The MONOID Package

Version 3.1.3

J. D. Mitchell

J. D. Mitchell — Email: jdm3@st-and.ac.uk

Copyright

Acknowledgements

The author would like to thank P. v. Bunau, A. Distler, S. Linton, J. Neubueser, V. Maltcev, M. Neuhoeffer, M. R. Quick, E. F. Robertson, and N. Ruskuc for their help and suggestions. Special thanks go to J. Araujo for his mathematical suggestions.

I would also like to acknowledge the support of the Centre of Algebra at the University of Lisbon, and of EPSRC grant number GR/S/56085/01.

Colophon

If you use the MONOID package, I would really appreciate it if you would let me know by sending me an email to jdm3@st-and.ac.uk. If you notice that there are any features missing that you think are important or if you find a bug, please let me know.

Contents

Chapter 1

The MONOID package

1.1 Introduction

This manual describes the MONOID package version 3.1.3 for computing with transformation semigroups. MONOID 3.1.3 is an updated version of the package with the same name for GAP 3; see http://schmidt.nuigalway.ie/monoid/index.html for more information about the original MONOID by Goetz Pfeiffer and Steve A. Linton, Edmund F. Robertson and Nik Ruskuc.

MONOID 3.1.3 retains all the functionality of the original MONOID package. In particular, MONOID 3.1.3 contains more efficient methods than those available in the GAP library for computing orbits, calculating Green's classes, finding the size, the elements, and testing membership in transformation semigroups; see Chapters 3 and 4. After MONOID has been loaded many of these methods are automatically used in preference to those in the library and do not need to be called explicitly by the user. These methods are described in [LPRRb] and the algorithms themselves are described in [LPRRa].

In addition, there are new methods for testing if a semigroup satisfies a particular property, such as if it is regular, simple, inverse, or completely regular, see Chapter 5; methods for computing the automorphism group of a transformation semigroup see Section 7.4; methods for finding homomorphisms and isomorphism between some types of semigroups see Chapter 7; and functions to create some well-known semigroups see Chapter 6. The property testing methods are described in [GM05] and the method for computing the automorphism group of a semigroup is described in [ABM07].

The MONOID package is written in GAP code only but relies on the GRAPE package Version 4.2 or higher in the methods for computing the automorphism group of a semigroup. The following functions can only be used fully if GRAPE is fully installed (and loaded):

- `AutomorphismGroup` (7.4.1) with argument satisfying `IsTransformationSemigroup` (**Reference: IsTransformationSemigroup**) or `IsReesZeroMatrixSemigroup` (**Reference: IsReesZeroMatrixSemigroup**)
- `RightTransStabAutoGroup` (7.5.9) with argument satisfying `IsReesZeroMatrixSemigroup` (**Reference: IsReesZeroMatrixSemigroup**)
- `RZMSGraph` (7.5.8)
- `RZMSInducedFunction` (7.5.6)
- `RZMStoRZMSInducedFunction` (7.5.7)

- `IsomorphismSemigroups` (7.7.5) with both arguments satisfying `IsReesZeroMatrixSemigroup` (**Reference: IsReesZeroMatrixSemigroup**)

Installation of GRAPE is described in the `README` file of the GRAPE distribution and in the section entitled 'Installing the GRAPE Package' of the GRAPE manual; see `http://www.maths.qmul.ac.uk/~leonard/grape/` or the main GAP webpages for more information.

If you want to take advantage of the online help facilities in MONOID, then the gapdoc package Version 1.1 or higher is also required; see `http://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc/` for further details of how to obtain and install gapdoc.

1.2 Installing MONOID

In this section we give a brief description of how to start using MONOID. If you have any problems getting MONOID working, then please email me directly at `jdm3@st-and.ac.uk`.

It is assumed that you have a working copy of GAP with version number 4.4.10 or higher. The most up-to-date version of GAP and instructions on how to install it can be obtained from the main GAP webpage `http://www.gap-system.org` Those functions in MONOID described in Chapter 7 relating to automorphism groups of semigroups require the GRAPE (for computing with graphs and groups) to be loaded. In particular, GRAPE must be installed in a UNIX operating system so that the automorphism group and isomorphism testing functions (for graphs) can be used.

Please go to `http://www.maths.qmul.ac.uk/~leonard/grape/` or the main GAP webpage for further details on how to obtain and install GRAPE.

The following is a summary of the steps that should lead to a successful installation of MONOID.

- download the package archive `monoid3r1p3.tar.gz` or `monoid3r1p3.tar.bz2` from `http://www-history.mcs.st-and.ac.uk/~jamesm/monoid/index.html`
- unzip & untar the file, this should create a directory called `MONOID`.
- move the directory `MONOID` into the `pkg` directory of your GAP directory (the one containing the directories `lib`, `doc`, `pkg`, and so on)
- start GAP in the usual way
- type `LoadPackage("monoid");`

Below is an example of an installation of MONOID in UNIX where `GAPROOT` should be substituted with the main GAP directory (the one containing the folders 'bin', 'lib', and so on) in your installation of GAP.

Example

```
> gunzip monoid3r1p3.tar.gz
> tar -xf monoid3r1p3.tar
> mv MONOID GAPROOT/pkg
> gap

[ ... ]

gap> LoadPackage("monoid");
```

```
Loading MONOID 3.1.3
by James Mitchell (www-groups.mcs.st-and.ac.uk/~jamesm)
For help, type: ?the monoid package
true
gap>
```

Presuming that the above steps can be completed successfully you will be running the MONOID package!

If you want to check that the package is working correctly, please see Section 1.3.

PLEASE NOTE: before you can used MONOID fully you must install GRAPE as described above.

1.3 Testing MONOID

In this section we describe how to test that MONOID is working as intended. To test that MONOID is installed correctly copy the following lines into GAP.

Example

```
LoadPackage( "monoid" );;
dirs := DirectoriesPackageLibrary( "monoid", "tst" );;
Read(Filename( dirs, "installtest.g" ) );
```

and press `return`. Please note that it will take a few moments before the tests are complete.

If the output looks like the following, then it is probable that you have a fully working copy of MONOID 3.1.3.

Example

```
gap> LoadPackage( "monoid" );;
gap> dirs := DirectoriesPackageLibrary( "monoid", "tst" );;
gap> Read( Filename( dirs, "installtest.g" ) );;
+ install_no_grape.tst 3.1.3
+ GAP4stones: 1
+ install_with_grape.tst 3.1.3
+ GAP4stones: 2
```

If you want to perform more extensive tests, then copy the following lines into GAP.

Example

```
LoadPackage( "monoid" );;
dirs := DirectoriesPackageLibrary( "monoid", "tst" );;
Read(Filename( dirs, "testall.g" ) );
```

Please note that these tests could take a long time to finish.

If something goes wrong, then please review the instructions in Section 1.2 and ensure that MONOID has been properly installed. If you continue having problems, please email me at jdm3@st-and.ac.uk.

1.4 Changes

- from 3.1.2 to 3.1.3: the method for *PreImagesRepresentative* for a semigroup homomorphism by function now tests whether the homomorphism is bijective and total before trying to find preimages. Some other minor corrections were made to the documentation and webpages.

- from 3.1.1 to 3.1.2:
  - the following new functions have been introduced: `TransformationActionNC` (2.1.6), `SmallestIdempotentPower` (2.2.6), `IsKerImgOfTransformation` (2.2.2), `TransformationByKernelAndImage` (2.1.1), `AllTransformationsWithKerAndImgNC` (2.1.2), `AsBooleanMatrix` (2.3.1), `KerImgOfTransformation` (2.2.3), `RandomIdempotent` (2.1.4), `InversesOfTransformation` (2.2.7), `KiselmanSemigroup` (6.1.3),
  - the following functions were renamed:
    - `PermRepTrans` was renamed `AsPermOfRange`
    - `ImagesTransformationMonoid` was renamed `ImagesOfTransSemigroup`
    - `GradedImagesTransformationMonoid` was renamed `GradedImagesOfTransSemigroup`
    - `KernelsTransformationMonoid` was renamed `KernelsOfTransSemigroup`
    - `GradedKernelsTransformationMonoid` was renamed `GradedKernelsOfTransSemigroup`
  - the following bugs were fixed:
    - a bug relating to the definition of the semigroup of order preserving functions was resolved
- from 3.1 to 3.1.1: fixed a bug that produced an error when loading MONOID with the GRAPE package present but not fully installed.
- from 2 to 3:
  - new methods for testing if a semigroup satisfies a particular property, such as if it is regular, simple, inverse, or completely regular, see Chapter 5;
  - implementations of new algorithms for computing the automorphism group of an arbitrary semigroup generated by transformations including an interactive function that allows the user to decide how the computation should proceed, see Chapter 7;
  - functions for finding automorphisms of Rees matrix semigroups and Rees `0`-matrix semigroups; see Section 7.5.
  - functions for defining homomorphisms and isomorphisms between some types of semigroups; see Chapter 7.

1.5 Forthcoming Features

The features are currently under development and will be available in a future version of MONOID:

- the number of special types of semigroups available in MONOID will be expanded to include all of the standard examples of transformation semigroups and some matrix semigroups.
- methods analogous to those used to find Green's relations and other structural properties of transformation semigroups in the current version of MONOID but for semigroups generated by partial transformations, binary relations, and matrix semigroups.

- a suite of functions for computing with inverse semigroups generated by partial bijections, including finding faithful representations of smaller degree and small generating sets.
- an algorithm for finding a small generating set of a semigroup.

Chapter 2

Transformations

The functions described in this section extend the functionality of GAP relating to transformations.

2.1 Creating Transformations

2.1.1 TransformationByKernelAndImage

◊ `TransformationByKernelAndImage(`*ker, img*`)` (operation)
◊ `TransformationByKernelAndImageNC(`*ker, img*`)` (operation)

returns the transformation `f` with kernel `ker` and image `img` where `(x)f=img[i]` for all `x` in `ker[i]`. The argument `ker` should be a set of sets that partition the set `1,...n` for some `n` and `img` should be a sublist of `1,...n`.

`TransformationByKernelAndImage` first checks that `ker` and `img` describe the kernel and image of a transformation whereas `TransformationByKernelAndImageNC` performs no such check.

Example

```
gap> TransformationByKernelAndImageNC([[1,2,3,4],[5,6,7],[8]],[1,2,8]);
Transformation( [ 1, 1, 1, 1, 2, 2, 2, 8 ] )
gap> TransformationByKernelAndImageNC([[1,6],[2,5],[3,4]], [4,5,6]);
Transformation( [ 4, 5, 6, 6, 5, 4 ] )
```

2.1.2 AllTransformationsWithKerAndImg

◊ `AllTransformationsWithKerAndImg(`*ker, img*`)` (operation)
◊ `AllTransformationsWithKerAndImgNC(`*ker, img*`)` (operation)

returns a list of all transformations with kernel `ker` and image `img`. The argument `ker` should be a set of sets that partition the set `1,...n` for some `n` and `img` should be a sublist of `1,...n`.

`AllTransformationsWithKerAndImg` first checks that `ker` and `img` describe the kernel and image of a transformation whereas `AllTransformationsWithKerAndImgNC` performs no such check.

Example

```
gap> AllTransformationsWithKerAndImg([[1,6],[2,5],[3,4]], [4,5,6]);
[ Transformation( [ 4, 5, 6, 6, 5, 4 ] ),
  Transformation( [ 6, 5, 4, 4, 5, 6 ] ),
  Transformation( [ 6, 4, 5, 5, 4, 6 ] ),
  Transformation( [ 4, 6, 5, 5, 6, 4 ] ),
```

```
      Transformation( [ 5, 6, 4, 4, 6, 5 ] ),
      Transformation( [ 5, 4, 6, 6, 4, 5 ] ) ]
```

2.1.3 Idempotent

◊ IdempotentNC(*ker, img*) (function)
◊ Idempotent(*ker, img*) (function)

IdempotentNC returns an idempotent with kernel ker and image img without checking IsTransversal (2.2.1) with arguments ker and im.

Idempotent returns an idempotent with kernel ker and image img after checking that IsTransversal (2.2.1) with arguments ker and im returns true.

Example

```
gap> g1:=Transformation([2,2,4,4,5,6]);;
gap> g2:=Transformation([5,3,4,4,6,6]);;
gap> ker:=KernelOfTransformation(g2*g1);;
gap> im:=ImageListOfTransformation(g2);;
gap> Idempotent(ker, im);
Error,  the image must be a transversal of the kernel
[ ... ]
gap> Idempotent([[1,2,3],[4,5],[6,7]], [1,5,6]);
Transformation( [ 1, 1, 1, 5, 5, 6, 6 ] )
gap> IdempotentNC([[1,2,3],[4,5],[6,7]], [1,5,6]);
Transformation( [ 1, 1, 1, 5, 5, 6, 6 ] )
```

2.1.4 RandomIdempotent

◊ RandomIdempotent(*arg*) (operation)
◊ RandomIdempotentNC(*arg*) (operation)

If the argument is a kernel, then a random idempotent is return that has that kernel. A *kernel* is a set of sets that partition the set 1,...n for some n and an *image* is a sublist of 1,...n.

If the first argument is an image img and the second a positive integer n, then a random idempotent of degree n is returned with image img.

The no check version does not check that the arguments can be the kernel and image of an idempotent.

Example

```
gap> RandomIdempotent([[1,2,3], [4,5], [6,7,8]], [1,2,3]);;
fail
gap> RandomIdempotent([1,2,3],5);
Transformation( [ 1, 2, 3, 1, 3 ] )
gap> RandomIdempotent([[1,6], [2,4], [3,5]]);
Transformation( [ 1, 2, 5, 2, 5, 1 ] )
```

2.1.5 RandomTransformation

◊ RandomTransformation(*arg*) (operation)
◊ RandomTransformationNC(*arg*) (operation)

These are new methods for the existing library function RandomTransformation.

If the first argument is a kernel and the second an image, then a random transformation is returned with this kernel and image.A *kernel* is a set of sets that partition the set 1,...n for some n and an *image* is a sublist of 1,...n.

If the argument is a kernel, then a random transformation is returned that has that kernel.

If the first argument is an image img and the second a positive integer n, then a random transformation of degree n is returned with image img.

The no check version does not check that the arguments can be the kernel and image of a transformation.

Example

```
gap> RandomTransformation([[1,2,3], [4,5], [6,7,8]], [1,2,3]);;
Transformation( [ 2, 2, 2, 1, 1, 3, 3, 3 ] )
gap> RandomTransformation([[1,2,3],[5,7],[4,6]]);
Transformation( [ 3, 3, 3, 6, 1, 6, 1 ] )
gap> RandomTransformation([[1,2,3],[5,7],[4,6]]);
Transformation( [ 4, 4, 4, 7, 3, 7, 3 ] )
gap> RandomTransformationNC([[1,2,3],[5,7],[4,6]]);
Transformation( [ 1, 1, 1, 7, 5, 7, 5 ] )
gap> RandomTransformation([1,2,3], 6);
Transformation( [ 2, 1, 2, 1, 1, 2 ] )
gap> RandomTransformationNC([1,2,3], 6);
Transformation( [ 3, 1, 2, 2, 1, 2 ] )
```

2.1.6 TransformationActionNC

◊ TransformationActionNC(*list, act, elm*) (operation)

returns the list list acted on by elm via the action act.

Example

```
gap> mat:=OneMutable(GeneratorsOfGroup(GL(3,3))[1]);
[ [ Z(3)^0, 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, 0*Z(3) ],
  [ 0*Z(3), 0*Z(3), Z(3)^0 ] ]
gap> mat[3][3]:=Z(3)*0;
0*Z(3)
gap> F:=BaseDomain(mat);
GF(3)
gap> TransformationActionNC(Elements(F^3), OnRight, mat);
Transformation( [ 1, 1, 1, 4, 4, 4, 7, 7, 7, 10, 10, 10, 13, 13, 13, 16, 16,
  16, 19, 19, 19, 22, 22, 22, 25, 25, 25 ] )
```

2.2 Properties & Attributes

2.2.1 IsTransversal

◊ IsTransversal(*list1, list2*) (function)

returns true if the list list2 is a transversal of the list of lists list1. That is, if every list in list1 contains exactly one element in list2.

Example

```
gap> g1:=Transformation([2,2,4,4,5,6]);;
gap> g2:=Transformation([5,3,4,4,6,6]);;
gap> ker:=KernelOfTransformation(g2*g1);
[ [ 1 ], [ 2, 3, 4 ], [ 5, 6 ] ]
gap> im:=ImageListOfTransformation(g2);
[ 5, 3, 4, 4, 6, 6 ]
gap> IsTransversal(ker, im);
false
gap> IsTransversal([[1,2,3],[4,5],[6,7]], [1,5,6]);
true
```

2.2.2 IsKerImgOfTransformation

◊ IsKerImgOfTransformation(*ker, img*) (function)

returns true if the arguments ker and img can be the kernel and image of a single transformation, respectively. The argument ker should be a set of sets that partition the set 1,...n for some n and img should be a sublist of 1,...n.

Example

```
gap> ker:=[[1,2,3],[5,6],[8]];
[ [ 1, 2, 3 ], [ 5, 6 ], [ 8 ] ]
gap> img:=[1,2,9];
[ 1, 2, 9 ]
gap> IsKerImgOfTransformation(ker,img);
false
gap> ker:=[[1,2,3,4],[5,6,7],[8]];
[ [ 1, 2, 3, 4 ], [ 5, 6, 7 ], [ 8 ] ]
gap> IsKerImgOfTransformation(ker,img);
false
gap> img:=[1,2,8];
[ 1, 2, 8 ]
gap> IsKerImgOfTransformation(ker,img);
true
```

2.2.3 KerImgOfTransformation

◊ KerImgOfTransformation(*f*) (operation)

returns the kernel and image set of the transformation f. These attributes of f can be obtain separately using KernelOfTransformation (**Reference: KernelOfTransformation**) and ImageSetOfTransformation (**Reference: ImageSetOfTransformation**), respectively.

Example

```
gap> t:=Transformation( [ 10, 8, 7, 2, 8, 2, 2, 6, 4, 1 ] );;
gap> KerImgOfTransformation(t);
[ [ [ 1 ], [ 2, 5 ], [ 3 ], [ 4, 6, 7 ], [ 8 ], [ 9 ], [ 10 ] ],
  [ 1, 2, 4, 6, 7, 8, 10 ] ]
```

2.2.4 IsRegularTransformation

◊ IsRegularTransformation(*S*, *f*) (operation)

if f is a regular element of the transformation semigroup S, then true is returned. Otherwise false is returned.

A transformation f is regular inside a transformation semigroup S if it lies inside a regular D-class. This is equivalent to the orbit of the image of f containing a transversal of the kernel of f.

