# idrel

## Identities among relations

2.48

27 August 2024

**Anne Heyworth**

**Chris Wensley**

**Chris Wensley**
Email: cdwensley.maths@btinternet.com
Homepage: https://github.com/cdwensley

# Abstract

IdRel is a GAP package 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 last author at cdwensley.maths@btinternet.com or submit an issue at the GitHub repository https://github.com/gap-packages/idrel/issues/.

# Copyright



# Acknowledgements

This documentation was prepared using the GAPDoc [LN17] and AutoDoc [GH17] packages.

The procedure used to produce new releases uses the package GitHubPagesForGAP [Hor14] and the package ReleaseTools.

# Contents


**1 Introduction** **4**
1.1 An illustrative example . . . . . 5

**2 Rewriting Systems** **7**
2.1 Monoid Presentations of FpGroups . . . . . 7
2.2 Rewriting systems for FpGroups . . . . . 9
2.3 Enumerating elements . . . . . 12

**3 Logged Rewriting Systems** **14**
3.1 Logged Knuth–Bendix Completion . . . . . 14
3.2 Logged reduction of a word . . . . . 17

**4 Monoid Polynomials** **19**
4.1 Construction of monoid polynomials . . . . . 19
4.2 Components of a polynomial . . . . . 20
4.3 Monoid Polynomial Operations . . . . . 21
4.4 Reduction of a Monoid Polynomial . . . . . 22

**5 Module Polynomials** **23**
5.1 Construction of module polynomials . . . . . 23
5.2 Components of a module polynomial . . . . . 24
5.3 Module Polynomial Operations . . . . . 25

**6 Identities Among Relators** **27**
6.1 Constructing identities . . . . . 27
6.2 Identities for $S_3$ . . . . . 30
6.3 Reducing identities . . . . . 32
6.4 The original approach . . . . . 36
6.5 Partial lists of elements . . . . . 38

**References** **39**

**Index** **40**

# 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 package may be obtained as a compressed tar file `idrel-version.number.tar.gz` by ftp from one of the following sites:

- the IdRel GitHub site: `https://github.com/gap-packages.github.io/idrel/`.
- any GAP archive, e.g. `https://www.gap-system.org/Packages/packages.html`;

The package also has a GitHub repository at: `https://github.com/gap-packages/idrel/` where issues can be raised.

## 1.1 An illustrative example

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 $(u_1^{-1}r_1u_1)\ldots(u_k^{-1}r_ku_k)$ 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 = F/[f^3,g^2,(fg)^2]`. In the listings below, `s3_Ri` is the `i`-th relator for `s3`, and `f1,f2` are the generators $f,g$ of $F$.

Example

```
gap> F := FreeGroup( 2 );;
gap> f := F.1;;  g := F.2;;
gap> rels3 := [ f^3, g^2, f*g*f*g ];
[ f1^3, f2^2, (f1*f2)^2 ]
gap> s3 := F/rels3;
<fp group on the generators [ f1, f2 ]>
gap> SetName( s3, "s3" );;
gap> IdentitiesAmongRelators( s3 );
[ [ [ -1, <identity ...> ], [ 1, s3_M1 ] ],
  [ [ -2, <identity ...> ], [ 2, s3_M2 ] ],
  [ [ -3, <identity ...> ], [ 3, s3_M1*s3_M2 ] ],
  [ [ 1, <identity ...> ], [ -3, s3_M1 ], [ 2, s3_M3*s3_M4 ], [ 1, s3_M4 ],
      [ -3, <identity ...> ], [ 2, s3_M3*s3_M4*s3_M3 ], [ 2, s3_M3 ],
      [ -3, s3_M3 ] ],
  [ [ 1, <identity ...> ], [ -3, s3_M2 ], [ 2, s3_M3*s3_M4*s3_M3*s3_M2 ],
      [ 2, s3_M3*s3_M2 ], [ 1, s3_M2 ], [ -3, <identity ...> ], [ 2, s3_M3 ],
      [ -3, s3_M3 ] ] ]
```

If we write $\rho = f^3$, $\sigma = g^2$, $\tau = (fg)^2$ then the first identity becomes $\rho^{-1}\rho^f$. Similarly, the second and third identities are the root identities $\sigma^{-1}\sigma^g$ and $\tau^{-1}\tau^{fg}$. The fourth identity, 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:

$$\rho\ (\tau^{-1})^f\ \sigma^{f^{-1}g^{-1}}\ \rho^{g^{-1}}\ (\tau^{-1})\ \sigma^{f^{-1}g^{-1}f^{-1}}\ \sigma^{f^{-1}}\ (\tau^{-1})^{f^{-1}}.$$

The fifth identity is equivalent to the fourth, as we shall show in section 6.2.

In order to form the *module of identities* for `s3` the identities are transformed into module polynomials. The first is $y_1 = \rho(f-1)$. The second and third are $y_2 = \sigma(g-1)$ and $y_3 = \tau(fg-1)$, while the fourth is $\rho(1+g^{-1}f)+\sigma(1+f^{-1}g^{-1}+f^{-1}g^{-1}f)-\tau(1++f+f^2)$. Note that, in

the fourth polynomial, the conjugators are converted to their normal forms in `s3`, namely $f^2 = f^{-1}$, $f^{-1}g^{-1}f = fg$, $g^{-1}f = gf$ and $fg^{-1}f = g$. Generators for this module are returned by the operation `IdentityYSequences`.

Example

```
gap> idyseq3 := IdentityYSequences( s3 );
[ ( s3_Y1*( -s3_M1), s3_R1*( s3_M1 - <identity ...>) ),
  ( s3_Y2*( <identity ...>), s3_R2*( s3_M2 - <identity ...>) ),
  ( s3_Y3*( s3_M1), s3_R3*( s3_M2 - s3_M1) ),
  ( s3_Y9*( -<identity ...>), s3_R1*( -s3_M2*s3_M1 - s3_M1) + s3_R2*( -s3_M1*s\
3_M2 - s3_M1 - <identity ...>) + s3_R3*( s3_M3 + s3_M2 + <identity ...>) ) ]
```

Further examples are given in chapter 6.

An extensive revision has been released with version 2.44. This has concentrated in the area of log sequences, adding many of the functions described in sections 6.2 and 6.3.

Work on revising Y-sequences is needed, but must wait for later versions.

# Chapter 2

# Rewriting Systems

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

## 2.1 Monoid Presentations of FpGroups

### 2.1.1 FreeRelatorGroup

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

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

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> relq8 := [ f^4, g^4, f*g*f*g^-1, f^2*g^2 ];;
gap> q8 := F/relq8;;
gap> SetName( q8, "q8" );;
gap> q8R := FreeRelatorGroup( q8 );
q8_R
gap> genq8R := GeneratorsOfGroup( q8R );
[ q8_R1, q8_R2, q8_R3, q8_R4]
gap> homq8R := FreeRelatorHomomorphism( q8 );
[ q8_R1, q8_R2, q8_R3, q8_R4 ] -> [ f1^4, f2^4, f1*f2*f1*f2^-1, f1^2*f2^2 ]
```

### 2.1.2 MonoidPresentationFpGroup

▷ `MonoidPresentationFpGroup(`*grp*`)` (attribute)
▷ `ArrangementOfMonoidGenerators(`*grp*`)` (attribute)
▷ `MonoidPresentationLabels(`*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 $gG, Gg$ specifying the inverses. The function MonoidPresentationFpGroup returns the monoid presentation derived in this way from an fp-presentation.

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 $G$'s. 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.

The attribute ArrangementOfMonoidGenerators will be discussed before the second example in the next section.

In the example below, the four monoid generators $a, b, A, B$ are named q8_M1, q8_M2, q8_M3, q8_M4 respectively.

Example

```
gap> mq8 := MonoidPresentationFpGroup( q8 );
monoid presentation with group relators
[ q8_M1^4, q8_M2^4, q8_M1*q8_M2*q8_M1*q8_M4, q8_M1^2*q8_M2^2 ]
gap> fmq8 := FreeGroupOfPresentation( mq8 );
<free group on the generators [ q8_M1, q8_M2, q8_M3, q8_M4 ]>
gap> genfmq8 := GeneratorsOfGroup( fmq8 );;
gap> gprels := GroupRelatorsOfPresentation( mq8 );
[ q8_M1^4, q8_M2^4, q8_M1*q8_M2*q8_M1*q8_M4, q8_M1^2*q8_M2^2 ]
gap> invrels := InverseRelatorsOfPresentation( mq8 );
[ q8_M1*q8_M3, q8_M2*q8_M4, q8_M3*q8_M1, q8_M4*q8_M2 ]
gap> hompres := HomomorphismOfPresentation( mq8 );
[ q8_M1, q8_M2, q8_M3, q8_M4 ] -> [ f1, f2, f1^-1, f2^-1 ]
```

### 2.1.3 PrintLnUsingLabels

▷ PrintLnUsingLabels(*args*) (function)
▷ PrintUsingLabels(*args*) (function)

The labels q8_M1, q8_M2, q8_M3, q8_M4 are rather unhelpful in lengthy output, so it is convenient to use $[a, b, A, B]$ as above. Then the function PrintUsingLabels takes as input a word in the monoid, the generators of the monoid, and a set of labels for these generators. This function also treats lists of words and lists of lists in a similar way. The function PrintLnUsingLabels does exactly the same, and then appends a newline.

Example

