Equations

Version 0.2.0

20 Dec 2017

Thorsten Groth

Thorsten Groth Email: thorsten.groth@mathematik.uni-goettingen.de

Copyright

Contents

Chapter 1

Installation

The package is installed by unpacking the archive in the `pkg/` directory of your GAP installation.

Example
```
gap> LoadPackage("equations");
true
```

Chapter 2

Example Session

We show some examples for using this package. The used methods are described in the latter chapter.

2.1 Normal form of equations

Let us consider some equations over the alternating group A_5. We start with defining the group in which our equations live in:

Example
```
gap> LoadPackage("equations");
true
gap> A5 := AlternatingGroup(5);
Alt( [ 1 .. 5 ] )
gap> EqG := EquationGroup(A5);
<free product group>
```

Now we enter the equation $E = X_2(1,2,3)X_1^{-1}X_2^{-1}(1,3)(4,5)X_3X_1X_3^{-1}$.

Example
```
gap> g := (1,2,3);;h := (1,3)(4,5);;
gap> vars := VariablesOfEquationGroup(EqG);
[ FreeProductElm([ X1 ]), FreeProductElm([ X2 ]), ... ]
gap> x1 := vars[1];; x2 := vars[2];; x3 := vars[3];;
gap> eq := Equation(x2*g*x1^-1*x2^-1*h*x3*x1*x3^-1);
Equation in [ X1, X2, X3 ]
gap> Print(eq);
FreeProductElm([ X2, (1,2,3), X1^-1*X2^-1, (1,3)(4,5), X3*X1*X3^-1 ])
```

Let us see what the normal form of this equation is:

Example
```
gap> Genus(eq);
1
gap> nf := NormalFormOfEquation(eq);
Equation in [ X1, X2, X3 ]
gap> Print(nf);
FreeProductElm([ X1^-1*X2^-1*X1*X2*X3^-1, (1,2,3), X3, (1,3)(4,5) ])
```

We know a solution for this normal form: $s: X_1 \mapsto (1,2,4),\ X_2 \mapsto (1,2,5),\ X_3 \mapsto ()$.

Example
```
gap> s:=EquationEvaluation(EqG,EquationVariables(eq),[(1,2,4),(1,2,5),()]);
[ X1, X2, X3 ]"->"[ (1,2,4), (1,2,5), () ]
gap> IsSolution(s,nf);
true
gap> nf^s;
()
gap> IsSolution(s,eq);
false
gap> eq^s;
(1,2,4,3,5)
```

Let us compute the solution for E.

Example
```
gap> sE:= NormalizingHomomorphism(nf)*s;
CompositionMapping( [ X1, X2, X3 ]->[ (1,2,4), (1,2,5), () ],
 CompositionMapping( [ X3, X2 ]->[ X2, X3 ], CompositionMapping(
CompositionMapping( CompositionMapping( [ X2 ]->[ (X2) ], [ X2 ]->[ X2^-1 ] ),
[ X3, X1 ]->[ <Free product element of length 5>,
  <Free product element of length 3> ] ), CompositionMapping( [ X1 ]->[ X1 ],
[ X3 ]->[ X3^-1 ] ) ) ) )
gap> IsSolution(sE,eq);
true;
gap> List(EquationVariables(eq),x->ImageElm(sE,x));
[ (2,3,4), (), (1,2,5,4,3) ]
```

Thus $s_E: X_1 \mapsto (2,3,4),\ X_2 \mapsto (),\ X_3 \mapsto (1,2,5,4,3)$ is a solution for the equaition E

2.2 Decomposition

Let us now study equations over groups acting on a rooted tree without having any explicitly given group in mind. Say $G \leq Aut(\{1,2\}^*)$ and $g,h \in G$ and assume we want to see how the decomposition Φ_γ of the equation $E = [X,Y]g^Z h$ looks like. This decomposition will depend on the activity of g and on γ_{act}.

Example
```
gap> F := FreeGroup("X","Y","Z");; x:=F.1; y:=F.2; z:=F.3;
X
Y
Z
gap> G := FreeGroup("g","h");; g:=G.1; h := G.2;
g
h
gap> S2 := [(),(1,2)];
gap> EqG := EquationGroup(G,F);;
gap> eq := Equation(EqG,[Comm(x,y)*z^-1,g,z,h]);
Equation in [ X, Y, Z ]
gap> PhiE := [];
[ ]
```

```
gap> for actg in S2 do
>       DeqG := DecompositionEquationGroup(EqG,2,[actg,()]);;
>       for gamma_act in Cartesian([S2,S2,S2]) do
>         Add(PhiE,DecompositionEquation(DeqG,eq,gamma_act));
>       od;
>    od;
gap> Print(PhiE[1]);
Equation([ FreeProductElm([ X1^-1*Y1^-1*X1*Y1*Z1^-1, g1, Z1, h1 ]),
      FreeProductElm([ X2^-1*Y2^-1*X2*Y2*Z2^-1, g2, Z2, h2 ]) ])
gap> Print(PhiE[16]);
Equation([ FreeProductElm([ X2^-1*Y1^-1*X1*Y2*Z2^-1, g2, Z1, h2 ]),
      FreeProductElm([ X1^-1*Y2^-1*X2*Y1*Z1^-1, g1, Z2, h1 ]) ])
```

