.4 IndeterminateAndExponentOfUnivariateMonomial

◊ `IndeterminateAndExponentOfUnivariateMonomial(mon)` (attribute)

Returns: List

Returns a list `[indet, exp]` where `indet` is the indeterminate of the univariate monomial *mon* and `exp` is the exponent of that indeterminate in the monomial. If *mon* is an element in the coefficient ring (i.e. the monomial contains no indeterminates) then the first element will be *mon* with an exponent of zero. If *mon* is not a univariate monomial, then `fail` is returned.

10.2.5 IndeterminatesOfPolynomial

◊ `IndeterminatesOfPolynomial(poly)` (attribute)

Returns: List

Returns a list of the indeterminates used in the polynomial *poly*.

10.2.6 ReduceIdeal

◊ `ReduceIdeal(I, O)` (operation)

◊ `ReduceIdeal(rels, O)` (operation)

Returns: Ideal or list

For an ideal *I* returns an ideal containing a reduced generating set for the ideal, i.e. one in which no monomial in a relation in *I* is divisible by the leading term of another polynomial in *I*. The monomial ordering to be used is specified by *O* (see (**Reference: Monomial Orderings**)). The ideal can instead be specified by a list of relations *rels*, in which case a reduced list of relations is returned.

10.2.7 ReducedPolynomialRingPresentation

◊ ReducedPolynomialRingPresentation(*R, I[, avoid]*) (operation)
◊ ReducedPolynomialRingPresentationMap(*R, I[, avoid]*) (operation)

Returns: List

For a polynomial ring *R* and a list of relations *I* in that ring, returns a list [S, J] representing a polynomial quotient ring S/J which is isomorphic to the ring R/I, but which involves the minimal number of ring indeterminates. The indeterminates in S will be distinct from thise in *R*, and an optional argument *avoid* can be used to give a list of further indeterminates to avoid when creating the ring S.

The extended version of this function, ReducedPolynomialRingPresentationMap, returns an additional third element to the list, which contains two lists giving the mapping between the new ring indeterminates and the old ring indeterminates. The first list is of polynomials in the original indeterminates, the second the equivalent polynomials in the new ring indeterminates.

10.2.8 Example: monomials, polynomials and ring presentations

A monomial is some product of ring indeterminates. A polynomial is a sum of monomials, where each monomial may also be multiplied by an element from the field of the polynomial. It can be useful to decompose polynomials as follows:

- decompose a polynomial into its individual terms (where a term is a product of a monomial and a field element). The function TermsOfPolynomial (10.2.1) does this.
- decompose a monomial into its component univariate monomials, each of which is some (power of) a single indeterminates. This operation is performed by UnivariateMonomialsOfMonomial (10.2.3).
- decompose a univariate monomial into it the indeterminates and exponent (IndeterminateAndExponentOfUnivariateMonomial (10.2.4)).

In the example below, we decompose $x + xy^2 + 3y^3$ into its three terms, and then further decompose the xy^2 term.

Example

```
gap> R := PolynomialRing(Integers, 2);;
gap> x := R.1;; y := R.2;;
gap> poly := x + x*y^2 + 3*y^3;
x_1*x_2^2+3*x_2^3+x_1
gap> terms := TermsOfPolynomial(poly);
[ [ x_1, 1 ], [ x_2^3, 3 ], [ x_1*x_2^2, 1 ] ]
gap> UnivariateMonomialsOfMonomial(terms[3][1]);
[ x_1, x_2^2 ]
gap> IndeterminateAndExponentOfUnivariateMonomial(last[2]);
[ x_2, 2 ]
```

HAPprime also provides some functions to help the generation of ring presentations. In the following example we consider the polynomial ring $\mathbb{Z}[w,x,y,z]$ an an ideal $I = \langle w^2+x, w^3+x^3 \rangle$. We first convert this ideal into reduced form (where no monomial in a polynomial is divisible by the leading term of any other polynomial). Then we calculate a reduced ring presentation for the quotient ring R/I, where we find that the indeterminate x is can be removed and a new ring $S/J = \mathbb{Z}[w,y,z]/\langle w^6 - w^3 \rangle$ is isomorphic to R/I