```
gap> q8labs := [ "a", "b", "A", "B" ];;
```

```
gap> SetMonoidPresentationLabels( q8, q8labs );;
gap> PrintLnUsingLabels( gprels, genfmq8, q8labs );
[ a^4, b^4, a*b*a*B, a^2*b^2 ]
```

### 2.1.4 InitialRulesOfPresentation

▷ `InitialRulesOfPresentation(`*mon*`)` (function)

The initial rules for *mq*8 are the four rules of the form $a^{-1} \to A$; the four rules of the form $aA \to id$; and the four relator rules of the form $a^4 \to id$.

Example

```
gap> q0 := InitialRulesOfPresentation( mq8 );;
gap> PrintLnUsingLabels( q0, genfmq8, q8labs );
[ [ a^-1, A ], [ b^-1, B ], [ A^-1, a ], [ B^-1, b ], [ a*A, id ],
[ b*B, id ], [ A*a, id ], [ B*b, id ], [ a^4, id ], [ a^2*b^2, id ],
[ a*b*a*B, id ], [ b^4, id ] ]
```

## 2.2 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.

### 2.2.1 RewritingSystemFpGroup

▷ `RewritingSystemFpGroup(`*grp*`)` (attribute)

This function attempts to return a complete rewrite system for the fp-group `G` obtained using the group's 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 20 rewrite rules in the rewriting system `rws`:

$$\begin{array}{cccccc} & a^{-1} \to A, & b^{-1} \to B, & A^{-1} \to a, & B^{-1} \to b, & \\ & aA \to \mathrm{id}, & \mathrm{bB} \to \mathrm{id}, & \mathrm{Aa} \to \mathrm{id}, & \mathrm{Bb} \to \mathrm{id}, & \\ ba \to aB, & b^2 \to a^2, & bA \to ab, & Ab \to aB, & A^2 \to a^2, & AB \to ab, \\ Ba \to ab, & BA \to aB, & B^2 \to a^2, & a^3 \to a, & a^2b \to B, & a^2B \to b. \end{array}$$

Example

```
gap> rws := RewritingSystemFpGroup( q8 );;
gap> Length( rws );
20
gap> PrintLnUsingLabels( rws, genfmq8, q8labs );
```

```
[ [ a^-1, A ], [ b^-1, B ], [ A^-1, a ], [ B^-1, b ], [ a*A, id ],
[ b*B, id ], [ A*a, id ], [ B*b, id ], [ b*a, a*B ], [ b^2, a^2 ],
[ b*A, a*b ], [ A*b, a*B ], [ A^2, a^2 ], [ A*B, a*b ], [ B*a, a*b ],
[ B*A, a*B ], [ B^2, a^2 ], [ a^3, A ], [ a^2*b, B ], [ a^2*B, b ] ]
```

The default ordering of the $2n$ monoid generators is $[g_1^+, g_2^+, \ldots, g_n^+, g_1^-, g_2^-, \ldots, g_n^-]$. In the case of the two-generator abelian group $T = \langle a, b \mid [a,b] \rangle$ the Knuth-Bendix process starts to generate infinite sets of relations such as $\{ab^m a^{-1} \to b^m,\ m \geqslant 1\}$.

If, using the `ArrangementOfMonoidGenerators` function, we specify the alternative ordering $[g_1^+, g_1^-, g_2^+, g_2^-]$, then a finite set of rules is obtained.

Example

```
gap> T := F/[Comm(f,g)];
<fp group of size infinity on the generators [ f1, f2 ]>
gap> SetName( T, "T" );
gap> SetArrangementOfMonoidGenerators( T, [1,-1,2,-2] );
gap> Tlabs := [ "x", "X", "y", "Y" ];;
gap> mT := MonoidPresentationFpGroup( T );
monoid presentation with group relators [ T_M2*T_M4*T_M1*T_M3 ]
gap> fgmT := FreeGroupOfPresentation( mT );;
gap> genfgmT := GeneratorsOfGroup( fgmT );;
gap> SetMonoidPresentationLabels( T, Tlabs );;
gap> rwsT := RewritingSystemFpGroup( T );;
gap> PrintLnUsingLabels( rwsT, genfgmT, Tlabs );
[ [ x^-1, X ], [ X^-1, x ], [ y^-1, Y ], [ Y^-1, y ], [ x*X, id ],
[ X*x, id ], [ y*Y, id ], [ Y*y, id ], [ y*x, x*y ], [ y*X, X*y ],
[ Y*x, x*Y ], [ Y*X, X*Y ] ]
```

The last four rules show that generators $x$ and $y$ commute.

### 2.2.2 OnePassReduceWord

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

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

These functions are called by the function `RewritingSystemFpGroup`.

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 our $q8$ example we see that $b^9a^{-9}$ reduces to $ab$.

Example

```
gap> a := genfmq8[1];;  b := genfmq8[2];;
gap> A := genfmq8[3];;  B := genfmq8[4];;
gap> w0 := b^9 * a^-9;;
gap> PrintLnUsingLabels( w0, genfmq8, q8labs );
b^9*a^-9
gap> w1 := OnePassReduceWord( w0, rws );;
gap> PrintLnUsingLabels( w1, genfmq8, q8labs );
B*b^5*a*b*a^-8
gap> w2 := ReduceWordKB( w0, rws );;
gap> PrintLnUsingLabels( w2, genfmq8, q8labs );
a*b
```

### 2.2.3 OnePassKB

▷ OnePassKB(*mon, rules*) (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 ShorterRule gives an ordering on rules. Rules $(g_1 g_2, id)$ 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 10 relators to the original 12.

Example

```
gap> q1 := OnePassKB( mq8, q0 );;
gap> Length( q1 );
22
gap> PrintLnUsingLabels( q1, genfmq8, q8labs );
[ [ a^-1, A ], [ b^-1, B ], [ A^-1, a ], [ B^-1, b ], [ a*A, id ],
[ b*B, id ], [ A*a, id ], [ B*b, id ], [ b^2, a^2 ], [ a^3, A ],
[ a^2*b, B ], [ a*b*a, b ], [ a*b^2, A ], [ b*a*B, A ], [ b^3, B ],
[ a*b^2, a^3 ], [ b*a*B, a^3 ], [ b^3, a^2*b ], [ a^4, id ],
[ a^2*b^2, id ], [ a*b*a*B, id ], [ b^4, id ] ]
```

### 2.2.4 RewriteReduce

▷ RewriteReduce(*mon, rules*) (operation)

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.

In the example the 22 rules in $q1$ are reduced to 13.

Example

```
gap> q2 := RewriteReduce( mq8, q1 );;
```

```
gap> Length( q2 );
13
gap> PrintLnUsingLabels( q2, genfmq8, q8labs );
[ [ a^-1, A ], [ b^-1, B ], [ A^-1, a ], [ B^-1, b ], [ a*A, id ],
[ b*B, id ], [ A*a, id ], [ B*b, id ], [ b^2, a^2 ], [ a^3, A ],
[ a^2*b, B ], [ a*b*a, b ], [ b*a*B, A ] ]
```

### 2.2.5 KnuthBendix

▷ KnuthBendix(*mon, rules*) (operation)

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 $(g_1 g_2, id)$ 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.

In the example the function KnuthBendix requires three instances of OnePassKB and RewriteReduce to form the complete rewrite system *rws* for the group shown above.

Example

```
gap> q3 := KnuthBendix( mq8, q0 );;
gap> Length( q3 );
20
gap> PrintLnUsingLabels( q3, genfmq8, q8labs );
[ [ a^-1, A ], [ b^-1, B ], [ A^-1, a ], [ B^-1, b ], [ a*A, id ],
[ b*B, id ], [ A*a, id ], [ B*b, id ], [ b*a, a*B ], [ b^2, a^2 ],
[ b*A, a*b ], [ A*b, a*B ], [ A^2, a^2 ], [ A*B, a*b ], [ B*a, a*b ],
[ B*A, a*B ], [ B^2, a^2 ], [ a^3, A ], [ a^2*b, B ], [ a^2*B, b ] ]
```

## 2.3 Enumerating elements

### 2.3.1 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. For q8 this list is

$$[\,\mathrm{id},\ a^{+},\ b^{+},\ a^{-},\ b^{-},\ a^{+2},\ a^{+}b^{+},\ a^{+}b^{-}\,].$$

Example

