JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

Computing in Direct Powers of Expanded Groups

(Rechnen in direkten Potenzen erweiterter Gruppen)

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

Computermathematik

Eingereicht von:
Stephan Zweckinger, BSc

Angefertigt am:
Institut für Algebra

Beurteilung:
Dipl.-Ing. Dr. Peter Mayr

Mitwirkung:
Assoz. Univ.-Prof. Dipl.-Ing. Dr. Erhard Aichinger

Linz, Mai 2013

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.
Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, Mai 2013

Stephan Zweckinger

Abstract

In this thesis we discuss expanded groups and the Subpower Intersection Problem for direct powers of groups and implement these concepts in GAP.

The first chapter is an introduction to universal algebra and GAP. We define algebraic structures, especially groups, expanded groups, direct powers and some concepts belonging to algebraic structures. Then we give an introduction to the programming language GAP which was developed for computations in algebra and we use GAP for proving a theorem about polynomial functions on groups of order 8.

In the second chapter we explore the theory of expanded groups, especially how to find the universe of an expanded group given by generators. After that we develop a data structure for computing with expanded groups in GAP and use this data structure for proving that not every normal subgroup of an expanded group is also an ideal and for counting the binary polynomials of a concrete expanded group.

The last chapter is about the Subpower Intersection Problem for direct powers of groups. We will first introduce the concept of strong generators for direct powers of groups and then discuss how to find strong generators of the intersection of direct powers of groups. Finally we implement the algorithms for solving the problem in GAP and discuss compatible functions as an application of the Subpower Intersection Problem.

Acknowledgements

I would like to thank my supervisor Peter Mayr for his advice, sharing his knowledge and answering all my questions.

I am very grateful for the financial support from the Austrian Science Fund: P24285.

Moreover I owe thanks to all colleagues from the Institute of Algebra for the discusssions about algebra, GAP and LaTeX.

Last but not least I want to thank my parents who enabled my whole education and always encouraged me.

Contents

Chapter 1

Universal Algebra and GAP

In this chapter we will introduce basic knowledge about algebraic structures, functions on algebraic structures and the programming language GAP.

1.1 Algebras

First we want to define operations:

Definition 1.1 (cf. [4, Chapter II, Definition 1.1])**.** Let A be a nonempty set and n a non-negative integer. We define $A^0 = \{\emptyset\}$ and, for $n > 0$, A^n as the set of n-tuples of elements from A. An *n-ary operation* (or *function*) *on* A is a function f from A^n to A. We call n the *arity* of f. By $f(a_1, \ldots, a_n)$ we denote the image of f evaluated at $(a_1, \ldots, a_n)$. A *finitary* operation is an n-ary operation for some n. A function f on A is called *unary, binary* or *ternary* if its arity is 1, 2 or 3, respectively. A unary function f is called *constant* if there exists an element $a \in A$ such that $f(x) = a$ for all $x \in A$.

A constant function is completly determined by the image $f(x)$ in A of an arbitrary element $x \in A$. If we talk about a constant function a, we mean that there is a constant function which maps every element to a and denote this function by a.

To every function belongs a function symbol:

Definition 1.2 (cf. [4, Chapter II, Definition 1.2])**.** A *type* (or *language*) of algebras is a set $\mathcal{F}$ of *function symbols* such that a non-negative integer n is assigned to each member f of $\mathcal{F}$. This integer is called the *arity* of f. f is said to be an *n-ary function symbol*. The subset of n-ary function symbols in $\mathcal{F}$ is denoted by $\mathcal{F}_n$.

An algebra is formed by a set and a family of functions:

Definition 1.3 (cf. [4, Chapter II, Definition 1.3])**.** If $\mathcal{F}$ is a type of algebras, then an *algebra* $\mathbf{A}$ with *type* $\mathcal{F}$ is an ordered pair $\langle A, F\rangle$ where A is a nonempty set and F is a family of finitary operations on A indexed by the type $\mathcal{F}$ such that corresponding to each n-ary function symbol f in $\mathcal{F}$ there is an n-ary operation $f^{\mathbf{A}}$ on A. The set A is called the *universe* of $\mathbf{A} = \langle A, F\rangle$ and the functions $f^{\mathbf{A}}$ ($f \in \mathcal{F}$) are called the *fundamental operations* of $\mathbf{A}$. An algebra $\mathbf{A}$ is *finite* if $|A|$ is finite and *trivial* if $|A| = 1$.

The order of an algebra $\mathbf{A}$, denoted $|\mathbf{A}|$, is the cardinality of A. By $x \in \mathbf{A}$ we mean $x \in A$. As suggested in [4, Chapter II, Definition 1.3] we also will write f instead of $f^{\mathbf{A}}$ and $\langle A, f_1, \ldots, f_k\rangle$ instead of $\langle A, F\rangle$ if F is finite with $\mathcal{F} = \{f_1, \ldots, f_k\}$.

Now we want to introduce the most important algebras for this thesis:

Definition 1.4 (cf. [4, pp. 27])**.** A *loop* $\mathbf{L}$ is an algebra $\langle L, \cdot, /, \backslash, 1\rangle$ with three binary operations $\cdot, /, \backslash$ and one constant operation 1 in which the following identities are true for all $x, y \in Q$:

(L1) $x\backslash(x \cdot y) = y$ and $(x \cdot y)/y = x$

(L2) $x \cdot (x\backslash y) = y$ and $(x/y) \cdot y = x$

(L3) $x \cdot 1 = 1 \cdot x = x$

Definition 1.5 (cf. [4, pp. 26–27])**.** A *group* $\mathbf{G}$ is an algebra $\langle G, \cdot, ^{-1}, 1\rangle$ with a binary operation $\cdot$, a unary operation $^{-1}$ and a constant function 1 such that the following identities are true for all $x, y, z \in G$:

(G1) $x \cdot (y \cdot z) = (x \cdot y) \cdot z$

(G2) $x \cdot 1 = 1 \cdot x = x$

(G3) $x \cdot x^{-1} = x^{-1} \cdot x = 1$

(G1) is called associativity. 1 is the *neutral element* of $\mathbf{G}$ and x^{-1} is the *inverse element* of x. A group $\mathbf{G}$ is *abelian* or *commutative* if

(G4) $x \cdot y = y \cdot x$

holds for all $x, y \in G$.

If we talk about a *group operation*, we usually mean the binary operation of a group. Note that the neutral element 1 and the inverse operation $^{-1}$ are uniquely determined by $\cdot$. Hence groups are also often denoted by $\langle G, \cdot \rangle$.

One of the most important class of groups are the permutation groups:

Example 1.6 (cf. [11, pp.6–7])**.** Let X be a set. We call a bijection $X \to X$ a permutation on X. For permutations on finite X we use the so-called cycle-notation, i.e. we write $(x_1, x_2, \ldots, x_n)$ if we want to express that x_1 is mapped to x_2, x_2 to x_3, $\ldots$, x_{n-1} to x_n and x_n to x_1. By () we denote the identity function. Let G be a set of permutations on some set X such that G is closed under the operation of functional composition $\circ$, the corresponding inverse operation $^{-1}$ and the constant operation (). Then $\langle G, \circ, ^{-1}, () \rangle$ is called a *permutation group*. The group of the set of all permutations, denoted by S_x, is called the *symmetric* group on X. For $n \in \mathbb{N}$ and $X = \{1, 2, \ldots, n\}$ we call the symmetric group on X the *symmetric group of degree* n, written as $S_n = \langle X, \circ, ^{-1}, () \rangle$.

Often we have two algebras where one universe is contained in the other universe:

Definition 1.7 (cf. [4, Chapter II, Definition 2.2])**.** Let $\mathbf{A}$ and $\mathbf{B}$ be two algebras of the same type $\mathcal{F}$. Then $\mathbf{B}$ is called a *subalgebra* of $\mathbf{A}$, written as $\mathbf{B} \leq \mathbf{A}$, if $B \subseteq A$ and every fundamental operation of $\mathbf{B}$ is the restriction of the corresponding operation of $\mathbf{A}$. A *subuniverse* of $\mathbf{A}$ is a subset B of A which is closed under the fundamental operations of $\mathbf{A}$.

