IdRel

A package for Identities among Relators

Version 2.31

01/06/2015

Anne Heyworth
Chris Wensley

Chris Wensley Email: c.d.wensley@bangor.ac.uk
Homepage: http://pages.bangor.ac.uk/~mas023/
Address: School of Computer Science, Bangor University,
Dean Street, Bangor, Gwynedd, LL57 1UT, U.K.

Abstract

The IdRel package was originally implemented in 1999, using the GAP 3 language, when the first author was studying for a Ph.D. in Bangor.

This package is designed to compute a minimal set of generators for the module of the identities among relators of a group presentation. It does this using

- rewriting and logged rewriting: a self-contained implementation of the Knuth-Bendix process using the monoid presentation associated to the group presentation;
- monoid polynomials: an implementation of the monoid ring;
- module polynomials: an implementation of the right module over this monoid generated by the relators.
- Y-sequences: used as a *rewriting* way of representing elements of a free crossed module (products of conjugates of group relators and inverse relators).

IdRel became an accepted GAP package in May 2015.

Bug reports, suggestions and comments are, of course, welcome. Please contact the second author at `c.d.wensley@bangor.ac.uk`.

Copyright

Acknowledgements

This documentation was prepared with the GAPDoc package of Frank Lübeck and Max Neunhöffer.

Contents

Chapter 1

Introduction

This manual describes the IdRel package for GAP 4.7 for computing the identities among relators of a group presentation using rewriting, logged rewriting, monoid polynomials, module polynomials and Y-sequences.

The theoretical background for these computations is contained in Brown and Huebschumann [BH82], Brown and Razak Salleh [BRS99] and is surveyed in the first author's thesis [Hey99].

IdRel is primarily designed for the computation of a minimal set of generators for the module of identities among relators. It also contains functions which compute logged rewrite systems for group presentations (and complete them where possible); functions for operations involving elements of monoid rings; and functions for operations with elements of right modules over monoid rings. The Y-sequences are used as a *rewriting* way of representing elements of a free crossed module (products of conjugates of group relators and inverse relators). The package is written entirely in GAP4, and requires no compilation.

The package is loaded into GAP with the `LoadPackage` command, and on-line help is available in the usual way.

Example

```
gap> LoadPackage( "idrel" );
gap> ?idrel
```

A pdf version of the IdRel manual is available in the `doc` directory of the home directory of IdRel. The information parameter `InfoIdRel` has default value `0`. When raised to a higher value, additional information is printed out. IdRel was originally developed in 1999 using GAP3, partially supported by a University of Wales Research Assistantship for the first author, Anne Heyworth.

If you use IdRel to solve a problem then please send a short email to the second author, to whom bug reports, suggestions and other comments should also be sent. You may reference the package by mentioning [HW03] and [Hey99].

The current version is 2.31 of 1st June 2015.

Chapter 2

Rewriting Systems

This chapter describes functions to construct rewriting systems for finitely presented groups which store rewriting information. The main example used is a presentation of the quaternion group q8 with generators a, b and relators $[a^4, b^4, abab^{-1}, a^2b^2]$.

2.1 Identity Y-sequences

A typical input for IdRel is an fp-group presentation. This requires a free group F on a set of generators and a set of relators R (words in the free group). The module of identities among relators for this presentation has as its elements the Peiffer equivalence classes of all products of conjugates of relators which represent the identity in the free group.

In this package the identities among relators are represented by Y-sequences, which are lists $[[r_1, u_1], \ldots, [r_k, u_k]]$ where $r_1, \ldots, r_k$ are the group relators or their inverses, and $u_1, \ldots, u_k$ are words in the free group F. A Y-sequence is evaluated in F as the product $(u_1^{-1} r_1 u_1) \ldots (u_k^{-1} r_k u_k)$ and is an identity Y-sequence if it evaluates to the identity in F. An identity Y-sequence represents an identity among the relators of the group presentation. The main function of the package is to produce a set of Y-sequences which generate the module of identites among relators, and further, that this set be minimal in the sense that every element in it is needed to generate the module.

Before starting on the main example, we consider a simpler example illustrating the use of IdRel. All the functions used are described in detail in this manual. We compute a reduced set of identities among relators for the presentation of the symmetric group s3 with generators a, b and relators $[a^3, b^2, (ab)^2]$. In the listing below, s3_M1 is the first monoid generator for s3, s3_R2 is the second relator, while s3_Y4 is the fourth Y-sequence for s3.

Example

```
gap> F := FreeGroup( 2 );;
gap> a := F.1;; b:= F.2;;
gap> rels3 := [ a^3 , b^2, a*b*a*b];
[ f1^3, f2^2, (f1*f2)^2 ]
gap> s3 := F/rels3;
<fp group on the generators [ f1, f2 ]>
gap> SetName( s3, "s3" );
gap> idy3 := IdentityYSequences( s3 );;
gap> Length( idy3 );
18
gap> Y4 := idy3[4];
```

```
[ [ s3_R1^-1, f1^-1 ], [ s3_R1, <identity ...> ] ]
gap> Y6 := idy3[6];
[ [ s3_R3^-1, f1^-1 ], [ s3_R1, <identity ...> ], [ s3_R3^-1, f1 ],
  [ s3_R2, f1^-1*f2^-1 ], [ s3_R1, f2^-1 ], [ s3_R3^-1, f1*f2^-1 ],
  [ s3_R2, <identity ...> ], [ s3_R2, f1^-1 ] ]
gap> Y7 := idy3[7];
[ [ s3_R3^-1, f1*f2^-1 ], [ s3_R2, <identity ...> ], [ s3_R3, <identity ...> ],
  [ s3_R2^-1, <identity ...> ] ]
gap> Y8 := idy3[8];
[ [ s3_R2^-1, f2^-1 ], [ s3_R2, <identity ...> ] ]
```

Of the 18 Y-sequences formed, 6 are empty, and Y4 is the *root identity* `((a^3)^-1)^(a^-1).(a^3)`. If we write $r = a^3, s = b^2, t = (ab)^2$ then Y4 is $(r^{-1})^{a^{-1}}r$. Similarly, Y8 is the second root identity $(s^{-1})^{b^{-1}}s$, while Y7 is the third root identity $(t^{-1})^{(ab)^{-1}}t$. The identity Y6, which is not a root identity, is obtained by walking around the Schreier diagram of the presentation, a somewhat truncated triangular prism. Taking the appropriate conjugate of each face in turn, we get: `Y6=(t^-1)^(a^-1).r.(t^-1)^a.s^(a^-1b^-1).r^(b^-1).(t^-1)^(ab^-1).s.s^(a^-1)`. These four identities generate the module of identities for s3.