```
gap> elmq8 := ElementsOfMonoidPresentation( q8 );;
gap> PrintLnUsingLabels( elmq8, genfmq8, q8labs );
[ id, a, b, A, B, a^2, a*b, a*B ]
```

# 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 $u = n_1^{w_1} \ldots n_k^{w_k} v$.

## 3.1 Logged Knuth-Bendix Completion

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

### 3.1.1 InitialLoggedRulesOfPresentation

▷ `InitialLoggedRulesOfPresentation(`*mon*`)` (function)

The 12 initial logged rules for *mq*8 correspond to the 12 initial rules in section 2.1.4. Rules of the form $g^{-1} \to G$ and $gG \to id$ apply to the monoid presentation, but not to the group presentation, so are given an empty logged component. The remaining four rules, which corresppond to the relators $r \in [a^4, b^4, abab^{-1}, a^2b^2]$ are given logged components $[r, [[n, id]], id]$ for $n \in [9..12]$.

Example

```
gap> r0 := InitialLoggedRulesOfPresentation( mq8 );;
gap> PrintLnUsingLabels( r0, genfmq8, q8labs );
[ [ a^-1, [ ], A ], [ b^-1, [ ], B ], [ A^-1, [ ], a ], [ B^-1,
[ ], b ], [ a*A, [ ], id ], [ b*B, [ ], id ], [ A*a, [ ], id ],
[ B*b, [ ], id ], [ a^4, [ [ 1, id ] ], id ], [ a^2*b^2, [ [ 4, id ] ], id ],
[ a*b*a*B, [ [ 3, id ] ], id ], [ b^4, [ [ 2, id ] ], id ] ]
```

### 3.1.2 LoggedOnePassKB

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

Given a logged rewrite system for the group `grp`, this function finds all the rules that would be added to complete the rewrite system of `OnePassKB` in 2.2.3, 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 apply one pass of the logged Knuth-Bendix procedure to the initial set of logged rules.

Example

```
gap> r1 := LoggedOnePassKB( mq8, r0 );;
gap> Length( r1 );
25
gap> PrintLnUsingLabels( r1, genfmq8, q8labs );
[ [ a^-1, [ ], A ], [ b^-1, [ ], B ], [ A^-1, [ ], a ], [ B^-1,
[ ], b ], [ a*A, [ ], id ], [ b*B, [ ], id ], [ A*a, [ ], id ],
[ B*b, [ ], id ], [ b^2, [ [ -4, id ], [ 2, A^2 ] ], a^2 ],
[ b^2, [ [ -1, id ], [ 4, A^2 ] ], a^2 ], [ a^3, [ [ 1, id ] ], A ],
[ a^3, [ [ 1, a ] ], A ], [ a^2*b, [ [ 4, id ] ], B ], [ a*b*a,
[ [ 3, id ] ], b ], [ a*b^2, [ [ 4, a ] ], A ], [ b*a*B, [
[ 3, a ] ], A ], [ b^3, [ [ 2, id ] ], B ], [ b^3, [ [ 2, b ] ], B ],
[ a*b^2, [ [ -1, id ], [ 4, A^3 ] ], a^3 ], [ b*a*B, [ [ -1, id ],
[ 3, A^3 ] ], a^3 ], [ b^3, [ [ -4, id ], [ 2, B*A^2 ] ], a^2*b ],
[ a^4, [ [ 1, id ] ], id ], [ a^2*b^2, [ [ 4, id ] ], id ],
[ a*b*a*B, [ [ 3, id ] ], id ], [ b^4, [ [ 2, id ] ], id ] ]
```

Note that $r1$ has length 25, three more than the length 22 of `q1` in 2.2.3. This because the three rules $b^2 \to a^2$; $a^3 \to A$; $b^3 \to B$ each appear twice, with alternative logged components.

If we write $a,b,A,B$ for `M1,M2,M3,M4` and label the four original relators as $q = a^4$, $r = b^4$, $s = abaB$, $t = a^2b^2$ then the ninth identity (for example) says that $b^2 = (t^{-1}r^{A^2})a^2$. To verify this, we may expand the right-hand side as follows:

$$(B^2A^2).a^2(b^4)A^2.a^2 \;=\; B^2(A^2a^2)b^4(A^2a^2) \;=\; B^2b^4 \;=\; b^2.$$

### 3.1.3 LoggedRewriteReduce

▷ `LoggedRewriteReduce(`*grp, loggedrules*`)` (operation)

The function `LoggedRewriteReduce` removes unnecessary rules from a logged rewrite system. It works on the same principle as `RewriteReduce` in 2.2.4. Note that $q2$ nd $r2$ both have length 13.

Example

```
gap> r2 := LoggedRewriteReduce( mq8, r1 );;
gap> Length( r2 );
13
gap> PrintLnUsingLabels( r2, genfmq8, q8labs );
[ [ a^-1, [ ], A ], [ b^-1, [ ], B ], [ A^-1, [ ], a ], [ B^-1,
[ ], b ], [ a*A, [ ], id ], [ b*B, [ ], id ], [ A*a, [ ], id ],
[ B*b, [ ], id ], [ b^2, [ [ -4, id ], [ 2, A^2 ] ], a^2 ],
[ a^3, [ [ 1, id ] ], A ], [ a^2*b, [ [ 4, id ] ], B ], [ a*b*a,
[ [ 3, id ] ], b ], [ b*a*B, [ [ 3, a ] ], A ] ]
```

### 3.1.4 LoggedKnuthBendix

▷ LoggedKnuthBendix(*grp*, *loggedrules*) (operation)

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

As a further example, consider the ninth rule in r3 which shows how $ba$ reduces to $aB$. For this rule [u,L,v] we will verify that $u = n_1^{w_1} n_2^{w_2} n_3^{w_3} v$, as in the introduction to this chapter. The rule is:

$$[ba, [[-11, id], [12, BA]], aB].$$

The relators are $-11 \equiv s^{-1} = bABA$ and $12 \equiv t = a^2b^2$. These are conjugated by the identity and $BA$ respectively. So the second and third parts of the rule expand to:

$$(bABA)(ab(aabb)BA)aB \;=\; bAB(Aa)baab(bB)(Aa)B \;=\; bA(Bb)aa(bB) \;=\; b(Aa)a \;=\; ba,$$

the first part of the rule.

Example

```
gap> r3 := LoggedKnuthBendix( mq8, r0 );;
gap> Length( r3 );
20
gap> PrintLnUsingLabels( r3, genfmq8, q8labs );
[ [ a^-1, [ ], A ], [ b^-1, [ ], B ], [ A^-1, [ ], a ], [ B^-1,
[ ], b ], [ a*A, [ ], id ], [ b*B, [ ], id ], [ A*a, [ ], id ],
[ B*b, [ ], id ], [ b*a, [ [ -3, id ], [ 4, B*A ] ], a*B ],
[ b^2, [ [ -4, id ], [ 2, A^2 ] ], a^2 ], [ b*A, [ [ -3, id ] ], a*b ],
[ A*b, [ [ -1, id ], [ 4, A ] ], a*B ], [ A^2, [ [ -1, id ] ], a^2 ],
[ A*B, [ [ -4, a ] ], a*b ], [ B*a, [ [ -4, id ], [ 3, A ] ], a*b ],
[ B*A, [ [ -3, a*b ] ], a*B ], [ B^2, [ [ -4, id ] ], a^2 ],
[ a^3, [ [ 1, id ] ], A ], [ a^2*b, [ [ 4, id ] ], B ], [ a^2*B,
[ [ -4, A^2 ], [ 1, id ] ], b ] ]
```

### 3.1.5 LoggedRewritingSystemFpGroup

▷ LoggedRewritingSystemFpGroup(*grp*) (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. Corresponding to the j-th relator rel of the group there is a logged rule [rel,[[j,id]],id]. For each inverse relator there is a logged rule [ gen*inv, [], id ]. The function then attempts a completion of the logged rewrite system. The rules in the final system are partially ordered by the function ShorterLoggedRule.

Example

```
gap> lrws := LoggedRewritingSystemFpGroup( q8 );;
gap> PrintLnUsingLabels( lrws, genfgmon, q8labs );
```

```
[ [ a^-1, [ ], A ], [ b^-1, [ ], B ], [ A^-1, [ ], a ], [ B^-1,
[ ], b ], [ a*A, [ ], id ], [ b*B, [ ], id ], [ A*a, [ ], id ],
[ B*b, [ ], id ], [ b*a, [ [ -3, id ], [ 4, B*A ] ], a*B ],
[ b^2, [ [ -4, id ], [ 2, A^2 ] ], a^2 ], [ b*A, [ [ -3, id ] ], a*b ],
[ A*b, [ [ -1, id ], [ 4, A ] ], a*B ], [ A^2, [ [ -1, id ] ], a^2 ],
[ A*B, [ [ -4, a ] ], a*b ], [ B*a, [ [ -4, id ], [ 3, A ] ], a*b ],
[ B*A, [ [ -3, a*b ] ], a*B ], [ B^2, [ [ -4, id ] ], a^2 ],
[ a^3, [ [ 1, id ] ], A ], [ a^2*b, [ [ 4, id ] ], B ], [ a^2*B,
[ [ -4, A^2 ], [ 1, id ] ], b ] ]
gap> Length( lrws );
16
```

Consider now the two-generator abelian group $T$ considered in the previous chapter (2.2.1). Using the alternative ordering on the monoid generators, `[ T_M1=` $a$, `T_M2=` $A$, `T_M3=` $b$, `T_M4=` $B$ `]`, we obtain the following set of 8 logged rules. The last of these may be checked as follows:

$$(ba(BAba)AB)ab \;=\; ba(B(A(b(aA)B)a)b)$$

and is a logged version of the rule $ba \to ab$.

Example