Example

```
gap> g1:=Transformation([2,2,4,4,5,6]);;
gap> g2:=Transformation([5,3,4,4,6,6]);;
gap> m1:=Monoid(g1,g2);;
gap> IsRegularTransformation(m1, g1);
true
gap> img:=ImageSetOfTransformation(g1);
[ 2, 4, 5, 6 ]
gap> ker:=KernelOfTransformation(g1);
[ [ 1, 2 ], [ 3, 4 ], [ 5 ], [ 6 ] ]
gap> ForAny(MonoidOrbit(m1, img), x-> IsTransversal(ker, x));
true
gap> IsRegularTransformation(m1, g2);
false
gap> IsRegularTransformation(FullTransformationSemigroup(6), g2);
true
```

2.2.5 IndexPeriodOfTransformation

◊ IndexPeriodOfTransformation(*f*) (attribute)

returns the minimum numbers m, r such that f^(m+r)=f^m; known as the *index* and *period* of the transformation.

Example

```
gap> f:=Transformation( [ 3, 4, 4, 6, 1, 3, 3, 7, 1 ] );;
gap> IndexPeriodOfTransformation(f);
[ 2, 3 ]
gap> f^2=f^5;
true
```

2.2.6 SmallestIdempotentPower

◊ SmallestIdempotentPower(*f*) (attribute)

returns the least natural number n such that the transformation f^n is an idempotent.

Example

```
gap> t:=Transformation( [ 6, 7, 4, 1, 7, 4, 6, 1, 3, 4 ] );;
gap> SmallestIdempotentPower(t);
6
gap> t:=Transformation( [ 6, 6, 6, 2, 7, 1, 5, 3, 10, 6 ] );;
gap> SmallestIdempotentPower(t);
4
```

2.2.7 InversesOfTransformation

◊ InversesOfTransformation(*S, f*) (operation)
◊ InversesOfTransformationNC(*S, f*) (operation)

returns a list of the inverses of the transformation f in the transformation semigroup S. The function InversesOfTransformationNC does not check that f is an element of S.

Example

```
gap> S:=Semigroup([ Transformation( [ 3, 1, 4, 2, 5, 2, 1, 6, 1 ] ),
  Transformation( [ 5, 7, 8, 8, 7, 5, 9, 1, 9 ] ),
  Transformation( [ 7, 6, 2, 8, 4, 7, 5, 8, 3 ] ) ]);;
gap> f:=Transformation( [ 3, 1, 4, 2, 5, 2, 1, 6, 1 ] );;
gap> InversesOfTransformationNC(S, f);
[  ]
gap> IsRegularTransformation(S, f);
false
gap> f:=Transformation( [ 1, 9, 7, 5, 5, 1, 9, 5, 1 ] );;
gap> inv:=InversesOfTransformation(S, f);
[ Transformation( [ 1, 5, 1, 1, 5, 1, 3, 1, 2 ] ),
  Transformation( [ 1, 5, 1, 2, 5, 1, 3, 2, 2 ] ),
  Transformation( [ 1, 2, 3, 5, 5, 1, 3, 5, 2 ] ) ]
gap> IsRegularTransformation(S, f);
true
```

2.3 Changing Representation

2.3.1 AsBooleanMatrix

◊ AsBooleanMatrix(*f[, n]*) (operation)

returns the transformation or permutation f represented as an n by n Boolean matrix where i,f(i)th entries equal 1 and all other entries are 0.

If f is a transformation, then n is the size of the domain of f.

If f is a permutation, then n is the number of points moved by f.

Example

```
gap> t:=Transformation( [ 4, 2, 2, 1 ] );;
gap> AsBooleanMatrix(t);
[ [ 0, 0, 0, 1 ], [ 0, 1, 0, 0 ], [ 0, 1, 0, 0 ], [ 1, 0, 0, 0 ] ]
gap> t:=(1,4,5);;
gap> AsBooleanMatrix(t);
[ [ 0, 0, 0, 1, 0 ], [ 0, 1, 0, 0, 0 ], [ 0, 0, 1, 0, 0 ], [ 0, 0, 0, 0, 1 ],
  [ 1, 0, 0, 0, 0 ] ]
gap> AsBooleanMatrix(t,3);
fail
gap> AsBooleanMatrix(t,5);
[ [ 0, 0, 0, 1, 0 ], [ 0, 1, 0, 0, 0 ], [ 0, 0, 1, 0, 0 ], [ 0, 0, 0, 0, 1 ],
  [ 1, 0, 0, 0, 0 ] ]
gap> AsBooleanMatrix(t,6);
[ [ 0, 0, 0, 1, 0, 0 ], [ 0, 1, 0, 0, 0, 0 ], [ 0, 0, 1, 0, 0, 0 ],
  [ 0, 0, 0, 0, 1, 0 ], [ 1, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 1 ] ]
```

2.3.2 AsPermOfRange

◊ AsPermOfRange(*x*) (operation)

converts a transformation x that is a permutation of its image into that permutation.

Example

```
gap> t:=Transformation([1,2,9,9,9,8,8,8,4]);
Transformation( [ 1, 2, 9, 9, 9, 8, 8, 8, 4 ] )
gap> AsPermOfRange(t);
(4,9)
gap> t*last;
Transformation( [ 1, 2, 4, 4, 4, 8, 8, 8, 9 ] )
gap> AsPermOfRange(last);
()
```

Chapter 3

Monoid Actions and Orbits

3.1 Introduction

The functions described in this section relate to how a transformation semigroup acts on its underlying set.

Let S be a transformation semigroup of degree n. Then the *orbit* of i in {1,...,n} is the set of elements j in {1,...,n} such that there exists f in S where (i)f=j. Note that the essential difference between monoid orbits and group orbits is that monoid orbits do not describe an equivalence relation on S. In particular, the relation described by monoid orbits is not symmetric.

The *strong orbit* of i in {1,...,n} is the set of elements j in {1,...,n} such that there exists f, g in S where (i)f=j and (j)g=i.

Recall that a *grading* is a function f from a transformation semigroup S of degree n to the natural numbers such that if s in S and X is a subset of {1,...,n}, then (Xs)f is at most (X)f.

3.2 Actions

In addition to the actions define in the reference manual (**Reference: Basic Actions**) the following two actions are available in MONOID.

3.2.1 OnTuplesOfSetsAntiAction

◊ OnTuplesOfSetsAntiAction(*tup, f*) (function)

returns the preimages of each of the sets in the tuple of sets tup under the transformation f. The tuple of sets tup can have any number of elements.

Example

```
gap> f:=Transformation( [ 8, 7, 5, 3, 1, 3, 8, 8 ] );;
gap> OnTuplesOfSetsAntiAction([ [ 1, 2 ], [ 3 ], [ 4 ], [ 5 ] ], f);
[ [ 5 ], [ 4, 6 ], [ 3 ] ]
```

3.2.2 OnKernelsAntiAction

◊ OnKernelsAntiAction(*ker, f*) (function)

returns the kernel of the product `f*g` of the transformation `f` with a transformation `g` having kernel `ker`.

Example

```
gap> f:=Transformation( [ 8, 7, 5, 3, 1, 3, 8, 8 ] );;
gap> OnKernelsAntiAction([ [ 1, 2 ], [ 3 ], [ 4 ], [ 5 ], [ 6, 7, 8 ] ], f);
[ [ 1, 2, 7, 8 ], [ 3 ], [ 4, 6 ], [ 5 ] ]
```

3.3 General Orbits

The following functions allow the calculation of arbitrary orbits in transformation semigroups. Several more specific orbits that are often useful are given in Section 3.4.

3.3.1 MonoidOrbit

◊ `MonoidOrbit(S, obj[, act])` (operation)

returns the orbit of the object `obj` under the action `act` of the transformation collection `S`. Usually, `obj` would be a point, list of points, or list of lists, and `act` would be `OnPoints` (**Reference: OnPoints**), `OnSets` (**Reference: OnSets**), `OnTuples` (**Reference: OnTuples**), or another action defined in (**Reference: Basic Actions**). The argument `act` can be any function.

If the optional third argument `act` is not given, then `OnPoints` (**Reference: OnPoints**), `OnSets` (**Reference: OnSets**), or `OnSetsSets` (**Reference: OnSetsSets**) is used as the default action depending on what `obj` is.

Further details can be found in Algorithm A and B of [LPRRa].

Example

```
gap> g1:=Transformation([3,3,2,6,2,4,4,6,3,4,6]);;
gap> g2:=Transformation([4,4,6,1,3,3,3,3,11,11,11]);;
gap> g3:=Transformation([2,2,3,4,4,6,6,6,6,6,11]);;
gap> S:=Monoid(g1,g2,g3);;
gap> MonoidOrbit(S, 1);
[ 1, 3, 4, 2, 6 ]
gap> MonoidOrbit(S, [1,2], OnSets);
[ [ 1, 2 ], [ 3 ], [ 4 ], [ 2 ], [ 6 ], [ 1 ] ]
gap> MonoidOrbit(S, [1,2], OnTuples);
[ [ 1, 2 ], [ 3, 3 ], [ 4, 4 ], [ 2, 2 ], [ 6, 6 ], [ 1, 1 ] ]
gap> MonoidOrbit(S, 2, OnPoints);
[ 2, 3, 4, 6, 1 ]
```

3.3.2 MonoidOrbits

◊ `MonoidOrbits(S, list[, act])` (operation)

returns a list of the orbits of the elements of `list` under the action `act` of the transformation collection `S` using the `MonoidOrbit` (3.3.1) function.

If the optional third argument `act` is not given, then `OnPoints` (**Reference: OnPoints**), `OnSets` (**Reference: OnSets**), or `OnSetsSets` (**Reference: OnSetsSets**) is used as the default action depending on what `obj` is.

Further details can be found in Algorithm A and B of [LPRRa].

Example

```
gap> g1:=Transformation([3,3,2,6,2,4,4,6,3,4,6]);;
gap> g2:=Transformation([4,4,6,1,3,3,3,3,11,11,11]);;
gap> g3:=Transformation([2,2,3,4,4,6,6,6,6,6,11]);;
gap> S:=Monoid(g1,g2,g3);;
gap> MonoidOrbits(S, [1,2]);
[ [ 1, 3, 4, 2, 6 ], [ 2, 3, 4, 6, 1 ] ]
gap> MonoidOrbits(S, [[1,2], [2,3]], OnSets);
[ [ [ 1, 2 ], [ 3 ], [ 4 ], [ 2 ], [ 6 ], [ 1 ] ],
  [ [ 2, 3 ], [ 4, 6 ], [ 1, 3 ] ] ]
```

3.3.3 StrongOrbit

◊ StrongOrbit(*S, obj[, act]*) (operation)

returns the strong orbit of obj under the action act of the transformation collection S. Usually, obj would be a point, list of points, or list of lists, and act would be OnPoints (**Reference: OnPoints**), OnSets (**Reference: OnSets**), OnTuples (**Reference: OnTuples**), or another action defined in (**Reference: Basic Actions**). The argument act can be any function.

If the optional third argument act is not given and obj is a point, then OnPoints (**Reference: OnPoints**) is the default action.

Further details can be found in Algorithm A and B of [LPRRa].

Example

```
gap> g1:=Transformation([3,3,2,6,2,4,4,6,3,4,6]);;
gap> g2:=Transformation([4,4,6,1,3,3,3,3,11,11,11]);;
gap> g3:=Transformation([2,2,3,4,4,6,6,6,6,6,11]);;
gap> S:=Monoid(g1,g2,g3);;
gap> StrongOrbit(S, 4, OnPoints);
[ 1, 3, 2, 4, 6 ]
gap> StrongOrbit(S, 4);
[ 1, 3, 2, 4, 6 ]
gap> StrongOrbit(S, [2,3], OnSets);
[ [ 2, 3 ], [ 4, 6 ], [ 1, 3 ] ]
gap> StrongOrbit(S, [2,3], OnTuples);
[ [ 2, 3 ], [ 3, 2 ], [ 4, 6 ], [ 6, 4 ], [ 1, 3 ], [ 3, 1 ] ]
```

3.3.4 StrongOrbits

◊ StrongOrbits(*S, list[, act]*) (operation)

returns a list of the strong orbits of the elements of list under the action act of the transformation collection S using the StrongOrbit (3.3.3) function.

If the optional third argument act is not given, then OnPoints (**Reference: OnPoints**), OnSets (**Reference: OnSets**), or OnSetsSets (**Reference: OnSetsSets**) is used as the default action depending on what obj is.

Further details can be found in Algorithm A and B of [LPRRa].

Example

```
gap> g1:=Transformation([3,3,2,6,2,4,4,6,3,4,6]);;
gap> g2:=Transformation([4,4,6,1,3,3,3,3,11,11,11]);;
```

```
gap> g3:=Transformation([2,2,3,4,4,6,6,6,6,6,11]);;
gap> S:=Monoid(g1,g2,g3);;
gap> StrongOrbits(S, [1..6]);
[ [ 1, 3, 2, 4, 6 ], [ 5 ] ]
gap> StrongOrbits(S, [[1,2],[2,3]], OnSets);
[ [ [ 1, 2 ] ], [ [ 2, 3 ], [ 4, 6 ], [ 1, 3 ] ] ]
gap> StrongOrbits(S, [[1,2],[2,3]], OnTuples);
[ [ [ 1, 2 ] ], [ [ 2, 3 ], [ 3, 2 ], [ 4, 6 ], [ 6, 4 ],
  [ 1, 3 ], [ 3, 1 ] ] ]
```

3.3.5 GradedOrbit

◊ GradedOrbit(*S, obj[, act], grad*) (operation)

returns the orbit of the object obj under the action act of the transformation collection S partitioned by the grading grad. That is, two elements lie in the same class if they have the same value under grad.

(Recall that a *grading* is a function f from a transformation semigroup S of degree n to the natural numbers such that if s in S and X is a subset of {1,...,n}, then (Xs)f is at most (X)f.)

Note that this function will not check if grad actually defines a grading or not.

If the optional third argument act is not given, then OnPoints (**Reference: OnPoints**), OnSets (**Reference: OnSets**), or OnSetsSets (**Reference: OnSetsSets**) is used as the default action depending on what obj is.

Further details can be found in Algorithm A and B of [LPRRa].

Example

```
gap> g1:=Transformation([3,3,2,6,2,4,4,6,3,4,6]);;
gap> g2:=Transformation([4,4,6,1,3,3,3,3,11,11,11]);;
gap> g3:=Transformation([2,2,3,4,4,6,6,6,6,6,11]);;
gap> S:=Monoid(g1,g2,g3);;
gap> GradedOrbit(S, [1,2], OnSets, Size);
[ [ [ 3 ], [ 4 ], [ 2 ], [ 6 ], [ 1 ] ], [ [ 1, 2 ] ] ]
gap> GradedOrbit(S, [1,2], Size);
[ [ [ 3 ], [ 4 ], [ 2 ], [ 6 ], [ 1 ] ], [ [ 1, 2 ] ] ]
gap> GradedOrbit(S, [1,3,4], OnTuples, function(x)
> if 1 in x then return 2;
> else return 1;
> fi;
> end);
[ [ [ 3, 2, 6 ], [ 2, 3, 4 ], [ 6, 4, 3 ], [ 4, 6, 2 ] ],
  [ [ 1, 3, 4 ], [ 4, 6, 1 ], [ 3, 1, 6 ] ] ]
```

3.3.6 ShortOrbit

◊ ShortOrbit(*S, obj[, act], grad*) (operation)

returns the elements of the orbit of obj under the action act of the transformation collection S with the same value as obj under the grading grad.

(Recall that a *grading* is a function f from a transformation semigroup S of degree n to the natural numbers such that if s in S and X is a subset of {1,...,n}, then (Xs)f is at most (X)f.)

Note that this function will not check if grad actually defines a grading or not.

If the optional third argument act is not given, then OnPoints (**Reference: OnPoints**), OnSets (**Reference: OnSets**), or OnSetsSets (**Reference: OnSetsSets**) is used as the default action depending on what obj is.

Further details can be found in Algorithm A and B of [LPRRa].

Example

```
gap> g1:=Transformation([3,3,2,6,2,4,4,6,3,4,6]);;
gap> g2:=Transformation([4,4,6,1,3,3,3,3,11,11,11]);;
gap> g3:=Transformation([2,2,3,4,4,6,6,6,6,6,11]);;
gap> S:=Monoid(g1,g2,g3);;
gap> ShortOrbit(S, [1,2], Size);
[ [ 1, 2 ] ]
gap> ShortOrbit(S, [2,4], Size);
[ [ 2, 4 ], [ 3, 6 ], [ 1, 4 ] ]
gap> ShortOrbit(S, [1,2], OnTuples, Size);
[ [ 1, 2 ], [ 3, 3 ], [ 4, 4 ], [ 2, 2 ], [ 6, 6 ], [ 1, 1 ] ]
```

3.3.7 ShortStrongOrbit

◊ ShortStrongOrbit(*S, obj[, act], grad*) (operation)

returns the strong orbit of obj under the action act of the transformation collection S. During the computation of this orbit the grading grad is used to quickly disregard elements that cannot be in the strong orbit as their value under grad is strictly less than the value of obj under grad. (Recall that a *grading* is a function f from a transformation semigroup S of degree n to the natural numbers such that if s in S and X is a subset of {1,...,n}, then (Xs)f is at most (X)f.)

Note that this function will not check if grad actually defines a grading or not.

If the optional third argument act is not given, then OnPoints (**Reference: OnPoints**), OnSets (**Reference: OnSets**), or OnSetsSets (**Reference: OnSetsSets**) is used as the default action depending on what obj is.

Further details can be found in Algorithm A and B of [LPRRa].

Example

```
gap> g1:=Transformation([3,3,2,6,2,4,4,6,3,4,6]);;
gap> g2:=Transformation([4,4,6,1,3,3,3,3,11,11,11]);;
gap> g3:=Transformation([2,2,3,4,4,6,6,6,6,6,11]);;
gap> S:=Monoid(g1,g2,g3);;
gap> ShortStrongOrbit(m8, [1,3,4], OnTuples, Size);
[ [ 1, 3, 4 ], [ 3, 2, 6 ], [ 2, 3, 4 ], [ 4, 6, 1 ], [ 6, 4, 3 ],
  [ 4, 6, 2 ], [ 3, 1, 6 ] ]
```

3.3.8 StrongOrbitsInForwardOrbit

◊ StrongOrbitsInForwardOrbit(*s, x, act*) (operation)

returns a list of the strong orbits contained in the orbit of x under the action act of the transformation collection s.