Example
```
gap> idrels3 := IdentitiesAmongRelators( s3 );;
gap> Display( idrels3[1] );
[ ( s3_Y4*( s3_M2*s3_M1), s3_R1*( s3_M1 - <identity ...>) ),
  ( s3_Y8*( s3_M2*s3_M1), s3_R2*( s3_M2 - <identity ...>) ),
  ( s3_Y7*( s3_M2*s3_M1), s3_R3*( s3_M2 - s3_M1) ),
  ( s3_Y6*( -s3_M2*s3_M1), s3_R1*( -s3_M2*s3_M1 - s3_M1) + s3_R2*( -s3_M1*s3_M\
2 - s3_M1 - <identity ...>) + s3_R3*( s3_M3 + s3_M2 + <identity ...>) )
  ]
```

2.2 Monoid Presentations of FpGroups

2.2.1 FreeRelatorGroup

▷ FreeRelatorGroup(*grp*) (attribute)
▷ FreeRelatorHomomorphism(*grp*) (attribute)

The function `FreeRelatorGroup` returns a free group on the set of relators of the given fp-group `G`. If `HasName(G)` is `true` then a name is automatically assigned to the free group.

The function `FreeRelatorHomomorphism` returns the group homomorphism from the free group on the relators to the free group on the generators of `G`, mapping each generator to the corresponding word.

Example
```
gap> F := FreeGroup( 2 );;
gap> a := F.1;; b:= F.2;;
gap> rels := [ a^4, b^4, a*b*a*b^-1, a^2*b^2];;
gap> q8 := F/rels;;
gap> SetName( q8, "q8" );
gap> frq8 := FreeRelatorGroup( q8 );
q8_R
gap> GeneratorsOfGroup( frq8 );
[ q8_R1, q8_R2, q8_R3, q8_R4]
gap> frhomq8 := FreeRelatorHomomorphism( q8 );
[ q8_R1, q8_R2, q8_R3, q8_R4] -> [ f1^4, f2^4, f1*f2*f1*f2^-1, f1^2*f2^2]
```

2.2.2 MonoidPresentationFpGroup

▷ `MonoidPresentationFpGroup(grp)` (attribute)
▷ `FreeGroupOfPresentation(mon)` (attribute)
▷ `GroupRelatorsOfPresentation(mon)` (attribute)
▷ `InverseRelatorsOfPresentation(mon)` (attribute)
▷ `HomomorphismOfPresentation(mon)` (attribute)

A monoid presentation for a finitely presented group `G` has two monoid generators g^+, g^- for each group generator g. The relators of the monoid presentation comprise the group relators, and relators g^+g^- specifying the inverses. The function `MonoidPresentationFpGroup` returns the monoid presentation derived in this way from an fp-presentation. (Note: this function should always be followed by a double semicolon – `MonoidPresentationFpGroup(G);;` – because an error occurs in attempting to display the results on the screen: the `ElementsFamily` needs to be fixed.)

The function `FreeGroupOfPresentation` returns the free group on the monoid generators.

The function `GroupRelatorsOfPresentation` returns those relators of the monoid which correspond to the relators of the group. All negative powers in the group relators are converted to positive powers of the g^-.

The function `InverseRelatorsOfPresentation` returns relators which specify the inverse pairs of the monoid generators.

The function `HomomorphismOfPresentation` returns the homomorphism from the free group of the monoid presentation to the free group of the group presentation.

In the example below, the four monoid generators a^+, b^+, a^-, b^- are named `q8_M1, q8_M2, q8_M3, q8_M4`.

Example

```
gap> mon := MonoidPresentationFpGroup( q8 );;
gap> fgmon := FreeGroupOfPresentation( mon);
<free group on the generators [ q8_M1, q8_M2, q8_M3, q8_M4]>
gap> genfgmon := GeneratorsOfGroup( fgmon);
[ q8_M1, q8_M2, q8_M3, q8_M4]
gap> gprels := GroupRelatorsOfPresentation( mon );
[ q8_M1^4, q8_M2^4, q8_M1*q8_M2*q8_M1*q8_M4, q8_M1^2*q8_M2^2]
gap> invrels := InverseRelatorsOfPresentation( mon);
[ q8_M1*q8_M3, q8_M2*q8_M4, q8_M3*q8_M1, q8_M4*q8_M2]
gap> hompres := HomomorphismOfPresentation( mon );
[ q8_M1, q8_M2, q8_M3, q8_M4] -> [ f1, f2, f1^-1, f2^-1 ]
```

2.3 Rewriting systems for FpGroups

These functions duplicate the standard Knuth Bendix functions which are available in the GAP library. There are two reasons for this: (1) these functions were first written before the standard functions were available; (2) we require logged versions of the functions, and these are most conveniently extended versions of the non-logged code.

2.3.1 RewritingSystemFpGroup

▷ `RewritingSystemFpGroup(grp)` (attribute)

This function attempts to return a complete rewrite system for the group `G` obtained from the monoid presentation `mon`, with a length-lexicographical ordering on the words in `fgmon`, by applying Knuth-Bendix completion. Such a rewrite system can be obtained for all finite groups. The rewrite rules are (partially) ordered, starting with the inverse relators, followed by the rules which reduce the word length the most.

In our q8 example there are 16 rewrite rules.

Example

```
gap> rws := RewritingSystemFpGroup( q8 );
[ [q8_M1*q8_M3, <identity ...>], [ q8_M2*q8_M4, <identity ...> ],
  [q8_M3*q8_M1, <identity ...>], [ q8_M4*q8_M2, <identity ...> ],
  [q8_M1^2*q8_M4, q8_M2], [q8_M1^2*q8_M2, q8_M4], [ q8_M1^3, q8_M3 ],
  [ q8_M4^2, q8_M1^2], [ q8_M4*q8_M3, q8_M1*q8_M4],
  [ q8_M4*q8_M1, q8_M1*q8_M2], [q8_M3*q8_M4, q8_M1*q8_M2],
  [ q8_M3^2, q8_M1^2], [q8_M3*q8_M2, q8_M1*q8_M4],
  [ q8_M2*q8_M3, q8_M1*q8_M2], [q8_M2^2, q8_M1^2],
  [ q8_M2*q8_M1, q8_M1*q8_M4] ]
```

The functions called by `RewritingSystemFpGroup` are as follows.