We see that for some (indeed for all but the first two cases) the states of the decomposition do not form independent systems. Let us see how an equivalent independent system looks like and find out which genus the corresponding equations have:

Example

```
gap> Apply(PhiE,E->DisjointFormOfDecomposedEquation(E));
gap> Print(PhiE[16]);
Equation([ FreeProductElm([ X2^-1*Y1^-1*Y2^-1*X2*Y1*Z1^-1, g1, Z2, h1, Y2*Z2^-1,
 g2, Z1, h2 ]), FreeProductElm([  ]) ])
gap> Genus(EquationComponent(PhiE[16],1));
2
gap> List(PhiE,E->Genus(EquationComponent(E,1)));
[ 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2 ]
```

2.3 Using the fr package

Finally let us do some computations in the Grigorchuk group. For example let us compute a solution for the equation $E = [X,Y]dacab$.

Example

```
gap> LoadPackage("fr");;
gap> G := GrigorchukGroup;;
gap> a:= G.1;; b:=G.2;; c:=G.3;; d:= G.4;;
gap> EqG := EquationGroup(G);;
gap> x:=EqG.5;y:=EqG.6;
(X1)
(X2)
gap> eq := Equation(Comm(x,y)*d*a*c*a*b);
<Equation in [ X1, X2 ]>
gap> gamma_a := GroupHomomorphismByImages(
>       Group(EquationVariables(eq)),SymmetricGroup(2),[(),(1,2)]);
[ X1, X2 ] -> [ (), (1,2) ]
gap> neq := DecompositionEquation(eq,gamma_a);
DecomposedEquation in [ Xn1, Xn2, Xn3, Xn4 ]
gap> deq := DisjointFormOfDecomposedEquation(neq);
DecomposedEquation in [ Xn2, Xn3, Xn4 ]
gap> nf := NormalFormOfEquation(EquationComponent(deq,1));
<Equation in [ Xn1, Xn2, Xn3 ]>
```

```
gap> s := EquationEvaluation(DecomposedEquationGroup(EqG),
>            EquationVariables(nf),[d,b,b]);
[ Xn1, Xn2, Xn3 ]->[ d, b, b ]
gap> IsSolution(s,nf);
true
gap> IsSolution(NormalizingHomomorphism(nf)*s,EquationComponent(deq,1));
true
gap> sol := DisjointFormHomomorphism(deq)*NormalizingHomomorphism(nf)*s;;
gap> ForAll(EquationComponents(neq),E->IsSolution(sol,E));
true;
gap> imgs := List(EquationVariables(neq),x->ImageElm(sol,x));
[ <Mealy element on alphabet [ 1 .. 2 ] with 6 states>,
  <Mealy element on alphabet [ 1 .. 2 ] with 7 states>, b^-1,
  <Mealy element on alphabet [ 1 .. 2 ] with 9 states> ]
gap> soleq := EquationEvaluation(EqG,EquationVariables(eq),
                [ComposeElement([imgs[1],imgs[2]],()),
                 ComposeElement([imgs[3],imgs[4]],(1,2))] );
[ X1, X2 ]->[ <Mealy element on alphabet [ 1 .. 2 ] with 9 states>,
  <Mealy element on alphabet [ 1 .. 2 ] with 10 states> ]
gap> IsSolution(soleq,eq);
true;
```

Chapter 3

FreeProducts

3.1 Construction

This package installs some new method for the command `FreeProduct`. Before it was only possible to construct free products of finitely presented groups.

3.1.1 FreeProductOp

▷ `FreeProductOp(`*`list, f.g., free, group`*`)` (operation)

Returns: The free product of all groups in *`list`*.

This is the method of choice if *`list`* contains at least one finetely generated free group but not only free groups.

3.1.2 FreeProductOp

▷ `FreeProductOp(`*`list, inf.g., free, group`*`)` (operation)

Returns: The free product of all groups in *`list`*.

We choose this is the method if *`list`* contains at least one infinetely generated free group but not only free groups.

3.1.3 FreeProductOp

▷ `FreeProductOp(`*`list, group`*`)` (operation)

Returns: The free product of all groups in *`list`*.

This method does always work. We refer to a more specific method if all of the groups are finitely presented. I.e. they are in the filter `IsFpGroup` and finitely generated.

If the resulting group was constructed by one of the new methods they will be in the following filter: `IsGeneralFreeProduct`

3.2 Filters

3.2.1 IsGeneralFreeProduct

▷ `IsGeneralFreeProduct(`*`obj`*`)` (filter)

Returns: `true` if *`obj`* is a general free product.