Note that the universe of every subalgebra is a subuniverse. Some subalgebras have special names, e.g. a subalgebra of a group is a *subgroup*.

Definition 1.8 (cf. [11, p.15])**.** A subuniverse H of a group $\mathbf{G} = \langle G, \cdot, ^{-1}, 1 \rangle$ is a *normal subuniverse*, denoted by $H \trianglelefteq \mathbf{G}$, if

$$x^{-1} \cdot h \cdot x \in H \text{ for all } x \in G \text{ and } h \in H.$$

Subalgebras are defined by a subset of the universe, while reducts are defined by a subset of the type:

Definition 1.9 ([4, p.30])**.** Let $\langle A, F^* \rangle$ be an algebra with type $\mathcal{F}^*$ and let $\mathcal{F} \subseteq \mathcal{F}^*$. If F is the restriction of F^* to $\mathcal{F}$, then the algebra $\langle A, F \rangle$ is called the *reduct* of the algebra $\langle A, F^* \rangle$ to $\mathcal{F}$. Conversely, $\langle A, F^* \rangle$ is called an expansion of $\langle A, F \rangle$.

Some mappings between algebras have special names:

Definition 1.10 (cf. [4, Chapter II, Definition 6.3])**.** Let $\mathbf{A}$ and $\mathbf{B}$ be two algebras of the same type $\mathcal{F}$. A mapping $\alpha : A \to B$ is called a *homomorphism* from $\mathbf{A}$ to $\mathbf{B}$ if

$$\alpha f^{\mathbf{A}}(a_1, \ldots, a_n) = f^{\mathbf{B}}(\alpha a_1, \ldots, \alpha a_n)$$

for each n-ary f in $\mathcal{F}$ and all $a_1, \ldots, a_n$ from A. An *isomorphism* is an homomorphism which is onto and one-to-one. If $\mathbf{A} = \mathbf{B}$, then a homomorphism is referred to as an *endomorphism* and an isomorphism is also called an *automorphism*.

We say shortly "$\alpha : \mathbf{A} \to \mathbf{B}$ is a homomorphism" if we mean that α is a homomorphism from $\mathbf{A}$ to $\mathbf{B}$.

Definition 1.11 (cf. [11, p. 5])**.** Two algebras $\mathbf{A}$ and $\mathbf{B}$ are said to be *isomorphic* if there exists an isomorphism from $\mathbf{A}$ to $\mathbf{B}$.

We use the notation $\mathbf{A} \cong \mathbf{B}$ to denote that $\mathbf{A}$ and $\mathbf{B}$ are isomorphic.

Example 1.12. Consider a set of permutations

$$X = \{(), (7,8), (6,7), (6,7,8), (6,8,7), (6,8)\}.$$

Remember, that the universe of S_3 is the set of permutations

$$Y = \{(), (2,3), (1,2), (1,2,3), (1,3,2), (1,3)\}.$$

We can define following isomorphism from S_X to S_3:

$$\begin{aligned} f : X &\to Y \\ () &\mapsto () \\ (7,8) &\mapsto (2,3) \\ (6,7) &\mapsto (1,2) \\ (6,7,8) &\mapsto (1,2,3) \\ (6,8,7) &\mapsto (1,3,2) \\ (6,8) &\mapsto (1,3) \end{aligned}$$

Hence the symmetric group on X is isomorphic to the symmetric group of order 3.

Next we want to introduce quotient algebras. For defining them we first have to define equivalence relations and congruences:

Definition 1.13 (cf. [4, Chapter I, Definition 4.4. and Def. 4.8.])**.** Let A be a set. A binary relation θ on A is a subset of A^2. If $(a, b) \in \theta$, we also write $a\,\theta\,b$. Such a relation θ is called an *equivalence relation* if it is

- reflexive, i.e. $\forall a \in A\colon a\,\theta\,a$,
- symmetric, i.e. $\forall a, b \in A\colon a\,\theta\,b \Rightarrow b\,\theta\,a$ and
- transitive, i.e. $\forall a, b, c \in A\colon a\,\theta\,b \wedge b\,\theta\,c \Rightarrow a\,\theta\,c$.

Let a be an element of A and let θ be an equivalence relation. The *coset* or *equivalence class of a modulo θ* is defined as the set $a/\theta = \{b \in A : (b, a) \in \theta\}$. The set $\{a/\theta : a \in A\}$ is denoted by A/θ.

Note that every element $b \in A$ is contained in exactly one element of A/θ. Therefore every coset can be represented by one element contained in the coset. Hence we call an element contained in a coset also a *coset representative*.

Definition 1.14 (cf. [4, Chapter II, Definition 5.1.])**.** Let $\mathbf{A}$ be an algebra of type $\mathcal{F}$ and let θ be an equivalence relation. Then θ is a *congruence* on $\mathbf{A}$ if θ satisfies the following *compatibility property* for all n-ary function symbols $f \in \mathcal{F}$ and all elements $a_1, \ldots, a_n, b_1, \ldots, b_n \in A$:
If $a_i\,\theta\,b_i$ holds for all $1 \leq i \leq n$, then

$$f^{\mathbf{A}}(a_1, \ldots, a_n)\,\theta\,f^{\mathbf{A}}(b_1, \ldots, b_n)$$

holds.

Now we can define quotient algebras:

Definition 1.15 (cf. [4, Chapter II, Definition 5.2.])**.** Let $\mathbf{A}$ be an algebra of type $\mathcal{F}$ with a congruence θ. Then the *quotient algebra of $\mathbf{A}$ by θ*, written $\mathbf{A}/\theta$, is the algebra of type $\mathcal{F}$ whose universe is the set of classes modulo θ. For every $n \in \mathbb{N}$ and every $f \in \mathcal{F}$, the fundamental operation $f^{\mathbf{A}/\theta}$ is defined by

$$\forall a_1, \ldots, a_n \in \mathbf{A}\colon f^{\mathbf{A}/\theta}(a_1/\theta, \ldots, a_n/\theta) = f^{\mathbf{A}}(b_1, \ldots, b_n)/\theta.$$

Note that $f^{\mathbf{A}/\theta}$ is well-defined since $f^{\mathbf{A}}$ is compatible with θ by Definition 1.14. An example for quotient algebras are the cosets of groups:

Example 1.16 (cf. [5, p.37])**.** Let H be a normal subuniverse of a group $\mathbf{G}$ and θ a congruence defined by $x\,\theta\,y$ for $x, y \in \mathbf{G}$ if and only if $x^{-1} \cdot y \in H$. Note that $x^{-1} \cdot y \in H$ is equivalent to $y \in xH$, where $xH = \{x \cdot u \mid u \in H\}$. Then $\mathbf{G}/\theta$ is a quotient algebra. We denote this quotient algebra by $\mathbf{G}/H$ and call it the *collection of (left) cosets* or the *factor group* of $\mathbf{G}$ over H.

One of the most important groups are the factor groups $\mathbb{Z}/n\mathbb{Z}$:

Example 1.17. Consider the infinite group of integers $\langle \mathbb{Z}, +, -, 0\rangle$ and $n \in \mathbb{N}$. By $n\mathbb{Z}$ we denote the set containing all integer multiples of n, i.e. $n\mathbb{Z} = \{\dots, -1n, 0, n, 2n, 3n, \dots\}$. Then $n\mathbb{Z}$ is a normal subuniverse of $\langle \mathbb{Z}, +, -, 0\rangle$. We denote the factor set $\mathbb{Z}/n\mathbb{Z}$ by $\mathbb{Z}_n$. For $i \in \{0, \dots, n-1\}$, the class $i + n\mathbb{Z}$ is denoted by i. If we consider the group operation $+$ as addition modulo n and the corresponding inverse and neutral elements, we get the abelian group $\langle \mathbb{Z}_n, +, -, 0\rangle$, which is isomorphic to the factor group $\langle \mathbb{Z}, +, -, 0\rangle / n\mathbb{Z}$.