2.3.2 OnePassReduceWord

▷ `OnePassReduceWord(word, rules)` (operation)

▷ `ReduceWordKB(word, rules)` (operation)

Assuming that `word` is an element of a free monoid and `rules` is a list of ordered pairs of such words, the function `OnePassReduceWord` searches the list of rules until it finds that the left-hand side of a `rule` is a `subword` of `word`, whereupon it replaces that `subword` with the right-hand side of the matching rule. The search is continued from the next `rule` in `rules`, but using the new `word`. When the end of `rules` is reached, one pass is considered to have been made and the reduced `word` is returned. If no matches are found then the original `word` is returned.

The function `ReduceWordKB` repeatedly applies the function `OnePassReduceWord` until the `word` remaining contains no left-hand side of a `rule` as a `subword`. If `rules` is a complete rewrite system, then the irreducible `word` that is returned is unique, otherwise the order of the rules in `rules` will determine which irreducible word is returned. In the example we see that b^9a^9 reduces to ba, and we shall see later that this is not a normal form.

Example

```
gap> monrels := Concatenation( gprels, invrels );
[ q8_M1^4, q8_M2^4, q8_M1*q8_M2*q8_M1*q8_M4, q8_M1^2*q8_M2^2, q8_M1*q8_M3,
  q8_M2*q8_M4, q8_M3*q8_M1, q8_M4*q8_M2]
gap> id := One( monrels[1] );;
gap> r0 := List( monrels, r -> [ r, id ] );
[ [ q8_M1^4, <identity ...> ], [ q8_M2^4, <identity. ..> ],
  [ q8_M1*q8_M2*q8_M1*q8_M4, <identity ...> ],
  [ q8_M1^2*q8_M2^2, <identity. ..>], [ q8_M1*q8_M3, <identity ...> ],
  [ q8_M2*q8_M4, <identity ...> ], [ q8_M3*q8_M1, <identity. ..>],
  [ q8_M4*q8_M2, <identity ...> ] ]
gap> ap := genfgmon[1];; bp := genfgmon[2];;   ## p for plus
gap> am := genfgmon[3];; bm := genfgmon[4];;   ## m for minus
gap> w0 := bp^9 * ap^9;
q8_M2^9*q8_M1^9
gap> w1 := OnePassReduceWord( w0, r0 );
q8_M2^5*q8_M1^5
gap> w2 := ReduceWordKB( w0, r0 );
q8_M2*q8_M1
```

2.3.3 OnePassKB

▷ `OnePassKB(`*rules*`)` (operation)

▷ `RewriteReduce(`*rules*`)` (operation)

▷ `KnuthBendix(`*rules*`)` (operation)

▷ `ShorterRule(`*rule1*`,` *rule2*`)` (operation)

The function `OnePassKB` implements the main loop of the Knuth-Bendix completion algorithm. Rules are compared with each other; all critical pairs are calculated; and the irreducible critical pairs are orientated with respect to the length-lexicographical ordering and added to the rewrite system.

The function `RewriteReduce` will remove unnecessary rules from a rewrite system. A rule is deemed to be unnecessary if it is implied by the other rules, i.e. if both sides can be reduced to the same thing by the remaining rules.

The function `KnuthBendix` implements the Knuth-Bendix algorithm, attempting to complete a rewrite system with respect to a length-lexicographic ordering. It calls first `OnePassKB`, which adds rules, and then (for efficiency) `RewriteReduce` which removes any unnecessary ones. This procedure is repeated until `OnePassKB` adds no more rules. It will not always terminate, but for many examples (all finite groups) it will be successful. The rewrite system returned is complete, that is: it will rewrite any given word in the free monoid to a unique irreducible; there is one irreducible for each element of the quotient monoid; and any two elements of the free monoid which are in the same class will rewrite to the same irreducible.

The function `ShorterRule` gives an ordering on rules. Rules (g_lg_2,id) that identify two generators (or one generator with the inverse of another) come first in the ordering. Otherwise one precedes another if it reduces the length of a word by a greater amount.

One pass of this procedure for our q8 example adds 13 relators to the original 8, and these 21 are then reduced to 9. A second pass and reduction gives a list of 16 rules which forms a complete rewrite system for the group. Now b^9a^9 reduces to ab^{-1}.

Example

```
gap> r1 := OnePassKB( r0 );
[ [ q8_M1^4, <identity ...> ], [ q8_M2^4, <identity ...> ],
  [ q8_M1*q8_M2*q8_M1*q8_M4, <identity ...> ],
  [ q8_M1^2*q8_M2^2, <identity. ..> ], [ q8_M1*q8_M3, <identity ...> ],
  [ q8_M2*q8_M4, <identity ...> ], [ q8_M3*q8_M1, <identity ...> ],
  [ q8_M4*q8_M2, <identity ...> ], [ q8_M2*q8_M1*q8_M4, q8_M1^3],
  [ q8_M1*q8_M2^2, q8_M1^3 ], [ q8_M2^2, q8_M1^2 ], [q8_M1^3, q8_M3 ],
  [ q8_M2^3, q8_M4 ], [ q8_M1*q8_M2*q8_M1, q8_M2],
  [ q8_M2^3, q8_M1^2*q8_M2], [ q8_M2^2, q8_M1^2 ], [q8_M1^2*q8_M2, q8_M4 ],
  [ q8_M1^3, q8_M3 ], [ q8_M2*q8_M1*q8_M4, q8_M3 ], [q8_M1*q8_M2^2, q8_M3 ],
  [ q8_M2^3, q8_M4 ] ]
gap> r1 := RewriteReduce( r1 );
[ [ q8_M1*q8_M3, <identity ...> ], [ q8_M2^2, q8_M1^2 ],
  [ q8_M2*q8_M4, <identity ...> ], [ q8_M3*q8_M1, <identity ...> ],
  [ q8_M4*q8_M2, <identity ...> ], [ q8_M1^3, q8_M3 ],
  [ q8_M1^2*q8_M2, q8_M4 ], [ q8_M1*q8_M2*q8_M1, q8_M2 ],
  [ q8_M2*q8_M1*q8_M4, q8_M3 ] ]
gap> r2 := KnuthBendix( r1 );
[ [ q8_M1*q8_M3, <identity ...> ], [ q8_M2*q8_M1, q8_M1*q8_M4 ],
  [ q8_M2^2, q8_M1^2], [ q8_M2*q8_M3, q8_M1*q8_M2 ],
  [ q8_M2*q8_M4, <identity ...> ], [ q8_M3*q8_M1, <identity ...> ],
  [ q8_M3*q8_M2, q8_M1*q8_M4 ], [ q8_M3^2, q8_M1^2 ],
  [ q8_M3*q8_M4, q8_M1*q8_M2 ], [ q8_M4*q8_M1, q8_M1*q8_M2 ],
  [ q8_M4*q8_M2, <identity ...> ], [ q8_M4*q8_M3, q8_M1*q8_M4 ],
  [ q8_M4^2, q8_M1^2], [ q8_M1^3, q8_M3 ], [q8_M1^2*q8_M2, q8_M4 ],
  [ q8_M1^2*q8_M4, q8_M2 ] ]
gap> w2 := ReduceWordKB( w0, r2 );
q8_M1*q8_M4
```

2.4 Enumerating elements

2.4.1 ElementsOfMonoidPresentation

▷ `ElementsOfMonoidPresentation(`*mon*`)` (attribute)

The function `ElementsOfMonoidPresentation` returns a list of normal forms for the elements of the group given by the monoid presentation `mon`. The normal forms are the least elements in each equivalence class (with respect to length-lex order). When `rules` is a complete rewrite system for `G` the list returned is a set of normal forms for the group elements.