This filter can be used to check whether a given group was created as general free product.

3.2.2 IsFreeProductElm

▷ `IsFreeProductElm(obj)` (filter)

3.2.3 IsFreeProductHomomorphism

▷ `IsFreeProductHomomorphism(obj)` (filter)

3.3 Construction

3.3.1 GeneralFreeProduct (group)

▷ `GeneralFreeProduct(group)` (operation)

Returns: A a new general free product isomorphic to *group*.

Takes a group which has free product information stored and returns a new group which lies in the filter `IsGeneralFreeProduct`. The returned groups represents the free product of the groups in `FreeProductInfo.groups`.

Example
```
gap> S2 := SymmetricGroup(2);; SetName(S2,"S2");
gap> S3 := SymmetricGroup(3);; SetName(F2,"F2");
gap> G := FreeProduct(S2,S3);
<fp group on the generators [ f1, f2, f3 ]>
gap> G := GeneralFreeProduct(G);
S2*S3
```

3.3.2 GeneratorsOfGroup (group)

▷ `GeneratorsOfGroup(group)` (operation)

Returns: The generators of *group*.

3.3.3 \= (G,H)

▷ `\=(group, group)` (operation)

Returns: True if the free factors of the groups *G* and *H* are equal.

3.4 Elements

3.4.1 FreeProductElm (group,list,list)

▷ `FreeProductElm(group, word, factors)` (operation)

▷ `FreeProductElmLetterRep(group, word, factors)` (operation)

Returns: A new element in the group *group*.

This function constructs a new free product element, belonging to the group *group*.

words is a dense list of elements of any of the factors of *group*.

factors is a list of integers. *word*[i] must lie in the factor *factors*[*i*] of *group*. If this is not the case an error is thrown.

FreeProductElmLetterRep does not simplify the word by multipliying neighbored equal factor elements but stores the letters as given.

Example
```
gap> F2 := FreeGroup(2);; SetName(F2,"F2");
gap> S4 := SymmetricGroup(4);; SetName(S4,"S4");
gap> G := FreeProduct(F2,S4);
F2*S4
gap> e := FreeProductElm(G,[F2.1,F2.2,(1,2),F2.1],[1,1,2,1]);
FreeProductElm of length 3
gap> Print(e^2);
FreeProductElm([ f1*f2, (1,2), f1^2*f2, (1,2), f1 ])
gap> Print(FreeProductElmLetterRep(G,[F2.1,F2.2,(1,2),F2.1],[1,1,2,1]));
FreeProductElm([ f1, f2, (1,2), f1 ])
```

There are two representations for this kind of elements.

3.4.2 IsFreeProductElmRep

▷ IsFreeProductElmRep(*obj*) (filter)
▷ IsFreeProductElmLetterRep(*obj*) (filter)

Returns: true if *obj* is a general free product element in standard/letter storing representation.

3.5 Basic operations

3.5.1 * (freeproductelm,freeproductelm)

▷ *(*e1*, *e2*) (operation)

Returns: The product of the two elements.

3.5.2 * (freeproductelm,group elm)

▷ *(*e1*, *e2*) (operation)

Returns: The product of *e1* and the image of *e2* under the embedding into the free product group.

Only works if *e2* lies in one of the free factors of the free product group.

3.5.3 InverseOp (freeproductelm)

▷ InverseOp(*elm*) (operation)

Returns: The inverse element

3.5.4 OneOp (freeproductelm)

▷ OneOp(*elm*) (operation)

Returns: The identity element

3.5.5 \= (freeproductelm,freeproductelm)

▷ \=(*e1*, *ee2*) (operation)

Returns: True if the two elements are equal.

#

3.5.6 Length (freeproductelm)

▷ Length(*e1*) (operation)

Returns: The length of the list that stores the elements of the free factors

3.5.7 \[\] (freeproductelm,integer)

▷ \[\](*e1*, *i*) (operation)

Returns: The free product element consisting only of the *i*-th entry of the underlying list of elements.

3.5.8 Position (freeproductelm)

▷ Position(*e1*) (operation)

Returns: The position of the element *e1* in the underlyig list.

Example
```
gap> F2 := FreeGroup(2);; SetName(F2,"F2");
gap> S4 := SymmetricGroup(4);; SetName(S4,"S4");
gap> G := FreeProduct(F2,S4);
F2*S4
gap> e := FreeProductElm(G,[F2.1,F2.2,(1,2),F2.1],[1,1,2,1]);;Print(e);
FreeProductElm([ f1*f2, (1,2), f1 ])
gap> Length(e);
3
gap> Position(e,(1,2));
2
gap> Print(e[1]);
FreeProductElm([ f1*f2 ])
```

3.6 Homomorphisms

3.6.1 FreeProductHomomorphism (group,group,list)

▷ FreeProductHomomorphism(*source*, *target*, *homs*) (operation)

Returns: A new group homomorphism from *source* to *target*.