Example

```
gap> gens:=[ Transformation( [ 1, 3, 4, 1 ] ), Transformation( [ 2, 4, 1, 2 ] ),
> Transformation( [ 3, 1, 1, 3 ] ), Transformation( [ 3, 3, 4, 1 ] ) ];;
```

```
gap> s:=Semigroup(gens);;
gap> StrongOrbitsInForwardOrbit(s, [1,3,4], OnSets);
[ [ [ 3 ], [ 1 ], [ 2 ], [ 4 ] ], [ [ 1, 4 ], [ 1, 3 ], [ 1, 2 ], [ 2, 4 ], [ 3, 4 ] ],
  [ [ 1, 3, 4 ] ] ]
```

3.4 Some Specific Orbits

The following specific orbits are used in the computation of Green's relations and to test if an arbitrary transformation semigroup has a particular property; see Chapter 4 and Chapter 5.

3.4.1 ImagesOfTransSemigroup

◊ ImagesOfTransSemigroup(*S[, n]*) (attribute)

returns the set of all the image sets that elements of S admit. That is, the union of the orbits of the image sets of the generators of S under the action OnSets (**Reference: OnSets**).

If the optional second argument n (a positive integer) is present, then the list of image sets of size n is returned. If you are only interested in the images of a given size, then the second version of the function will likely be faster.

Example

```
gap>  S:=Semigroup([ Transformation( [ 6, 4, 4, 4, 6, 1 ] ),
> Transformation( [ 6, 5, 1, 6, 2, 2 ] ) ];;
gap> ImagesOfTransSemigroup(S, 6);
[  ]
gap> ImagesOfTransSemigroup(S, 5);
[  ]
gap> ImagesOfTransSemigroup(S, 4);
[ [ 1, 2, 5, 6 ] ]
gap> ImagesOfTransSemigroup(S, 3);
[ [ 1, 4, 6 ], [ 2, 5, 6 ] ]
gap> ImagesOfTransSemigroup(S, 2);
[ [ 1, 4 ], [ 2, 5 ], [ 2, 6 ], [ 4, 6 ] ]
gap> ImagesOfTransSemigroup(S, 1);
[ [ 1 ], [ 2 ], [ 4 ], [ 5 ], [ 6 ] ]
gap> ImagesOfTransSemigroup(S);
[ [ 1 ], [ 1, 2, 5, 6 ], [ 1, 4 ], [ 1, 4, 6 ], [ 2 ], [ 2, 5 ], [ 2, 5, 6 ],
  [ 2, 6 ], [ 4 ], [ 4, 6 ], [ 5 ], [ 6 ] ]
```

3.4.2 GradedImagesOfTransSemigroup

◊ GradedImagesOfTransSemigroup(*S*) (attribute)

returns the set of all the image sets that elements of S admit in a list where the ith entry contains all the images with size i (including the empty list when there are no image sets with size i).

This is just the union of the orbits of the images of the generators of S under the action OnSets (**Reference: OnSets**) graded according to size.

Example

```
gap> gens:=[ Transformation( [ 1, 5, 1, 1, 1 ] ),
> Transformation( [ 4, 4, 5, 2, 2 ] ) ];;
gap> S:=Semigroup(gens);;
gap> GradedImagesOfTransSemigroup(S);
[ [ [ 1 ], [ 4 ], [ 2 ], [ 5 ] ], [ [ 1, 5 ], [ 2, 4 ] ], [ [ 2, 4, 5 ] ],
[  ], [ [ 1 .. 5 ] ] ]
```

3.4.3 KernelsOfTransSemigroup

◊ KernelsOfTransSemigroup(*S[, n]*) (attribute)

returns the set of all the kernels that elements of S admit. This is just the union of the orbits of the kernels of the generators of S under the action OnKernelsAntiAction (3.2.2).

If the optional second argument n (a positive integer) is present, then the list of image sets of size n is returned. If you are only interested in the images of a given size, then the second version of the function will likely be faster.

Example

```
gap>  S:=Semigroup([ Transformation( [ 2, 4, 1, 2 ] ),
> Transformation( [ 3, 3, 4, 1 ] ) ]);
gap>  KernelsOfTransSemigroup(S);
[ [ [ 1, 2 ], [ 3 ], [ 4 ] ], [ [ 1, 2 ], [ 3, 4 ] ], [ [ 1, 2, 3 ],
[ 4 ] ],
  [ [ 1, 2, 3, 4 ] ], [ [ 1, 2, 4 ], [ 3 ] ], [ [ 1, 3, 4 ], [ 2 ] ],
  [ [ 1, 4 ], [ 2 ], [ 3 ] ], [ [ 1, 4 ], [ 2, 3 ] ] ]
gap>  KernelsOfTransSemigroup(S,1);
[ [ [ 1, 2, 3, 4 ] ] ]
gap>  KernelsOfTransSemigroup(S,2);
[ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 1, 2, 3 ], [ 4 ] ], [ [ 1, 2, 4 ], [ 3 ] ],
  [ [ 1, 3, 4 ], [ 2 ] ], [ [ 1, 4 ], [ 2, 3 ] ] ]
gap>  KernelsOfTransSemigroup(S,3);
[ [ [ 1, 2 ], [ 3 ], [ 4 ] ], [ [ 1, 4 ], [ 2 ], [ 3 ] ] ]
gap>  KernelsOfTransSemigroup(S,4);
[  ]
```

3.4.4 GradedKernelsOfTransSemigroup

◊ GradedKernelsOfTransSemigroup(*S*) (attribute)

returns the set of all the kernels that elements of S admit in a list where the ith entry contains all the kernels with i classes (including the empty list when there are no kernels with i classes).

This is just the union of the orbits of the kernels of the generators of S under the action OnKernelsAntiAction (3.2.2) graded according to size.

Example

```
gap> gens:=[ Transformation( [ 1, 1, 2, 1, 4 ] ),
> Transformation( [ 2, 5, 3, 2, 3 ] ) ];;
gap> S:=Semigroup(gens);;
gap> GradedKernelsOfTransSemigroup(S);
[ [ [ [ 1, 2, 3, 4, 5 ] ] ],
  [ [ [ 1, 2, 4, 5 ], [ 3 ] ], [ [ 1, 4 ], [ 2, 3, 5 ] ],
```

```
          [ [ 1, 2, 4 ], [ 3, 5 ] ] ],
      [ [ [ 1, 2, 4 ], [ 3 ], [ 5 ] ], [ [ 1, 4 ], [ 2 ], [ 3, 5 ] ] ], [  ],
      [ [ [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ] ] ] ]
```

3.4.5 StrongOrbitOfImage

◇ `StrongOrbitOfImage(`*S*`, `*f*`)` (operation)

returns a triple where

- the first entry `imgs` is the strong orbit of the image set `A` of `f` under the action of `S`. That is, the set of image sets `B` of elements of `S` such that there exist `g,h` in `S` with `OnSets(A, g)=B` and `OnSet(B, h)=A`. If the strong orbit of the image of `f` has already been calculated, then the image of `f` might not be the first entry in the list `imgs`.
- the second entry is a list of permutations `mults` such that `OnSets(imgs[i], mults[i])=imgs[1]`.
- the third entry is the Right Schutzenberger group associated to the first entry in the list `imgs` (that is, the group of permutations arising from elements of the semigroup that stabilise the set `imgs[1]`).

Example

```
gap> gens:=[ Transformation( [ 3, 5, 2, 5, 1 ] ),
> Transformation( [ 4, 3, 2, 1, 5 ] ) ];;
gap> S:=Semigroup(gens);;
gap> f:=Transformation( [ 2, 1, 1, 1, 5 ] );;
gap> StrongOrbitOfImage(S, f);
[ [ [ 1, 2, 5 ], [ 1, 3, 5 ], [ 1, 2, 3 ], [ 2, 3, 5 ], [ 2, 3, 4 ],
      [ 2, 4, 5 ], [ 3, 4, 5 ] ],
  [ (), (1,5,2,3), (1,2)(3,5,4), (1,3,2,5), (1,3)(2,5,4), (1,3,4,5,2),
      (1,3,2,4) ], Group([ (), (2,5), (1,5) ]) ]
```

3.4.6 StrongOrbitsOfImages

◇ `StrongOrbitsOfImages(`*S*`)` (attribute)

this is a mutable attribute that stores the result of `StrongOrbitOfImage` (3.4.5) every time it is used. If `StrongOrbitOfImage` (3.4.5) has not been invoked for `S`, then an error is returned.

Example

```
gap> gens:=[ Transformation( [ 3, 5, 2, 5, 1 ] ),
> Transformation( [ 4, 3, 2, 1, 5 ] ) ];;
gap> S:=Semigroup(gens);;
gap> f:=Transformation( [ 2, 1, 1, 1, 5 ] );;
gap> StrongOrbitOfImage(S, f);;
gap> StrongOrbitsOfImages(S);
[ [ [ [ 1, 2, 5 ], [ 1, 3, 5 ], [ 1, 2, 3 ], [ 2, 3, 5 ], [ 2, 3, 4 ],
          [ 2, 4, 5 ], [ 3, 4, 5 ] ] ],
  [ [ (), (1,5,2,3), (1,2)(3,5,4), (1,3,2,5), (1,3)(2,5,4), (1,3,4,5,2),
          (1,3,2,4) ] ], [ Group([ (), (2,5), (1,5) ]) ] ]
gap> f:=Transformation( [ 5, 5, 5, 5, 2 ] );
```

```
gap> StrongOrbitOfImage(S, f);;
gap> StrongOrbitsOfImages(S);
[ [ [ [ 1, 2, 5 ], [ 1, 3, 5 ], [ 1, 2, 3 ], [ 2, 3, 5 ], [ 2, 3, 4 ],
          [ 2, 4, 5 ], [ 3, 4, 5 ] ],
      [ [ 2, 5 ], [ 1, 5 ], [ 1, 3 ], [ 2, 3 ], [ 2, 4 ], [ 4, 5 ], [ 3, 5 ],
          [ 1, 2 ], [ 3, 4 ] ] ],
  [ [ (), (1,5,2,3), (1,2)(3,5,4), (1,3,2,5), (1,3)(2,5,4), (1,3,4,5,2),
          (1,3,2,4) ],
      [ (), (1,5,2), (1,2)(3,5,4), (2,5,4,3), (2,5,4), (2,3,4,5), (2,3),
          (1,5,4,3), (2,3)(4,5) ] ],
  [ Group([ (), (2,5), (1,5) ]), Group([ (), (2,5) ]) ] ]
```

Chapter 4

Green's Relations

4.1 Introduction

This chapter contains instructions on how to use the functions for computing Green's relations and related notions for transformation semigroups and monoids that are implemented in MONOID.

The theory behind these algorithms is developed in [LPRRb] and the algorithms themselves are described in [LPRRa]. Another reference is [LM90].

Green's relations can be calculated when MONOID is loaded using the same commands that you would used when MONOID is not loaded; see (**Reference: Semigroups**). For example, in GAP with the MONOID package loaded:

Example

```
gap> a:=Transformation([2,1,1,2,1]);;
gap> b:=Transformation([3,4,3,4,4]);;
gap> c:=Transformation([3,4,3,4,3]);;
gap> d:=Transformation([4,3,3,4,4]);;
gap> S:=Semigroup(a,b,c,d);
<semigroup with 4 generators>
gap> GreensRClasses(S);
[ {Transformation( [ 2, 1, 1, 2, 1 ] )}, {Transformation( [ 1, 2, 1, 2, 2 ] )}
  , {Transformation( [ 1, 2, 1, 2, 1 ] )},
{Transformation( [ 2, 1, 1, 2, 2 ] )} ]
```

Without the MONOID package loaded:

Example

```
gap> a:=Transformation([2,1,1,2,1]);;
gap> b:=Transformation([3,4,3,4,4]);;
gap> c:=Transformation([3,4,3,4,3]);;
gap> d:=Transformation([4,3,3,4,4]);;
gap> S:=Semigroup(a,b,c,d);
<semigroup with 4 generators>
gap> GreensRClasses(S);
[ {Transformation( [ 1, 2, 1, 2, 1 ] )}, {Transformation( [ 1, 2, 1, 2, 2 ] )}
  , {Transformation( [ 1, 2, 2, 1, 1 ] )},
{Transformation( [ 1, 2, 2, 1, 2 ] )} ]
```

The only noticable differences are the representatives of the classes and the order the classes appear in the list. These differences are caused by the differences in the methods for `GreensRClasses` in MONOID and the GAP library.

Most of the commands in this section relate to how Green's relations are calculated in MONOID. Although some of the commands might be used for other purposes, if all that is required is to calculate Green's classes, relations and so on, then this is done in the exactly the same way as described in the GAP manual; see (**Reference: Green's Relations**).

Due to inherent difficulties with computing Green's L- and D-classes, the methods used to compute with Green's R-classes are the most efficient in MONOID. Thus wherever possible it is advisable to use the commands relating to Green's R-classes rather than those relating to Green's L-classes, D-classes, or H-classes.

For small examples of semigroups, say with fewer than 40 elements, it may be more efficient to use the methods from the library than those in MONOID. This stems from the fact that there are higher overheads in the methods used in MONOID. In either case, with such small examples computing Green's relations does not take much time.

The methods in MONOID allow the computation of individual Green's classes without the need to compute all the elements of the underlying semigroup. It is also possible to compute all the R-classes, the number of elements and test membership in a transformation semigroup without computing all the elements. This may be useful if you want to study a very large semigroup where computing all the elements of the semigroup is infeasible.

4.2 Data Structures

4.2.1 GreensData

◊ `GreensData(C)` (operation)

- if `C` satisfies `IsGreensRClass` (**Reference: IsGreensRClass**), then `GreensData` returns the value of `GreensRClassData` (4.2.2) with argument `C`.
- if `C` satisfies `IsGreensLClass` (**Reference: IsGreensLClass**), then `GreensData` returns the value of `GreensLClassData` (4.2.3) with argument `C`.
- if `C` satisfies `IsGreensHClass` (**Reference: IsGreensHClass**), then `GreensData` returns the value of `GreensHClassData` (4.2.4) with argument `C`.
- if `C` satisfies `IsGreensDClass` (**Reference: IsGreensDClass**), then `GreensData` returns the value of `GreensDClassData` (4.2.5) with argument `C`.

4.2.2 GreensRClassData

◊ `GreensRClassData(C)` (attribute)

if `C` satisfies `IsGreensRClass` (**Reference: IsGreensRClass**), then `GreensRClassData` returns an object in the category `IsGreensRClassData` (4.2.6) with representation `IsGreensRClassDataRep` (4.2.8) and the following four components:

- `rep` the representative of the *R*-class.
- `strongorb` the strong orbit of the image of `rep` under the action of the semigroup on sets.

- `perms` a list of permutations that map the image of `rep` to the corresponding image in `strongorb`, that is, `OnSets(strongorb[i], perms[i])=strongorb[1]`.

- `schutz` the group of permutations arising from elements of the semigroup that stabilise the image of `rep` (called the *(generalized) right Schutzenberger group*).

The components `strongorb`, `perms`, and `schutz` are obtained using the function `StrongOrbitOfImage` (3.4.5). Further details can be found in Algorithm C, D, E, F, and U of [LPRRa].

Example

```
gap> a:=Transformation( [ 2, 1, 4, 5, 6, 3 ] );;
gap> b:=Transformation( [ 2, 3, 1, 5, 4, 1 ] );;
gap> M:=Semigroup(a,b);;
gap> rc:=GreensRClassOfElement(M, a*b*a);
{Transformation( [ 4, 1, 6, 5, 2, 2 ] )}
gap> GreensRClassData(rc);
GreensRClassData( Transformation( [ 4, 1, 6, 5, 2, 2 ] ), [ [ 1, 2, 4, 5, 6 ],
[ 1, 2, 3, 5, 6 ], [ 1, 2, 3, 4, 6 ], [ 1, 2, 3, 4, 5 ] ], [ (), (1,2)
(3,6,5,4), (3,5)(4,6), (1,6,3,2)(4,5) ], Group( [ (), (2,4,6), (2,6,4),
(1,2,6)(4,5) ] ) )
```

4.2.3 GreensLClassData

◊ `GreensLClassData(`*S*`,` *f*`)` (attribute)

if `C` satisfies `IsGreensLClass` (**Reference: IsGreensLClass**), then `GreensLClassData` returns an object in the category `IsGreensLClassData` (4.2.6) with representation `IsGreensLClassDataRep` (4.2.8) and the following five components:

- `rep` the representative of the L-class.

- `strongorb` the strong orbit of the kernel of `rep` under the action of the semigroup by `OnTuplesOfSetsAntiAction` (3.2.1).

- `relts` a list of relations such that

  ```
  KernelOfTransformation(relts[i]*x)=strongorb[1]
  ```

  whenever `x` has

  ```
  KernelOfTransformation(x)=strongorb[i].
  ```

- `invrelts` the inverses of the relations `relts`, so that

  ```
  KernelOfTransformation(invrelts[i]*rep)=strongorb[i].
  ```

- `schutz` the (generalised) left Schutzenberger group.

Further details can be found in Algorithm G, H, I, and J of [LPRRa].

Example

```
gap> gens:=[ Transformation( [ 4, 1, 4, 5, 3 ] ),
> Transformation( [ 5, 3, 5, 4, 3 ] ) ];;
gap> S:=Semigroup(gens);;
gap> C:=GreensLClassOfElement(S, gens[1]*gens[2]*gens[1]);
{Transformation( [ 5, 3, 5, 4, 3 ] )}
gap> GreensLClassData(C);
GreensLClassData( Transformation( [ 5, 3, 5, 4, 3 ] ),
[ [ [ 1, 3 ], [ 2, 5 ], [ 4 ] ] ], [ Binary Relation on 5 points ],
[ Binary Relation on 5 points ], Group( [ (), (3,5,4), (3,5) ] ) )
```

4.2.4 GreensHClassData

◇ GreensHClassData(*S*, *f*) (attribute)

if C satisfies IsGreensHClass (**Reference: IsGreensHClass**), then GreensLClassData returns an object in the category IsGreensHClassData (4.2.6) with representation IsGreensHClassDataRep (4.2.8) and the following two components:

- rep the representative of the *H*-class.
- schutz the intersection of the left Schutzenberger group and right Schutzenberger group of the *L*-class and *R*-class containing the representative rep (that is, the intersection of the schutz component of GreensRClassData (4.2.2) and the schutz component of GreensLClassData (4.2.3)).

Further details can be found in Algorithm K, L, M, and N of [LPRRa].