Example

```
gap> elq8 := Elements( q8 );
[ <identity ...>, f1, f1^3, f2, f1^2*f2, f1^2, f1*f2, f1^3*f2 ]
gap> elmonq8 := ElementsOfMonoidPresentation( monq8 );
[ <identity. ..>, q8_M1, q8_M2, q8_M3, q8_M4, q8_M1^2, q8_M1*q8_M2,
  q8_M1*q8_M4 ]
```

Chapter 3

Logged Rewriting Systems

A *logged rewrite system* is associated with a group presentation. Each *logged rewrite rule* contains, in addition to the standard rewrite rule, a record or *log component* which expresses the rule in terms of the original relators of the group. We represent such a rule by a triple `[ u, [L1,L2,..,Lk], v]`, where `[u,v]` is a rewrite rule and $L_i = [n_i, w_i]$ where n_i is a group relator and w_i is a word. These three components obey the identity $u = n_1^{w_1} \ldots n_k^{w_k} v$.

Rules of the form g^+g^- are relevant to the monoid presentation, but not to the group presentation, so are given an empty logged component.

3.1 Logged Knuth-Bendix Completion

The functions in this section are the logged versions of those in the previous chapter.

3.1.1 LoggedOnePassKB

▷ `LoggedOnePassKB(`*`loggedrules`*`)` (operation)

Given a logged rewrite system, this function finds all the rules that would be added to complete the rewrite system in `OnePassKB`, and also the logs which relate the new rules to the originals. The result of applying this function to `loggedrules` is to add new logged rules to the system without changing the monoid it defines.

In the example, we first convert the presentation for q8 into an initial set of logged rules, and then apply one pass of Knuth-Bendix.

Example

```
gap> l0 := ListWithIdenticalEntries( 8, 0 );;
gap> for j in [1..8] do
>        r := r0[j];
>        if ( j<5 ) then
>           l0[j] := [ r[1], [ [j,id] ], r[2] ];
>        else
>           l0[j] := [ r[1], [ ], r[2] ];
>        fi;
>    od;
gap> l0;
[ [ q8_M1^4, [ [ 1, <identity ...>] ], <identity. ..> ],
    [ q8_M2^4, [ [ 2, <identity ...>] ], <identity ...> ],
    [ q8_M1*q8_M2*q8_M1*q8_M4, [ [ 3, <identity ...> ] ], <identity ...> ],
    [ q8_M1^2*q8_M2^2, [ [ 4, <identity ...> ] ], <identity ...> ],
    [ q8_M1*q8_M3, [ ], <identity ...> ], [ q8_M2*q8_M4, [ ], <identity ...> ],
    [ q8_M3*q8_M1, [ ], <identity ...> ], [ q8_M4*q8_M2, [ ], <identity ...> ] ]
gap> l1 := LoggedOnePassKB( l0 );;
gap> Length( l1 );
21
```

3.1.2 LoggedKnuthBendix

▷ LoggedKnuthBendix(*loggedrules*) (operation)
▷ LoggedRewriteReduce(*loggedrules*) (operation)

The function LoggedRewriteReduce removes unnecessary rules from a logged rewrite system. It works on the same principle as RewriteReduce.

The function LoggedKnuthBendix repeatedly applies LoggedOnePassKB and LoggedRewriteReduce until no new rules are added and no unnecessary ones are included. The output is a reduced complete logged rewrite system.

Example

```
gap> l1 := LoggedRewriteReduce( l1 );
[ [ q8_M1*q8_M3, [ ], <identity ...> ],
  [ q8_M2^2, [ [ -4, <identity ...> ], [ 2, q8_M1^-2 ] ], q8_M1^2 ],
  [ q8_M2*q8_M4, [ ], <identity ...> ], [ q8_M3*q8_M1, [ ], <identity ...> ],
      [ q8_M4*q8_M2, [ ], <identity ...> ],
  [ q8_M1^3, [ [ 1, <identity. ..> ] ], q8_M3 ],
  [ q8_M1^2*q8_M2, [ [ 4, <identity. ..> ] ], q8_M4 ],
  [ q8_M1*q8_M2*q8_M1, [ [ 3, <identity ...> ] ], q8_M2 ],
  [ q8_M2*q8_M1*q8_M4, [ [ 3, q8_M3^-1 ] ], q8_M3] ]
gap> l2 := LoggedKnuthBendix( l1 );
[ [ q8_M1*q8_M3, [ ], <identity ...> ],
  [ q8_M2*q8_M1, [ [ 3, q8_M3^-1 ], [-1, <identity. ..> ], [ 4, q8_M1^-1 ] ],
      q8_M1*q8_M4 ],
  [ q8_M2^2, [ [ -4, <identity ...> ], [2, q8_M1^-2] ], q8_M1^2 ],
  [ q8_M2*q8_M3, [ [ -3, <identity ...> ] ], q8_M1*q8_M2 ],
  [ q8_M2*q8_M4, [ ], <identity ...> ], [ q8_M3*q8_M1, [ ], <identity ...> ],
  [ q8_M3*q8_M2, [ [ -1, <identity ...>], [4, q8_M1^-1 ] ], q8_M1*q8_M4 ],
  [ q8_M3^2, [ [ -1, <identity ...>] ], q8_M1^2 ],
  [ q8_M3*q8_M4, [ [ -1, <identity ...>], [ -2, q8_M1^-2],
        [ 4, <identity ...> ], [ 3, q8_M3^-1*q8_M2^-1 ],
        [ -3, <identity. ..> ] ], q8_M1*q8_M2 ],
  [ q8_M4*q8_M1, [ [ -4, <identity ...> ], [ 3, q8_M1^-1 ] ], q8_M1*q8_M2 ],
  [ q8_M4*q8_M2, [ ], <identity ...> ],
  [ q8_M4*q8_M3, [ [ -3, q8_M3^-1*q8_M4^-1] ], q8_M1*q8_M4 ],
  [ q8_M4^2, [ [ -4, <identity. ..> ] ], q8_M1^2 ],
  [ q8_M1^3, [ [ 1, <identity ...> ] ], q8_M3 ],
  [ q8_M1^2*q8_M2, [ [4, <identity. ..> ] ], q8_M4 ],
  [ q8_M1^2*q8_M4, [ [ -4, q8_M1^-2], [ 1, <identity ...> ] ], q8_M2 ] ]
```