It is also possible to describe an algebra by some elements of the algebra:

Definition 1.18 (cf. [4, Chapter II, Definition 3.4])**.** Let $\mathbf{A}$ be an algebra and let $X \subseteq A$. Define

$$\langle X\rangle = \bigcap \{B : X \subseteq B \text{ and } B \text{ is a subuniverse of } \mathbf{A}\}.$$

Then $\langle X\rangle$ is a subuniverse of $\mathbf{A}$, namely the smallest subuniverse of $\mathbf{A}$ that contains X. We call $\langle X\rangle$ the subuniverse generated by X and X the generators of $\langle X\rangle$.

We write $\langle g_1, \dots, g_k\rangle$ for the algebra generated by the set $\{g_1, \dots, g_k\}$. The corresponding operations should be clear from the context. The group generated by the set $\{1\}$ only containing the neutral element is called the *identity group*. We denote this group by $\mathbf{I}$.

Example 1.19 (cf. [11, p.9])**.** Let $\langle G, \cdot, ^{-1}, 1\rangle$ be a group. If there exists an element $g \in G$, such that the whole group can be generated by this element, then the group is called a *cyclic group*. Important examples of cyclic groups are the groups $\langle \mathbb{Z}_n, +, -, 0\rangle$ defined in Example 1.17, because every element in $\mathbb{Z}_n$ can be generated by the single element 1.

An example which summarizes cyclic groups and isomorphic groups is following lemma:

Lemma 1.20. *Every finite cyclic group G is isomorphic to $\mathbb{Z}_n$ for some $n \in N$.*

Proof. Let g be an element of G such that g generates every other element of G and let n be the order of G. Then

$$\begin{aligned} f : \mathbb{Z}_n &\to G \\ k &\mapsto g^k \end{aligned}$$

is an isomorphism from $\mathbb{Z}_n$ to G. □

A group together with additional operations forms an expanded group:

Definition 1.21 (cf. [3, p. 3])**.** An *expanded group* is an expansion of a group (that is, an algebra that has a group as reduct).

Rings are examples of expanded groups:

Definition 1.22 (cf. [10, p. 14])**.** An algebra $\langle G, +, -, 0, \cdot \rangle$ (or shortened $\langle G, +, \cdot \rangle$) is called a *ring* if $\langle G, +, -, 0 \rangle$ is an abelian group and

(R1) $\forall x, y, z \in G \colon x \cdot (y \cdot z) = (x \cdot y) \cdot z$,

(R2) $\forall x, y, z \in G \colon x \cdot (y + z) = (x \cdot y) + (x \cdot z)$,

(R3) $\forall x, y, z \in G \colon (y + z) \cdot x = (y \cdot x) + (z \cdot x)$.

Example 1.23 (cf. [10, p. 14])**.** The integers with addition and multiplication form a ring.

Example 1.24. We define a function f by

$$f : \mathbb{Z}_6 \to \mathbb{Z}_6, x \mapsto \begin{cases} 2 & \text{if } x \in 2\mathbb{Z}_6 = \{0, 2, 4\}, \\ 0 & \text{otherwise.} \end{cases}$$

Then $\langle \mathbb{Z}_6, +, -, 0, f \rangle$ is an expanded group with $\langle \mathbb{Z}_6, +, -, 0 \rangle$ being a cyclic group.

Next we introduce products of algebras:

Definition 1.25 (cf. [4, Chapter II, Definition 7.8])**.** Let $(\mathbf{A}_i)_{i \in I}$ be an indexed family of algebras of type $\mathcal{F}$. The *direct product* $\mathbf{A} = \prod_{i \in I} A_i$ is an algebra with universe $\prod_{i \in I} A_i$ and operations defined componentwise, i.e. for $f \in \mathcal{F}_n$ and $a_1, \ldots, a_n \in \prod_{i \in I} A_i$,

$$f^{\mathbf{A}}(a_1, \ldots, a_n)(i) = f^{\mathbf{A}_i}(a_1(i), \ldots, a_n(i))$$

for $i \in I$. We denote the i-th component of $a \in \prod_{i \in I} A_i$ by $a(i)$ or a_i and call it the *i-th coordinate* of a. The empty product $\prod \emptyset$ is the trivial algebra with universe $\{\emptyset\}$.
Projection maps

$$\pi_j : \prod_{i \in I} A_i \to A_j$$

are defined for $j \in I$ by

$$\pi_j(a) = a(j)$$

and give surjective homomorphisms

$$\pi_j : \prod_{i \in I} \mathbf{A}_i \to \mathbf{A}_j.$$

If $\mathbf{A}_i = \mathbf{A}$ for all $i \in I$, then we write $\mathbf{A}^n$ for the direct product of n such algebras and call it the n-th *direct power* of $\mathbf{A}$. $\mathbf{A}^0$ is the trivial algebra.

Example 1.26. Consider an algebra $\mathbf{G}$ with universe G. Let $n \in \mathbb{N}$. Consider the set of n-ary operations on G as G^{G^n}. Then $\mathbf{G}^{G^n}$ is a direct power of $\mathbf{G}$ and hence forms an algebra.

Theorem 1.27. *Let $\mathbf{G}$ be a group and let $n \in \mathbb{N}_0$. Then $\mathbf{G}^n$ is a group.*

Proof. We show that $\mathbf{G}^n$ fulfills (G1):

$$\forall \bar{x}, \bar{y}, \bar{z} \in \mathbf{G}^n \colon \bar{x} \cdot (\bar{y} \cdot \bar{z}) = (\bar{x} \cdot \bar{y}) \cdot \bar{z}.$$

As we know, $\mathbf{G}$ is a group and the group operation $\cdot$ fulfills

$$\forall x_i, y_i, z_i \in \mathbf{G} \colon x_i \cdot (y_i \cdot z_i) = (x_i \cdot y_i) \cdot z_i$$

in every component of $\mathbf{G}^n$:

$$\begin{aligned} \bar{x} \cdot (\bar{y} \cdot \bar{z}) &= (x_1, \ldots, x_n) \cdot ((y_1, \ldots, y_n) \cdot (z_1, \ldots, z_n)) \\ &= (x_1 \cdot (y_1 \cdot z_1), \ldots, x_n \cdot (y_n \cdot z_n)) \\ &= ((x_1 \cdot y_1) \cdot z_1, \ldots, (x_n \cdot y_n) \cdot z_n) \\ &= (\bar{x} \cdot \bar{y}) \cdot \bar{z} \end{aligned}$$

Similarly we can show (G2) and (G3). Therefore $\mathbf{G}^n$ is a group. □

Example 1.28. For p a prime consider the group $(\mathbb{Z}_p, +, -, 0)$. Then the direct product $(\mathbb{Z}_p, +, -, 0) \times (\mathbb{Z}_p, +, -, 0)$ (or shortened $\mathbb{Z}_p \times \mathbb{Z}_p$ or $\mathbb{Z}_p^2$) with component-wise defined operations, i.e. $(a_1, a_2) + (b_1, b_2) = (a_1 + b_1, a_2 + b_2)$, $-(a_1, a_2) = (-a_1, -a_2)$ and the neutral element $(0, 0)$, is again a group.

The last definitions in this section will lead us to the concept of polynomial equivalence:

Definition 1.29 (cf. [4, p.69])**.** Let X be a set of (distinct) objects called *variables*. Let $\mathcal{F}$ be a type of algebras. The set $T(X)$ of *terms of type $\mathcal{F}$ over X* is the smallest set such that

(i) $X \subseteq T(X)$,

(ii) if $p_1, \ldots, p_n \in T(X)$ and $f \in \mathcal{F}_n$, then $f(p_1, \ldots, p_n) \in T(X)$.