This function constructs a new group homomorphism from the general free product group *source* to the general free product group *target* by mapping the factor i by the group homomorphism *homs*[i] to the ith factor of *target*.

homs is a dense list of group homomorphisms where the source of *homs*[i] must be the ith factor of *source* and the range of *homs*[i] must be the ith factor of *target*.

Example
```
gap> F2 := FreeGroup(2);; SetName(F2,"F2");
gap> S4 := SymmetricGroup(4);; SetName(S4,"S4");
```

```
gap> A4 := AlternatingGroup(4);; SetName(A4,"A4");
gap> G := FreeProduct(F2,S4); H := FreeProduct(F2,A4);
F2*S4
F2*A4
gap> hf := GroupHomomorphismByImages(F2,F2,[F2.2,F2.1]);;
gap> hg := GroupHomomorphismByFunction(S4,A4,s->Comm(s,S4.2));;
gap> h := FreeProductHomomorphism(G,H,[hf,hg]);
<mapping: F2*S4 -> F2*A4 >
gap> e := FreeProductElm(G,[F2.1,F2.2,(1,2),F2.1],[1,1,2,1]);
FreeProductElm of length 3
gap> Print(e^h);
FreeProductElm([ f2*f1*f2 ])
```

3.6.2 IsGeneralFreeProductRep

▷ `IsGeneralFreeProductRep(`*obj*`)` (filter)

Returns: `true` if *obj* is a general free product element in standard/letter storing representation.

3.7 Other operations

3.7.1 Abs (assocword)

▷ `Abs(`*obj*`)` (operation)

Returns: An assocword without inverses of generators

In the word *obj* all occurencies of inverse generators are replaced by the coresponding generators.

Example

```
gap> F2 := FreeGroup(2);; SetName(F2,"F2");
gap> w := F2.1^-1*F2.2*F2.1*F2.2^-1;
f1^-1*f2*f1*f2^-1
gap> Abs(w);
(f1*f2)^2
```

3.7.2 \in (elm,list)

▷ `\in(`*elm*`, `*list*`)` (operation)

Returns: true if elm is in the infinite list *list*

Chapter 4

Equations

We fix a set $\mathscr{X}$ and call its elements *variables*. We assume that $\mathscr{X}$ is infinite countable, is well ordered, and its family of finite subsets is also well ordered, by size and then lexicographic order. We denote by $F_{\mathscr{X}}$ the free group on the generating set $\mathscr{X}$.

Let G be a group. The *equation group* will be the free product $G * F_{\mathscr{X}}$ and the elements belonging to G will be called *constants*.

A G-equation is an element E of the group $F_{\mathscr{X}} * G$ regarded as a reduced word. For E a G-equation, its set of *variables* $\mathrm{Var}(E) \subset \mathscr{X}$ is the set of symbols in $\mathscr{X}$ that occur in it; namely, $\mathrm{Var}(E)$ is the minimal subset of $\mathscr{X}$ such that E belongs to $F_{\mathrm{Var}(E)} * G$.

A *quadratic* equation is an equation in which each variable $X \in \mathrm{Var}(E)$ occurs exactly twice. A quadratic equation is called *oriented* if for each variable $X \in$ Var(X) both letters X and X^{-1} occure in the reduced word E.

4.1 Construction

4.1.1 IsEquationGroup

▷ `IsEquationGroup(obj)` (filter)

Returns: `true` if *obj* is a general free product over to groups `G,F` where `F` is a free group.

The free factor `F` represents the group of variables for the equations.

4.1.2 EquationGroup (group,group)

▷ `EquationGroup(G, F)` (operation)

Returns: A a new *G*-group for equations over *G*.

Uses the `FreeProduct` method to create the free product object. The second argument *F* must be a free group.

Example
```
gap> S2 := SymmetricGroup(2);; SetName(S2,"S2");
gap> F := FreeGroup(infinity,"xn",["x1","x2"]);;SetName(F,"F");
gap> EqG := EquationGroup(S2,F);
S2*F
```

4.1.3 EquationGroup (group)

▷ `EquationGroup(G)` (operation)

Returns: A a new `G`-group for equations over `G`.

Uses the `FreeProduct` method to create the free product object of the given group and the free group on infinitely many generators

Example
```
gap> S2 := SymmetricGroup(2);; SetName(S2,"S2");
gap> EqG := EquationGroup(S2);
S2*Free(oo)
```

4.1.4 VariablesOfEquationGroup (group)

▷ `VariablesOfEquationGroup(G)` (attribute)

Returns: A list of the embedded free generators of the free facotor

If the equation group `G` was constructed with an infinitely generated free group as the group of variables, this returns an infinite list of generators.