```
gap> lrwsT := LoggedRewritingSystemFpGroup( T );;
gap> PrintLnUsingLabels( lrwsT, genfgmonT, Tlabs );
[ [ x^-1, [ ], X ], [ X^-1, [ ], x ], [ y^-1, [ ], Y ], [ Y^-1,
[ ], y ], [ x*X, [ ], id ], [ X*x, [ ], id ], [ y*Y, [ ], id ],
[ Y*y, [ ], id ], [ y*x, [ [ -1, X*Y ] ], x*y ], [ y*X, [ [ 1, Y ] ], X*y ],
[ Y*x, [ [ 1, X ] ], x*Y ], [ Y*X, [ [ -1, id ] ], X*Y ] ]
```

## 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 pass of the word (possibly involving several reductions) (as does `OnePassReduceWord` in 2.2.2) and records this, using the log part of the rule(s) 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 2.2.3. In the example we perform logged reductions of $w_0 = a^9b^{-9}$ corresponding to the ordinary reductions performed in the previous chapter (section 2.2.2).

In order to clarify the following output, note that, in the log below, $b^9a^{-9}$ reduces to $Bb^5aba^{-8}$ in `lw1`, just as in section 2.2.2. This may be checked by cancelling terms in:

$$(b^2A^2)(a^2.b^4.A^2)(a^2b^6.bABA.b^6A^2)(a^2b^2)Bb^5aba^{-8} \;=\; b^9a^9.$$

The corresponding expansion of `lw2` is too lengthy to include here. (It's hard to believe that the logged part of this identity is the simplest possible. Further investigation is needed to determine whether or not this logged part can be simplified.)

Example

```
gap> PrintLnUsingLabels( w0, genfmq8, q8labs );
b^9*a^-9
gap> lw1 := LoggedOnePassReduceWord( w0, lrws );;
gap> PrintLnUsingLabels( lw1, genfmq8, q8labs );
[ [ [ -4, id ], [ 2, A^2 ], [ -3, b^-6*a^-2 ], [ 4, id ] ],
B*b^5*a*b*a^-8 ]
gap> lw2 := LoggedReduceWordKB( w0, lrws );;
gap> PrintLnUsingLabels( lw2, genfmq8, q8labs );
[ [ [ -4, id ], [ 2, A^2 ], [ -3, b^-6*a^-2 ], [ 4, id ], [ -3, b^-3 ],
[ 4, B*A*b^-3 ], [ -4, id ], [ 2, A^2 ], [ -3, B^-1*a^-1*b^-1*a^-2 ],
[ -4, a^-1*b^-1*a^-2 ], [ 3, A*a^-1*b^-1*a^-2 ], [ 4, id ],
[ -4, a^-2*B^-1 ], [ 2, A^2*a^-2*B^-1 ], [ -4, id ], [ 3, A ],
[ 1, b^-1*a^-1 ], [ -3, a^-1 ], [ -1, b^-1*a^-2 ], [ 4, id ],
[ -3, a*b ], [ -3, a*b*a^-1 ], [ -4, A^2 ], [ 1, id ], [ -3, id ] ], a*b ]
```

# 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> relq8 := 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, relq8 );;
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 (for monoid polynomials)

▷ 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 (for monoid polynomials)

▷ `Length(`*poly*`)` (method)

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

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> Length( pr );
4
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 := genfmq8;;
gap> mp1 := MonoidPolyFromCoeffsWords( [9,-7,5],
>               [ M[1]*M[3], M[2]^3, M[4]*M[3]*M[2] ] );;
gap> PrintUsingLabels( mp1, genfmq8, q8labs );
5*B*A*b + -7*b^3 + 9*a*A
gap> rmp1 := ReduceMonoidPoly( mp1, r2 );;
gap> PrintUsingLabels( rmp1, genfmq8, q8labs );
-7*B + 5*a + 9*id
```

# 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 (with input gens, polys)

▷ `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> q8R := FreeRelatorGroup( q8 );;
gap> genq8R := GeneratorsOfGroup( q8R );
[ q8_R1, q8_R2, q8_R3, q8_R4 ]
gap> q8Rlabs := [ "q", "r", "s", "t" ];;
gap> Print( rmp1, "\n" );
 - 7*q8_M4 + 5*q8_M1 + 9*<identity ...>
gap> M := GeneratorsOfGroup( fmq8 );
[ q8_M1, q8_M2, q8_M3, q8_M4 ]
```

```
gap> mp2 := MonoidPolyFromCoeffsWords( [4,-5], [ M[4], M[1] ] );;
gap> Print( mp2, "\n" );
4*q8_M4 - 5*q8_M1
gap> zeromp := ZeroModulePoly( q8R, freeq8 );
zero modpoly
gap> s1 := ModulePoly( [ genq8R[4], genq8R[1] ], [ rmp1, mp2 ] );
q8_R1*(4*q8_M4 - 5*q8_M1) + q8_R4*( - 7*q8_M4 + 5*q8_M1 + 9*<identity ...>)
```

### 5.1.2 PrintLnModulePoly (input object, [gens,labels] for the group, ditto relators)

▷ PrintLnModulePoly(*obj, gens1, labs1, gens2, labs2*) (operation)
▷ PrintModulePoly(*obj, gens1, labs1, gens2, labs2*) (operation)

The function PrintModulePoly prints a module polynomial, using the function PrintUsingLabels. Two lists of labels are involved: those for the fp-group being investigated, and those for the free relator group of this group. The function PrintLnModulePoly does exactly the same, and then appends a newline.

Example

```
gap> q8Rlabs := [ "q", "r", "s", "t" ];;
gap> PrintLnModulePoly( s1, genfgmon, q8labs, genq8R, q8Rlabs );
q*(4*B + -5*a) + t*(-7*B + 5*a + 9*id)
gap> s2 := ModulePoly( [ genq8R[3], genq8R[2], genq8R[1] ],
>       [ -1*rmp1, 3*mp2, (rmp1+mp2) ] );;
gap> PrintLnModulePoly( s2, genfgmon, q8labs, genq8R, q8Rlabs );
q*(-3*B + 9*id) + r*(12*B + -15*a) + s*(7*B + -5*a + -9*id)
```

## 5.2 Components of a module polynomial

### 5.2.1 Terms (for module polynomials)

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

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.

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.

Example

```
gap> [ Length(s1), Length(s2) ];
[ 2, 3 ]
gap> One( s1 );
<identity ...>
gap> terms := Terms( s1 );;
gap> for t in terms do
>        PrintModulePolyTerm( t, genfmq8, q8labs, genq8R, q8Rlabs );
>        Print( "\n" );
>    od;
q*(4*B + -5*a)
t*(-7*B + 5*a + 9*id)
gap> t1 := LeadTerm( s1 );;
gap> PrintModulePolyTerm( t1, genfmq8, q8labs, genq8R, q8Rlabs );
t*(-7*B + 5*a + 9*id)
gap> t2 := LeadTerm( s2 );;
gap> PrintModulePolyTerm( t2, genfmq8, q8labs, genq8R, q8Rlabs );
s*(7*B + -5*a + -9*id)
gap> p1 := LeadMonoidPoly( s1 );
 - 7*q8_M4 + 5*q8_M1 + 9*<identity ...>
gap> p2 := LeadMonoidPoly( s2 );
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> s0 := AddTermModulePoly( s1, genq8R[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)
```

# Chapter 6

# Identities Among Relators

The identities among the relators for a finitely presented group $G$ are constructed as logged module polynomials. The procedure, described in [HW03] and based on work in [BRS99], is to construct a full set of *group relator 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.

When a reduced set of polynomials has been obtained, the relator sequences from which they were formed are returned as the *identities among relators* for $G$.

## 6.1 Constructing identities

### 6.1.1 RootIdentities

▷ `RootIdentities(grp)` (attribute)
▷ `RootPositions(grp)` (attribute)

The *root identities* of $G$ are identities of the form $R^{-1}R^w$ where $R = w^n$ is a proper power relator and $n > 1$. (For equivalent forms invert, or permute the factors cyclically, or act with $w^{-1}$.)

For $S_3 = \langle f, g \mid \rho = f^3, \sigma = g^2, \tau = (fg)^2 \rangle$ all three relators are proper powers: $[1 \equiv \rho = f^3, 2 \equiv \sigma = g^2, 3 \equiv \tau = (fg)^2]$. So the root identities are $\rho^{-1}\rho^a$, $\sigma^{-1}\sigma^b$ and $\tau^{-1}\tau^{ab}$.

For $Q_8 = \langle a, b \mid q = a^4, r = b^4, s = abab^{-1}, t = a^2b^2 \rangle$ only two of the four relators are proper powers, so the root identities are $q^{-1}q^a$ and $r^{-1}r^b$.

In the example we see that the attribute `RootIdentities` returns a list which includes $R^{-1}R^{w^{-1}}$ as well as $R^{-1}R^w$. Relator $\rho^{-1}\rho^f$ is stored as `[[-1,id],[1,f]]`, etc.

The `RootPositions` attribute is a boolean list specifying which of the relators are proper powers.

Example