A term p is called n-ary if the number of variables appearing explicitly in p is less than or equal to n.

As in [4, p.69] we will prefer the infix notation for binary symbols, e.g. $p_1 \cdot p_2$ instead of $\cdot(p_1, p_2)$. With $p(x_1, \ldots, x_n)$ we mean that the variables occurring in p are among $x_1, \ldots, x_n$.

Example 1.30. Consider a type $\mathcal{F}$ which contains two binary symbols $\cdot$ and $+$ and a set $X = \{w, x, y, z\}$. Then

$$w + (x \cdot y) + z$$

is a term of type $\mathcal{F}$ over X.

From a term we can also define a term function:

Definition 1.31 (cf. [4, Chapter II, Definition 10.2])**.** Let $p(x_1, \ldots, x_n)$ be a term of type $\mathcal{F}$ over $X = \{x_1, \ldots, x_n\}$ and let $\mathbf{A}$ be an algebra of type $\mathcal{F}$. We define a mapping $p^{\mathbf{A}} : A^n \to A$ as follows:

(i) If p is a variable x_i, then $p^{\mathbf{A}}$ is the projection map, i.e.

$$p^{\mathbf{A}}(a_1, \ldots, a_n) = a_i$$

for all $a_1, \ldots, a_n \in A$.

(ii) If p is of the form $f(p_1(x_1, \ldots, x_n), \ldots, p_k(x_1, \ldots, x_n))$, where $f \in \mathcal{F}_k$, then

$$p^{\mathbf{A}}(a_1, \ldots, a_n) = f^{\mathbf{A}}(p_1^{\mathbf{A}}(a_1, \ldots, a_n), \ldots, p_k^{\mathbf{A}}(a_1, \ldots, a_n)).$$

Particularly, if $p = f$ for some $f \in \mathcal{F}$, then $p^{\mathbf{A}} = f^{\mathbf{A}}$.
$p^{\mathbf{A}}$ is the *term function* or *term operation* on $\mathbf{A}$ induced by the term p. For simplicity, we often omit the superscript $\mathbf{A}$.

Example 1.32. Again consider a type $\mathcal{F}$ which contains two binary symbols $\cdot$ and $+$. Let $\mathbf{A} = \langle A, F \rangle$ be an algebra with type $\mathcal{F}$. Then the term function induced by the term $x_1 + (x_2 \cdot x_3) + x_4$ is the mapping

$$\begin{aligned} p^{\mathbf{A}} &: A^4 \to A \\ (x_1, x_2, x_3, x_4) &\mapsto x_1 + (x_2 \cdot x_3) + x_4 \end{aligned}$$

Moreover we define the length of a term:

Definition 1.33. Let p be a term. We denote the *length of a term* p by $L(t)$ and define

(i) $L(p) = 1$ if $p = x_i$,

(ii) $L(p) = 1 + \sum_{i=1}^{k} p_i$ if p is of the form $f(p_1(x_1, \ldots, x_n), \ldots, p_k(x_1, \ldots, x_n))$.

Example 1.34. Consider the term

$$p = x_1 + (x_2 \cdot x_3) + x_4.$$

Then

$$\begin{aligned} L(p) &= L(x_1 + (x_2 \cdot x_3) + x_4) \\ &= 1 + L(x_1) + L((x_2 \cdot x_3) + x_4) \\ &= 1 + 1 + 1 + L(x_2 \cdot x_3) + L(x_4) \\ &= 1 + 1 + 1 + 1 + L(x_2) + L(x_3) + 1 \\ &= 1 + 1 + 1 + 1 + 1 + 1 + 1 \\ &= 7. \end{aligned}$$

We can generate a set containing all basic operations and projections of the algebra which is closed under composition of those functions:

Definition 1.35 (cf. [10, Definition 4.1 and 4.2.])**.** Let A be a set. Let $O_A := \bigcup_{n \in \mathbb{N}} A^{A^n}$ be the set of all operations on A. A subset $C \subseteq O_A$ is a clone if

1. $\forall n \in \mathbb{N} \forall i \in \{1, \ldots, n\} : \Pi_i : A^n \to A, (x_1, \ldots, x_n) \mapsto x_i$ is in C.

2. $\forall n, m \in \mathbb{N} \forall f, g_1, \ldots, g_n \in C$: if f is n-ary and $g_1, \ldots, g_n$ are m-ary, then

$$\begin{aligned} f(g_1, \ldots, g_n) : \quad & A^m \to A \\ & (x_1, \ldots, x_m) \mapsto f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m)) \end{aligned}$$

is in C.

The *clone of term operations* (or *clone of term functions*) of an algebra $\mathbf{A} = \langle A, F \rangle$ is the smallest clone on the set A that contains the operations of F. We denote the whole clone of term functions of $\mathbf{A}$ by $\mathrm{Clo}(\mathbf{A})$ and the set of n-ary operations in $\mathrm{Clo}(\mathbf{A})$ by $\mathrm{Clo}_n(\mathbf{A})$.

Note that the elements of $\mathrm{Clo}_n(\mathbf{A})$ are exactly the operations induced by terms of type $\mathcal{F}$ over $\{x_1, \ldots, x_n\}$.

Example 1.36 (cf. [10, p. 144])**.** Consider the group $\mathbb{Z}_2 = \langle\{0,1\}, +, -, 0\rangle$. We want to determine all elements of $\mathrm{Clo}_2(\mathbb{Z}_2)$. $\mathrm{Clo}_2(\mathbb{Z}_2)$ must contain the two projection operations $f_1(x,y) = x$ and $f_2(x,y) = y$ and the group operations $f_3(x,y) = x + y$, $f_4(x,y) = 0$ and $f_5(x,y) = -x$. Remember that $1 + 1 = 0$. Hence $x = -x$ and $f_1(x,y) = f_5(x,y)$. We have to check whether the set $\{f_1, f_2, f_3, f_4\}$ is closed under composition, i.e.

$$\forall x, y \in \mathbb{Z}_2 \forall i, j, k \in \{1,2,3,4\} \exists l \in \{1,2,3,4\} \colon f_i(f_j(x,y,), f_k(x,y)) = f_l(x,y).$$

By some manual work, e.g.

$$f_2(f_4(x,y,), f_3(x,y)) = f_2(0, x+y) = x + y = f_2(x,y)$$

we get that $\{f_1, f_2, f_3, f_4\}$ is closed under composition and therefore

$$\mathrm{Clo}_2(\mathbb{Z}_2) = \{f_1, f_2, f_3, f_4\}.$$

The most important form of term functions in this thesis are the polynomial functions:

Definition 1.37 (cf. [4, Chapter II, Definition 13.3] and [10, Definition 4.4])**.** Let $\mathbf{A}$ be an algebra of type $\mathcal{F}$ with universe A. We define $\mathcal{F}_A$ as the union of $\mathcal{F}$ and the set of symbols c_a for constant functions for each $a \in A$. Let $\mathbf{A}_A$ be the algebra of type $\mathcal{F}_A$, i.e. $\mathbf{A}$ with a constant operation corresponding to each element of A. We call the terms of type $\mathcal{F}_A$ *polynomials* of $\mathbf{A}$. For a polynomial p of $\mathbf{A}$ we denote the induced polynomial function by $p^{\mathbf{A}}$. The *clone of polynomial functions* of an algebra $\mathbf{A}$, denoted by $\mathrm{Pol}(\mathbf{A})$, is the smallest clone on the universe A that contains all basic operations of $\mathbf{A}$ and all constant operations. $\mathrm{Pol}_n(\mathbf{A})$ is the set of all n-ary polynomial functions.