4.1.5 ConstantsOfEquationGroup (group)

▷ `ConstantsOfEquationGroup(G)` (attribute)

Returns: The image of the embedding of the group of constants in `G`

4.1.6 Equation (group,list)

▷ `Equation(G, L)` (operation)

Returns: A a new element of the equation group `G`

Creates a `FreeProductElm` from the list `L`. By default this elements will be cyclicaly reduced.

Example
```
gap> F2 := FreeGroup(2);; SetName(F2,"F2");
gap> S4 := SymmetricGroup(4);; SetName(S4,"S4");
gap> G := EquationGroup(S4,F2);
S4*F2
gap> e := Equation(G,[F2.1,F2.2,(1,2),F2.1]);
Equation in [ f1, f2 ]
gap> Print(e);
FreeProductElm([ f1*f2, (1,2), f1 ])
```

4.1.7 Equation (free product elm)

▷ `Equation(elm)` (operation)

Returns: A a new element of the equation group `G`

Creates a `FreeProductElm` from the `FreeProductElm` `elm`. By default this elements will be cyclicaly reduced.

Example
```
gap> G := EquationGroup(SymmetricGroup(4));
<free product group>
gap> e := Equation(G.4*G.1*G.2*G.3);
<Equation in [ X1, X2 ]>
```

```
gap> Print(e);
FreeProductElm([ X2, (2,3,4), X1 ])
```

4.1.8 EquationVariables (groupelement)

▷ `EquationVariables(E)` (attribute)

Returns: A list of all variables occuring in E.

The elements of the result are elements of the group of variables in the `EquationGroup`. See in contrast the attribute `EquationVariablesEmbedded`.

4.1.9 EquationVariablesEmbedded (groupelement)

▷ `EquationVariablesEmbedded(E)` (attribute)

Returns: A list of all variables occuring in E.

The elements of the result are elements of the `EquationGroup`. and thus `FreeProductElm`s of length 1. See in contrast the attribute`EquationVariables`.

4.1.10 EquationLetterRep (equation)

▷ `EquationLetterRep(E)` (attribute)

Returns: A a new element of the equation group G in letter representation which is equal to E

In the standard representation of an equation the elements of the free group that are not devided by a constant are collected. In the letter representation they are seperate letters.

Example
```
gap> F2 := FreeGroup(2);; SetName(F2,"F2");
gap> S4 := SymmetricGroup(4);; SetName(S4,"S4");
gap> G := EquationGroup(S4,F2);
S4*F2
gap> e := Equation(G,[F2.1,F2.2,(1,2),F2.1]);
Equation in [ f1, f2 ]
gap> Print(e);
FreeProductElm([ f1*f2, (1,2), f1 ])
gap> Print(EquationLetterRep(e));
FreeProductElm([ f1, f2, (1,2), f1 ])
```

4.1.11 EquationLetterRep (group,list)

▷ `EquationLetterRep(G, L)` (attribute)

Returns: Creates a new equation in letter representation

4.1.12 IsQuadraticEquation (equation)

▷ `IsQuadraticEquation(E)` (property)

Returns: `true` if E is an quadratic equation.

4.1.13 IsOrientedEquation (equation)

▷ `IsOrientedEquation(E)` (property)

Returns: `true` if E is an oriented quadratic equation.

4.2 Homomorphisms

An *evaluation* is a G-homomorphism $e{:}F_{\mathscr{X}} * G \to G$. A *solution* of an equation E is an evaluation s satisfying $s(E) = 1$. If a solution exists for E then the equation E is called *solvable*. The set of elements $X \in \mathscr{X}$ with $s(X) \neq 1$ is called the *support* of the solution.

4.2.1 EquationHomomorphism (group,list,list)

▷ `EquationHomomorphism(G, vars, imgs)` (operation)

Returns: A a new homomorphism from `G` to `G`

If `G` is the group $H * F_X$ the result of this command is a H-homomorphism that maps the i-th variable of the list `vars` to the i-th member of `imgs`. Therefore `vars` can be a list without duplicates of variables. The list `imgs` can contain elements of the following type:

- Element of the group F_X
- Elements of the group H
- Lists of elements from the groups F_X and H. The list is then regarded as the corresponding word in G
- Elements of the group G

Example

```
gap> F3 := FreeGroup(3);; SetName(F3,"F3");
gap> S4 := SymmetricGroup(4);; SetName(S4,"S4");
gap> G := EquationGroup(S4,F3);
S4*F3
gap> e := Equation(G,[Comm(F3.2,F3.1)*F3.3^2,(1,2)]);
Equation in [ f1, f2, f3 ]
gap>  h := EquationHomomorphism(G,[F3.1,F3.2,F3.3],
> [F3.1*F3.2*F3.3,(F3.2*F3.3)^(F3.1*F3.2*F3.3),(F3.2^-1*F3.1^-1)^F3.3]);
[ f1, f2, f3 ]"->"[ f1*f2*f3, f3^-1*f2^-1*f1^-1*f2*f3*f1*f2*f3, f3^-1*f2^-1*f1^-1*f3 ]
gap> Print(e^h);
FreeProductElm([ f1^2*f2^2*f3^2, (1,2) ])
```

4.2.2 EquationEvaluation (group,list,list)

▷ `EquationEvaluation(G, vars, imgs)` (operation)

Returns: A a new evaluation from `G`

Works the same as *EquationHomomorphism* but the target of the homomorphism is the group of constants and all variables which are not specified in in `vars` are maped to the identity. Hence the only allowed input for `imgs` are elements of the group of constants.