Example

```
gap> gens:=[ Transformation( [ 2, 2, 5, 2, 3 ] ),
> Transformation( [ 2, 5, 3, 5, 3 ] ) ];;
gap> S:=Semigroup(gens);;
gap> f:=Transformation( [ 5, 5, 3, 5, 3 ] );;
gap> GreensHClassData(GreensHClassOfElement(S, f));
GreensHClassData( Transformation( [ 5, 5, 3, 5, 3 ] ), Group( () ) )
```

4.2.5 GreensDClassData

◇ GreensDClassData(*S*, *f*) (attribute)

if C satisfies IsGreensDClass (**Reference: IsGreensDClass**), then GreensDClassData returns an object in the category IsGreensDClassData (4.2.6) with representation IsGreensDClassDataRep (4.2.8) and the following five components:

- rep the representative of the *D*-class.
- R the result of GreensRClassData (4.2.2) with argument rep.
- L the result of GreensLClassData (4.2.3) with argument rep.
- H the result of GreensHClassData (4.2.4) with argument rep.

- `cosets` a transversal of right cosets of the Schutzenberger group of `H` in the Schutzenberger group of `R`.

Note that only the first three components are displayed. Further details can be found in Algorithm O, P, Q, R, S, and T of [LPRRa].

Example

```
gap> gens:=[ Transformation( [ 4, 1, 5, 2, 4 ] ),
> Transformation( [ 4, 4, 1, 5, 3 ] ) ];;
gap> S:=Semigroup(gens);;
gap> f:=Transformation( [ 5, 5, 3, 3, 3 ] );;
gap> GreensDClassData(GreensDClassOfElement(S, f));
GreensDClassData( Transformation( [ 5, 5, 3, 3, 3
] ), GreensRClassData( Transformation( [ 5, 5, 3, 3, 3
] ) ), GreensLClassData( Transformation( [ 5, 5, 3, 3, 3 ] ) ) )
```

4.2.6 IsGreensData

◊ `IsGreensData(obj)` (Category)
◊ `IsGreensRClassData(obj)` (Category)
◊ `IsGreensLClassData(obj)` (Category)
◊ `IsGreensHClassData(obj)` (Category)
◊ `IsGreensDClassData(obj)` (Category)

returns `true` if `obj` lies in the category `IsGreensData`, `IsGreensRClassData`, `IsGreensLClassData`, `IsGreensHClassData`, `IsGreensDClassData`, respectively. The objects created using the functions `RClassData` (4.2.7), `LClassData` (4.2.7), `HClassData` (4.2.7), and `DClassData` (4.2.7) lie in the categories `IsGreensRClassData`, `IsGreensLClassData`, `IsGreensHClassData`, `IsGreensDClassData`, respectively, and all these categories are contained in the category `IsGreensData`.

4.2.7 XClassData

◊ `RClassData(rec)` (function)
◊ `LClassData(rec)` (function)
◊ `HClassData(rec)` (function)
◊ `DClassData(rec)` (function)

These function construct objects in the categories `IsGreensRClassData` (4.2.6), `IsGreensLClassData` (4.2.6), `IsGreensHClassData` (4.2.6), and `IsGreensDClassData` (4.2.6) subcategories of `IsGreensData` (4.2.6) using the output from `GreensRClassData` (4.2.2), `GreensLClassData` (4.2.3), `GreensHClassData` (4.2.4), `GreensDClassData` (4.2.5).

4.2.8 IsGreensXClassDataRep

◊ `IsGreensRClassDataRep(obj)` (Representation)
◊ `IsGreensLClassDataRep(obj)` (Representation)
◊ `IsGreensHClassDataRep(obj)` (Representation)
◊ `IsGreensDClassDataRep(obj)` (Representation)

returns `true` if `obj` has `IsGreensRClassDataRep`, `IsGreensLClassDataRep`, `IsGreensHClassDataRep`, or `IsGreensDClassDataRep`, respectively as a representation.

These representations each have several components detailed in `GreensRClassData` (4.2.2), `GreensLClassData` (4.2.3), `GreensHClassData` (4.2.4), `GreensDClassData` (4.2.5), respectively.

4.2.9 IsAssociatedSemigpTransSemigp

◊ `IsAssociatedSemigpTransSemigp(C)` (property)

returns `true` if `C` is a Green's class of a transformation semigroup and returns `false` otherwise.

This attribute is required so that a Green's class knowns that it belongs to a transformation semigroup, so that the methods defined in the MONOID are used in preference to those in the library.

Example

```
gap> a:=Transformation( [ 2, 1, 4, 5, 6, 3 ] );;
gap> b:=Transformation( [ 2, 3, 1, 5, 4, 1 ] );;
gap> M:=Semigroup(a,b);;
gap> GreensLClassOfElement(M,a);
{Transformation( [ 2, 1, 4, 5, 6, 3 ] )}
gap> IsAssociatedSemigpTransSemigp(last);
true
gap> f:=FreeSemigroup(3);;
gap> a:=f.1;; b:=f.2;; c:=f.3;;
gap> s:=f/[[a^2, a], [b^2,b], [c^2,c], [a*b,a], [b*a,b], [a*c,a],
> [c*a,c], [b*c,b],[c*b,c]];
<fp semigroup on the generators [ s1, s2, s3 ]>
gap> Size(s);
3
gap> GreensLClassOfElement(s,a);
{s1}
gap> IsAssociatedSemigpTransSemigp(last);
false
```

4.2.10 SchutzenbergerGroup

◊ `SchutzenbergerGroup(D)` (attribute)

if `D` satisfies `IsGreensRClassData` (4.2.6), `IsGreensLClassData` (4.2.6), `IsGreensHClassData` (4.2.6), or `IsGreensDClassData` (4.2.6), then `SchutzenbergerGroup` returns the `schutz` component of `D`.

If `D` satisfies `IsGreensRClass` (**Reference: IsGreensRClass**), `IsGreensLClass` (**Reference: IsGreensLClass**), `IsGreensHClass` (**Reference: IsGreensHClass**), `IsGreensDClass` (**Reference: IsGreensDClass**), then `SchutzenbergerGroup` returns the `schutz` component of `GreensData` with argument `D`.

Example

```
gap> gens:=[ Transformation( [ 4, 4, 3, 5, 3 ] ),
> Transformation( [ 5, 1, 1, 4, 1 ] ),
> Transformation( [ 5, 5, 4, 4, 5 ] ) ];;
gap> f:=Transformation( [ 4, 5, 5, 5, 5 ] );;
gap> SchutzenbergerGroup(GreensDClassOfElement(S, f));
Group([ (), (4,5) ])
```

```
gap> SchutzenbergerGroup(GreensRClassOfElement(S, f));
Group([ (), (4,5) ])
gap> SchutzenbergerGroup(GreensLClassOfElement(S, f));
Group([ (), (4,5) ])
gap> SchutzenbergerGroup(GreensHClassOfElement(S, f));
Group([ (), (4,5) ])
```

4.2.11 Idempotents

◇ `Idempotents(X[, n])` (function)

returns a list of the idempotents in the transformation semigroup or Green's class `X`.

If the optional second argument `n` is present, then a list of the idempotents in `S` of rank `n` is returned. If you are only interested in the idempotents of a given rank, then the second version of the function will likely be faster.

Example

```
gap> S:=Semigroup([ Transformation( [ 2, 3, 4, 1 ] ),
> Transformation( [ 3, 3, 1, 1 ] ) ]);;
gap> Idempotents(S, 1);
[  ]
gap> Idempotents(S, 2);
[ Transformation( [ 1, 1, 3, 3 ] ), Transformation( [ 1, 3, 3, 1 ] ),
  Transformation( [ 2, 2, 4, 4 ] ), Transformation( [ 4, 2, 2, 4 ] ) ]
gap> Idempotents(S, 3);
[  ]
gap> Idempotents(S, 4);
[ Transformation( [ 1, 2, 3, 4 ] ) ]
gap> Idempotents(S);
[ Transformation( [ 1, 1, 3, 3 ] ), Transformation( [ 1, 2, 3, 4 ] ),
  Transformation( [ 1, 3, 3, 1 ] ), Transformation( [ 2, 2, 4, 4 ] ),
  Transformation( [ 4, 2, 2, 4 ] ) ]
```

4.2.12 PartialOrderOfDClasses

◇ `PartialOrderOfDClasses(S)` (attribute)

returns the partial order of the `D`-classes of `S` as a directed graph in GRAPE, if it is installed, using the command

Example

```
Graph(Group(()), [1..Length(GreensDClasses(S))], OnPoints, function(x,y)
return y in poset[x]; end, true); ;
```

where `y` in `poset[x]` if and only if `S^1yS^1` is a subset of `S^1 xS^1`.

If GRAPE is not loaded, then the list `poset` is returned instead.

Chapter 5

Properties of Semigroups

5.1 Introduction

In this section we give the theoretical results and the corresponding GAP functions that can be used to determine whether a set of transformations generates a semigroup of a given type. Let `S` be a semigroup. Then

- `S` is a *left zero semigroup* if `xy=x` for all `x,y` in `S`.
- `S` is a *right zero semigroup* if `xy=y` for all `x,y` in `S`.
- `S` is *commutative* if `xy=yx` for all `x,y` in `S`.
- `S` is *simple* if it has no proper two-sided ideals.
- `S` is *regular* if for all `x` in `S` there exists `y` in `S` such that `xyx=x`.
- `S` is *completely regular* if every element of `S` lies in a subgroup.
- `S` is an *inverse semigroup* if for all elements `x` in `S` there exists a unique semigroup inverse, that is, a unique element `y` such that `xyx=x` and `yxy=y`.
- `S` is a *Clifford semigroup* if it is a regular semigroup whose idempotents are central, that is, for all `e` in `S` with `e^2=e` and `x` in `S` we have that `ex=xe`.
- `S` is a *band* if every element is an idempotent, that is, `x^2=x` for all `x` in `S`.
- `S` is a *rectangular band* if for all `x,y,z` in `S` we have that `x^2=x` and `xyz=xz`.
- `S` is a *semiband* if it is generated by its idempotent elements, that is, elements satisfying `x^2=x`.
- `S` is an *orthodox semigroup* if its idempotents (elements satisfying `x^2=x`) form a subsemigroup.
- `S` is a *zero semigroup* if there exists an element `0` in `S` such that `xy=0` for all `x,y` in `S`.
- `S` is a *zero group* if there exists an element `0` in `S` such that `S` without `0` is a group and for all `x` in `S` we have that `x0=0x=0`.

The following results provide efficient methods to determine if an arbitrary transformation semigroup is a left zero, right zero, simple, completely regular, inverse or Clifford semigroup. Proofs of these results can be found in [GM05].

Let S be a semigroup generated by a set of transformations U on a finite set. Then the following hold:

- S is a left zero semigroup if and only if for all f, g in U the image of f equals the image of g and f^2=f.
- S is a right zero semigroup if and only if for all f, g in U the kernel of f equals the kernel of g and f^2=f.
- S is simple if and only if for all f, g in U every class of the kernel of f contains exactly 1 element of the image of g.
- S is completely regular if and only if for all f in U and g in S, every class of the kernel of f contains at most 1 element of the set found by applying g to the image of f.
- S is inverse if and only if it is regular and there is a bijection \phi from the set of kernels of elements of S to the set of images of elements of S such that every class of a kernel K contains exactly 1 element in (K)\phi.
- S is a Clifford semigroup if and only if for all f, g in U
  - f permutes its image
  - f commutes with the power of g that acts as the identity on its image.

It is straightforward to verify that a transformation semigroup S generated by U is a group if and only if for all f, g in U

- the kernel of f equals the kernel of g.
- the image of f equals the image of g.
- f permutes its image.

At first glance it might not be obvious why these conditions are an improvement over the original definitions. The main point is that it can be easily determined whether a semigroup S generated by a set U of mappings satisfies these conditions by considering the generators U and their action on the underlying set only.

5.2 Property Tests

5.2.1 IsCompletelyRegularSemigroup

◊ IsCompletelyRegularSemigroup(*S*) (property)

returns true if the transformation semigroup S is completely regular and false otherwise.
A semigroup is *completely regular* if every element is contained in a subgroup.

Example

```
gap> gens:=[ Transformation( [ 1, 2, 4, 3, 6, 5, 4 ] ),
>  Transformation( [ 1, 2, 5, 6, 3, 4, 5 ] ),
>  Transformation( [ 2, 1, 2, 2, 2, 2, 2 ] ) ];;
gap> S:=Semigroup(gens);;
gap> IsCompletelyRegularSemigroup(S);
true
gap> S:=RandomSemigroup(5,5);;
gap> IsSimpleSemigroup(S);
false
```

5.2.2 IsSimpleSemigroup

◊ IsSimpleSemigroup(*S*) (property)
◊ IsCompletelySimpleSemigroup(*S*) (property)

returns true if the transformation semigroup S is simple and false otherwise.

A semigroup is *simple* if it has no proper 2-sided ideals. A semigroup is *completely simple* if it is simple and possesses minimal left and right ideals. A finite semigroup is simple if and only if it is completely simple.

Example

```
gap> gens:=[ Transformation( [ 2, 2, 4, 4, 6, 6, 8, 8, 10, 10, 12, 12, 2 ] ),
>  Transformation( [ 1, 1, 3, 3, 5, 5, 7, 7, 9, 9, 11, 11, 3 ] ),
>  Transformation( [ 1, 7, 3, 9, 5, 11, 7, 1, 9, 3, 11, 5, 5 ] ),
>  Transformation( [ 7, 7, 9, 9, 11, 11, 1, 1, 3, 3, 5, 5, 7 ] ) ];;
gap> S:=Semigroup(gens);;
gap> IsSimpleSemigroup(S);
true
gap> IsCompletelySimpleSemigroup(S);
true
gap> S:=RandomSemigroup(5,5);;
gap> IsSimpleSemigroup(S);
false
```

5.2.3 IsGroupAsSemigroup

◊ IsGroupAsSemigroup(*S*) (property)

returns true if the transformation semigroup S is a group and false otherwise.

Example

```
gap> gens:=[ Transformation( [ 2, 4, 5, 3, 7, 8, 6, 9, 1 ] ),
>  Transformation( [ 3, 5, 6, 7, 8, 1, 9, 2, 4 ] ) ];;
gap> S:=Semigroup(gens);;
gap> IsGroupAsSemigroup(S);
true
```

5.2.4 IsCommutativeSemigroup

◊ IsCommutativeSemigroup(*S*) (property)

returns `true` if the transformation semigroup `S` is commutative and `false` otherwise. The function `IsCommutative` (**Reference: IsCommutative**) can also be used to test if a semigroup is commutative.

A semigroup `S` is *commutative* if `xy=yx` for all `x,y` in `S`.

Example

```
gap> gens:=[ Transformation( [ 2, 4, 5, 3, 7, 8, 6, 9, 1 ] ),
>  Transformation( [ 3, 5, 6, 7, 8, 1, 9, 2, 4 ] ) ];;
gap> S:=Semigroup(gens);;
gap> IsCommutativeSemigroup(S);
true
gap> IsCommutative(S);
true
```

5.2.5 IsRegularSemigroup

◊ `IsRegularSemigroup(`*S*`)` (property)

returns `true` if the transformation semigroup `S` is a regular semigroup and `false` otherwise. The algorithm used here is essentially the same algorithm as that used for `GreensRClasses` (**Reference: GreensRClasses**) in MONOID. If `S` is regular, then `S` will have the attribute `GreensRClasses` after `IsRegularSemigroup` is invoked.

A semigroup `S` is *regular* if for all `x` in `S` there exists `y` in `S` such that `xyx=x`.

Example

```
gap> IsRegularSemigroup(FullTransformationSemigroup(5));
true
```

5.2.6 IsInverseSemigroup

◊ `IsInverseSemigroup(`*S*`)` (property)

returns `true` if the transformation semigroup `S` is an inverse semigroup and `false` otherwise.

A semigroup `S` is an *inverse semigroup* if every element `x` in `S` has a unique semigroup inverse, that is, a unique element `y` such that `xyx=x` and `yxy=y`.

Example

```
gap> gens:=[Transformation([1,2,4,5,6,3,7,8]),
> Transformation([3,3,4,5,6,2,7,8]),
>Transformation([1,2,5,3,6,8,4,4])];;
gap> S:=Semigroup(gens);;
gap> IsInverseSemigroup(S);
true
```

5.2.7 IsCliffordSemigroup

◊ `IsCliffordSemigroup(`*S*`)` (property)

returns `true` if the transformation semigroup `S` is a Clifford semigroup and `false` otherwise.

A semigroup `S` is a *Clifford semigroup* if it is a regular semigroup whose idempotents are central, that is, for all `e` in `S` with `e^2=e` and `x` in `S` we have that `ex=xe`.

Example

```
gap> gens:=[Transformation([1,2,4,5,6,3,7,8]),
> Transformation([3,3,4,5,6,2,7,8]),
>Transformation([1,2,5,3,6,8,4,4])];;
gap> S:=Semigroup(gens);;
gap> IsCliffordSemigroup(S);
true
```

5.2.8 IsBand

◊ IsBand(*S*) (property)

returns `true` if the transformation semigroup `S` is a band and `false` otherwise.
A semigroup `S` is a *band* if every element is an idempotent, that is, `x^2=x` for all `x` in `S`.

Example

```
gap> gens:=[ Transformation( [ 1, 1, 1, 4, 4, 4, 7, 7, 7, 1 ] ),
> Transformation( [ 2, 2, 2, 5, 5, 5, 8, 8, 8, 2 ] ),
> Transformation( [ 3, 3, 3, 6, 6, 6, 9, 9, 9, 3 ] ),
> Transformation( [ 1, 1, 1, 4, 4, 4, 7, 7, 7, 4 ] ),
> Transformation( [ 1, 1, 1, 4, 4, 4, 7, 7, 7, 7 ] ) ];;
gap> S:=Semigroup(gens);;
gap> IsBand(S);
true
```

5.2.9 IsRectangularBand

◊ IsRectangularBand(*S*) (property)

returns `true` if the transformation semigroup `S` is a rectangular band and `false` otherwise.
A semigroup `S` is a *rectangular band* if for all `x,y,z` in `S` we have that `x^2=x` and `xyz=xz`.

Example

```
gap> gens:=[ Transformation( [ 1, 1, 1, 4, 4, 4, 7, 7, 7, 1 ] ),
> Transformation( [ 2, 2, 2, 5, 5, 5, 8, 8, 8, 2 ] ),
> Transformation( [ 3, 3, 3, 6, 6, 6, 9, 9, 9, 3 ] ),
> Transformation( [ 1, 1, 1, 4, 4, 4, 7, 7, 7, 4 ] ),
> Transformation( [ 1, 1, 1, 4, 4, 4, 7, 7, 7, 7 ] ) ];;
gap> S:=Semigroup(gens);;
gap> IsRectangularBand(S);
true
```

5.2.10 IsSemiBand

◊ IsSemiBand(*S*) (property)

returns `true` if the transformation semigroup `S` is a semiband and `false` otherwise.
A semigroup `S` is a *semiband* if it is generated by its idempotent elements, that is, elements satisfying `x^2=x`.