Example 1.38. Consider the ring $A := \langle\mathbb{Z}, +, \cdot\rangle$. The set of polynomials of $\langle\mathbb{Z}, +, \cdot\rangle$ over the variables $\{x_1, \ldots, x_n\}$ is usually denoted by $\mathbb{Z}[x_1, \ldots, x_n]$. For unary polynomials $p \in \mathbb{Z}[x]$ there exist $n \in \mathbb{N}$ and $a_0, \ldots, a_n \in \mathbb{Z}$ such that $p = a_0 + a_1 x + \ldots a_n x^n$. Then p induces the polynomial function

$$p^{\mathbf{A}} : \mathbb{Z} \to \mathbb{Z}, x \mapsto a_0 + a_1 x + \ldots a_n x^n.$$

We can define an equivalence relation based on the set of polynomial functions of algebras.

Definition 1.39 (cf. [4, Chapter II, Definition 13.3])**.** Two algebras on the same universe are *polynomially equivalent* if they have the same set of polynomial functions.

Note that polynomial equivalent algebras may have distinct types.

Example 1.40. The inverse element of the group replaces the right division / and the left division \ in the definition of a loop, i.e. $x/y = x \cdot y^{-1}$ and $x\backslash y = x^{-1} \cdot y$. Hence every group is polynomial equivalent to a loop. Loops and groups have different languages.

1.2 GAP

The following section is based on the information provided by [6] on `http://www.gap-system.org`.

For computing in groups, expanded groups and direct products of groups or expanded groups we use the programming language GAP. GAP, which is the abbreviation of "Groups, Algorithms and Programming", is a free and open software project for computing in group theory. Until now it was coordinated by different universities, starting 1984 at RWTH Aachen. GAP is an extensible language, that means, we can write our own GAP programs and publish them as so-called "packages" which are distributed together with GAP. An example for such a package is `SONATA` [1] which was developed at the Institute of Algebra at the Johannes Kepler University Linz for computations with near-rings.

In this thesis we work with GAP 4.5.6. [6] and the additional package `SONATA`, Version 2.6 [1]. Therefore it is recommended to install also those versions or higher versions for using the developed programs.

One possibility to use GAP is running the program in a terminal and writing the commands directly there. Every GAP command, which ends always with a semicolon, used in the terminal will be displayed in the following form:

```
gap>command;
Output
```

With two semicolons we can suppress the output. We will use that if the output of a command is not of special interest and displaying the output would need a lot of space.

It is also possible to write some code in a text file and then import it to GAP running in a terminal by the command `Read("filename")`. Comments are denoted by `#`. GAP will ignore everything after the comment symbol in the

same line.

With the command := we can define symbols, e.g.

- functions,

```
gap> f := function(x, y) return x^y; end;
function( x, y ) ... end
gap> f(2, 3);
8
```

- variables.

```
gap> a:= 4;
4
gap> b:= 6;
6
gap> c := a*b;
24
```

In GAP we can compute with many different data types:

- Integers can be used e.g. with the standard operations `*, /, +, -`.

```
gap> 5+8;
13
gap> 10*7;
70
```

- Boolean values are `true` and `false`. We can combine them by logical connectives like `and` and `or`.

```
gap> false or true;
true
gap> false and true;
false
```

- Permutations are represented in cyclic notation and can be composed by the operation `*`.

```
gap> (1,2,3)*(1,2);
(2,3)
```

- Lists are collections of objects. Elements can occur more than once and every element is at a certain position in the list. The objects are enclosed in square brackets and separated by commas.

```
gap> [1..4];
[ 1 .. 4 ]
gap> List([1..4], x-> x^2);
[ 1, 4, 9, 16 ]
```

- Sets are sorted lists containing each element only once.

```
Set([2,2,4,3,5]);
[ 2, 3, 4, 5 ]
```

In GAP there exist many operators, e.g. `+, -, *, /, =,<, >`. The meaning of the operators depends on the data type for which they are used, e.g. `*` used for integers is the usual integer multiplication, whereas `*` used for permutations is the functional composition.

There are a lot of algebraic structures implemented in GAP. Most important for us are groups. For example, we can define cyclic groups and symmetric groups with special commands. It is possible to return a set which generates a group, to check whether a group is subgroup of another group, generate a group by generators and much more.

```
gap> Z4 := CyclicGroup(4);
<pc group of size 4 with 2 generators>
gap> Z2:= CyclicGroup(2);
<pc group of size 2 with 1 generators>
gap> IsSubgroup(Z4, Z2);
false
gap> S3 := SymmetricGroup(3);
Sym( [ 1 .. 3 ] )
gap> GeneratorsOfGroup(S3);
[ (1,2,3), (1,2) ]
gap> G := Group((1,2,3));
Group([ (1,2,3) ])
gap> IsCyclic(G);
true
```

Normally GAP displays only names for the group elements,

```
gap> G:= DihedralGroup(8);
<pc group of size 8 with 3 generators>
gap> AsSortedList(G);
[ <identity> of ..., f1, f2, f3, f1*f2, f1*f3, f2*f3, f1*f2*f3 ]
```

which cannot be used as GAP input:

```
gap> f1 in G;
false
```

Luckily we can solve this problem by using an isomorphic representation of the group. The package `SONATA` contains a function `AsPermGroup(G)` which yields a permutation group isomorphic to `G`. Some groups, like the symmetric groups, are always represented as permutation groups. For those groups we do not have to convert them by `AsPermGroup`.

```
gap> LoadPackage("sonata");;
#I  You may wish to install the xgap package
#I  and enjoy the graphic capabilities of SONATA.

  ------------------------------------------------------------------------
 /        ___
||       /   \                 /\    Version 2.6
||      ||   ||  |\    |      /  \               /\       Erhard Aichinger
 \___   ||   ||  |\\   |     /____\_____________/__\      Franz Binder
     \  ||   ||  | \\  |    /      \     ||    /    \     Juergen Ecker
     ||  \___/   |  \\ |   /        \    ||   /      \    Peter Mayr
     ||          |   \\|  /          \   ||               Christof Noebauer
 \___/           |    \|                 ||

 System    Of   Nearrings     And      Their Applications
 Info: http://www.algebra.uni-linz.ac.at/Sonata/

gap> G := AsPermGroup(DihedralGroup(8));
Group([ (1,2)(3,8)(4,6)(5,7), (1,3,4,7)(2,5,6,8),
(1,4)(2,6)(3,7)(5,8) ])
gap> AsSortedList(G);
[ (), (1,2)(3,8)(4,6)(5,7), (1,3,4,7)(2,5,6,8),
  (1,4)(2,6)(3,7)(5,8), (1,5)(2,3)(4,8)(6,7),
  (1,6)(2,4)(3,5)(7,8), (1,7,4,3)(2,8,6,5),
  (1,8)(2,7)(3,6)(4,5) ]
gap> (1,2)(3,8)(4,6)(5,7) in G;
true
```

Another interesting fact is the representation of direct products in GAP. Consider two symmetric groups S_n and S_m. Then the direct product $S_n \times S_m$ is built by mapping all elements $(x_1, \ldots, x_k)$ of S_m to elements $(x_1 + n, \ldots, x_k + n)$, whereas the elements of S_n are mapped to itself, which can be seen in following example:

```
gap> GeneratorsOfGroup(SymmetricGroup(3));
[ (1,2,3), (1,2) ]
gap> GeneratorsOfGroup(SymmetricGroup(4));
[ (1,2,3,4), (1,2) ]
gap> dp := DirectProduct(SymmetricGroup(4), SymmetricGroup(3));
Group([ (1,2,3,4), (1,2), (5,6,7), (5,6) ])
```

For converting the i-th coordinate of a direct product to an element of the corresponding factor or vice versa GAP provides the mappings `Projection` and `Embedding`:

```
gap> Image(Embedding(dp, 2), (1,3,2));
(5,7,6)
gap> Image(Projection(dp, 2), (5,7,6));
(1,3,2)
```

Finally we want to write a function, which takes a direct product and a tuple and converts this tuple to an element of the direct product:

```
1  AsDPElement := function (DP, tuple)
2  #
3     local i, result;
4  #
5      i := 0; result := Identity (DP);
6      for i in [1..Length (tuple)] do
7         result := result * Image (Embedding (DP, i), tuple [i]);
8      od;
9      return result;
10 end;
```