3.2 Logged reduction of a word

3.2.1 LoggedReduceWordKB

▷ LoggedReduceWordKB(*word, loggedrules*) (operation)
▷ LoggedOnePassReduceWord(*word, loggedrules*) (operation)
▷ ShorterLoggedRule(*logrule1, logrule2*) (operation)

Given a word and a logged rewrite system, the function LoggedOnePassReduceWord makes one reduction of the word (as in OnePassReduceWord) and records this, using the log part of the rule used and the position in the original word of the replaced part.

The function LoggedReduceWordKB repeatedly applies OnePassLoggedReduceWord until the word can no longer be reduced. Each step of the reduction is logged, showing how the original word can be expressed in terms of the original relators and the irreducible word. When loggedrules is complete the reduced word is a unique normal form for that group element. The log of the reduction depends on the order in which the rules are applied.

The function ShorterLoggedrule decides whether one logged rule is better than another, using the same criteria as ShorterRule. In the example we perform logged reductions of w0 corresponding to the ordinary reductions performed in the previous chapter.

Example
```
gap> w0;
q8_M2^9*q8_M1^9
gap> lw1 := LoggedOnePassReduceWord( w0, l0 );
[ [ [ 1, q8_M2^-9 ], [ 2, <identity ...> ] ], q8_M2^5*q8_M1^5 ]
gap> lw2 := LoggedReduceWordKB( w0, l0 );
[ [ [ 1, q8_M2^-9 ], [ 2, <identity ...> ], [ 1, q8_M2^-5 ],
      [ 2, <identity ...> ] ], q8_M2*q8_M1 ]
gap> lw2 := LoggedReduceWordKB( w0, l2 );
[ [ [ 3, q8_M3^-1*q8_M2^-8 ], [ -1, q8_M2^-8 ], [ 4, q8_M1^-1*q8_M2^-8 ],
      [ -4, <identity ...> ], [ 2, q8_M1^-2 ],
      [ -4, q8_M1^-1*q8_M2^-6*q8_M1^-2 ], [ 3, q8_M1^-2*q8_M2^-6*q8_M1^-2 ],
      [ 1, q8_M2^-1*q8_M1^-2*q8_M2^-6*q8_M1^-2 ], [ 4, <identity ...> ],
      [ 3, q8_M3^-1*q8_M2^-4*q8_M4^-1 ], [ -1, q8_M2^-4*q8_M4^-1 ],
      [ 4, q8_M1^-1*q8_M2^-4*q8_M4^-1 ], [ -4, q8_M4^-1 ],
      [ 2, q8_M1^-2*q8_M4^-1 ],
      [ -3, q8_M1^-1*q8_M4^-1*q8_M1^-1*q8_M2^-2*q8_M1^-2*q8_M4^-1 ],
      [ -4, <identity ...> ], [ 3, q8_M1^-1 ],
      [ 1, q8_M2^-1*q8_M1^-2*q8_M4^-1*q8_M1^-1*q8_M2^-1*(q8_M2^-1*q8_M1^-1)^2
         ], [ 4, q8_M4^-1*q8_M1^-1*q8_M2^-1*(q8_M2^-1*q8_M1^-1)^2 ],
      [ 3, q8_M3^-1*q8_M1^-1 ], [ -1, q8_M1^-1 ], [ 4, q8_M1^-2 ],
      [ -4, q8_M4^-1*q8_M1^-2 ], [ 2, q8_M1^-2*q8_M4^-1*q8_M1^-2 ],
      [ -4, q8_M1^-2 ], [ 3, q8_M1^-3 ], [ -4, q8_M1^-2*q8_M2^-1*q8_M1^-3 ],
      [ 1, <identity ...> ], [ 3, q8_M3^-2 ], [ -1, q8_M3^-1 ],
      [ 4, q8_M1^-1*q8_M3^-1 ], [ -4, <identity ...> ], [ 3, q8_M1^-1 ],
      [ 3, q8_M3^-1*q8_M1^-1 ], [ -1, q8_M1^-1 ], [ 4, q8_M1^-2 ],
      [ -4, q8_M1^-2 ], [ 3, q8_M1^-3 ], [ 1, <identity ...> ],
      [ -1, <identity ...> ], [ 4, q8_M1^-1 ] ], q8_M1*q8_M4 ]
```

3.2.2 LoggedRewritingSystemFpGroup

▷ LoggedRewritingSystemFpGroup(*loggedrules*) (attribute)

Given a group presentation, the function LoggedRewritingSystemFpGroup determines a logged rewrite system based on the relators. The initial logged rewrite system associated with a group presentation consists of two types of rule. These are logged versions of the two types of rule in the monoid presentation. For each relator rel of the group there is a logged rule [rel, [[1, rel]], id]. For each inverse relator there is a logged rule [gen*inv, [], id]. It then attempts a completion of the logged rewrite system. The rules in the final system are partially ordered by the function ShorterLoggedRule.

Example

```
gap> LoggedRewritingSystemFpGroup( q8 );
[ [ q8_M4*q8_M2, [ ], <identity ...> ], [ q8_M3*q8_M1, [ ], <identity ...> ],
    [ q8_M2*q8_M4, [ ], <identity ...> ],
  [ q8_M1*q8_M3, [ ], <identity ...> ],
  [ q8_M1^2*q8_M4, [ [ -8, q8_M1^-2 ], [ 5, <identity ...> ] ], q8_M2 ],
  [ q8_M1^2*q8_M2, [ [ 8, <identity ...> ] ], q8_M4 ],
  [ q8_M1^3, [ [ 5, <identity ...> ] ], q8_M3 ],
  [ q8_M4^2, [ [ -8, <identity ...> ] ], q8_M1^2 ],
  [ q8_M4*q8_M3, [ [ -7, q8_M3^-1*q8_M4^-1 ] ], q8_M1*q8_M4 ],
  [ q8_M4*q8_M1, [ [ -8, <identity. ..> ], [ 7, q8_M1^-1 ] ], q8_M1*q8_M2 ],
  [ q8_M3*q8_M4,
      [ [ -5, <identity ...>], [ -6, q8_M1^-2 ], [ 8, <identity ...> ],
          [ 7, q8_M3^-1*q8_M2^-1 ], [ -7, <identity. ..> ] ], q8_M1*q8_M2 ],
  [ q8_M3^2, [ [ -5, <identity ...>] ], q8_M1^2 ],
  [ q8_M3*q8_M2, [ [ -5, <identity. ..> ], [ 8, q8_M1^-1 ] ], q8_M1*q8_M4 ],
  [ q8_M2*q8_M3, [ [ -7, <identity ...> ] ], q8_M1*q8_M2 ],
  [ q8_M2^2, [ [ -a, <identity ...> ], [ 6, q8_M1^-2 ] ], q8_M1^2 ],
  [ q8_M2*q8_M1, [ [ 7, q8_M3^-1 ], [ -5, <identity ...> ], [ a, q8_M1^-1 ] ],
      q8_M1*q8_M4 ] ]
```