Example

```
gap> S:=FullTransformationSemigroup(4);;
gap> x:=Transformation( [ 1, 2, 3, 1 ] );;
gap> D:=GreensDClassOfElement(S, x);;
gap> T:=Semigroup(Elements(D));;
gap> IsSemiBand(T);
true
```

5.2.11 IsOrthodoxSemigroup

◊ IsOrthodoxSemigroup(*S*) (property)

returns true if the transformation semigroup S is orthodox and false otherwise.
A semigroup is an *orthodox semigroup* if its idempotent elements form a subsemigroup.

Example

```
gap> gens:=[ Transformation( [ 1, 1, 1, 4, 5, 4 ] ),
>  Transformation( [ 1, 2, 3, 1, 1, 2 ] ),
>  Transformation( [ 1, 2, 3, 1, 1, 3 ] ),
>  Transformation( [ 5, 5, 5, 5, 5, 5 ] ) ];;
gap> S:=Semigroup(gens);;
gap> IsOrthodoxSemigroup(S);
true
```

5.2.12 IsRightZeroSemigroup

◊ IsRightZeroSemigroup(*S*) (property)

returns true if the transformation semigroup S is a right zero semigroup and false otherwise.
A semigroup S is a *right zero semigroup* if xy=y for all x,y in S.

Example

```
gap> gens:=[ Transformation( [ 2, 1, 4, 3, 5 ] ),
>  Transformation( [ 3, 2, 3, 1, 1 ] ) ];;
gap> S:=Semigroup(gens);;
gap> IsRightZeroSemigroup(S);
false
gap> gens:=[Transformation( [ 1, 2, 3, 3, 1 ] ),
>  Transformation( [ 1, 2, 4, 4, 1 ] )];;
gap> S:=Semigroup(gens);;
gap> IsRightZeroSemigroup(S);
true
```

5.2.13 IsLeftZeroSemigroup

◊ IsLeftZeroSemigroup(*S*) (property)

returns true if the transformation semigroup S is a left zero semigroup and false otherwise.
A semigroup S is a *left zero semigroup* if xy=x for all x,y in S.

Example

```
gap> gens:=[ Transformation( [ 2, 1, 4, 3, 5 ] ),
>  Transformation( [ 3, 2, 3, 1, 1 ] ) ];;
```

```
gap> S:=Semigroup(gens);;
gap> IsRightZeroSemigroup(S);
false
gap> gens:=[Transformation( [ 1, 2, 3, 3, 1 ] ),
> Transformation( [ 1, 2, 3, 3, 3 ] ) ];;
gap> S:=Semigroup(gens);;
gap> IsLeftZeroSemigroup(S);
true
```

5.2.14 IsZeroSemigroup

◊ IsZeroSemigroup(*S*) (property)

returns true if the transformation semigroup S is a zero semigroup or if S was created using the ZeroSemigroup (6.2.1) command. Otherwise false is returned.

A semigroup S is a *zero semigroup* if there exists an element 0 in S such that xy=0 for all x,y in S.

Example

```
gap> gens:=[ Transformation( [ 4, 7, 6, 3, 1, 5, 3, 6, 5, 9 ] ),
> Transformation( [ 5, 3, 5, 1, 9, 3, 8, 7, 4, 3 ] ),
> Transformation( [ 5, 10, 10, 1, 7, 6, 6, 8, 7, 7 ] ),
> Transformation( [ 7, 4, 3, 3, 2, 2, 3, 2, 9, 3 ] ),
> Transformation( [ 8, 1, 3, 4, 9, 6, 3, 7, 1, 6 ] ) ];;
gap> S:=Semigroup(gens);;
gap> IsZeroSemigroup(S);
false
```

5.2.15 IsZeroGroup

◊ IsZeroGroup(*S*) (property)

returns true if the transformation semigroup S is a zero group or if S was created using the ZeroGroup (6.2.3) command. Otherwise false is returned.

A semigroup S S is a *zero group* if there exists an element 0 in S such that S without 0 is a group and for all x in S we have that x0=0x=0.

Example

```
gap> S:=ZeroGroup(DihedralGroup(10));;
gap> iso:=IsomorphismTransformationSemigroup(S);;
gap> T:=Range(iso);;
gap> IsZeroGroup(T);
true
```

5.2.16 MultiplicativeZero

◊ MultiplicativeZero(*S*) (property)

returns the multiplicative zero of the transformation semigroup S if it has one and returns fail otherwise.

Example

```
gap> gens:=[ Transformation( [ 1, 4, 2, 6, 6, 5, 2 ] ),
> Transformation( [ 1, 6, 3, 6, 2, 1, 6 ] ) ];;
gap> S:=Semigroup(gens);;
gap> MultiplicativeZero(S);
Transformation( [ 1, 1, 1, 1, 1, 1, 1 ] )
```

Chapter 6

Special Classes of Semigroup

In this chapter functions for creating certain semigroups are given.

6.1 Some Classes of Semigroup

6.1.1 SingularSemigroup

◊ `SingularSemigroup(`*n*`)` (function)

creates the semigroup of singular transformations of degree `n`. That is, the semigroup of all transformations of the `n`-element set `{1,2,...,n}` that are non-invertible.

This semigroup is known to be regular, idempotent generated (satisfies `IsSemiBand` (5.2.10)), and has size `n^n-n!`.

The generators used here are the idempotents of rank `n-1`, so there are `n(n-1)` generators in total.

Example

```
gap> S:=SingularSemigroup(6);
<semigroup with 30 generators>
gap> Size(S);
45936
gap> IsRegularSemigroup(S);
true
gap> IsSemiBand(S);
true
```

6.1.2 OrderPreservingSemigroup

◊ `OrderPreservingSemigroup(`*n*`)` (operation)

returns the semigroup of order preserving transformations of the `n`- element set `{1,2,...,n}`. That is, the mappings `f` such that `i` is at most `j` implies `f(i)` is at most `f(j)` for all `i,j` in `{1,2,...,n}`.

This semigroup is known to be regular, idempotent generated (satisfies `IsSemiBand` (5.2.10)), and has size `Binomial(2*n-1, n-1)`. The generators and relations used here are those specified by Aizenstat as given in [AR00] and [GH92]. That is, `OrderPreservingSemigroup(n)` has the `2n-2` idempotent generators

Example
```
u_2:=Transformation([2,2,3,..,n]), u_3:=Transformation([1,3,3,..,n]), ...
v_n-2:=Transformation([1,2,2,...,n]), v_n-3:=Transformation
([1,2,3,3,...,n]), ...
```

and the presentation obtained using `IsomorphismFpMonoid` (7.7.4) has relations

Example
```
v_n-i u_i = u_i v_n-i+1 (i=2,..., n-1)
u_n-i v_i = v_i u_n-i+1 (i=2,...,n-1),
v_n-i u_i = u_i (i=1,...,n-1),
u_n-i v_i = v_i (i=1,...,n-1),
u_i v_j = v_j u_i (i,j=1,...,n-1; not j=n-i, n-i+1),
u_1 u_2 u_1 = u_1 u_2,
v_1 v_2 v_1 = v_1 v_2.
```

Example
```
gap> S:=OrderPreservingSemigroup(5);
<monoid with 8 generators>
gap> IsSemiBand(S);
true
gap> IsRegularSemigroup(S);
true
gap> Size(S)=Binomial(2*5-1, 5-1);
true
```

6.1.3 KiselmanSemigroup

◇ `KiselmanSemigroup(`*n*`)` (operation)

returns the Kiselman semigroup with `n` generators. That is, the semigroup defined in [KM05] with the presentation

$$< a_1, a_2, ..., a_n | a_i^2 = a_i (i = 1, ...n) a_i a_j a_i = a_j a_i a_j = a_j a_i (1 <= i < j <= n) > .$$

Example
```
gap> S:=KiselmanSemigroup(3);
<fp monoid on the generators [ m1, m2, m3 ]>
gap> Elements(S);
[ <identity ...>, m1, m2, m3, m1*m2, m1*m3, m2*m1, m2*m3, m3*m1, m3*m2,
  m1*m2*m3, m1*m3*m2, m2*m1*m3, m2*m3*m1, m3*m1*m2, m3*m2*m1, m2*m1*m3*m2,
  m2*m3*m1*m2 ]
gap> Idempotents(S);
[ 1, m1, m2*m1, m3*m2*m1, m3*m1, m2, m3*m2, m3 ]
gap> SetInfoLevel(InfoMonoidAutos, 0);
gap> AutomorphismGroup(Range(IsomorphismTransformationSemigroup(S)));
<group of size 1 with 1 generators>
```

6.2 Zero Groups and Zero Semigroups

6.2.1 ZeroSemigroup

◊ ZeroSemigroup(*n*) (operation)

returns the *zero semigroup* S of order n. That is, the unique semigroup up to isomorphism of order n such that there exists an element 0 in S such that xy=0 for all x,y in S.

A zero semigroup is generated by its nonzero elements, has trivial Green's relations, and is not regular.

Example

```
gap> S:=ZeroSemigroup(10);
<zero semigroup with 10 elements>
gap> Size(S);
10
gap> GeneratorsOfSemigroup(S);
[ z1, z2, z3, z4, z5, z6, z7, z8, z9 ]
gap> Idempotents(S);
[ 0 ]
gap> IsZeroSemigroup(S);
true
gap> GreensRClasses(S);
[ {0}, {z1}, {z2}, {z3}, {z4}, {z5}, {z6}, {z7}, {z8}, {z9} ]
```

6.2.2 ZeroSemigroupElt

◊ ZeroSemigroupElt(*n*) (operation)

returns the zero semigroup element zn where n is a positive integer and z0 is the multiplicative zero.

The zero semigroup element zn belongs to every zero semigroup with degree at least n.

Example

```
gap> ZeroSemigroupElt(0);
0
gap> ZeroSemigroupElt(4);
z4
```

6.2.3 ZeroGroup

◊ ZeroGroup(*G*) (operation)

returns the monoid obtained by adjoining a zero element to G. That is, the monoid S obtained by adjoining a zero element 0 to G with g0=0g=0 for all g in S.

Example

```
gap> S:=ZeroGroup(CyclicGroup(10));
<zero group with 3 generators>
gap> IsRegularSemigroup(S);
true
gap> Elements(S);
[ 0, <identity> of ..., f1, f2, f1*f2, f2^2, f1*f2^2, f2^3, f1*f2^3, f2^4,
```

```
    f1*f2^4 ]
gap> GreensRClasses(S);
[ {<adjoined zero>}, {ZeroGroup(<identity> of ...)} ]
```

6.2.4 ZeroGroupElt

◇ ZeroGroupElt(*g*) (operation)

returns the zero group element corresponding to the group element g. The function ZeroGroupElt is only used to create an object in the correct category during the creation of a zero group using ZeroGroup (6.2.3).

Example

```
gap> ZeroGroupElt(Random(DihedralGroup(10)));;
gap> IsZeroGroupElt(last);
true
```

6.2.5 UnderlyingGroupOfZG

◇ UnderlyingGroupOfZG(*ZG*) (attribute)

returns the group from which the zero group ZG was constructed.

Example

```
gap> G:=DihedralGroup(10);;
gap> S:=ZeroGroup(G);;
gap> UnderlyingGroupOfZG(S)=G;
true
```

6.2.6 UnderlyingGroupEltOfZGElt

◇ UnderlyingGroupEltOfZGElt(*g*) (attribute)

returns the group element from which the zero group element g was constructed.

Example

```
gap> G:=DihedralGroup(10);;
gap> S:=ZeroGroup(G);;
gap> Elements(S);
[ 0, <identity> of ..., f1, f2, f1*f2, f2^2, f1*f2^2, f2^3, f1*f2^3, f2^4,
  f1*f2^4 ]
gap> x:=last[5];
f1*f2
gap> UnderlyingGroupEltOfZGElt(x);
f1*f2
```

6.3 Random Semigroups

6.3.1 RandomMonoid

◇ RandomMonoid(*m*, *n*) (function)

returns a random transformation monoid of degree n with m generators.

Example

```
gap> S:=RandomMonoid(5,5);
<semigroup with 5 generators>
```

6.3.2 RandomSemigroup

◊ RandomSemigroup(*m, n*) (function)

returns a random transformation semigroup of degree n with m generators.

Example

```
gap> S:=RandomSemigroup(5,5);
<semigroup with 5 generators>
```

6.3.3 RandomReesMatrixSemigroup

◊ RandomReesMatrixSemigroup(*i, j, deg*) (function)

returns a random Rees matrix semigroup with an i by j sandwich matrix over a permutation group with maximum degree deg.

Example

```
gap> S:=RandomReesMatrixSemigroup(4,5,5);
Rees Matrix Semigroup over Group([ (1,5,3,4), (1,3,4,2,5) ])
[ [ (), (), (), (), () ],
[ (), (1,3,5)(2,4), (1,3,5)(2,4), (1,5,3), (1,5,3) ],
[ (), (1,3,5), (1,5,3)(2,4), (), (1,5,3) ],
[ (), (), (1,3,5)(2,4), (2,4), (2,4) ] ]
```

6.3.4 RandomReesZeroMatrixSemigroup

◊ RandomReesZeroMatrixSemigroup(*i, j, deg*) (function)

returns a random Rees 0-matrix semigroup with an i by j sandwich matrix over a permutation group with maximum degree deg.

Example

```
gap> S:=RandomReesZeroMatrixSemigroup(2,3,2);
Rees Zero Matrix Semigroup over <zero group with 2 generators>
gap> SandwichMatrixOfReesZeroMatrixSemigroup(S);
[ [ 0, (), 0 ], [ 0, 0, 0 ] ]
```

Chapter 7

Semigroup Homomorphisms

7.1 Introduction

In this chapter we give instructions on how to create semigroup homomorphisms using MONOID in several different ways.

In Section 7.2, we give functions for creating arbitrary semigroup homomorphism specified by a function on the elements, the images of the generators, or the images of all the semigroup elements. These functions were written to support the functions for computing the automorphism group of an arbitrary transformation semigroup and to specify isomorphisms between different classes of semigroup, such as finitely presented semigroups and transformation semigroups.

In Section 7.3, we show how to specify and compute the inner automorphisms of a transformation semigroup. The functions that can be used to find the entire automorphism group of an arbitrary transformation semigroup are given in Section 7.4. The `AutomorphismGroup` (7.4.1) has an interactive mode that allows the user to decide how the computation should proceed. This can be invoked by using the command `SetInfoLevel(InfoMonoidAutos, 4);` see `InfoMonoidAutos` (7.1.1).

In Section 7.5, commands for creating automorphisms and finding all automorphisms of Rees matrix semigroups and Rees `0`-matrix semigroups are given.

In Section 7.6, functions for specifying the automorphisms of a zero group are given.

In the final section (7.7), functions for finding isomorphisms between various kinds of semigroups are given.

The methods behind the commands in this chapter are taken from [ABM07].

PLEASE NOTE: the following functions can only be used fully if GRAPE is fully installed (and loaded):

- `AutomorphismGroup` (7.4.1) with argument satisfying `IsTransformationSemigroup` (**Reference: IsTransformationSemigroup**) or `IsReesZeroMatrixSemigroup` (**Reference: IsReesZeroMatrixSemigroup**)
- `RightTransStabAutoGroup` (7.5.9) with argument satisfying `IsReesZeroMatrixSemigroup` (**Reference: IsReesZeroMatrixSemigroup**)
- `RZMSGraph` (7.5.8)
- `RZMSInducedFunction` (7.5.6)
- `RZMStoRZMSInducedFunction` (7.5.7)

- IsomorphismSemigroups (7.7.5) with both arguments satisfying IsReesZeroMatrixSemigroup (**Reference: IsReesZeroMatrixSemigroup**)

Please see Chapter 1 for further details on how to obtain GRAPE.

7.1.1 InfoMonoidAutos

◊ InfoMonoidAutos (info class)

This is the InfoClass for the functions in this chapter. Setting the value of InfoMonoidAutos to 1, 2, 3, or 4 using the command SetInfoLevel (**Reference: SetInfoLevel**) will give different levels of information about what GAP is doing during a computation. In particular, if the level of InfoMonoidAutos is set to 4, then AutomorphismGroup (7.4.1) runs in interactive mode.

7.2 Creating Homomorphisms

The principal functions for creating arbitrary semigroup homomorphisms are the following three.

7.2.1 SemigroupHomomorphismByFunction

◊ SemigroupHomomorphismByFunction(*S, T, func*) (operation)
◊ SemigroupHomomorphismByFunctionNC(*S, T, func*) (operation)

returns a semigroup homomorphism with representation IsSemigroupHomomorphismByFunctionRep from the semigroup S to the semigroup T defined by the function func.

SemigroupHomomorphismByFunction will find an isomorphism from S to a finitely presented semigroup or monoid (using IsomorphismFpSemigroup (7.7.3) or IsomorphismFpMonoid (7.7.4)) and then check that the list of values under func of the generators of S satisfy the relations of this presentation.

SemigroupHomomorphismByFunctionNC does not check that func defines a homomorphism and, in this case S and T can be semigroups, $\mathcal{D}$-classes, $\mathcal{H}$-classes or any combination of these.

Example

```
gap> gens:=[ Transformation( [ 1, 4, 3, 5, 2 ] ),
> Transformation( [ 2, 3, 1, 1, 2 ] ) ];;
gap> S:=Semigroup(gens);;
gap> gens:=[ Transformation( [ 1, 5, 1, 2, 1 ] ),
> Transformation( [ 5, 1, 4, 3, 2 ] ) ];;
gap> T:=Semigroup(gens);;
gap> idem:=Random(Idempotents(T));;
gap> hom:=SemigroupHomomorphismByFunction(S, T, x-> idem);
SemigroupHomomorphism ( <semigroup with 2 generators>-><semigroup with
2 generators>)
gap> hom:=SemigroupHomomorphismByFunctionNC(S, T, x-> idem);
SemigroupHomomorphism ( <semigroup with 2 generators>-><semigroup with
2 generators>)
```

7.2.2 SemigroupHomomorphismByImagesOfGens

◊ SemigroupHomomorphismByImagesOfGens(*S, T, list*) (operation)
◊ SemigroupHomomorphismByImagesOfGensNC(*S, T, list*) (operation)

returns a semigroup homomorphism with representation IsSemigroupHomomorphismByImagesOfGensRep from S to T where the image of the ith generator of S is the ith position in list.