Note that all local variables used in a function have to be declared as `local`. After defining that function we can use it like every other GAP command:

```
gap> AsDPElement(dp, [(3,4), (1,2)]);
(3,4)(5,6)
```

1.3 An Example Combining Theory and GAP

As we saw in the last section, a polynomial function is a composition of projection functions and constant functions. Consider a group $\mathbf{G} = \langle G, \cdot, ^{-1}, 1\rangle$. We want to compute all binary polynomial functions f of $\mathbf{G}$ and check whether one of them fulfills the group properties:

(G1) $\forall x, y, z \in G\colon f(x, f(y, z)) = f(f(x, y), z)$ (associativity)

(G2) $\forall x \in G\colon f(1, x) = f(x, 1) = 1$ (neutral element)

(G3) $\forall x \in G \exists y \in G\colon f(x, y) = 1$ (inverse element)

Clearly the original group operation $\cdot$ and its opposite $(x, y) \mapsto y \cdot x$ are group operations in $\mathrm{Pol}_2(\mathbf{G})$. We are interested in whether there are any more.
In this example we will concentrate on the groups of order 8 up to isomorphism which are:

(i) $\langle \mathbb{Z}_8, +, -, 0\rangle$

(ii) $\langle \mathbb{Z}_4 \times \mathbb{Z}_2, +, -, 0\rangle$

(iii) $\langle \mathbb{Z}_2 \times \mathbb{Z}_2 \times \mathbb{Z}_2, +, -, 0\rangle$

(iv) D_8, the dihedral group with 8 elements

(v) Q_8, the quaternion group with 8 elements

For more information about those groups see [11] or every other good book about group theory. While D_8 and Q_8 are not abelian, the other three groups are abelian groups. We also will check if the polynomials are commutative, i.e.

(G4) $\forall x, y \in G\colon f(x, y) = f(y, x)$.

At the beginning of our computations we define the group `G` as D_8. (See Appendix A for the same computations with respect to Q_8). Moreover we define variables: the size `s` of the group, the generators `gens` of the group and a sorted list `t` with all the elements of the group. `H` is a list containing all possible tuples of size 2 with elements of `G`. Those tuples are ordered lexicographically like the elements of `G` in `t`.

```
gap> G:= DihedralGroup(8);
<pc group of size 8 with 3 generators>
gap> s := Size(G);
```

```
8
gap> gens:=GeneratorsOfGroup(G);
[ f1, f2, f3 ]
gap> t:= AsSortedList(G);
[ <identity> of ..., f1, f2, f3, f1*f2, f1*f3, f2*f3, f1*f2*f3 ]
gap> H:=Tuples(G,2);;
```

We define the projection functions and constant functions as tuples: The i-th element of such a tuple is the element to which the i-th element of `H` is mapped to. Note that $\mathrm{Pol}_2(\langle G, \cdot \rangle)$ forms a subgroup of $\mathbf{G}^{G^2}$. This subgroup is generated by the projections π_1, π_2 and the constant operations on G^2. Hence we define those generators, the corresponding group and a list containing all elements of that group:

```
gap> gen := [];
[  ]
gap> p1:= Tuple(List(H, x->x[1]));;
gap> p2:= Tuple(List(H, x->x[2]));;
gap> Add(gen, p1);
gap> Add(gen, p2);
gap> for i in [1..Size(gens)] do
> c := Tuple(List([1..Size(H)], x-> gens[i]));
> Add(gen, c);
> od;
gap> pol := AsSortedList(Group(gen));;
```

Finally we define functions for checking the group properties (G1)-(G4). With the GAP command `Filtered`, which returns only the elements of a list fulfilling a certain property, we want to find the functions, which fulfill the group properties. Moreover we use the command `Position`, which returns the position, in which a certain element occurs in a list (or fail, if it does not occur in the list). Note the fact, that the result of $f(x, y)$, where x is the i-th and y is the j-th element of `G`, is the $((i-1) * |\mathtt{G}| + j)$-th entry of the list representing f. We prepare the functions in a text-file called `FilterGroupProperties.gap`:

```
1 # Remember: s is order of G, t is a sorted list containing the
     elements of G, pol is a sorted list containing tuples
     which represent binary polynomial functions on G;
2 # check whether f(x,1) = f(1,x) = x for all x
3 FilterNeutral := function(pol, t, s)
4 #
```

```
5    local j
6  #
7  for j in [1..s] do
8    pol := Filtered(pol, x-> (x[j] = x[(j-1)*s+1] and x[j] =
        t[j]));
9  od;
10 return pol;
11 end;
12
13 # check whether for all x there exists a y such that f(x,y) =
     f(y, x) = 1 for all x
14 FilterInverse := function(pol, t, s, G)
15 #
16   local i
17 #
18 for i in [1..s] do
19   pol := Filtered(pol, x->
        Position(x{[1+(i-1)*s..i*s]},Identity(G))<> fail and
        Position(x{[i, i+s .. i+(s-1)*s]},Identity(G))<> fail);
20 od;
21 return pol;
22 end;
23
24 # check whether f(x, f(y,z)) = f(f(x,y), z) for all x, y, z
25 FilterAssociative := function(pol, t, s)
26 #
27   local i, j, k
28 #
29 for i in [1..s] do
30   for j in [1..s] do
31     for k in [1..s] do
32       pol := Filtered(pol, x-> x[(Position(t,
            x[(i-1)*s+j])-1)*s + k] = x[(i-1)*s+Position(t,
            x[(j-1)*s+k])]);
33     od;
34   od;
35 od;
36 return pol;
37 end;
38
39 # check whether f(x,y) = f(y,x) for all x, y
```

```
40 FilterCommutative := function(pol, t, s)
41 #
42    local i
43 #
44 for i in [1..s] do
45    for j in [1..s] do
46       pol := Filtered(pol, x-> (x[(i-1)*s+j] = x[(j-1)*s+i]));
47    od;
48 od;
49 return pol;
50 end;
```

With the command `Read` we can read the textfile and then use the defined functions:

```
gap> Read("FilterGroupProperties.gap");
gap> Size(pol);
4096
gap> pol := FilterNeutral(pol, t, s);;
gap> Size(pol);
2
gap> pol := FilterInverse(pol, t, s, G);;
gap> Size(pol);
2
gap> pol := FilterAssociative(pol, t, s);;
gap> Size(pol);
2
gap> f1 := Tuple(List(H, x -> x[1]*x[2]));;
gap> f2 := Tuple(List(H, x -> x[2]*x[1]));;
gap> f1 = pol[1];
true
gap> f2 = pol[2];
true
gap> pol := FilterCommutative(pol, t, s);;
gap> Size(pol);
0
```

Our computations show that in both groups D_8 and Q_8 there are exactly two polynomial functions fulfilling the group properties, namely the group operation $f(x, y) = x \cdot y$ and the opposite operation $f(x, y) = y \cdot x$. Only those two polynomial functions have the same neutral element 1 as the group operation $\cdot$.