```
gap> roots3 := RootIdentities(s3);
[ [ [ -1, <identity ...> ], [ 1, s3_M1 ] ],
  [ [ -1, <identity ...> ], [ 1, s3_M3 ] ],
  [ [ -2, <identity ...> ], [ 2, s3_M2 ] ],
  [ [ -2, <identity ...> ], [ 2, s3_M4 ] ],
  [ [ -3, <identity ...> ], [ 3, s3_M1*s3_M2 ] ],
  [ [ -3, <identity ...> ], [ 3, s3_M4*s3_M3 ] ] ]
gap> RootPositions(s3);
[ true, true, true ]
```

```
gap> PrintLnUsingLabels( RootIdentities(q8), genfmq8, q8labs );
[ [ [ -1, id ], [ 1, a ] ], [ [ -1, id ], [ 1, A ] ], [ [ -2, id ],
[ 2, b ] ], [ [ -2, id ], [ 2, B ] ] ]
gap> RootPositions(q8);
[ true, true, false, false ]
```

### 6.1.2 IdentityRelatorSequences

▷ `IdentityRelatorSequences(grp)` (attribute)

To construct the *identity relator sequences* for a group $G$ we apply each relator $R$ at each non-identity element $x$, reducing the resulting words using the logged rewrite system.

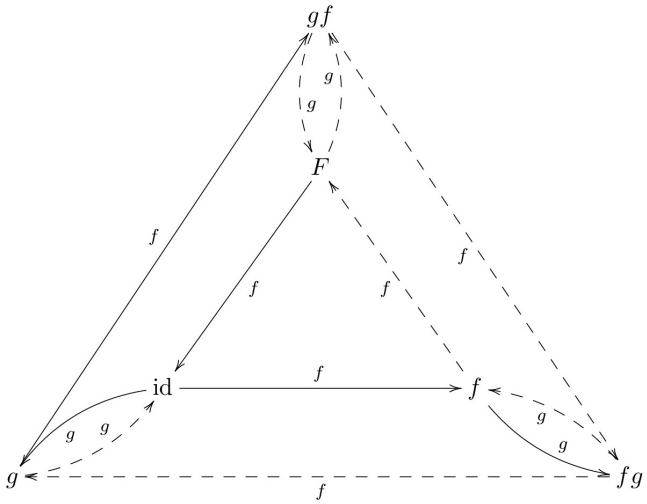
With the `s3` example, the monoid presentation has generators $\{f,g,F,G\}$ and relators

$$[\, fF,\ gG,\ Ff,\ Gg,\ f^3,\ g^2,\ (fg)^2\,]\,,$$

and the elements are $\{\mathrm{id}, f, g, F, fg, gf\}$. The logged rewriting system has relations

$$\begin{array}{llllll} f^{-1}=F, & g^{-1}=[-2,\mathrm{id}]g, & F^{-1}=f, & G^{-1}=g, & G=[-2,id]g, & \\ fF=\mathrm{id}, & g^2=[2,\mathrm{id}]\mathrm{id}, & Ff=\mathrm{id}, & f^2=[1,\mathrm{id}]F, & F^2=[-1,\mathrm{id}]f, & \\ gF=[-3,\mathrm{id}][2,FGF]fg, & Fg=[-3,f][2,FG]gf, & & & & \\ fgf=[-2,FGF][3,\mathrm{id}]g, & gfg=[3,f]F & & & & \end{array}$$

Here is the Cayley graphs of $S_3$, with the solid arrows forming the spanning tree:




Applying $R=\tau=(fg)^2$ at $x=f$ gives the cycle (top right-hand quadrilateral):

$$f \xrightarrow{f} F \xrightarrow{g} gf \xrightarrow{f} fg \xrightarrow{g} f\,.$$

Each of these edges has a non-trivial logged rewrite, particularly the third edge where $gff \to gF \to fg$. Combining this log information we obtain:

$$[\tau,F]f \;=\; f.\tau \;=\; [1,\mathrm{id}].[-3,f][2,FG].[1,G][-3,\mathrm{id}][2,FGF].[2,F]f.$$

Expanding $[1,\mathrm{id}][-3,f][2,FG][1,G][-3,\mathrm{id}][2,FGF][2,F][-3,F]$ gives

$$fff.FGFGFf.gfggFG.gfffG.GFGF.fgfggFGF.fggF.fGFGFF$$

which cancels to the identity, as expected. Converting this back to the group presentation, we obtain the fourth identity given in the introduction (1.1):

$$\iota_{(\tau,f)} = \rho\,(\tau^{-1})^{f}\,\sigma^{FG}\,\rho^{G}\,(\tau^{-1})\,\sigma^{FGF}\,\sigma^{F}\,(\tau^{-1})^{F}\,.$$

The operation `IdentityRelatorSequences` returns a list which omits any duplicates or empty lists. For the `s3` example, all of the possible $5*3=15$ sequences are added to the root identities.

Example

```
gap> ms3 := MonoidPresentationFpGroup( s3 );;
gap> fms3 := FreeGroupOfPresentation( ms3 );;
gap> genfms3 := GeneratorsOfGroup(fms3 );;
gap> s3labs := ["f","g","F","G"];;
gap> SetMonoidPresentationLabels( ms3, s3labs );;
gap> idss3 := IdentityRelatorSequences( s3 );;
gap> lenidss3 := Length( idss3 );
17
gap> List( idss3, L -> Length(L) );
[ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4, 6, 8, 8 ]
gap>  for i in [1..Length(idss3)] do
>       PrintLnUsingLabels( idss3[i], genfms3, s3labs );
>     od;
[ [ -3, id ], [ 3, f*g ] ]
[ [ -3, id ], [ 3, G*F ] ]
[ [ -2, id ], [ 2, g ] ]
[ [ -2, id ], [ 2, G ] ]
[ [ -1, id ], [ 1, f ] ]
[ [ -1, id ], [ 1, F ] ]
[ [ 1, id ], [ -1, f ] ]
[ [ 1, id ], [ -1, F ] ]
[ [ 1, G ], [ -1, F*G ] ]
[ [ 2, id ], [ -2, G ] ]
[ [ 2, F ], [ -2, G*F ] ]
[ [ 3, f ], [ -3, G ] ]
[ [ -3, f ], [ 2, F*G ], [ 3, f ], [ -2, f ] ]
[ [ -2, F*G*F ], [ 3, id ], [ 2, id ], [ -3, G*F ] ]
[ [ -2, F*G*F ], [ 3, id ], [ 1, G ], [ -3, id ], [ 2, F*G*F ],
[ -1, G*F ] ]
[ [ 1, id ], [ -3, f ], [ 2, F*G ], [ 1, G ], [ -3, id ],
[ 2, F*G*F ], [ 2, F ], [ -3, F ] ]
[ [ 1, G ], [ -3, id ], [ 2, F*G*F ], [ 2, F ], [ 1, id ],
[ -3, f ], [ 2, F*G ], [ -3, F*G ] ]
```

### 6.1.3 LogSequenceLessThan

▷ `LogSequenceLessThan(J, K)` (operation)

This is an operation used to sort lists of identity sequences. First the lengths of sequences `J, K` are compared. If the lengths are equal then the sequences are compared as lists. The list `idss3` is sorted using this function.

Example

```
gap> LogSequenceLessThan( idss3[7], idss3[8] );
true
```

### 6.1.4 ExpandLogSequence

▷ `ExpandLogSequence(`*mG, L*`)` (operation)

This operation takes a log sequence $L$, writes each term as a conjugate of a relator, takes the product of all of these, and then cancels consecutive inverse generators to return a word in the free group of the presentation. This is precisely what we did by hand with $\iota_{(\tau,f)}$ on the previous page. If the sequence is an identity sequence the identity element should be returned, so this provides a useful check.

Example

```
gap> ExpandLogSequence( ms3, idss3[17] );
<identity ...>
```

## 6.2 Identities for $S_3$

We now return to the example considered in section 1.1. In the previous section we have constructed 17 identity sequences, and we now wish to reduce this number to find a minimal set.

### 6.2.1 ReduceLogSequences

▷ `ReduceLogSequences(`*G, ids*`)` (operation)

This operation applies a collection of operations, which will be described in the following section, to reduce the list `idss3` from 17 to 5 identities.

Example

```
gap> ridss3 := ReduceLogSequences( s3, idss3 );;
gap> lenridss3 := Length( ridss3 );
5
gap> for i in [1..lenridss3] do
>        PrintLnUsingLabels( ridss3[i], genfms3, s3labs );
>    od;
[ [ -3, id ], [ 3, f*g ] ]
[ [ -2, id ], [ 2, g ] ]
[ [ -1, id ], [ 1, f ] ]
[ [ 1, id ], [ -3, f ], [ 2, F*G ], [ 1, G ], [ -3, id ],
[ 2, F*G*F ], [ 2, F ], [ -3, F ] ]
```