Example

```
gap> F3 := FreeGroup(3);; SetName(F3,"F3");
gap> S4 := SymmetricGroup(4);; SetName(S4,"S4");
gap> G := EquationGroup(S4,F3);
S4*F3
gap> e := Equation(G,[Comm(F3.2,F3.1)*F3.3^2,(1,2,3)]);
Equation in [ f1, f2, f3 ]
gap>  h := EquationHomomorphism(G,[F3.1,F3.2,F3.3],[(),(),(1,2,3)]);
[ f1, f2, f3 ]"->"[ (), (), (1,3,2) ]
```

```
gap>  he := EquationEvaluation(G,[F3.1,F3.2,F3.3],[(),(),(1,2,3)]);
MappingByFunction( S4*F3, S4, function( q ) ... end )
gap> e^he;
()
gap> IsSolution(he,e);
true
```

4.3 Normal Form

For $m,n \geq 0$, $X_i, Y_i, Z_i \in \mathscr{X}$ and $c_i \in G$ the following two kinds of equations are called in *normal form*:

$$\begin{aligned} O_{n,m}: &\quad [X_1,Y_1][X_2,Y_2]\cdots[X_n,Y_n]c_1^{Z_1}\cdots c_{m-1}^{Z_{m-1}}c_m \\ U_{n,m}: &\quad X_1^2X_2^2\cdots X_n^2c_1^{Z_1}\cdots c_{m-1}^{Z_{m-1}}c_m \,. \end{aligned}$$

The form $O_{n,m}$ is called the oriented case and $U_{n,m}$ for $n > 0$ the unoriented case. The parameter n is referred to as *genus* of the normal form of an equation. The pair (n,m) will be called the *signature* of the quadratic equation. It was proven by Commerford and Edmunds ([CJE81]) that every quadratic equation is isomorphic to one of the form $O_{n,m}$ or $U_{n,m}$ by an G-isomorphism.

4.3.1 NormalFormOfEquation (equation)

▷ `NormalFormOfEquation(E)` (attribute)

Returns: The normal form of the equation `E`

The argument `E` needs to be a quadratic equation. For each such equation there exists an equivalent equation in normal form.

The result is an equation in one of the forms $O_{n,m}, U_{n,m}$ equivalent to the equation `E`. The resulting equation has the attributes *NormalizingHomomorphism* and *NormalizingInverseHomomorphism* storing in the first case the homomorphism that maps `E` to the result and in the second case the inverse of this homomorphism.

4.3.2 NormalizingHomomorphism (equation)

▷ `NormalizingHomomorphism(E)` (attribute)

Returns: The `EquationHomomorphism` that maps to `E`

Only available if `E` was obtained via `NormalFormOfEquation`.

4.3.3 NormalizingInverseHomomorphism (equation)

▷ `NormalizingInverseHomomorphism(E)` (attribute)

Returns: The `EquationHomomorphism` that maps from `E`

Only available if `E` was obtained via `NormalFormOfEquation`. This is the inverse homomorphism to `NormalizingInverseHomomorphism(E)` #

Example

```
gap> F3 := FreeGroup("x","y","z");; SetName(F3,"F3");
gap> S4 := SymmetricGroup(4);; SetName(S4,"S4");
gap> G := EquationGroup(S4,F3);
S4*F3
gap> e := Equation(G,[Comm(F3.2,F3.1)*F3.3^2,(1,2)]);
Equation in [ x, y, z ]
```

```
gap>  nf := NormalFormOfEquation(e);;
gap> Print(nf);
FreeProductElm([ x^2*y^2*z^2, (1,2) ])
gap> e^NormalizingHomomorphism(nf)=nf;
true
gap> nf^NormalizingInverseHomomorphism(nf)=e;
true
```

4.3.4 Genus (equation)

▷ `Genus(E)` (operation)

Returns: The integer that is the genus of the equation

4.3.5 EquationSignature (equation)

▷ `EquationSignature(E)` (operation)

Returns: The list `[n,m]` of integers that is the signature of the equation

Chapter 5

FR-Equations

5.1 Decomposable equations

For self-similar groups one strategy to solve equations is to consider the inherit equations by passing to states. To use this methods the package FR ([Bar16]) from Laurent Bartholdi is needed.