SemigroupHomomorphismByImagesOfGens will find an isomorphism from S to a finitely presented semigroup or monoid (using IsomorphismFpSemigroup (7.7.3) or IsomorphismFpMonoid (7.7.4)) and then check that list satisfies the relations of this presentation.

SemigroupHomomorphismByImagesOfGensNC does not check that list induces a homomorphism.

Example

```
gap> gens:=[ Transformation( [ 1, 4, 3, 5, 2 ] ),
> Transformation( [ 2, 3, 1, 1, 2 ] ) ];;
gap> S:=Semigroup(gens);;
gap> gens:=[ Transformation( [ 1, 5, 1, 2, 1 ] ),
> Transformation( [ 5, 1, 4, 3, 2 ] ) ];;
gap> T:=Semigroup(gens);;
gap> SemigroupHomomorphismByImagesOfGens(S, T, GeneratorsOfSemigroup(T));
fail
gap> SemigroupHomomorphismByImagesOfGens(S, S, GeneratorsOfSemigroup(S));
SemigroupHomomorphismByImagesOfGens ( <trans. semigroup of size 161 with
2 generators>-><trans. semigroup of size 161 with 2 generators>)
```

7.2.3 SemigroupHomomorphismByImages

◊ SemigroupHomomorphismByImages(*S, T, list*) (operation)
◊ SemigroupHomomorphismByImagesNC(*S, T, list*) (operation)

returns a semigroup homomorphism with representation IsSemigroupHomomorphismByImagesRep from S to T where the image of the ith element of S is the ith position in list.

SemigroupHomomorphismByImages will find an isomorphism from S to a finitely presented semigroup or monoid (using IsomorphismFpSemigroup (7.7.3) or IsomorphismFpMonoid (7.7.4)) and then check that list satisfies the relations of this presentation.

SemigroupHomomorphismByImagesNC does not check that list induces a homomorphism.

Example

```
gap> gens:=[ Transformation( [ 2, 3, 4, 2, 4 ] ),
> Transformation( [ 3, 4, 2, 1, 4 ] ) ];;
gap> S:=Semigroup(gens);;
gap> gens:=[ Transformation( [ 2, 4, 4, 1, 2 ] ),
> Transformation( [ 5, 1, 1, 5, 1 ] ) ];;
gap> T:=Semigroup(gens);;
gap> idem:=Transformation( [ 5, 5, 5, 5, 5 ] );;
gap> list:=List([1..Size(S)], x-> idem);;
gap> hom:=SemigroupHomomorphismByImages(S, T, list);
SemigroupHomomorphismByImagesOfGens ( <trans. semigroup of size 164 with
2 generators>-><trans. semigroup with 2 generators>)
gap> SemigroupHomomorphismByImagesNC(S, T, list);
```

```
SemigroupHomomorphismByImages ( <trans. semigroup of size 164 with
2 generators>-><trans. semigroup with 2 generators>)
```

7.3 Inner Automorphisms

7.3.1 InnerAutomorphismOfSemigroup

◊ InnerAutomorphismOfSemigroup(*S, perm*) (operation)
◊ InnerAutomorphismOfSemigroupNC(*S, perm*) (operation)

returns the inner automorphism of the transformation semigroup S given by the permutation perm. The degree of perm should be at most the degree of S.

The notion of inner automorphisms of semigroups differs from the notion of the same name for groups. Indeed, if S is a semigroup of transformations of degree n, then g in the symmetric group S_n induces an inner automorphism of S if the mapping that takes s to g^-1sg for all s in S is an automorphism of S.

InnerAutomorphismOfSemigroup checks that the mapping induced by perm is an automorphism and InnerAutomorphismOfSemigroupNC only creates the appropriate object without performing a check that the permutation actually induces an automorphism.

Example

```
gap> gens:=[ Transformation( [ 6, 2, 7, 5, 3, 5, 4 ] ),
> Transformation( [ 7, 7, 5, 7, 2, 4, 3 ] ) ];;
gap> S:=Monoid(gens);;
gap> InnerAutomorphismOfSemigroup(S, (1,2,3,4,5));
fail
gap> InnerAutomorphismOfSemigroupNC(S, (1,2,3,4,5));
^(1,2,3,4,5)
gap> InnerAutomorphismOfSemigroup(S, ());
^()
```

7.3.2 ConjugatorOfInnerAutomorphismOfSemigroup

◊ ConjugatorOfInnerAutomorphismOfSemigroup(*f*) (attribute)

returns the permutation perm used to construct the inner automorphism f of a semigroup; see InnerAutomorphismOfSemigroup (7.3.1) for further details.

Example

```
gap> S:=RandomSemigroup(3,8);;
gap> f:=InnerAutomorphismOfSemigroupNC(S, (1,2)(3,4));
^(1,2)(3,4)
gap> ConjugatorOfInnerAutomorphismOfSemigroup(f);
(1,2)(3,4)
```

7.3.3 IsInnerAutomorphismOfSemigroup

◊ IsInnerAutomorphismOfSemigroup(*f*) (property)

returns `true` if the general mapping `f` is an inner automorphism of a semigroup; see `InnerAutomorphismOfSemigroup` (7.3.1) for further details.

Example

```
gap> S:=RandomSemigroup(2,9);;
gap> f:=InnerAutomorphismOfSemigroupNC(S, (1,2)(3,4));
^(1,2)(3,4)
gap> IsInnerAutomorphismOfSemigroup(f);
true
```

7.3.4 InnerAutomorphismsOfSemigroup

◊ `InnerAutomorphismsOfSemigroup(`*S*`)` (attribute)

`InnerAutomorphismsOfSemigroup` returns the group of inner automorphisms of the transformation semigroup `S`.

The same result can be obtained by applying `InnerAutomorphismsAutomorphismGroup` (7.3.6) to the result of `AutomorphismGroup` (7.4.1) of `S`. It is possible that the inner automorphism of `S` have been calculated at the same time as the entire automorphism group of `S` but it might not be. If the degree of `S` is high, then this function may take a long time to return a value.

The notion of inner automorphisms of semigroups differs from the notion of the same name for groups. Indeed, if `S` is a semigroup of transformations of degree `n`, then `g` in the symmetric group `S_n` induces an inner automorphism of `S` if the mapping that takes `s` to `g^-1sg` for all `s` in `S` is an automorphism of `S`.

Example

```
gap> x:=Transformation([2,3,4,5,6,7,8,9,1]);;
gap> y:=Transformation([4,2,3,4,5,6,7,8,9]);;
gap> S:=Semigroup(x,y);;
gap> G:=InnerAutomorphismsOfSemigroup(S);
<group of size 54 with 2 generators>
```

7.3.5 InnerAutomorphismsOfSemigroupInGroup

◊ `InnerAutomorphismsOfSemigroupInGroup(`*S, G[, bval]*`)` (operation)

`InnerAutomorphismsOfSemigroupInGroup` returns the group of inner automorphisms of the transformation semigroup `S` that also belong to the group `G`. The default setting is that the inner automorphisms of `S` are calculated first, then filtered to see which elements also belong to `G`.

If the optional argument `bval` is present and `true`, then the filtering is done as the inner automorphisms are found rather than after they have all been found. Otherwise, then this is equivalent to doing `InnerAutomorphismsOfSemigroupInGroup(S, G)`.

If `InfoMonoidAutos` (7.1.1) is set to level `4`, then a prompt will appear during the procedure to let you decide when the filtering should be done. In this case the value of `bval` is irrelevant.

Example

```
gap> gens:=[ Transformation( [1,8,11,2,5,16,13,14,3,6,15,10,7,4,9,12 ] ),
>   Transformation( [1,16,9,6,5,8,13,12,15,2,3,4,7,10,11,14] ),
>   Transformation( [1,3,7,9,1,15,5,11,13,11,13,3,5,15,7,9 ] ) ];;
gap> S:=Semigroup(gens);;
gap> InnerAutomorphismsOfSemigroup(S);
```

```
<group of size 16 with 3 generators>
gap> G:=Group(SemigroupHomomorphismByImagesOfGensNC(S, S, gens));
<group with 1 generators>
gap> InnerAutomorphismsOfSemigroupInGroup(S, G);
<group of size 1 with 1 generators>
gap> InnerAutomorphismsOfSemigroupInGroup(S, G, true);
<group of size 1 with 1 generators>
gap> InnerAutomorphismsOfSemigroupInGroup(S, G, false);
<group of size 1 with 1 generators>
```

7.3.6 InnerAutomorphismsAutomorphismGroup

◊ InnerAutomorphismsAutomorphismGroup(*autgroup*) (attribute)

If autgroup satisfies IsAutomorphismGroupOfSemigroup (7.4.3) then, this attribute stores the subgroup of inner automorphisms of the original semigroup.

It is possible that the inner automorphisms of autgroup have been calculated at the same time as autgroup was calculated but they might not be. If the degree of underlying semigroup is high, then this function may take a long time to return a value.

The notion of inner automorphisms of semigroups differs from the notion of the same name for groups. Indeed, if S is a semigroup of transformations of degree n, then g in the symmetric group S_n induces an inner automorphism of S if the mapping that takes s to g^-1sg for all s in S is an automorphism of S.

If autgroup satisfies IsAutomorphismGroupOfZeroGroup (7.4.5), then InnerAutomorphismsAutomorphismGroup returns the subgroup of inner automorphisms inside the automorphism group of the zero group by computing the inner automorphisms of the underlying group. Note that in this case the notion of inner automorphisms corresponds to that of the group theoretic notion.

Example

```
gap> g1:=Transformation([3,3,2,6,2,4,4,6]);;
gap> g2:=Transformation([5,1,7,8,7,5,8,1]);;
gap> m6:=Semigroup(g1,g2);;
gap> A:=AutomorphismGroup(m6);
<group of size 12 with 2 generators>
gap> InnerAutomorphismsAutomorphismGroup(A);
<group of size 12 with 2 generators>
gap> last=InnerAutomorphismsOfSemigroup(m6);
```

7.3.7 IsInnerAutomorphismsOfSemigroup

◊ IsInnerAutomorphismsOfSemigroup(*G*) (property)

returns true if G is the inner automorphism group of a transformation semigroup.

The notion of inner automorphisms of semigroups differs from the notion of the same name for groups. Indeed, if S is a semigroup of transformations of degree n, then g in the symmetric group S_n induces an inner automorphism of S if the mapping that takes s to g^-1sg for all s in S is an automorphism of S.

Note that this property is set to `true` when the computation of the inner automorphisms is performed. Otherwise, there is no method to check if an arbitrary group satisfies this property.

Example

```
gap> S:=RandomSemigroup(5,5);
<semigroup with 5 generators>
gap> I:=InnerAutomorphismsOfSemigroup(S);;
gap> IsInnerAutomorphismsOfSemigroup(I);
true
```

7.3.8 IsInnerAutomorphismsOfZeroGroup

◊ `IsInnerAutomorphismsOfZeroGroup(`*G*`)` (property)

returns `true` if `G` is the inner automorphism group of a zero group. This property is set to `true` when the computation of the inner automorphism group of the zero group is performed. Otherwise, there is no method to check if an arbitrary group satisfies this property.

Every inner automorphism of a zero group is just an inner automorphism of the underlying group that fixes the zero element. So, this notion of inner automorphism corresponds to the notion of inner automorphisms of a group.

Example

```
gap> zg:=ZeroGroup(CyclicGroup(70));
<zero group with 4 generators>
gap> I:=InnerAutomorphismsAutomorphismGroup(AutomorphismGroup(zg));
<group of size 1 with 1 generators>
gap> IsInnerAutomorphismsOfZeroGroup(I);
true
```

7.4 Automorphism Groups

7.4.1 AutomorphismGroup

◊ `AutomorphismGroup(`*S*`)` (attribute)

`AutomorphismGroup` returns the group of automorphisms of the transformation semigroup, zero group, zero semigroup, Rees matrix semigroup, or Rees 0-matrix semigroup `S`; that is, semigroups satisfying the properties `IsTransformationSemigroup` (**Reference: IsTransformationSemigroup**), `IsZeroGroup` (5.2.15), `IsZeroSemigroup` (5.2.14), `IsReesMatrixSemigroup` (**Reference: IsReesMatrixSemigroup**), or `IsReesZeroMatrixSemigroup` (**Reference: IsReesZeroMatrixSemigroup**).

If `S` is a transformation semigroup, then `AutomorphismGroup` computes the automorphism group of `S` using the algorithm described in [ABM07].

If `S` is a (completely) simple transformation semigroup, then the automorphism group is computed by passing to an isomorphic Rees matrix semigroup. If `S` is a transformation group, then the automorphism group is computed by passing to an isomorphic permutation group. If `S` has order $<$`10` and

knows its Cayley table (MultiplicationTable (**Reference: MultiplicationTable**)), then the automorphism group is calculated by finding the setwise stabilizer of the Cayley table in the symmetric group of degree |S| under the action on the Cayley table.

If S is a zero group, then AutomorphismGroup computes the automorphism group of the underlying group. Obviously, every automorphism of a zero group is the extension of an automorphism of the underlying group that fixes the zero element.

If S is a zero semigroup, then every permutation of the elements of S that fixes the zero element is an automorphism. Thus the automorphism group of a zero semigroup of order n is isomorphic to the symmetric group on n-1 elements.

If S is a Rees matrix semigroup or a Rees 0-matrix semigroup, then the automorphism group of S is calculated using the algorithm described in [ABM07, Section 2]. In this case, the returned group has as many generators as elements. This may be changed in the future.

If InfoMonoidAutos (7.1.1) is set to level 4, then prompts will appear during the procedure to allow you interactive control over the computation.

PLEASE NOTE: if grape is not loaded, then this function will not work when S satisfies IsTransformationSemigroup (**Reference: IsTransformationSemigroup**) or IsReesZeroMatrixSemigroup (**Reference: IsReesZeroMatrixSemigroup**).

Example

```
gap> g1:=Transformation([5,4,4,2,1]);;
gap> g2:=Transformation([2,5,5,4,1]);;
gap> m2:=Monoid(g1,g2);;
gap> IsTransformationSemigroup(m2);
true
gap> AutomorphismGroup(m2);
<group of size 24 with 5 generators>
gap> IsAutomorphismGroupOfSemigroup(last);
true
gap> zg:=ZeroGroup(CyclicGroup(70));
<zero group with 4 generators>
gap> IsZeroGroup(zg);
true
gap> AutomorphismGroup(zg);
<group with 3 generators>
gap> IsAutomorphismGroupOfZeroGroup(last);
true
gap> InnerAutomorphismsOfSemigroup(zg);
<group of size 1 with 1 generators>
gap> InnerAutomorphismsAutomorphismGroup(AutomorphismGroup(zg));
<group of size 1 with 1 generators>
gap> last2=InnerAutomorphismsAutomorphismGroup(AutomorphismGroup(zg));
true
gap> S:=ZeroSemigroup(10);
<zero semigroup with 10 elements>
gap> Size(S);
10
gap> Elements(S);
[ 0, z1, z2, z3, z4, z5, z6, z7, z8, z9 ]
gap> A:=AutomorphismGroup(S);
<group with 2 generators>
gap> IsAutomorphismGroupOfZeroSemigroup(A);
true
```

```
gap> Factorial(9)=Size(A);
true
gap> G:=Group([ (2,5)(3,4) ]);;
gap> mat:=[ [ (), (), (), (), () ],
>   [ (), (), (2,5)(3,4), (2,5)(3,4), () ],
>   [ (), (), (), (2,5)(3,4), (2,5)(3,4) ],
>   [ (), (2,5)(3,4), (), (2,5)(3,4), () ],
>   [ (), (2,5)(3,4), (), (2,5)(3,4), () ] ];;
gap> rms:=ReesMatrixSemigroup(G, mat);
Rees Matrix Semigroup over Group([ (2,5)(3,4) ])
gap> A:=AutomorphismGroup(rms);
<group of size 12 with 12 generators>
gap> IsAutomorphismGroupOfRMS(A);
true
gap> G:=ZeroGroup(Group([ (1,3)(2,5), (1,3,2,5) ]));;
gap> elts:=Elements(G);;
gap> mat:=[ [ elts[7], elts[1], elts[9], elts[1], elts[1] ],
>   [ elts[1], elts[1], elts[1], elts[9], elts[1] ],
>   [ elts[9], elts[1], elts[1], elts[4], elts[9] ],
>   [ elts[1], elts[1], elts[1], elts[1], elts[1] ],
>   [ elts[1], elts[5], elts[1], elts[1], elts[1] ] ];;
gap> rzms:=ReesZeroMatrixSemigroup(G, mat);;
gap> AutomorphismGroup(rzms);
gap> IsAutomorphismGroupOfRZMS(A);
true
<group of size 512 with 512 generators>
```

7.4.2 AutomorphismsSemigroupInGroup

◊ AutomorphismsSemigroupInGroup(*S, G[, bvals]*) (operation)

AutomorphismsSemigroupInGroup returns the group of automorphisms of the transformation semigroup S that also belong to the group G. If the value of G is fail, then AutomorphismsSemigroupInGroup returns the same value as AutomorphismGroup (7.4.1). The default setting is that the automorphisms of S are calculated first, then filtered to see which elements also belong to G.

The optional argument bvals is a list of 5 Boolean variables that correspond to the following options:

- if bvals[1] is true, then GAP will run a cheap check to see if all the automorphisms are inner. Note that this can return false when all the automorphisms are inner, that is the condition is sufficient but not necessary. The default setting is false.
- if bvals[2] is true, then GAP will try to compute the inner automorphisms of S before computing the entire automorphism group. For semigroups of large degree this may not be sensible. The default setting is false.
- if bvals[3] is true, then GAP will test elements in the inner automorphism search space to see if they are in G as the inner automorphisms are found rather than after they have all been found. The default setting is false.

- if `bvals[4]` is `true`, then GAP will test elements in the outer (i.e. not inner) automorphism search space to see if they are in `G` as they are found rather than after they have all been found. The default setting is `false`.
- if `bvals[5]` is `true`, then GAP will keep track of non-automorphisms in the search for outer automorphisms. The default setting is `false`.

PLEASE NOTE: if grape is not loaded, then this function will not work when `S` satisfies `IsTransformationSemigroup` (**Reference: IsTransformationSemigroup**) or `IsReesZeroMatrixSemigroup` (**Reference: IsReesZeroMatrixSemigroup**).