Chapter 4

Monoid Polynomials

This chapter describes functions to compute with elements of a free noncommutative algebra. The elements of the algebra are sums of rational multiples of words in a free monoid. These are called *monoid polynomials*, and are stored as lists of pairs `[coefficient, word]`.

4.1 Construction of monoid polynomials

4.1.1 MonoidPolyFromCoeffsWords

▷ `MonoidPolyFromCoeffsWords(`*`coeffs, words`*`)` (operation)
▷ `MonoidPoly(`*`terms`*`)` (operation)
▷ `ZeroMonoidPoly(`*`F`*`)` (operation)

There are two ways to input a monoid polynomial: by listing the coefficients and then the words; or by listing the terms as a list of pairs `[coefficient, word]`. If a word occurs more than once in the input list, the coefficients will be added so that the terms of the monoid polynomial recorded do not contain any duplicates. The zero monoid polynomial is the polynomial with no terms.

Example

```
gap> rels := RelatorsOfFpGroup( q8 );
[ f1^4, f2^4, f1*f2*f1*f2^-1, f1^2*f2^2 ]
gap> freeq8 := FreeGroupOfFpGroup( q8 );;
gap> gens := GeneratorsOfGroup( freeq8 );;
gap> famfree := ElementsFamily( FamilyObj( freeq8 ) );;
gap> famfree!.monoidPolyFam := MonoidPolyFam;;
gap> cg := [6,7];;
gap> pg := MonoidPolyFromCoeffsWords( cg, gens );;
gap> Print( pg, "\n" );
7*f2 + 6*f1
gap> cr := [3,4,-5,-2];;
gap> pr := MonoidPolyFromCoeffsWords( cr, rels );;
gap> Print( pr, "\n" );
4*f2^4 - 5*f1*f2*f1*f2^-1 - 2*f1^2*f2^2 + 3*f1^4
gap> Print( ZeroMonoidPoly( freeq8 ), "\n" );
zero monpoly
```

4.2 Components of a polynomial

4.2.1 Terms

▷ `Terms(poly)` (attribute)
▷ `Coeffs(poly)` (attribute)
▷ `Words(poly)` (attribute)
▷ `LeadTerm(poly)` (attribute)
▷ `LeadCoeffMonoidPoly(poly)` (attribute)

The function `Terms` returns the terms of a polynomial as a list of pairs of the form `[word, coefficient]`. The function `Coeffs` returns the coefficients of a polynomial as a list, and the function `Words` returns the words of a polynomial as a list. The function `LeadTerm` returns the term of the polynomial whose word component is the largest with respect to the length-lexicographical ordering. The function `LeadCoeffMonoidPoly` returns the coefficient of the leading term of a polynomial.

Example

```
gap> Coeffs( pr );
[ 4, -5, -2, 3 ]
gap> Terms( pr );
[ [ 4, f2^4 ], [ -5, f1*f2*f1*f2^-1 ], [ -2, f1^2*f2^2 ], [ 3, f1^4 ] ]
gap> Words( pr );
[ f2^4, f1*f2*f1*f2^-1, f1^2*f2^2, f1^4 ]
gap> LeadTerm( pr );
[ 4, f2^4]
gap> LeadCoeffMonoidPoly( pr );
4
```

4.2.2 Monic

▷ `Monic(poly)` (operation)

A monoid polynomial is called *monic* if the coefficient of its leading polynomial is one. The function `Monic` converts a polynomial into a monic polynomial by dividing all the coefficients by the leading coefficient.

Example

```
gap> mpr := Monic( pr );;
gap> Print( mpr, "\n" );
f2^4 - 5/4*f1*f2*f1*f2^-1 - 1/2*f1^2*f2^2 + 3/4*f1^4
```

4.2.3 AddTermMonoidPoly

▷ `AddTermMonoidPoly(poly, coeff, word)` (operation)

The function `AddTermMonoidPoly` adds a new term, given by its coeffiecient and word, to an existing polynomial.

Example

```
gap> w := gens[1]^gens[2];
f2^-1*f1*f2
gap> cw := 3/4;;
gap> wpg:= AddTermMonoidPoly( pg, cw, w);;
gap> Print( wpg, "\n" );
3/4*f2^-1*f1*f2 + 7*f2 + 6*f1
```

4.3 Monoid Polynomial Operations

Tests for equality and arithmetic operations are performed in the usual way.

The operation `poly1 = poly2` returns `true` if the monoid polynomials have the same terms, and `false` otherwise. Multiplication of a monoid polynomial (on the left or right) by a coefficient; the addition or subtraction of two monoid polynomials; multiplication (on the right) of a monoid polynomial by a word; and multiplication of two monoid polynomials; are all implemented.

Example

```
gap> [ pg = pg, pg = pr ];
[ true, false ]
gap> prcw := pr*cw;;
gap> Print( prcw, "\n" );
3*f2^4 - 15/4*f1*f2*f1*f2^-1 - 3/2*f1^2*f2^2 + 9/4*f1^4
gap> cwpr := cw*pr;;
gap> Print( cwpr, "\n" );
3*f2^4 - 15/4*f1*f2*f1*f2^-1 - 3/2*f1^2*f2^2 + 9/4*f1^4
gap> [ pr = prcw, prcw = cwpr ];
[ false, true ]
gap> Print( pg + pr, "\n" );
4*f2^4 - 5*f1*f2*f1*f2^-1 - 2*f1^2*f2^2 + 3*f1^4 + 7*f2 + 6*f1
gap> Print( pg - pr, "\n" );
- 4*f2^4 + 5*f1*f2*f1*f2^-1 + 2*f1^2*f2^2 - 3*f1^4 + 7*f2 + 6*f1
gap> Print( pg * w, "\n" );
6*f1*f2^-1*f1*f2 + 7*f1*f2
gap> Print( pg * pr, "\n" );
28*f2^5 - 35*(f2*f1)^2*f2^-1 - 14*f2*f1^2*f2^2 + 21*f2*f1^4 + 24*f1*f2^4 -
30*f1^2*f2*f1*f2^-1 - 12*f1^3*f2^2 + 18*f1^5
```

4.3.1 Length

▷ `Length(`*poly*`)` (attribute)

This function returns the number of distinct terms in the monoid polynomial.