```
[ [ 1, id ], [ -3, g ], [ 2, F*G*F*g ], [ 2, F*g ], [ 1, g ],
[ -3, id ], [ 2, F ], [ -3, F ] ]
```

We wish to show that the fifth of these identities is a combination of the first four. Recall that the fourth identity was obtained by applying $R = \tau = (fg)^2$ at $x = f$. The fifth comes from applying $R = \tau$ at $x = gf$, so this is the same cycle but with a different start point.

### 6.2.2 ConjugateByWordLogSequence

▷ `ConjugateByWordLogSequence(`*mG, K, w*`)` (operation)

This operation conjugates every term in a log sequence by a word in the generators. In the example we conjugate the fifth identity `K5` by $G$. It then becomes apparent that the fourth identity `K4` has the form `[ A, B, [ -3, F ] ]` while `K5` has the form `[ B, A, [ -3, FG ] ]`, where the `F` and the `GF` are the inverses of the vertices where the cycle starts.

Example

```
gap> K4 := ShallowCopy( ridss3[4] );;
gap> PrintLnUsingLabels( K4, genfms3, s3labs );
[ [ 1, id ], [ -3, f ], [ 2, F*G ], [ 1, G ], [ -3, id ],
[ 2, F*G*F ], [ 2, F ], [ -3, F ] ]
gap> L5 := ShallowCopy( ridss3[5] );;
gap> K5 := ConjugateByWordLogSequence( ms3, L5, genfms3[4] );;
gap> PrintLnUsingLabels( K5, genfms3, s3labs );
[ [ 1, G ], [ -3, id ], [ 2, F*G*F ], [ 2, F ], [ 1, id ],
[ -3, f ], [ 2, F*G ], [ -3, F*G ] ]
gap> A := K4{[1..3]};;
gap> PrintLnUsingLabels( A, genfms3, s3labs );
[ [ 1, id ], [ -3, f ], [ 2, F*G ] ]
gap> B := K4{[4..7]};;
gap> PrintLnUsingLabels( B, genfms3, s3labs );
[ [ 1, G ], [ -3, id ], [ 2, F*G*F ], [ 2, F ] ]
gap> PositionSublist( K5, A );
5
gap> PositionSublist( K5, B );
1
```

### 6.2.3 ChangeStartLogSequence

▷ `ChangeStartLogSequence(`*mon, K, p*`)` (operation)

The start point of an identity log sequence can be chosen at random (since every conjugate of an identity is that identity). This operation permutes a given sequence $K$ so as to start at the $p$-th position.

In our example we wish to show that `K4` and `K5` are equivalent up to root identities. To do this we first replace `K4` by `J4 = [ B, [ -3, F ], A ]`.

Example

```
gap> J4 := ChangeStartLogSequence( ms3, K4, 4 );;
```

```
gap> PrintLnUsingLabels( J4, genfms3, s3labs );
[ [ 1, G ], [ -3, id ], [ 2, F*G*F ], [ 2, F ], [ -3, F ],
[ 1, id ], [ -3, f ], [ 2, F*G ] ]
```

### 6.2.4 InverseLogSequence

▷ `InverseLogSequence(`*K*`)` (operation)

To invert a log sequence we reverse the order of the terms and replace each `[m,w]` by `[-m,w]`. We continue our example by replacing `J4` by its inverse.

Example

```
gap> J4 := InverseLogSequence( J4 );;
gap> PrintLnUsingLabels( J4, genfms3, s3labs );
[ [ -2, F*G ], [ 3, f ], [ -1, id ], [ 3, F ], [ -2, F ], [ -2, F*G*F ],
[ 3, id ], [ -1, G ] ]
```

### 6.2.5 CancelImmediateInversesLogSequence

▷ `CancelImmediateInversesLogSequence(`*K*`)` (attribute)
▷ `CancelInversesLogSequence(`*mG, K*`)` (operation)

Concatenating `J4` and `K5`, we get `[ A^-1, [ 3, F ], B^-1, B, A, [ -3, FG ] ]`, with length 16. Cancelling immediate inverses removes the `[ B^-1, B ]`. Cancelling inverses gets rid of the terms `a^-1` and `A`, converting `[ 3, F ]` into `[ 3, fgFG ] = [ 3, FG ]`. Conjugating with `gf` produces the third root identity `[ [ 3, fg ], [ -3, id ] ]`, which then cancels.

Example

```
gap> J4K5 := Concatenation( J4, K5 );;
gap> J4K5 := CancelImmediateInversesLogSequence( J4K5 );;
gap> PrintLnUsingLabels( J4K5, genfms3, s3labs );
[ [ -2, F*G ], [ 3, f ], [ -1, id ], [ 3, F ], [ 1, id ],
[ -3, f ], [ 2, F*G ], [ -3, F*G ] ]
gap> J4K5 := CancelInversesLogSequence( ms3, J4K5 );
[ ]
```

## 6.3 Reducing identities

In this section we list some further operations which may be used to simplify the list of identities returned by `IdentityRelatorSequences`. We will use our $Q_8$ presentation in the examples.

Example

```
gap> mq8 := MonoidPresentationFpGroup( q8 );;
gap> fmq8 := FreeGroupOfPresentation( mq8 );;
gap> genfmq8 := GeneratorsOfGroup(fmq8 );;
```

```
gap> q8labs := ["a","b","A","B"];;
gap> SetMonoidPresentationLabels( mq8, q8labs );;
gap> idsq8 := IdentityRelatorSequences( q8 );;
gap> lenidsq8 := Length( idsq8 );
28
gap> List( idsq8, L -> Length(L) );
[ 2, 2, 2, 2, 2, 2, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 6, 6, 7, 8, 8, 8,
  9, 10, 10 ]
```

### 6.3.1 LogSequenceRewriteRules

▷ LogSequenceRewriteRules(*mG*) (attribute)

The root identity $R^{-1}R^w$ may be converted into the rewrite rule $R^w \to R$.

Example

```
gap> rulesq8 := LogSequenceRewriteRules( mq8 );;
gap> for i in [1..8] do
>        PrintLnUsingLabels( rulesq8[i], genfmq8, q8labs );
>    od;
[ [ 1, a ], [ 1, id ] ]
[ [ -1, a ], [ -1, id ] ]
[ [ 1, A ], [ 1, id ] ]
[ [ -1, A ], [ -1, id ] ]
[ [ 2, b ], [ 2, id ] ]
[ [ -2, b ], [ -2, id ] ]
[ [ 2, B ], [ 2, id ] ]
[ [ -2, B ], [ -2, id ] ]
[ [ 3, a*b*a*B ], [ 3, id ] ]
[ [ 3, b*A*B*A ], [ 3, id ] ]
[ [ -3, a*b*a*B ], [ -3, id ] ]
[ [ -3, b*A*B*A ], [ -3, id ] ]
[ [ 4, a^2*b^2 ], [ 4, id ] ]
[ [ 4, B^2*A^2 ], [ 4, id ] ]
[ [ -4, a^2*b^2 ], [ -4, id ] ]
[ [ -4, B^2*A^2 ], [ -4, id ] ]
```

### 6.3.2 OnePassReduceLogSequence

▷ OnePassReduceLogSequence(*J, rules*) (operation)

The rewrite rules returned by LogSequenceRewriteRules may be used to simplify other identity sequences. In the example the fourth rule $(q^{-1})^A \to q^{-1}$, applied twice, reduces $(q^{-1})^{A^2}$ to $q^{-1}$.

Example

```
gap> J7 := idsq8[7];
[ [ 1, <identity ...> ], [ -1, q8_M3^2 ] ]
gap> OnePassReduceLogSequence( J7, rulesq8 );
```

```
[ [ 1, <identity ...> ], [ -1, <identity ...> ] ]
```

The operation ReduceLogSequences, described in subsection 6.2.1, applied to the list idsq8 reduces the 28 identities to 15.

Example