Example

```
gap> g1:=Transformation([5,4,4,2,1]);;
gap> g2:=Transformation([2,5,5,4,1]);;
gap> m2:=Monoid(g1,g2);;
gap> A:=AutomorphismsSemigroupInGroup(m2, fail,
> [false, true, true, false, true]);
<group of size 24 with 3 generators>
gap> g1:=Transformation([3,3,2,6,2,4,4,6,3,4,6]);;
gap> g2:=Transformation([4,4,6,1,3,3,3,3,11,11,11]);;
gap> m7:=Monoid(g1,g2);;
gap> A:=AutomorphismsSemigroupInGroup(m7, fail,
> [false, true, false, false, true]);
<group of size 2 with 2 generators>
gap> imgs:=[ [ Transformation( [ 1, 1, 5, 4, 3, 6, 7, 8, 9, 10, 11, 12 ] ),
>       Transformation( [ 1, 1, 5, 7, 4, 3, 6, 8, 9, 10, 11, 12 ] ),
>       Transformation( [ 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 8 ] ) ],
>   [ Transformation( [ 1, 1, 5, 4, 3, 6, 7, 8, 9, 10, 11, 12 ] ),
>       Transformation( [ 1, 1, 5, 3, 7, 4, 6, 8, 9, 10, 11, 12 ] ),
>       Transformation( [ 1, 2, 3, 4, 5, 6, 7, 11, 12, 8, 9, 10 ] ) ] ];;
gap> gens:=List(imgs, x-> SemigroupHomomorphismByImagesOfGensNC(S, S, x));;
gap> G:=Group(gens);
<group with 2 generators>
gap> A:=AutomorphismsSemigroupInGroup(S, G,
> [false, false, false, true, false]);
<group of size 48 with 4 generators>
gap> Size(G);
48
gap> A:=AutomorphismsSemigroupInGroup(S, G);
<group of size 48 with 4 generators>
gap> gens:=[ Transformation( [ 1, 1, 4, 3, 5, 6, 7, 8, 9, 10, 11, 12 ] ),
>   Transformation( [ 1, 1, 4, 5, 6, 7, 3, 8, 9, 10, 11, 12 ] ),
>   Transformation( [ 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 8 ] ) ];;
gap> S:=Semigroup(gens);;
gap> A:=AutomorphismsSemigroupInGroup(S, G);
<group of size 48 with 4 generators>
gap> HasAutomorphismGroup(S);
true
gap> AutomorphismGroup(S);
<group of size 480 with 7 generators>
```

7.4.3 IsAutomorphismGroupOfSemigroup

◊ IsAutomorphismGroupOfSemigroup(*G*) (property)

returns true if G is the automorphism group of a semigroup. Note that this property is set to true when the computation of the automorphism group is performed. Otherwise, there is no method to check if an arbitrary group satisfies this property; see AutomorphismGroup (7.4.1) for an example of the usage of this command.

7.4.4 IsAutomorphismGroupOfSimpleSemigp

◊ IsAutomorphismGroupOfSimpleSemigp(*G*) (property)

returns true if G is the automorphism group of a simple transformation semigroup. This property is set to true when the computation of the automorphism group of the simple transformation semigroup is performed. Otherwise, there is no method to check if an arbitrary group satisfies this property; see AutomorphismGroup (7.4.1) for an example of the usage of this command.

7.4.5 IsAutomorphismGroupOfZeroGroup

◊ IsAutomorphismGroupOfZeroGroup(*G*) (property)

returns true if G is the automorphism group of a zero group. This property is set to true when the computation of the automorphism group of the zero group is performed. Otherwise, there is no method to check if an arbitrary group satisfies this property; see AutomorphismGroup (7.4.1) for an example of the usage of this command.

Every automorphism of a zero group is just an automorphism of the underlying group that fixes the zero element.

7.4.6 IsAutomorphismGroupOfZeroSemigroup

◊ IsAutomorphismGroupOfZeroSemigroup(*G*) (property)

returns true if G is the automorphism group of a zero semigroup. This property is set to true when the computation of the automorphism group of the zero semigroup is performed. Otherwise, there is no method to check if an arbitrary group satisfies this property; see AutomorphismGroup (7.4.1) for an example of the usage of this command.

Every permutation of a zero semigroup that fixes the zero element is an automorphism. Thus the automorphism group of a zero semigroup of order n is isomorphic to the symmetric group on n-1 elements.

7.4.7 IsAutomorphismGroupOfRMS

◊ IsAutomorphismGroupOfRMS(*G*) (property)

returns true if G is the automorphism group of a Rees matrix semigroup; that is, a semigroup created using the command ReesMatrixSemigroup (**Reference: ReesMatrixSemigroup**) and/or satisfying IsReesMatrixSemigroup (**Reference: IsReesMatrixSemigroup**).

Note that this property is set to true when the computation of the automorphism group is performed. Otherwise, there is no method to check if an arbitrary group satisfies this property; see AutomorphismGroup (7.4.1) for an example of the usage of this command.

7.4.8 IsAutomorphismGroupOfRZMS

◊ IsAutomorphismGroupOfRZMS(*G*) (property)

returns true if G is the automorphism group of a Rees matrix semigroup; that is, a semigroup created using the command ReesZeroMatrixSemigroup (**Reference: ReesZeroMatrixSemigroup**) and/or satisfying IsReesZeroMatrixSemigroup (**Reference: IsReesZeroMatrixSemigroup**).

Note that this property is set to true when the computation of the automorphism group is performed. Otherwise, there is no method to check if an arbitrary group satisfies this property; see AutomorphismGroup (7.4.1) for an example of the usage of this command.

7.5 Rees Matrix Semigroups

7.5.1 RMSIsoByTriple

◊ RMSIsoByTriple(*rms1, rms2, triple*) (function)

this is a function to create an isomorphism between the Rees matrix semigroups rms1 and rms2 defined by triple. The first component of triple should be an isomorphism from the underlying group of rms1 to the underlying group of rms2, the second component should be an isomorphism from the graph associated to the matrix of rms1 to the graph associated with the matrix of rms2, and the third component should be a function (given as a list of image elements) from the index sets of rms1 to the underlying group of rms2; see [ABM07, Section 2] for further details.

Note that this function only creates an object with representation IsRMSIsoByTripleRep (7.5.3) and does not check that triple actually defines an isomorphism from rms1 to rms2 or that the arguments even make sense. To create an isomorphism from rms1 to rms2 use IsomorphismSemigroups (7.7.5).

Example

```
gap> G:=Group((1,4,3,5,2));;
gap> mat:=[ [ (), (), () ], [ (), (1,4,3,5,2), () ], [ (), (1,3,2,4,5), () ] ];;
gap> rms:=ReesMatrixSemigroup(G, mat);;
gap> l:=(4,6);;
gap> g:=GroupHomomorphismByImages(G, G, [(1,4,3,5,2)], [(1,2,5,3,4)]);
[ (1,4,3,5,2) ] -> [ (1,2,5,3,4) ]
gap> map:=[(), (1,5,4,2,3), (), (), (), () ];;
gap> RMSIsoByTriple(rms, rms, [l, g, map]);
[ (4,6), GroupHomomorphismByImages( Group( [ (1,4,3,5,2) ] ), Group(
  [ (1,4,3,5,2) ] ), [ (1,4,3,5,2) ], [ (1,2,5,3,4) ] ),
  [ (), (1,5,4,2,3), (), (), (), () ] ]
gap> IsRMSIsoByTripleRep(last);
true
gap> #the previous actually defines an automorphism of rms
gap> #on the other hand, the next example is nonsense but no error
gap> #is given
gap> RMSIsoByTriple(rms, rms, [l, g, [()]]);
```

```
[ (4,6), GroupHomomorphismByImages( Group( [ (1,4,3,5,2) ] ), Group(
  [ (1,4,3,5,2) ] ), [ (1,4,3,5,2) ], [ (1,2,5,3,4) ] ), [ () ] ]
```

7.5.2 RZMSIsoByTriple

◊ RZMSIsoByTriple(*rzms1, rzms2, triple*) (function)

this is a function to create an isomorphism between the Rees 0-matrix semigroups rzms1 and rzms2 defined by triple. The first component of triple should be an isomorphism from the underlying zero group of rzms1 to the underlying zero group of rzms2, the second component should be an isomorphism from the graph associated to the matrix of rzms1 to the graph associated with the matrix of rzms2, and the third component should be a function (given as a list of image elements) from the index sets of rzms1 to the underlying zero group of rzms2; see [ABM07, Section 2] for further details.

Note that this function only creates an object with representation IsRZMSIsoByTripleRep (7.5.4) and does not check that triple actually defines an isomorphism from rzms1 to rzms2 or that the arguments even make sense. To create an isomorphism from rzms1 to rzms2 use IsomorphismSemigroups (7.7.5).

Example

```
gap> G:=Group((1,4,3,5,2));;
gap> ZG:=ZeroGroup(G);
<zero group with 2 generators>
gap> mat:=[ [ (), (), () ], [ (), (1,4,3,5,2), () ], [ (), (1,3,2,4,5), () ] ];;
gap> mat:=List(mat, x-> List(x, ZeroGroupElt));
[ [ (), (), () ], [ (), (1,4,3,5,2), () ], [ (), (1,3,2,4,5), () ] ]
gap> rms:=ReesZeroMatrixSemigroup(ZG, mat);
Rees Zero Matrix Semigroup over <zero group with 2 generators>
gap> l:=(4,6);;
gap> g:=GroupHomomorphismByImages(G, G, [(1,4,3,5,2)], [(1,2,5,3,4)]);
[ (1,4,3,5,2) ] -> [ (1,2,5,3,4) ]
gap> g:=ZeroGroupAutomorphism(ZG, g);
<mapping: <zero group with 2 generators> -> <zero group with 2 generators> >
gap>  map:=List([(), (1,5,4,2,3), (), (), (), () ], ZeroGroupElt);;
gap> RZMSIsoByTriple(rms, rms, [l, g, map]);
[ (4,6), <mapping: <zero group with 2 generators> -> <zero group with
  2 generators> >,
[ ZeroGroup(()), ZeroGroup((1,5,4,2,3)), ZeroGroup(()), ZeroGroup(()),
    ZeroGroup(()), ZeroGroup(()) ] ]
gap> RZMSIsoByTriple(rms, rms, [l, g, [()]]);
[ (4,6), <mapping: <zero group with 2 generators> -> <zero group with
  2 generators> >, [ () ] ]
gap> IsRZMSIsoByTripleRep(last);
true
```

7.5.3 IsRMSIsoByTripleRep

◊ IsRMSIsoByTripleRep(*f*) (Representation)

returns true if the object f is represented as an isomorphism of Rees matrix semigroups by a triple; as explained in [ABM07, Section 2]; see RMSIsoByTriple (7.5.1) for an example of the usage of this command.

7.5.4 IsRZMSIsoByTripleRep

◊ IsRZMSIsoByTripleRep(*f*) (Representation)

returns true if the object f is represented as an isomorphism of Rees matrix semigroups by a triple; as explained in [ABM07, Section 2]; see RZMSIsoByTriple (7.5.2) for an example of the usage of this command.

7.5.5 RMSInducedFunction

◊ RMSInducedFunction(*RMS, lambda, gamma, g*) (operation)

lambda is an automorphism of the graph associated to the Rees matrix semigroup RMS, gamma an automorphism of the underlying group of RMS, and g an element of the underlying group of RMS. The function RMSInducedFunction attempts to find the function determined by lambda and gamma from the union of the index sets I and J to the group G of the Rees matrix semigroup RMS over G, I, and J with respect to P where the first element is given by the element g. If a conflict is found, then false is returned together with the induced map; see [ABM07, Section 2] for further details.

Example

```
gap> G:=Group([ (1,2) ]);;
gap> mat:=[ [ (), (), () ], [ (), (1,2), () ], [ (), (1,2), (1,2) ],
>    [ (), (), () ], [ (), (1,2), () ] ];;
gap> rms:=ReesMatrixSemigroup(G, mat);;
gap> l:=(1,2)(4,5,6);
(1,2)(4,5,6)
gap> gam:=One(AutomorphismGroup(G));
IdentityMapping( Group([ (1,2) ]) )
gap> g:=(1,2);
gap> RMSInducedFunction(rms, l, gam, g);
[ false, [ (1,2), (), (), (), (), (1,2), (1,2), () ] ]
gap> RMSInducedFunction(rms, (4,7), gam, ());
[ true, [ (), (), (), (), (), (), (), () ] ]
```

7.5.6 RZMSInducedFunction

◊ RZMSInducedFunction(*RZMS, lambda, gamma, g, comp*) (operation)

lambda is an automorphism of the graph associated to the Rees 0- matrix semigroup RZMS, gamma an automorphism of the underlying zero group of RZMS, comp is a connected component of the graph associated to RZMS, and g is an element of the underlying zero group of RZMS. The function RZMSInducedFunction attempts to find the partial function determined by lambda and gamma from comp to the zero group G^0 of G of the Rees 0-matrix semigroup RZMS over G^0, I, and J with respect to P where the image of the first element in comp is given by the element g. If a conflict is found, then fail is returned; see [ABM07, Section 2] for further details.

PLEASE NOTE: if grape is not loaded, then this function will not work.

Example

```
gap> zg:=ZeroGroup(Group(()));;
gap> z:=Elements(zg)[1];
0
gap> x:=Elements(zg)[2];
```

```
()
gap> mat:=[ [ z, z, z ], [ x, z, z ], [ x, x, z ] ];;
gap> rzms:=ReesZeroMatrixSemigroup(zg, mat);;
gap> RZMSInducedFunction(rzms, (), One(AutomorphismGroup(zg)), x,
> [1,2,5,6])
[ (), (),,, (), () ]
gap> RZMSInducedFunction(rzms, (), One(AutomorphismGroup(zg)), x, [3]);
[ ,, () ]
gap> RZMSInducedFunction(rzms, (), One(AutomorphismGroup(zg)), x, [4]);
[ ,,, () ]
gap> zg:=ZeroGroup(Group([ (1,5,2,3), (1,4)(2,3) ]));;
gap> elts:=Elements(zg);;
gap> mat:=[ [ elts[1], elts[1], elts[11], elts[1], elts[1] ],
>    [ elts[1], elts[13], elts[21], elts[1], elts[1] ],
>    [ elts[1], elts[16], elts[1], elts[16], elts[3] ],
>    [ elts[10], elts[17], elts[1], elts[1], elts[1] ],
>    [ elts[1], elts[1], elts[1], elts[4], elts[1] ] ];
gap> rzms:=ReesZeroMatrixSemigroup(zg, mat);
gap> RZMSInducedFunction(rzms, (), Random(AutomorphismGroup(zg)),
> Random(elts), [1..10])=fail;
false
```

7.5.7 RZMStoRZMSInducedFunction

◊ RZMStoRZMSInducedFunction(*RZMS1, RZMS2, lambda, gamma, elts*) (operation)

`lambda` is an automorphism of the graph associated to the Rees 0- matrix semigroup `RZMS1` composed with isomorphism from that graph to the graph of `RZMS2`, `gamma` an automorphism of the underlying zero group of `RZMS1`, and `elts` is a list of elements of the underlying zero group of `RZMS2`. The function `RZMStoRZMSInducedFunction` attempts to find the function determined by `lambda` and `gamma` from the union of the index sets `I` and `J` of `RZMS1` to the zero group `G^0` of the Rees 0-matrix semigroup `RZMS2` over the zero group `G^0`, sets `I` and `J`, and matrix `P` where the image of the first element in the `i`th connected component of the associated graph of `RZMS1` is given by `elts[i]`. If a conflict is found, then `false` is returned; see [ABM07, Section 2] for further details.

PLEASE NOTE: if grape is not loaded, then this function will not work.

Example

```
gap> gens:=[ Transformation( [ 4, 4, 8, 8, 8, 8, 4, 8 ] ),
  Transformation( [ 8, 2, 8, 2, 5, 5, 8, 8 ] ),
  Transformation( [ 8, 8, 3, 7, 8, 3, 7, 8 ] ),
  Transformation( [ 8, 6, 6, 8, 6, 8, 8, 8 ] ) ];;
gap> S:=Semigroup(gens);;
gap> D:=GreensDClasses(S);;
gap> rms1:=Range(IsomorphismReesMatrixSemigroupOfDClass(D[1]));
Rees Zero Matrix Semigroup over <zero group with 2 generators>
gap> rms2:=Range(IsomorphismReesMatrixSemigroupOfDClass(D[4]));
Rees Zero Matrix Semigroup over <zero group with 2 generators>
gap> gam:=One(AutomorphismGroup
> (UnderlyingSemigroupOfReesZeroMatrixSemigroup(Group(rms1))));
IdentityMapping( <zero group with 2 generators> )
gap> g:=One(UnderlyingSemigroupOfReesZeroMatrixSemigroup(rms2));
```

```
()
gap> RZMStoRZMSInducedFunction(rms1, rms2, (2,3)(5,6), gam, [g]);
[ (), (), (), (), (), () ]
```

7.5.8 RZMSGraph

◇ RZMSGraph(*rzms*) (attribute)

if rzms is a Rees 0-matrix semigroup over a zero group G^0, 3 index sets I and J, and matrix P, then RZMSGraph returns the undirected bipartite graph with |I|+|J| vertices and edge (i,j) if and only if i<|I|+1, j>|I| and p_{j-|I|, i} is not zero.

The returned object is a simple undirected graph created in GRAPE using the command

$$Graph(Group(()), [1..n+m], OnPoints, adj, true);$$

where adj is true if and only if i<|I|+1, j>|I| and p_{j-|I|, i} is not zero.

PLEASE NOTE: if grape is not loaded, then this function will not work.

Example

```
gap> zg:=ZeroGroup(Group(()));;
gap> z:=Elements(zg)[1];
0
gap> x:=Elements(zg)[2];
()
gap> mat:=[ [ 0, 0, 0 ], [ (), 0, 0 ], [ (), (), 0 ] ];;
gap> rzms:=ReesZeroMatrixSemigroup(zg, mat);;
gap> RZMSGraph(rzms);
rec( isGraph := true, order := 6, group := Group(()),
  schreierVector := [ -1, -2, -3, -4, -5, -6 ],
  adjacencies := [ [ 5, 6 ], [ 6 ], [  ], [  ], [ 1 ], [ 1, 2 ] ],
  representatives := [ 1, 2, 3, 4, 5, 6 ], names := [ 1, 2, 3, 4, 5, 6 ] )
gap> UndirectedEdges(last);
[ [ 1, 5 ], [ 1, 6 ], [ 2, 6 ] ]
```

7.5.9 RightTransStabAutoGroup

◇ RightTransStabAutoGroup(*S, elts, func*) (operation)

returns a right transversal of the stabilizer w.r.t the action func of the elements elts in the automorphism group of the zero semigroup, Rees matrix semigroup, or Rees 0-matrix semigroup S. That is, S satisfying IsZeroSemigroup (5.2.14), IsReesMatrixSemigroup (**Reference: IsReesMatrixSemigroup**), or IsReesZeroMatrixSemigroup (**Reference: IsReesZeroMatrixSemigroup**).