Theorem 1.41. *Let* $\mathbf{G} = \langle G, \cdot, ^{-1}, 1\rangle$ *be a group of order 8, and let* $*$ *be in* $\mathrm{Pol}_2(\mathbf{G})$ *such that* $\forall x \in G\colon x * 1 = 1 * x = x$*. Then* $\forall x, y \in G\colon : x * y = x \cdot y$ *or* $\forall x, y \in G\colon : x * y = y \cdot x$*.*

Proof. By computing in GAP we already proved the theorem for the non-abelian groups D_8 and Q_8.
Let $\mathbf{G}$ be one of the tree abelian groups of order 8. Then there exist $a, b \in \mathbb{N}_0$ and $c \in \mathbf{G}$ for every polynomial function f, such that

$$\forall x, y \in \mathbf{G}\colon f(x, y) = ax + by + c.$$

From (G2) we know that $f(0, 0) = 0$, hence $c = 0$. Also from (G2) we get $f(x, 0) = x$ which leads us to $a = 1$ and $f(0, y)$ which leads us to $b = 1$. Therefore $f(x, y) = x + y = y + x$ which is the group operation itself. □

Corollary 1.42. *Let* $\mathbf{G} = \langle G, \cdot, ^{-1}, 1\rangle$ *be a group of order 8, and let* $*, i, e \in \mathrm{Pol}(\mathbf{G})$ *such that* $\mathbf{G}' := \langle G, *, i, e\rangle$ *is a group. Then* $\mathbf{G}$ *and* $\mathbf{G}'$ *are isomorphic.*

Proof. Define $x \odot y := xe^{-1}y, h(x) := ex^{-1}e$. Then $\odot \in \mathrm{Pol}_2(\mathbf{G})$ and $h \in \mathrm{Pol}_1(\mathbf{G})$. $\mathbf{G}'' := \langle G, \odot, h, e\rangle$ is a group, isomorphic to $\mathbf{G}$. Due to $\mathrm{Pol}(\mathbf{G}) = \mathrm{Pol}(\mathbf{G}'') = \mathrm{Pol}(\mathbf{G}')$ we can use Theorem 1.41. Hence either $x * y = x \odot y$ or $x * y = y \odot x$. In both cases $\mathbf{G}'$ is isomorphic to $\mathbf{G}''$, thus to $\mathbf{G}$. □

Chapter 2

Expanded Groups

In the following chapter we will discuss some facts about expanded groups and implement a data structure in GAP which we can use for computations with expanded groups.

2.1 Generators of Expanded Groups

As we have seen in Definition 1.21, an expanded group $\langle G, \cdot, ^{-1}, 1, f_1, \dots, f_l \rangle$ is an algebra consisting of a group $\langle G, \cdot, ^{-1}, 1 \rangle$ and additional operations $f_1, \dots, f_l$, i.e. G is closed under the operations $\cdot, ^{-1}, 1, f_1, \dots, f_l$.

Given an expanded group $\mathbf{A} = \langle A, \cdot, ^{-1}, 1, f_1, \dots, f_l \rangle$ and some generators of a subalgebra $\mathbf{B} = \langle B, \cdot, ^{-1}, 1, f_1, \dots, f_l \rangle \leq \mathbf{A}$, i.e. every element of $\mathbf{B}$ can be obtained from the generators by iteratively applying the fundamental operations and B is closed under all expanded group operations, we want to find $|B|$ or we want to check whether some $x \in A$ is also contained in B. We can solve those problems if we can compute the universe of $\mathbf{B}$.

Definition 2.1. We say a subset S of G generates $\langle G, \cdot, ^{-1}, 1, f_1, \dots, f_l \rangle$ as a group (or S is a set of *group generators*) if S generates the group reduct $\langle G, \cdot, ^{-1}, 1 \rangle$.

Example 2.2. The ring of unary polynomial functions with integer coefficients $\langle \mathbb{Z}[x], +, \cdot \rangle$ is generated by the set $\{1, x\}$ as a ring. It is not possible to generate the group $\langle \mathbb{Z}[x], + \rangle$ with 1 and x. A set of group generators of $\langle \mathbb{Z}[x], + \rangle$ is $\{1, x, x^2, x^3, x^4, \dots\}$.

First we want to list the elements of a group generated by a set. Suppose a group $\mathbf{G} = \langle G, \cdot, ^{-1}, 1 \rangle$ and a set of generators $S = \{h_1, \dots, h_n\} \subseteq G$ of the subgroup $\mathbf{H} = \langle H, \cdot, ^{-1}, 1 \rangle$. The universe H consists of the neutral element,

all elements of S and their inverse elements and all elements which can be obtained by iteratively applying the fundamental operations to the elements of S. Hence we can compute the universe by the following algorithm:

Algorithm 1 *GroupUniverseByGenerators*$(S, \langle G, \cdot, ^{-1}, 1\rangle)$

Input: set of generators S and a finite group $\langle G, \cdot, ^{-1}, 1\rangle$, where $S \subseteq G$
Output: set H such that $\langle H, \cdot, ^{-1}, 1\rangle$ is subgroup of $\langle G, \cdot, ^{-1}, 1\rangle$ and is generated by S

$H \leftarrow \{1\} \cup S \cup \{s^{-1} | s \in S\}$
while there exist elements $s \notin H$ and $s_1, s_2 \in H$ such that $s = s_1 \cdot s_2$ **do**
    Add s and s^{-1} to H
end while
return H

If we also check whether H is closed under all additional operations $f_1, \ldots, f_l$ and add missing elements, we get an algorithm for computing the universe H of an expanded group $\mathbf{H} = \langle H, \cdot, ^{-1}, 1, f_1, \ldots, f_l\rangle$ generated by S:

Algorithm 2 *ExpandedGroupUniverseByGenerators*$(S, \langle G, \cdot, ^{-1}, 1, f_1, \ldots, f_l\rangle)$

Input: set of generators S and a finite expanded group $\langle G, \cdot, ^{-1}, 1, f_1, \ldots, f_l\rangle$, where $S \subseteq G$
Output: subuniverse H of the expanded group $\langle G, \cdot, ^{-1}, 1, f_1, \ldots, f_l\rangle$ that is generated by S

$H \leftarrow$ GroupUniverseByGenerators$(S, \langle G, \cdot, ^{-1}, 1\rangle)$
while there exist elements $s \notin H$ and $s_1, \ldots, s_k \in H$ and a k-ary function $f \in \{f_1, \ldots, f_l\}$ such that $s = f(s_1, \ldots, s_k)$ **do**
    $H \leftarrow$ GroupUniverseByGenerators$(H \cup \{s\}, \langle G, \cdot, ^{-1}, 1\rangle)$
end while
return H

Moreover we assume an algorithm *GeneratorsOfGroup*$(G, \cdot)$, which takes a group universe and a group operation and returns a relatively small set, which generates the whole group universe. Such an algorithm is implemented in GAP. We will not discuss the details and refer to the GAP implementation. By composing the two algorithms as

GeneratorsOfGroup(
    ExpandedGroupUniverseByGenerators($S, \langle G, \cdot, ^{-1}, 1, f_1, \ldots, f_l\rangle), \cdot)$)

we can transform a set S of generators of an expanded group $\langle G, \cdot, ^{-1}, 1, f_1, \ldots, f_l \rangle$ to a set of group generators S'.

In Algorithm 2 we have to check

$$\forall i \in \{1, \ldots, l\} \forall s_1, \ldots, s_k \in H \colon f_i(s_1, \ldots, s_k) \in H,$$

which is a lot of effort. For special functions, so-called functions with finite degree, it is not necessary to consider all elements of H.

2.2 Functions with Finite Degree

In this section we will follow [9] and [12]. To simplify notation we will treat all functions in this section as unary functions $f(x)$. By rewriting every function input $x \in S$ as tuple $(x_1, \ldots, x_k) \in S^k$ the following results are also valid for k-ary functions. As always, let's start with a definition:

Definition 2.3. Let $\mathbf{G} = \langle G, \cdot, ^{-1}, 1 \rangle$ be a group and let f be a function from G to G. We define the *degree of* f as the smallest $d \in \mathbb{N}_0$ such that for all $n \in \mathbb{N}$ and for all $x_1, \ldots, x_n \in G$:

$$f(\prod_{i=1}^{n} x_i) \text{ is a product of functions } f(\prod_{i \in S} x_i) \text{ or } f(\prod_{i \in S} x_i)^{-1}$$

for $S \subseteq \{1, \ldots, n\}, |S| \leq d$. If no such d exists, we say that the degree of the function is *infinite*.The degree of a function f is denoted by $\deg(f)$.