Let G be a group which lies in the filter *IsFRGroup* and which admitts an embedding $\psi: G \to \tilde{G} \wr S_n$ where $\tilde{G}$ is the group generated by the states of the group G. Note that if G is a *self-similar* group then $G \simeq \tilde{G}$. Further let F_X be the free group on the generating set X. Given an equation group $G * F_X$ we will the fix n natural embeddings $\varphi_i: F \to F_{X^n}$ and call the group $(\tilde{G} * F_{X^n}) \wr S_n)$ the *decomposition equation group* of $G * F_X$. The decomposition of an equation e with variables $x_1, \ldots, x_k$ with respect to a choice of activities $\sigma(x_i) \in S_n$ for each variable x_i is the image of e under the homomorphism

$$\begin{aligned} \Phi_\sigma: G * F_X &\to (\tilde{G} * (F_{X^n}) \wr S_n \\ x_i &\mapsto \varphi_i(x_i) \cdot \sigma(x_i) \\ g &\mapsto \psi_i(x_i) \end{aligned}$$

5.1.1 DecomposedEquationGroup (group)

▷ `DecomposedEquationGroup(G)` (attribute)

Returns: A new *EquationGroup*.

This method needs `G` to be an equation group where the group of constants is an fr-group. For `G` a group with free constant group see `DecomposedEquationGroup` (5.1.1). If F is the free group on the generating set X then the free group on the gerating set X^n is isomorphic to F^{*n} the n-fold free product of F .

This method returns the *EquationGroup* $G * F^{*n}$.

▷ `DecomposedEquationGroup(G, deg, acts)` (operation)

Returns: A new *EquationGroup*.

This method needs `G` to be an equation group where the group of constants is a free group on $n < \infty$ generators. The integer `deg` is the number of states each element will have. The list `acts` should be of length n and all elements should be permutation of `deg` elements. These will represent the activity of the generators of the free group.

5.1.2 DecompositionEquation (equation,group homorphism)

▷ `DecompositionEquation(E, sigma)` (operation)

Returns: A new equation in `G` which is the decomposed of the equation `E`.

The equation E needs to be a member of a EquationGroup $H = K * F$ where K is an FRGroup.

The argument *sigma* needs to be a group homomorphism $\sigma\colon F \to S_n$. Alternatively it can be a list of elements of S_n it is then regarded as the group homomorphism that maps the i-th variable of eq to the i-th element of the list.

The representation of the returned equation stores a list of words such that the i-th word represents an element in $G * \phi_i(F)$.

Example
```
gap> F := FreeGroup(1);; SetName(F,"F");
gap> G := EquationGroup(GrigorchukGroup,F);
GrigorchukGroup*F
gap>  sigma := GroupHomomorphismByImages(F,SymmetricGroup(2),[(1,2)]);
[ f1 ] -> [ (1,2) ]
gap>  e := Equation(G,[F.1^2,GrigorchukGroup.2]);
Equation in [ f1 ]
gap>  de := DecompositionEquation(e,sigma);
DecomposedEquation in [ f11, f12 ]
gap> Print(de);
Equation([ FreeProductElm([ f11*f12,a ]), FreeProductElm([ f12*f11,c ]) ])
```

5.1.3 DecompositionEquation (EquationGroup,equation,group homomorphism)

▷ DecompositionEquation(*G, E, sigma*) (operation)

Returns: A new equation in G which is the decomposed of the equation E.

The group G needs to be a *DecompositionEqationGroup(H)*, the equation E needs to be a member of the EquationGroup $H = K * F$.

The argument *sigma* needs to be a group homomorphism $\sigma\colon F \to S_n$. Alternatively it can be a list of elements of S_n it is then regarded as the group homomorphism that maps the i-th variable of eq to the i-th element of the list.

The representation of the returned equation stores a list of words such that the i-th word represents an element in $G * \phi_i(F)$.

Example
```
gap> F := FreeGroup("x1","x2");; SetName(F,"F");
gap> G := FreeGroup("g");; SetName(G,"G");
gap> eG := EquationGroup(G,F);
G*F
gap> DeG := DecompositionEquationGroup(eG,2,[(1,2)]);
G*G*F*F
gap>  e := Equation(eG,[Comm(F.1,F.2),G.1^2]);
Equation in [ x1, x2 ]
gap>  Print(DecompositionEquation(DeG,e,[(),()]));
Equation([ FreeProductElm([ x11^-1*x21^-1*x11*x21, g1*g2 ]),
 FreeProductElm([ x12^-1*x22^-1*x12*x22, g2*g1 ]) ])
```

5.1.4 EquationComponent (equation,int)

▷ EquationComponent(*E, i*) (operation)

Returns: The i-th component of the decomposed equation E.

Denote by p_i the natural projection $(G * F_{X^n})^n \rtimes S_n \to G * F_{X^n}$ to the i-th factor of the product. Given a decomposed Equation E and an integer $0 < i \leq n$ this method returns $p_i(E)$.