Example

```
gap> Length( pr );
4
```

The boolean function poly1 > poly2 returns true if the first polynomial has more terms than the second. If the polynomials are the same length it will compare their leading terms. If the leading word of the first is lengthlexicographically greater than the leading word of the second, or if the words are equal but the coefficient of the first is greater than the coefficient of the second then true is returned. If the leading terms are equal then the next terms are compared in the same way. If all terms are the same then false is returned.

Example

```
gap> [ pr > 3*pr, pr > pg ];
[ false, true ]
```

4.4 Reduction of a Monoid Polynomial

4.4.1 ReduceMonoidPoly

▷ ReduceMonoidPoly(*poly, rules*) (operation)

Recall that the words of a monoid polynomial are elements of a free monoid. Given a rewrite system (set of rules) on the free monoid the words can be reduced. This allows us to simulate calculation in monoid rings where the monoid is given by a complete presentation. This function reduces the words of the polynomial (elements of the free monoid) with respect to the complete rewrite system. The words of the reduced polynomial are normal forms for the elements of the monoid presented by that rewite system. The list of rules r2 is displayed in section 2.3.3.

Example

```
gap> M := genfgmon;;
gap> mp1 := MonoidPolyFromCoeffsWords(
>            [9,-7,5], [M[1]*M[3], M[2]^3, M[4]*M[3]*M[2]] );;
gap> Print( mp1, "\n" );
5*q8_M4*q8_M3*q8_M2 - 7*q8_M2^3 + 9*q8_M1*q8_M3
gap> rmp1 := ReduceMonoidPoly( mp1, r2 );;
gap> Print( rmp1, "\n" );
 - 7*q8_M4 + 5*q8_M1 + 9*<identity ...>
```

Chapter 5

Module Polynomials

In this chapter we consider finitely generated modules over the monoid rings considered previously. We call an element of this module a *module polynomial*, and we describe functions to construct module polynomials and the standard algebraic operations for such polynomials.

A module polynomial `modpoly` is recorded as a list of pairs, `[ gen, monpoly ]`, where `gen` is a module generator (basis element), and `monpoly` is a monoid polynomial. The module polynomial is printed as the formal sum of monoid polynomial multiples of the generators. Note that the monoid polynomials are the coefficients of the module polynomials and appear to the right of the generator, as we choose to work with right modules.

The examples we are aiming for are the identities among the relators of a finitely presented group (see section 5.4).

5.1 Construction of module polynomials

5.1.1 ModulePoly

▷ `ModulePoly(`*gens*`,` *monpolys*`)` (operation)
▷ `ModulePoly(`*args*`)` (operation)
▷ `ZeroModulePoly(`*Fgens*`,` *Fmon*`)` (operation)

The function `ModulePoly` returns a module polynomial. The terms of the polynomial may be input as a list of generators followed by a list of monoid polynomials or as one list of `[generator, monoid polynomial]` pairs.

Assuming that `Fgens` is the free group on the module generators and `Fmon` is the free group on the monoid generators, the function `ZeroModulePoly` returns the zero module polynomial, which has no terms, and is an element of the module.

Example

```
gap> frq8 := FreeRelatorGroup( q8 );;
gap> genfrq8 := GeneratorsOfGroup( frq8 );
[ q8_R1, q8_R2, q8_R3, q8_R4 ]
gap> Print( rmp1, "\n" );
 - 7*q8_M4 + 5*q8_M1 + 9*<identity ...>
gap> mp2 := MonoidPolyFromCoeffsWords( [4,-5], [ M[4], M[1] ] );;
gap> Print( mp2, "\n" );
4*q8_M4 - 5*q8_M1
```

```
gap> s1 := ModulePoly( [ genfrq8[4], genfrq8[1] ], [ rmp1, mp2 ] );
q8_R1*(4*q8_M4 - 5*q8_M1) + q8_R4*( - 7*q8_M4 + 5*q8_M1 + 9*<identity ...>)
gap> s2 := ModulePoly( [ genfrq8[3], genfrq8[2], genfrq8[1] ],
>        [ -1*rmp1, 3*mp2, (rmp1+mp2) ] );
q8_R1*( - 3*q8_M4 + 9*<identity ...>) + q8_R2*(12*q8_M4 - 15*q8_M1) + q8_R3*(
7*q8_M4 - 5*q8_M1 - 9*<identity ...>)
gap> zeromp := ZeroModulePoly( frq8, freeq8 );
zero modpoly
```

5.2 Components of a module polynomial

5.2.1 Terms

▷ Terms(*modpoly*) (attribute)
▷ LeadTerm(*modpoly*) (attribute)
▷ LeadMonoidPoly(*modpoly*) (attribute)
▷ One(*modpoly*) (attribute)
▷ Length(*modpoly*) (attribute)

The function Length counts the number of module generators which occur in modpoly (a generator occurs in a polynomial if it has nonzero coefficient). The function One returns the identity in the free group on the generators.

The function Terms returns the terms of a module polynomial as a list of pairs. In LeadTerm, the generators are ordered, and the term of modpoly with the highest value generator is defined to be the leading term. The monoid polynomial (coefficient) part of the leading term is returned by the function LeadMonoidPoly.

Example

```
gap> [ Length(s1), Length(s2) ];
[ 2, 3 ]
gap> One( s1 );
<identity ...>
gap> Terms( s1 );
[ [ q8_R1, <monpoly> ], [ q8_R4, <monpoly> ] ]
gap> Print( LeadTerm( s1 ), "\n" );
[ q8_R4,  - 7*q8_M4 + 5*q8_M1 + 9*<identity ...> ]
gap> Print( LeadTerm( s2 ), "\n" );
[ q8_R3, 7*q8_M4 - 5*q8_M1 - 9*<identity ...> ]
gap> Print( LeadMonoidPoly( s1 ), "\n" );
 - 7*q8_M4 + 5*q8_M1 + 9*<identity ...>
gap> Print( LeadMonoidPoly( s2 ), "\n" );
7*q8_M4 - 5*q8_M1 - 9*<identity ...>
```

5.3 Module Polynomial Operations

5.3.1 AddTermModulePoly

▷ `AddTermModulePoly(modpoly, gen, monpoly)` (operation)

The function `AddTermModulePoly` adds a term `[gen, monpoly]` to a module polynomial `modpoly`.