```
gap> ridsq8 := ReduceLogSequences( q8, idsq8 );;
gap> lenrids := Length( ridsq8 );
15
gap> for i in [1..lenrids] do
>        PrintLnUsingLabels( ridsq8[i], genfmq8, q8labs );
>    od;
[ [ -2, id ], [ 2, b ] ]
[ [ -1, id ], [ 1, a ] ]
[ [ -4, id ], [ 2, A^2 ], [ 1, id ], [ -4, a^2 ] ]
[ [ -4, id ], [ 3, A ], [ 4, a ], [ -3, b ] ]
[ [ 1, id ], [ -4, id ], [ 2, A^2 ], [ -4, A^2 ] ]
[ [ -4, id ], [ 3, A ], [ 3, id ], [ 2, id ], [ -4, b ] ]
[ [ -3, id ], [ 4, B*A ], [ -4, A ], [ 1, id ], [ -3, a ] ]
[ [ -3, id ], [ 4, B*A ], [ -4, A^2 ], [ 1, id ], [ -3, B ] ]
[ [ -4, id ], [ 3, A ], [ -4, A ], [ 2, A^3 ], [ 1, id ],
[ -3, b ] ]
[ [ -4, id ], [ 4, B*A^2 ], [ -4, A^2 ], [ 1, id ], [ 2, id ],
[ -4, b ] ]
[ [ -3, id ], [ 4, B*A ], [ -4, A ], [ 3, A^2 ], [ 4, id ],
[ -4, B ] ]
[ [ -4, id ], [ 3, A ], [ 4, B*A ], [ -4, A ], [ 1, id ],
[ -3, a ], [ 4, B ], [ -1, b ] ]
[ [ -3, id ], [ 4, B*A ], [ -4, A ], [ 3, A^2 ], [ 4, B*A^2 ],
[ -4, A^2 ], [ 1, id ], [ -1, B ] ]
[ [ 4, id ], [ -4, b ], [ 1, b ], [ -3, a^2*b ], [ 4, B*a*b ],
[ -4, a*b ], [ 3, b ], [ -1, id ] ]
[ [ -3, id ], [ 4, B*A ], [ -4, A ], [ 1, id ], [ -4, a ],
[ 2, A ], [ 1, id ], [ -4, a^2 ], [ -3, B ] ]
```

We now demonstrate that this list may be reduced further.

### 6.3.3 MoveRightLogSequence

▷ MoveRightLogSequence(*mG, J, L, q*) (operation)
▷ MoveLeftLogSequence(*mG, J, L, q*) (operation)
▷ SwapLogSequence(*mG, J, p, q*) (operation)

The terms in an identity sequence may be interchanged because

$$R^w Q^v = Q^v R^{wQ^v} = Q^{v(R^w)^{-1}} R^w.$$

In the first two of these three operations L = $[p..r]$ is a range specifying a sublist K=J{[p..r]} of J, and $l$ is the length of J. The operation MoveRightLogSequence(mG,J,L,q), with $0 < p < q$ and

$q+r \leq p+l$, moves sublist K to the $q$-th position, conjugating entries in $J\{[p+1 \ldots q]\}$ and moving them all to the left.

Similarly MoveLeftLogSequence(mG,J,L,q), with $0 < q < p$ nd $r \leq l$, moves sublist K to the $q$-th position, conjugating entries in $J\{[q \ldots p-1]\}$ and moving them all to the right.

The operation SwapLogSequence(mG,J,p,q) with $p < q$ swaps a pair of terms in a sequence J by calling the two previous commands.

In all three operations the procedure is completed by a call to OnePassReduceLogSequence.

In the example the third identity is converted into the fifth by moving the third term one place right and then changing the start position, so it may be omitted.

Example

```
gap> J3 := ShallowCopy( ridsq8[3] );;
gap> PrintLnUsingLabels( J3, genfmq8, q8labs );
[ [ -4, id ], [ 2, A^2 ], [ 1, id ], [ -4, a^2 ] ]
gap> K3 := MoveRightLogSequence( mq8, J3, [3], 4 );;
gap> PrintLnUsingLabels( K3, genfmq8, q8labs );
[ [ -4, id ], [ 2, A^2 ], [ -4, A^2 ], [ 1, id ] ]
gap> J5 := ShallowCopy( ridsq8[5] );;
gap> PrintLnUsingLabels( J5, genfmq8, q8labs );
[ [ 1, id ], [ -4, id ], [ 2, A^2 ], [ -4, A^2 ] ]
gap> J5 = ChangeStartLogSequence( mq8, K3, 4 );
true
```

### 6.3.4 SubstituteLogSubsequence

▷ SubstituteLogSubsequence(*mG, K, J1, J2*) (operation)

If we move the second term in J5 to the right, we find that sublist U = [[1,id],[2,id]] is equal to V = [[4,A^2],[4,id]], with both expanding to $a^4b^4$.

Now U appears in the tenth identity, and if we replace it with V and then cancel, we obtain the empty list. So the tenth identity may be omitted.

Example

```
gap> K5 := MoveRightLogSequence( mq8, J5, [2], 3 );;
gap> PrintLnUsingLabels( K5, genfmq8, q8labs );
[ [ 1, id ], [ 2, id ], [ -4, id ], [ -4, A^2 ] ]
gap> K5a := K5{[1..2]};;
gap> K5b := InverseLogSequence( K5{[3..4]} );;
gap> K5a;K5b;
[ [ 1, <identity ...> ], [ 2, <identity ...> ] ]
[ [ 4, q8_M3^2 ], [ 4, <identity ...> ] ]
gap> J10 := ShallowCopy( ridsq8[10] );;
gap> PrintLnUsingLabels( J10, genfmq8, q8labs );
[ [ -4, id ], [ 4, B*A^2 ], [ -4, A^2 ], [ 1, id ], [ 2, id ],
[ -4, b ] ]
gap> K10 := SubstituteLogSubsequence( mq8, J10, K5a, K5b );;
gap> PrintLnUsingLabels( K10, genfmq8, q8labs );
[ [ -4, id ], [ 4, B*A^2 ], [ -4, A^2 ], [ 4, A^2 ], [ 4, id ],
[ -4, b ] ]
```

```
gap> CancelInversesLogSequence( mq8, K10 );
[  ]
```

Similarly, we may reduce the ninth identity. Initially, U does not appear as a sublist of J9. Swapping the fourth and fifth terms and conjugating by $A$ produces U, which is then replaced by V. After a cancellation, we obtain a conjugate of the fourth identity.

Example

```
gap> J9 := ShallowCopy( ridsq8[9] );;
gap> PrintLnUsingLabels( J9, genfmq8, q8labs );
[ [ -4, id ], [ 3, A ], [ -4, A ], [ 2, A^3 ], [ 1, id ],
[ -3, b ] ]
gap> K9 := MoveLeftLogSequence( mq8, J9, [5], 4 );;
gap> PrintLnUsingLabels( K9, genfmq8, q8labs );
[ [ -4, id ], [ 3, A ], [ -4, A ], [ 1, id ], [ 2, a ], [ -3, b ] ]
gap> L9 := ConjugateByWordLogSequence( mq8, K9, genfmq8[3] );;
gap> PrintLnUsingLabels( L9, genfmq8, q8labs );
[ [ -4, A ], [ 3, A^2 ], [ -4, A^2 ], [ 1, id ], [ 2, id ],
[ -3, b*A ] ]
gap> M9 := SubstituteLogSubsequence( mq8, L9, K5a, K5b);;
gap> PrintLnUsingLabels( M9, genfmq8, q8labs );
[ [ -4, A ], [ 3, A^2 ], [ -4, A^2 ], [ 4, A^2 ], [ 4, id ],
[ -3, b*A ] ]
gap> N9 := CancelInversesLogSequence( mq8, M9 );;
gap> PrintLnUsingLabels( N9, genfmq8, q8labs );
[ [ -4, A ], [ 3, A^2 ], [ 4, id ], [ -3, b*A ] ]
gap> P9 := ConjugateByWordLogSequence( mq8, N9, genfmq8[1] );;
gap> PrintLnUsingLabels( P9, genfmq8, q8labs );
[ [ -4, id ], [ 3, A ], [ 4, a ], [ -3, b ] ]
gap> P9 = ridsq8[4];
true
```

We will not, for now, attempt to reduce the list of identities further.

## 6.4 The original approach

This section describes the approach used from the earliest versions of IdRel up to version 2.38 in 2017. For version 2.39 the methods were revised so as to produce some data for infinite groups. This experimental work is described in later sections.

### 6.4.1 IdentitiesAmongRelators

▷ IdentitiesAmongRelators(*grp*) (attribute)

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

Why idrelq8 in the following example is shorter than ridsq8 above remains to be investigated!

Example