Example

```
gap> S:=ZeroSemigroup(6);
<zero semigroup with 6 elements>
gap> elts:=Elements(S);
[ 0, z1, z2, z3, z4, z5 ]
gap> Length(RightTransStabAutoGroup(S, [elts[1]], OnSets));
1
gap> Length(RightTransStabAutoGroup(S, [elts[1], elts[2]], OnSets));
```

```
5
gap> Length(RightTransStabAutoGroup(S, [elts[1], elts[2]], OnTuples));
5
gap> G:=Group([ (1,2) ]);;
gap> mat:=[ [ (), (), () ], [ (), (1,2), () ], [ (), (1,2), (1,2) ],
>    [ (), (), () ], [ (), (1,2), () ] ];;
gap> rms:=ReesMatrixSemigroup(G, mat);;
gap> Size(rms);
30
gap> GeneratorsOfSemigroup(rms);
[ (1,(),2), (1,(),3), (1,(),4), (1,(),5), (2,(),1), (3,(),1), (1,(1,2),1) ]
gap> Length(RightTransStabAutoGroup(rms, last, OnSets));
4
gap> Length(RightTransStabAutoGroup(rms, GeneratorsOfSemigroup(rms),
> OnTuples));
8
gap> G:=ZeroGroup(Group([ (1,3) ]));;
gap> z:=MultiplicativeZero(G);; x:=Elements(G)[2];;
gap> mat:=[ [ z, z, z ], [ z, z, z ], [ z, z, z ], [ z, z, z ], [ z, x, z ] ];;
gap> rzms:=ReesZeroMatrixSemigroup(G, mat);
gap> Size(rzms);
31
gap> Size(GeneratorsOfSemigroup(rzms));
6
gap> Length(RightTransStabAutoGroup(rzms, GeneratorsOfSemigroup(rzms),
> OnSets));
512
gap> A:=AutomorphismGroup(rzms);
<group of size 3072 with 3072 generators>
```

7.6 Zero Groups

7.6.1 ZeroGroupAutomorphism

◊ ZeroGroupAutomorphism(*ZG, f*) (function)

converts the group automorphism f of the underlying group of the zero group ZG into an automorphism of the zero group ZG.

Example

```
gap> G:=Random(AllGroups(20));
<pc group of size 20 with 3 generators>
gap> A:=AutomorphismGroup(G);
<group with 2 generators>
gap> f:=Random(A);
[ f1*f2^4*f3 ] -> [ f1*f2^2 ]
gap> ZG:=ZeroGroup(G);
<zero group with 4 generators>
gap> ZeroGroupAutomorphism(ZG, f);
<mapping: <zero group with 4 generators> -> <zero group with 4 generators> >
gap> IsZeroGroupAutomorphismRep(last);
true
```

```
gap> UnderlyingGroupAutoOfZeroGroupAuto(last2)=f;
true
```

7.6.2 IsZeroGroupAutomorphismRep

◊ IsZeroGroupAutomorphismRep(*f*) (Representation)

returns true if the object f is represented as an automorphism of a zero group; see ZeroGroupAutomorphism (7.6.1) for an example of the usage of this command.

7.6.3 UnderlyingGroupAutoOfZeroGroupAuto

◊ UnderlyingGroupAutoOfZeroGroupAuto(*f*) (attribute)

returns the underlying group automorphism of the zero group automorphism f. That is, the restriction of f to its source without the zero; see ZeroGroupAutomorphism (7.6.1) for an example of the usage of this command.

7.7 Isomorphisms

7.7.1 IsomorphismAutomorphismGroupOfRMS

◊ IsomorphismAutomorphismGroupOfRMS(*G*) (attribute)

if G is the automorphism group of a simple transformation semigroup, then IsomorphismAutomorphismGroupOfRMS returns a GroupHomomorphismByImages (**Reference: GroupHomomorphismByImages**) from the automorphism group of G to the automorphism group of an isomorphic Rees matrix semigroup, obtained by using IsomorphismReesMatrixSemigroup (7.7.7).

Example

```
gap> g1:=Transformation([1,2,2,1,2]);;
gap> g2:=Transformation([3,4,3,4,4]);;
gap> g3:=Transformation([3,4,3,4,3]);;
gap> g4:=Transformation([4,3,3,4,4]);;
gap> cs5:=Semigroup(g1,g2,g3,g4);;
gap> AutomorphismGroup(cs5);
<group of size 16 with 3 generators>
gap> IsomorphismAutomorphismGroupOfRMS(last);
[ SemigroupHomomorphism ( <semigroup with 4 generators>-><semigroup with
    4 generators>), SemigroupHomomorphism ( <semigroup with
    4 generators>-><semigroup with 4 generators>),
  SemigroupHomomorphism ( <semigroup with 4 generators>-><semigroup with
    4 generators>) ] ->
[ [ (1,4)(2,3)(5,6), IdentityMapping( Group( [ (1,2) ] ) ),
      [ (), (1,2), (1,2), (), (), () ] ],
  [ (1,3,4,2), IdentityMapping( Group( [ (1,2) ] ) ),
      [ (), (), (), (), (), (1,2) ] ],
  [ (1,3)(2,4), IdentityMapping( Group( [ (1,2) ] ) ),
```

```
        [ (), (), (), (), (), (1,2) ] ] ]
```

7.7.2 IsomorphismPermGroup

◊ IsomorphismPermGroup(*G*) (attribute)

if G satisfies IsAutomorphismGroupOfSimpleSemigp (7.4.4), then IsomorphismPermGroup returns an isomorphism from G to a permutation group by composing the result f of IsomorphismAutomorphismGroupOfRMS (7.7.1) on G with the result of IsomorphismPermGroup on Range(f).

if G satisfies IsAutomorphismGroupOfRMS (7.4.7) or IsAutomorphismGroupOfRZMS (7.4.8), then IsomorphismPermGroup returns an isomorphism from G to a permutation group acting either on the elements of S or on itself, whichever gives a permutation group of lower degree.

if G is a transformation semigroup that satisfies IsGroupAsSemigroup (5.2.3), then IsomorphismPermGroup returns an isomorphism from G to the permutation group obtained by applying AsPermOfRange (2.3.2) to any element of G.

if G is a group H-class of a transformation semigroup, then IsomorphismPermGroup returns an isomorphism from G to the permutation group obtained by applying AsPermOfRange (2.3.2) to any element of G.

Example

```
gap> g1:=Transformation([3,3,2,6,2,4,4,6]);;
gap> g2:=Transformation([5,1,7,8,7,5,8,1]);;
gap> cs1:=Semigroup(g1,g2);;
gap> AutomorphismGroup(cs1);
<group of size 12 with 2 generators>
gap> IsomorphismPermGroup(last);
[ SemigroupHomomorphism ( <semigroup with 2 generators>-><semigroup with
    2 generators>), SemigroupHomomorphism ( <semigroup with
    2 generators>-><semigroup with 2 generators>) ] ->
[ (1,11,2,12,3,10)(4,8,5,9,6,7), (1,6)(2,5)(3,4)(7,10)(8,12)(9,11) ]
gap> Size(cs1);
96
gap> a:=IdempotentNC([[1,3,4],[2,5],[6],[7],[8]],[3,5,6,7,8])*(3,5);;
gap> b:=IdempotentNC([[1,3,4],[2,5],[6],[7],[8]],[3,5,6,7,8])*(3,6,7,8);;
gap> S:=Semigroup(a,b);;
gap> IsGroupAsTransSemigroup(S);
true
gap> IsomorphismPermGroup(S);
SemigroupHomomorphism ( <semigroup with 2 generators>->Group(
[ (3,5), (3,6,7,8) ]))
gap> gens:=[Transformation([3,5,3,3,5,6]), Transformation([6,2,4,2,2,6])];;
gap> S:=Semigroup(gens);;
gap> H:=GroupHClassOfGreensDClass(GreensDClassOfElement(S, Elements(S)[1]));
{Transformation( [ 2, 2, 2, 2, 2, 6 ] )}
gap> IsomorphismPermGroup(H);
SemigroupHomomorphism ( {Transformation( [ 2, 2, 2, 2, 2, 6 ] )}->Group(()))
```

7.7.3 IsomorphismFpSemigroup

◊ IsomorphismFpSemigroup(*S*) (attribute)

returns an isomorphism to a finitely presented semigroup from the transformation semigroup S. This currently works by running the function FroidurePinExtendedAlg (FroidurePinExtendedAlg???) in the library.

If S satisfies IsMonoid (**Reference: IsMonoid**), use the command IsomorphismFpMonoid (7.7.4) instead.

Example

```
gap> gens:=[ Transformation( [1,8,11,2,5,16,13,14,3,6,15,10,7,4,9,12 ] ),
>   Transformation( [1,16,9,6,5,8,13,12,15,2,3,4,7,10,11,14] ),
>   Transformation( [1,3,7,9,1,15,5,11,13,11,13,3,5,15,7,9 ] ) ];
gap> S:=Semigroup(gens);
<semigroup with 3 generators>
gap> IsomorphismFpSemigroup(last);
SemigroupHomomorphismByImages ( <trans. semigroup of size 16 with
3 generators>->Semigroup( [ s1, s2, s3 ] ))
```

7.7.4 IsomorphismFpMonoid

◊ IsomorphismFpMonoid(*S*) (attribute)

returns an isomorphism to a finitely presented monoid from the transformation monoid S. Currently works by running the function FroidurePinExtendedAlg (FroidurePinExtendedAlg???) in the library.

If S satisfies IsSemigroup (**Reference: IsSemigroup**), use the command IsomorphismFpSemigroup (7.7.3) instead.

Example

```
gap> x:=Transformation([2,3,4,5,6,7,8,9,1]);;
gap> y:=Transformation([4,2,3,4,5,6,7,8,9]);;
gap> S:=Monoid(x,y);;
gap> IsomorphismFpMonoid(last);
SemigroupHomomorphismByImages ( <trans. semigroup of size 40266 with
3 generators>->Monoid( [ m1, m2 ], ... ))
gap> Length(RelationsOfFpMonoid(Range(last)));
932
```

7.7.5 IsomorphismSemigroups

◊ IsomorphismSemigroups(*S*, *T*) (operation)

this operation returns an isomorphism from the semigroup S to the semigroup T if one exists and returns fail otherwise.

PLEASE NOTE: this function currently only works for zero groups, zero semigroups, Rees matrix semigroups, and Rees 0-matrix semigroups.

PLEASE NOTE: if grape is not loaded, then this function will not work when S and T satisfy IsReesZeroMatrixSemigroup (**Reference: IsReesZeroMatrixSemigroup**).

Example

```
gap> ZG1:=ZeroGroup(Group((1,2,3,5,4)));
<zero group with 2 generators>
gap> ZG2:=ZeroGroup(Group((1,2,3,4,5)));
<zero group with 2 generators>
gap> IsomorphismSemigroups(ZG1, ZG2);
SemigroupHomomorphismByImagesOfGens ( <zero group with
2 generators>-><zero group with 2 generators>)
gap> ZG2:=ZeroGroup(Group((1,2,3,4)));
<zero group with 2 generators>
gap> IsomorphismSemigroups(ZG1, ZG2);
fail
gap> IsomorphismSemigroups(ZeroSemigroup(5),ZeroSemigroup(5));
IdentityMapping( <zero semigroup with 5 elements> )
gap> IsomorphismSemigroups(ZeroSemigroup(5),ZeroSemigroup(6));
fail
gap> gens:=[ Transformation( [ 4, 4, 8, 8, 8, 8, 4, 8 ] ),
>   Transformation( [ 8, 2, 8, 2, 5, 5, 8, 8 ] ),
>   Transformation( [ 8, 8, 3, 7, 8, 3, 7, 8 ] ),
>   Transformation( [ 8, 6, 6, 8, 6, 8, 8, 8 ] ) ];;
gap> S:=Semigroup(gens);;
gap> D:=GreensDClasses(S);;
gap> rms1:=Range(IsomorphismReesMatrixSemigroupOfDClass(D[1]));;
gap> rms2:=Range(IsomorphismReesMatrixSemigroupOfDClass(D[4]));;
gap> IsomorphismSemigroups(rms1, rms2);
[ (2,3)(5,6), IdentityMapping( <zero group with 2 generators> ),
  [ ZeroGroup(()), ZeroGroup(()), ZeroGroup(()), ZeroGroup(()),
      ZeroGroup(()), ZeroGroup(()) ] ]
gap> IsomorphismSemigroups(rms2, rms1);
[ (2,3)(5,6), IdentityMapping( <zero group with 2 generators> ),
  [ ZeroGroup(()), ZeroGroup(()), ZeroGroup(()), ZeroGroup(()),
      ZeroGroup(()), ZeroGroup(()) ] ]
gap> rms2:=Range(IsomorphismReesMatrixSemigroupOfDClass(D[2]));
Group(())
gap> IsomorphismSemigroups(rms2, rms1);
fail
gap> rms2:=RandomReesZeroMatrixSemigroup(5,5,5);
Rees Zero Matrix Semigroup over <zero group with 2 generators>
gap> IsomorphismSemigroups(rms2, rms1);
fail
gap> rms2:=RandomReesMatrixSemigroup(5,5,5);
Rees Matrix Semigroup over Group([ (1,2)(3,4,5), (2,4,3), (1,4,5,3),
  (1,4,5,2) ])
gap> IsomorphismSemigroups(rms2, rms1);
fail
gap> IsomorphismSemigroups(rms1, rms2);
fail
```

7.7.6 IsomorphismReesMatrixSemigroupOfDClass

◊ IsomorphismReesMatrixSemigroupOfDClass(*D*) (attribute)

The *principal factor* of the D-class D is the semigroup with elements D and 0 and multiplication x*y defined to be the product xy in the semigroup containing D if xy in D and 0 otherwise.

IsomorphismReesMatrixSemigroupOfDClass returns an isomorphism from the principal factor of the D-class D to a Rees matrix, Rees 0-matrix or zero semigroup, as given by the Rees-Suschewitsch Theorem; see [How95, Theorem 3.2.3].

Example

```
gap> g1:=Transformation( [ 4, 6, 3, 8, 5, 6, 10, 4, 3, 7 ] );;
gap> g2:=Transformation( [ 5, 6, 6, 3, 8, 6, 3, 7, 8, 4 ] );;
gap> g3:=Transformation( [ 8, 6, 3, 2, 8, 10, 9, 2, 6, 2 ] );;
gap> m23:=Monoid(g1,g2,g3);;
gap> D:=GreensDClasses(m23)[17];
{Transformation( [ 7, 6, 6, 6, 7, 4, 8, 6, 6, 6 ] )}
gap> IsomorphismReesMatrixSemigroupOfDClass(D);
SemigroupHomomorphism ( {Transformation( [ 7, 6, 6, 6, 7, 4, 8, 6, 6, 6
 ] )}->zero semigroup with 3 elements)
gap> D:=GreensDClasses(m23)[77];
{Transformation( [ 6, 6, 6, 6, 6, 6, 6, 6, 6, 6 ] )}
gap> IsomorphismReesMatrixSemigroupOfDClass(D);
SemigroupHomomorphism ( {Transformation( [ 6, 6, 6, 6, 6, 6, 6, 6, 6, 6
 ] )}->Rees Matrix Semigroup over Group(())
gap> D:=GreensDClasses(m23)[1];
{Transformation( [ 1 .. 10 ] )}
gap> IsomorphismReesMatrixSemigroupOfDClass(D);
SemigroupHomomorphism ( {Transformation( [ 1 .. 10 ] )}->Group(())
gap> D:=GreensDClasses(m23)[23];
{Transformation( [ 6, 7, 3, 6, 6, 6, 6, 6, 7, 6 ] )}
gap> IsomorphismReesMatrixSemigroupOfDClass(D);
SemigroupHomomorphism ( {Transformation( [ 6, 7, 3, 6, 6, 6, 6, 6, 7, 6
 ] )}->Rees Zero Matrix Semigroup over <zero group with 3 generators>)
```

7.7.7 IsomorphismReesMatrixSemigroup

◊ IsomorphismReesMatrixSemigroup(*S*) (operation)

returns an isomorphism from the (completely) simple transformation semigroup S to a Rees matrix semigroup, as given by the Rees-Suschewitsch Theorem; see [How95, Theorem 3.2.3].

Example

```
gap> g1:=Transformation( [ 2, 3, 4, 5, 1, 8, 7, 6, 2, 7 ] );;
gap> g2:=Transformation( [ 2, 3, 4, 5, 6, 8, 7, 1, 2, 2 ] );;
gap> cs2:=Semigroup(g1,g2);;
gap> IsomorphismReesMatrixSemigroup(cs2);
SemigroupHomomorphism ( <semigroup with
2 generators>->Rees Matrix Semigroup over Group(
[ (2,5)(3,8)(4,6), (1,6,3)(5,8) ]))
```

References

[ABM07] J. Araujo, P. v. Bunau, and J. D. Mitchell. Computing automorphisms of semigroups. 2007. submitted for publication. 7, 48, 54, 55, 59, 60, 61, 62

[AR00] Robert E. Arthur and N. Ruskuc. Presentations for two extensions of the monoid of order-preserving mappings on a finite chain. *Southeast Asian Bull. Math.*, 24(1):1–7, 2000. 43

[GH92] Gracinda M. S. Gomes and John M. Howie. On the ranks of certain semigroups of order-preserving transformations. *Semigroup Forum*, 45(3):272–282, 1992. 43

[GM05] R. Gray and J. D. Mitchell. Largest subsemigroups of the full transformation monoid. *Discrete Math.*, 2005. to appear. 7, 36

[How95] John M. Howie. *Fundamentals of semigroup theory*, volume 12 of *London Mathematical Society Monographs. New Series*. The Clarendon Press Oxford University Press, New York, 1995. Oxford Science Publications. 69

[KM05] G. Kudryavtseva and V. Mazorchuk. On kiselman's semigroup. 2005. arXiv:math/0511374v1. 44

[LM90] G. Lallement and R. McFadden. On the determination of Green's relations in finite transformation semigroups. *J. Symbolic Comput.*, 10(5):481–498, 1990. 28

[LPRRa] S. A. Linton, G. Pfeiffer, E. F. Robertson, and N. Ruskuc. Computing transformation semigroups. *J. Symbolic Comput.*, 33:145–162. 7, 20, 21, 22, 23, 28, 30, 31, 32

[LPRRb] S. A. Linton, G. Pfeiffer, E. F. Robertson, and N. Ruskuc. Groups and actions in transformation semigroups. *Math. Z.*, 228:435–450. 7, 28

Index