Example

```
gap> R := PolynomialRing(Integers, 4);;
gap> w := R.1;; x := R.2;;
gap> I := [w^2 + x, w^3 + x^3];
[ x_1^2+x_2, x_1^3+x_2^3 ]
gap> ReduceIdeal(I, MonomialLexOrdering());
[ x_1^2+x_2, -x_2^3+x_1*x_2 ]
gap>
gap> ReducedPolynomialRingPresentation(R, I);
[ Integers[x_5,x_6,x_7], [ x_5^6-x_5^3 ] ]
```

10.3 Singular

10.3.1 SingularSetNormalFormIdeal

◊ `SingularSetNormalFormIdeal( I )` (operation)
◊ `SingularSetNormalFormIdealNC( I )` (operation)

Returns: nothing

Sets the ideal to be used by singular for any subsequent calls to `SingularPolynomialNormalForm` (10.3.2) to be `I`. After calling this function, the singular base ring and term ordering (see `SingularBaseRing` (**singular: SingularBaseRing**) and `TermOrdering` (**singular: TermOrdering**)) will be set to be that of the ring containing `I`, so an additional call to `SingularSetBaseRing` (**singular: SingularSetBaseRing**) is not necessary.

The standard form of this function ensures that `I` is a reduced Gröbner basis with respect to the value of `TermOrdering` (**singular: TermOrdering**) for the ring containing the ideal, while the `NC` assumes that `I` is already such a Gröbner basis.

10.3.2 SingularPolynomialNormalForm

◊ `SingularPolynomialNormalForm( poly[, I] )` (operation)

Returns: Polynomial

Returns the normal form of the polynomial `poly` after reduction by the ideal `I`. The ideal can either be passed to this function, in which case it is converted to a Gröbner basis (with respect to the term ordering of the ideal's ring - see `TermOrdering` (**singular: TermOrdering**)), or the ideal to use can be set first be calling `SingularSetNormalFormIdeal` (10.3.1), which is more efficient for repeated use of this function (the latter function also sets the base ring and term ordering).

10.3.3 SingularGroebnerBasis

◊ `SingularGroebnerBasis( I )` (attribute)

Returns: List

Returns a list of relations which form a Gröbner basis for the ideal `I` given the `TermOrdering` (**singular: TermOrdering**) associated with the ring containing `I`. This function is the same as the singular function `GroebnerBasis` (**singular: GroebnerBasis**), but fixes a bug in that package when using unusual term ordering.

10.3.4 SingularReducedGroebnerBasis

◊ `SingularReducedGroebnerBasis(I)` (attribute)

Returns: List

Returns a list of relations which form a reduced Gröbner basis for the ideal `I` given the `TermOrdering` (**singular: TermOrdering**) associated with the ring containing `I`. This function is the equivalent of the singular function `GroebnerBasis` (**singular: GroebnerBasis**) (and uses that function), but ensures that a reduced basis is returned.

10.4 Groups

Small groups in GAP can be indexed by their small groups library number (**Reference: Small Groups**). An alternative indexing scheme, the Hall-Senior number, is used by Jon Carlson to publish his cohomology ring calculations at `http://www.math.uga.edu/~lvalero/cohointro.html`. To allow comparison with these results, we provide a function that converts from the GAP small groups library numbers to Hall-Senior number for the groups of order 8, 16, 32 and 64.

10.4.1 HallSeniorNumber

◊ `HallSeniorNumber(order, i)` (attribute)

◊ `HallSeniorNumber(G)` (attribute)

Returns: Integer

Returns the Hall-Senior number for a small group (of order 8, 16, 32 or 64). The group can be specified an `order`, `i` pair from the GAP (**Reference: Small Groups**) library, or as a group `G`, in which case `IdSmallGroup` (**Reference: IdSmallGroup**) is used to identify the group.

Example

```
gap> HallSeniorNumber(32, 5);
20
gap> HallSeniorNumber(SmallGroup(64, 1));
11
```

Index