```
gap> idrelq8 := IdentitiesAmongRelators( q8 );;
gap> Length( idrelq8 );
14
gap> for i in [1..14] do
>       PrintLnUsingLabels( idrelq8[i], genfmq8, q8labs );
>    od;
[ [ -1, id ], [ 1, a ] ]
[ [ -2, id ], [ 2, b ] ]
[ [ -4, id ], [ 3, A ], [ 3, id ], [ 2, id ], [ -4, b ] ]
[ [ -4, id ], [ 2, A^2 ], [ 1, id ], [ -4, a^2 ] ]
[ [ 1, id ], [ -4, id ], [ 2, A^2 ], [ -4, A^2 ] ]
[ [ -3, id ], [ 4, B*A ], [ -4, A ], [ 1, id ], [ -3, a ] ]
[ [ -4, id ], [ 3, A ], [ 4, a ], [ -3, b ] ]
[ [ -3, id ], [ 4, B*A ], [ -4, A^2 ], [ 1, id ], [ -3, B ] ]
[ [ -4, id ], [ 4, B*A^2 ], [ -4, A^2 ], [ 1, id ], [ 2, id ],
[ -4, b ] ]
[ [ -3, id ], [ 4, B*A ], [ -4, A ], [ 3, A^2 ], [ 4, id ],
[ -4, B ] ]
[ [ -4, id ], [ 3, A ], [ -4, A ], [ 2, A^3 ], [ 1, id ],
[ -3, b ] ]
[ [ -3, id ], [ 4, B*A ], [ -4, A ], [ 1, id ], [ -4, a ],
[ 2, A ], [ 1, id ], [ -4, a^2 ], [ -3, B ] ]
[ [ -4, id ], [ 3, A ], [ 4, B*A ], [ -4, A ], [ 1, id ],
[ -3, a ], [ 4, B ], [ -1, b ] ]
[ [ -3, id ], [ 4, B*A ], [ -4, A ], [ 3, A^2 ], [ 4, B*A^2 ],
[ -4, A^2 ], [ 1, id ], [ -1, B ] ]
```

### 6.4.2 IdentityYSequences

▷ `IdentityYSequences(`*grp*`)` (attribute)

These identities are then transformed into module polynomials

$$\rho(a+ba)+\sigma(\mathrm{id}+ab+ba)-\tau(\mathrm{id}+a+A)\ ,$$

where the monoid elements are transformed into their normal forms.

The collection of saturated sets of these module polynomials is then reduced as far as possible, and the minimal set obtained returned as the `IdentityYSequences` of the group. The group relator sequences corresponding to these module polynomials form the `IdentitiesAmongRelators` for the group.

Example

```
gap> idyseq8 := IdentityYSequences( q8 );;
gap> for y in idyseq8 do
>       PrintLnYSequence( y, genfmq8, q8labs, genq8R, q8Rlabs );
>    od;
q8_Y2*(1*A), q^-1*(-1*A) + q*(1*id))
q8_Y1*(1*B), r^-1*(-1*B) + r*(1*id))
```

```
q8_Y6*(-1*id), r*(-1*id) + s*(-1*A + -1*id) + t^-1*(1*b + 1*id))
q8_Y3*(-1*a), q*(-1*a) + r*(-1*A) + t^-1*(1*A + 1*a))
q8_Y5*(-1*a), q*(-1*a) + r*(-1*A) + t^-1*(1*A + 1*a))
q8_Y7*(1*a*b), q*(1*a*b) + s^-1*(-1*a*b + -1*B) + t^-1*(-1*b) + t*(1*id))
q8_Y4*(1*A), s^-1*(-1*a*b) + s*(1*a^2) + t^-1*(-1*A) + t*(1*id))
q8_Y8*(1*a*b), q*(1*a*b) + s^-1*(-1*a*b + -1*A) + t^-1*(-1*a*B) + t*(1*id))
q8_Y10*(1*B), q*(1*B) + r*(1*B) + t^-1*(-1*B + -1*b + -1*id) + t*(1*id))
q8_Y11*(1*b), s^-1*(-1*b) + s*(1*B) + t^-1*(-1*a*B + -1*id) + t*(1*b + 1*a))
q8_Y9*(-1*a), q*(-1*a) + r*(-1*a^2) + s^-1*(1*a*B) + s*(-1*id) + t^-1*(1*a +
1*id))
q8_Y15*(1*a*b), q*(2*a*b) + r*(1*b) + s^-1*(-1*a*b + -1*A) + t^-1*(-1*a*B +
-1*B + -1*b) + t*(1*id))
q8_Y12*(1*b), q^-1*(-1*a^2) + q*(1*b) + s^-1*(-1*a*b) + s*(1*a*B) + t^-1*(
-1*a*B + -1*b) + t*(1*a + 1*id))
q8_Y13*(1*a*b), q^-1*(-1*A) + q*(1*a*b) + s^-1*(-1*a*b) + s*(1*a*B) + t^-1*(
-1*a*B + -1*b) + t*(1*a + 1*id))
```

## 6.5 Partial lists of elements

As we have seen, the procedure for obtaining identities involves applying each relator at each element of the group. Since this will not terminate when the group is infinite, we include an operation to construct words up to a given length in the monoid representation of the group.

### 6.5.1 PartialElementsOfMonoidRepresentation

▷ `PartialElementsOfMonoidRepresentation(`*G, len*`)` (operation)

As an example we take the group $\langle u, v, w \mid u^3, v^2, w^2, (uv)^2, (vw)^2 \rangle$.

Example

```
gap> F := FreeGroup(3);;
gap> u := F.1;;  v := F.2;;  w := F.3;;
gap> rels := [ u^3, v^2, w^2, (u*v)^2, (v*w)^2 ];;
gap> q0 := F/rels;;
gap> SetArrangementOfMonoidGenerators( q0, [1,-1,2,-2,3,-3] );
gap> SetName( q0, "q0" );
gap> mq0 := MonoidPresentationFpGroup( q0 );;
gap> fmq0 := FreeGroupOfPresentation( mq0 );;
gap> genfmq0 := GeneratorsOfGroup( fmq0 );;
gap> q0labs := ["u","U","v","V","w","W"];;
gap> SetMonoidPresentationLabels( mq0, q0labs );;
gap> lrws := LoggedRewritingSystemFpGroup( q0 );;
gap> pe1 := PartialElementsOfMonoidPresentation( q0, 1 );;
gap> PrintLnUsingLabels( pe1, genfmq0, q0labs );
[ id, u, U, v, w ]
gap> pe2 := PartialElementsOfMonoidPresentation( q0, 2 );;
gap> PrintLnUsingLabels( pe2, genfmq0, q0labs );
[ id, u, U, v, w, u*v, u*w, U*v, U*w, v*w, w*u, w*U ]
```

# 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, 27

[GH17] S. Gutsche and M. Horn. *AutoDoc - Generate documentation from GAP source code (Version 2017.09.15)*, 2017. GAP package, https://github.com/gap-packages/AutoDoc. 2

[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. https://www.researchgate.net/profile/Anne-Heyworth/research. 4

[Hor14] M. Horn. *GitHubPagesForGAP - Template for easily using GitHub Pages within GAP packages (Version 0.1)*, 2014. GAP package, https://github.com/fingolfin/GitHubPagesForGAP/. 2

[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, 27, 36

[LN17] F. Lübeck and M. Neunhöffer. *GAPDoc (version 1.6)*. RWTH Aachen, 2017. GAP package, https://www.math.rwth-aachen.de/~Frank.Luebeck/gap/GAPDoc/index.html. 2

# Index


=,+,* for module polynomials, 25
=,+,* for monoid polynomials, 21

AddTermModulePoly, 25
AddTermMonoidPoly, 20
ArrangementOfMonoidGenerators, 7

CancelImmediateInversesLogSequence, 32
CancelInversesLogSequence, 32
ChangeStartLogSequence, 31
Coeffs, 20
ConjugateByWordLogSequence, 31

ElementsOfMonoidPresentation, 12
ExpandLogSequence, 30

FreeGroupOfPresentation, 7
FreeRelatorGroup, 7
FreeRelatorHomomorphism, 7

GroupRelatorsOfPresentation, 8

HomomorphismOfPresentation, 8

IdentitiesAmongRelators, 36
IdentityRelatorSequences, 28
IdentityYSequences, 37
InitialLoggedRulesOfPresentation, 14
InitialRulesOfPresentation, 9
InverseLogSequence, 32
InverseRelatorsOfPresentation, 8

KnuthBendix, 12

LeadCoeffMonoidPoly, 20
LeadMonoidPoly, 24
LeadTerm
    for module polynomials, 24
    for monoid polynomials, 20
Length
    for module polynomials, 24
    for monoid polynomials, 21
LoggedKnuthBendix, 16
LoggedOnePassKB, 14
LoggedOnePassReduceWord, 17
LoggedReduceWordKB, 17
LoggedRewriteReduce, 15
LoggedRewritingSystemFpGroup, 16
LogSequenceLessThan, 29
LogSequenceRewriteRules, 33

ModulePoly
    with input [gen,poly] list, 23
    with input gens, polys, 23
Monic, 20
MonoidPoly, 19
MonoidPolyFromCoeffsWords, 19
MonoidPresentationFpGroup, 7
MonoidPresentationLabels, 7
MoveLeftLogSequence, 34
MoveRightLogSequence, 34

One, 24
OnePassKB, 11
OnePassReduceLogSequence, 33
OnePassReduceWord, 10

PartialElementsOfMonoidRepresentation, 38
PrintLnModulePoly
    input object, [gens,labels] for the group, ditto relators, 24
PrintLnUsingLabels, 8
PrintModulePoly
    input object, [gens,labels] for the group, ditto relators, 24
PrintUsingLabels, 8

ReduceLogSequences, 30
ReduceMonoidPoly, 22
ReduceWordKB, 10

RewriteReduce, 11
RewritingSystemFpGroup, 9
RootIdentities, 27
RootPositions, 27

ShorterLoggedRule, 17
SubstituteLogSubsequence, 35
SwapLogSequence, 34

Terms
    for module polynomials, 24
    for monoid polynomials, 20

Words, 20

ZeroModulePoly, 23
ZeroMonoidPoly, 19