Tests for equality and arithmetic operations are performed in the usual way. Module polynomials may be added or subtracted. A module polynomial can also be multiplied on the right by a word or by a scalar. The effect of this is to multiply the monoid polynomial parts of each term by the word or scalar. This is made clearer in the example.

Example
```
gap> mp0 := MonoidPolyFromCoeffsWords( [6], [ M[2] ] );;
gap> Print( mp0, "\n" );
6*q8_M2
gap> s0 := AddTermModulePoly( s1, genfrq8[3], mp0 );
q8_R1*(4*q8_M4 - 5*q8_M1) + q8_R3*(6*q8_M2) + q8_R4*( - 7*q8_M4 + 5*q8_M1 +
9*<identity ...>)
gap> Print( s1 + s2, "\n" );
q8_R1*( q8_M4 - 5*q8_M1 + 9*<identity ...>) + q8_R2*(12*q8_M4 -
15*q8_M1) + q8_R3*(7*q8_M4 - 5*q8_M1 - 9*<identity ...>) + q8_R4*( -
7*q8_M4 + 5*q8_M1 + 9*<identity ...>)
gap> Print( s1 - s0, "\n" );
q8_R3*( - 6*q8_M2)
gap> Print( s1 * 1/2, "\n" );
q8_R1*(2*q8_M4 - 5/2*q8_M1) + q8_R4*( - 7/2*q8_M4 + 5/2*q8_M1 + 9/
2*<identity ...>)
gap> Print( s1 * M[1], "\n" );
q8_R1*(4*q8_M4*q8_M1 - 5*q8_M1^2) + q8_R4*( - 7*q8_M4*q8_M1 + 5*q8_M1^2 +
9*q8_M1)
```

5.4 Identities among relators

5.4.1 IdentityYSequences

▷ `IdentityYSequences(grp)` (attribute)
▷ `IdentityModulePolynomials(grp)` (operation)
▷ `IdentitiesAmongRelators(grp)` (attribute)

The identities among the relators for a finitely presented group are constructed as logged module polynomials. The procedure, described in [HW03] and based on work in [BRS99], is to construct a full set of Y-sequences for the group; convert these into module polynomials (eliminating empty sequences); and then apply simplification rules (including the primary identity property) to eliminate obvious duplicates and conjugates.

It is *not* guaranteed that a minimal set of identities is obtained. For q8 a set of seven identities is obtained, whereas a minimal set contains only six. See Example 5.1 of [HW03] for further details.

Example

```
gap> yseqs := IdentityYSequences( q8 );;
gap> Length( yseqs );
32
gap> polys := IdentityModulePolys( q8 );;
gap> Length( polys );
22
gap> idsq8 := IdentitiesAmongRelators( q8 );;
gap> Length( idsq8 );
2
gap> Length( idsq8[1] );
7
gap> Display( idsq8[1] );
[ ( q8_Y3*( q8_M1*q8_M4), q8_R1*( q8_M1 - <identity ...>) ),
  ( q8_Y10*( -q8_M1*q8_M4), q8_R2*( q8_M2 - <identity ...>) ),
  ( q8_Y17*( <identity ...>), q8_R1*( -q8_M3 - q8_M2) + q8_R3*( q8_M1^2 + q8_M\
3 + q8_M1 + <identity ...>) ),
  ( q8_Y31*( q8_M1*q8_M4), q8_R3*( q8_M3 - q8_M2) + q8_R4*( q8_M1 - <identity \
...>) ),
  ( q8_Y32*( -q8_M1*q8_M4), q8_R2*( -q8_M1^2) + q8_R3*( -q8_M3 - <identity ...\
>) + q8_R4*( q8_M2 + <identity ...>) ),
  ( q8_Y12*( q8_M1*q8_M4), q8_R1*( -q8_M2) + q8_R3*( q8_M1*q8_M2 + q8_M4) + q8\
_R4*( q8_M2 - <identity ...>) ),
  ( q8_Y16*( -<identity ...>), q8_R1*( -<identity ...>) + q8_R2*( -q8_M1) + q8\
_R4*( q8_M3 + q8_M1) ) ]
  ]
```

5.4.2 RootIdentities

▷ `RootIdentities(`*grp*`)` (attribute)

The *root identities* are identities of the form $r^w r^{-1}$ where $r = w^n$ is a relator and $n > 1$.

For q8 only two of the four relators are proper powers, $q = a^4$ and $r = b^4$, so the root identities are $q^a q^{-1}$ and $r^b r^{-1}$.

Example

```
gap> RootIdentities( q8 );
[ ( q8_Y3*( q8_M1*q8_M4), q8_R1*( q8_M1 - <identity ...>) ),
  ( q8_Y10*( -q8_M1*q8_M4), q8_R2*( q8_M2 - <identity ...>) ) ]
gap> RootIdentities(s3);
[ ( s3_Y4*( s3_M2*s3_M1), s3_R1*( s3_M1 - <identity ...>) ),
  ( s3_Y8*( s3_M2*s3_M1), s3_R2*( s3_M2 - <identity ...>) ),
  ( s3_Y7*( s3_M2*s3_M1), s3_R3*( s3_M2 - s3_M1) ) ]
```

References

[BH82] R. Brown and J. Huebschumann. Identities among relations. In R. Brown and T. L. Thickstun, editors, *Low-Dimensional Topology*, volume 46 of *London Math. Soc. Lecture Note Series*, page 153–202. Cambridge University Press, 1982. 4

[BRS99] R. Brown and A. Razak Salleh. On the computation of identities among relations and of free crossed resolutions of groups. *London Math. Soc. J. Comput. Math.*, 2:28–61, 1999. 4, 21

[Hey99] A. Heyworth. *Applications of Rewriting Systems and Groebner Bases to Computing Kan Extensions and Identities Among Relations*. PhD thesis, University of Wales, Bangor, 1999. `http://www.maths.bangor.ac.uk/research/ftp/theses/heyworth.ps.gz`. 4

[HW03] A. Heyworth and C. D. Wensley. Logged rewriting and identities among relators. In C. M. Campbell, E. F. Robertson, and G. C. Smith, editors, *Groups St Andrews 2001 in Oxford*, volume 304 of *London Math. Soc. Lecture Note Series*, page 256–276. Cambridge University Press, 2003. 4, 21

Index