▷ `EquationComponents(E)` (operation)

Returns: The list of all components of the decomposed equation `E`.

Denote by p_i the natural projection $p_i\colon (G * F_{X^n})^n \rtimes S_n \to G * F_{X^n}$ to the i-th factor of the product. Given a decomposed Equation `E` this method returns the list $[p_1(E), p_2(E), \ldots, p_n(E)]$.

▷ `EquationActivity(E)` (operation)

Returns: The activity of the decomposed equation `E`.

Denote by *act* the natural projection $(G * F_{X^n}) \wr S_n \to S_n$. Given a decomposed Equation `E` this method returns $act(E)$.

5.1.5 DecomposedEquationDisjointForm (equation)

▷ `DecomposedEquationDisjointForm(E)` (attribute)

Returns: A decomposed equation that is a disjoint system.

If `E` is a decomposed equation there may be an overlap of the set of variables of some components. If `E` is a quadratic equation there is an equation homomorphism φ that maps each component to a new quadratic equation. Hence all maped components have pairwise disjoint sets of variables. This method computes such an homomorphism φ such that the solvability of the system of components remains unchanged. If s is a solution for the new system of components, then $s \circ \varphi$ is a solution for the old system.

5.1.6 DisjointFormOfDecomposedEquation (equation)

▷ `DisjointFormOfDecomposedEquation(E)` (attribute)

Returns: A decomposed Equation with disjoint components.

If `E` is a decomposed equation there may be an overlap of the set of variables of some components. If `E` is a quadratic equation there is an equation homomorphism φ that maps each component to a new quadratic equation. Hence all maped components have pairwise disjoint sets of variables. This method computes such an homomorphism φ such that the solvability of the system of components remains unchanged. If s is a solution for the new system of components, then $s \circ \varphi$ is a solution for the old system.

The method returns a the neq decomposed equation, that has the attribute `DisjointFormHomomorphism` that is the the homomorphism φ.

5.1.7 DisjointFormHomomorphism (equation)

▷ `DisjointFormHomomorphism(E)` (attribute)

Returns: The homomorphism that maps to `E`.

Only available if `E` was obtained via the method `DisjointFormOfDecomposedEquation`.

5.1.8 LiftSolution (equation,equation,equationhom,equationhom)

▷ `LiftSolution(DE, E, sigma, sol)` (operation)

Returns: An evaluation for E `eq`.

Given an equation `E` and a solution `sol` for its decomposed equation `DE` under the decomposition with activity `sigma` this method computes a solution for the equation `E`.

Note that the solution not neccecarily maps to the group of constants of `E` but can map to the group where all elements of the group of constants can appear as states. If the group of constants is layered, this two groups will coincide.

Example

```
gap> F := FreeGroup(2);; SetName(F,"F");
gap> Gr := GrigorchukGroup;; a:=Gr.1;; d:=Gr.4;;
gap> G := EquationGroup(Gr,F);;
gap> DG := DecompositionEquationGroup(G);;
gap>  sigma := GroupHomomorphismByImages(F,SymmetricGroup(2),[(1,2),()]);
[ f1, f2 ] -> [ (1,2), () ]
gap>  e := Equation(G,[Comm(F.1,F.2),Comm(d,a)]);
Equation in [ f1, f2 ]
gap>  de := DecompositionEquation(DG,e,sigma);
DecomposedEquation in [ f11, f21, f12, f22 ]
gap> dedj := DecomposedEquationDisjointForm(de);
rec( eq := DecomposedEquation in [ f11, f12, f22 ],
  hom := [ f21 ]->[ FreeProductElm of length 3 ] )
gap> EquationComponents(dedj.eq);
[ Equation in [ f11, f12, f22 ], Equation in [  ] ]
gap> s := EquationEvaluation(DG,EquationVariables(dedj.eq),[One(Gr),One(Gr),Gr.2]);
MappingByFunction( GrigorchukGroup*F*F, GrigorchukGroup, function( q ) ... end )
gap> IsSolution(s,EquationComponent(dedj.eq,1));
true
gap> ns := dedj.hom*s;; IsEvaluation(ns);
true
gap> ForAll(EquationComponents(de),F->IsSolution(ns,F));
true
gap> ls := LiftSolution(de,e,sigma,ns);;
gap> IsSolution(ls,e);
true
gap> ForAll(EquationVariables(e),x->Equation(G,[x])^ls in Gr); # only good luck
true
```

References

[Bar16] L. Bartholdi. FR, computations with functionally recursive groups, Version 2.3.6. \verb+https://www.gap-system.org/Packages/fr.html+, Apr 2016. GAP package. 20

[CJE81] L. P. Comerford Jr. and C. C. Edmunds. Quadratic equations over free groups and free products. *J. Algebra*, 68(2):276–297, 1981. 18

Index