The easiest examples are constant functions and homomorphisms:

Example 2.4. Constant functions have degree 0. Consider the constant function $f : G \to G, x \mapsto c$. Then

$$f(x_1 \cdot \ldots \cdot x_k) = c = f(1).$$

Example 2.5. Non-constant homomorphisms are examples for functions of degree 1. Let f be a non-constant homomorphism from $\langle G, \cdot, ^{-1}, 1 \rangle$ to $\langle G, \cdot, ^{-1}, 1 \rangle$. Then

$$f(x_1 \cdot \ldots \cdot x_n) = f(x_1) \cdot \ldots \cdot f(x_n)$$

and $|\{x_i\}| = 1$ for all $i \in \{1, \ldots, n\}$.

For abelian groups we can check the degree of a function using following theorem:

Theorem 2.6. *If $\langle G, \cdot, ^{-1}, 1\rangle$ is an abelian group and f an operation on G, then $deg(f)$ is the smallest $d \in \mathbb{N}_0$ such that*

$$\prod_{S\subseteq\{1,\dots,d+1\}} (f(\prod_{i\in S} x_i))^{(-1)^{|S|}} = 1$$

for all $x_1, \dots, x_{d+1} \in G$.

Proof. We want to show that the definition of finite degree, i.e. $\exists r \in \mathbb{N} \exists S_1, \dots, S_r \subsetneq \{1, \dots, d+1\} \exists e_1, \dots, e_r \in \{-1, 1\}$ such that

$$f(\prod_{i=1}^{d+1} x_i) = \prod_{j=1}^{r} (f(\prod_{i\in S_j} x_i))^{e_j}. \tag{2.1}$$

is equivalent to

$$\prod_{S\subseteq\{1,\dots,d+1\}} (f(\prod_{i\in S} x_i))^{(-1)^{|S|}} = 1 \tag{2.2}$$

for abelian groups.

From (2.2) follows directly

$$\prod_{S\subsetneq\{1,\dots,d+1\}} (f(\prod_{i\in S} x_i))^{(-1)^{d-|S|}} = f(\prod_{i=1}^{d+1} x_i).$$

Hence (2.2) implies (2.1).

Assume that (2.1) holds. Let $T \subseteq \{1, \dots, d+1\}$. From (2.1) follows

$$f(\prod_{i\in T} x_i) = \prod_{j=1}^{r} (f(\prod_{i\in S_j\cap T} x_i))^{e_j}.$$

Hence

$$\prod_{T\subseteq\{1,\dots,d+1\}} (f(\prod_{i\in T} x_i))^{(-1)^{|T|}} = \prod_{T\subseteq\{1,\dots,d+1\}} \Big(\prod_{j=1}^{r} \big(f(\prod_{i\in S_j\cap T} x_i)\big)^{e_j}\Big)^{(-1)^{|T|}}.$$

Since $\langle G, \cdot, ^{-1}, 1\rangle$ is abelian, we have

$$\prod_{T\subseteq\{1,\dots,d+1\}} (f(\prod_{i\in T} x_i))^{(-1)^{|T|}} = \prod_{j=1}^{r} \prod_{T\subseteq\{1,\dots,d+1\}} \big(f(\prod_{i\in S_j\cap T} x_i)\big)^{e_j+(-1)^{|T|}}.$$

For $j \in \{1, \ldots, r\}$, let $k \in \{1, \ldots, d+1\} \setminus S_j$. Then $S_j \cap T = S_j \cap (T \cup \{k_j\})$. Hence

$$\begin{aligned}
&\prod_{T \subseteq \{1,\ldots,d+1\}} (f(\prod_{i \in T} x_i))^{(-1)^{|T|}} \\
=&\prod_{j=1}^{r} \prod_{T \subseteq \{1,\ldots,d+1\} \setminus \{k_j\}} \left((f(\prod_{i \in S_j \cap T} x_i))^{e_j + (-1)^{|T|}} \cdot (f(\prod_{i \in S_j \cap (T \cup \{k_j\})} x_i))^{e_j + (-1)^{|T|+1}} \right) \\
=&1
\end{aligned}$$

□

We can use this theorem to implement a function for finding the degree of functions on abelian groups in GAP:
`FindDegree(G, f, k, max)` returns the degree of the `k`-ary function `f` on `G`, if the degree is less or equal to `max`. Otherwise it returns `false`. See Appendix B for the concrete implementation and Appendix F for the GAP documentation of the function.

```
gap> Read("FindDegree.gap");
gap> f1 := function(x) return x[1]*x[2]; end;
function( x ) ... end
gap> f2 := function(x) return Identity(x); end;
function( x ) ... end
gap> FindDegree(CyclicGroup(5), f1, 2, 10);
1
gap> FindDegree(CyclicGroup(5), f2, 1, 10);
0
gap> IsAbelian(SymmetricGroup(5));
false
gap> FindDegree(SymmetricGroup(5), f1, 2, 3);
Error, Group is not abelian called from
<function "FindDegree">( <arguments> )
 called from read-eval loop at line 4 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
```

Note that it is not possible to compute the degree of a function on a non-abelian group with the implemented algorithm.

Consider an expanded group $\langle G, \cdot, ^{-1}, 1, f_1, \ldots, f_l \rangle$ and a set $\{a_1, \ldots, a_n\} \subseteq G$. In Algorithm 2 we tested whether the group S generated by $a_1, \ldots, a_n$ is closed under an additional operation f of the expanded

group by checking every element of S, i.e.

$$f(x) \in S \text{ for all } x \in S.$$

If an operation f of an expanded group has finite degree d, it is sufficient to check whether

$$f(\prod_{i=1}^{d} c_i) \in S \text{ for all } c_i \in \{a_1^{\pm 1}, \ldots, a_n^{\pm 1}\}.$$

Hence we get following algorithm for finding the universe of an expanded group with additional operations of finite degree:

Algorithm 3 *ExpGroupUniverseByGeneratorsFiniteDeg*$(S, \langle G, \cdot, ^{-1}, 1, f_1, \ldots, f_l\rangle)$

Input: $S \subseteq G$ and a finite expanded group $\langle G, \cdot, ^{-1}, 1, f_1, \ldots, f_l\rangle$, where $f_1, \ldots, f_n$ are unary and have finite degree

Output: set H such that $\langle H, \cdot, ^{-1}, 1, f_1, \ldots, f_l\rangle$ is a subalgebra of the expanded group $\langle G, \cdot, ^{-1}, 1, f_1, \ldots, f_l\rangle$ and is generated by S

```
H ← GroupUniverseByGenerators(S,⟨G, ·, ^-1, 1⟩)
while there exist elements s ∉ H and a function f_i ∈ {f_1, ..., f_l} with
finite degree d_i and c_1, ..., c_{d_i} ∈ S such that s = f(∏_{i=1}^{d_i} c_i^{±1}) do
    H ← GroupUniverseByGenerators(H ∪ {s},⟨G, ·, ^-1, 1⟩)
    S ← S ∪ {s}
end while
return H
```

Remember that we simplified the notation and used only unary functions. Of course, the algorithm above is also valid for k-ary functions by rewriting every function input as tuple.

2.3 Direct Powers and Ideals of Expanded Groups

In Definition 1.25 we already defined direct powers of algebras and thereby of expanded groups. Then following theorems tell us what the generators of a direct product of expanded groups are.

Theorem 2.7. *Let* $\mathbf{A} = \langle A, \cdot, ^{-1}, 1_A, f_1, \ldots, f_n\rangle$ *and* $\mathbf{B} = \langle B, \cdot, ^{-1}, 1_B, f_1, \ldots, f_n\rangle$ *be two expanded groups of the same type* $\mathcal{F}$ *and*

with generators $\{a_1, \ldots, a_k\}$ *and* $\{b_1, \ldots, b_l\}$*, respectively. Then* $\mathbf{A} \times \mathbf{B}$ *is generated by the set*

$$\begin{aligned} G := \{(a_i, 1_B) \mid i \in \{1, \ldots, k\}\} \cup \{(1_A, b_i) \mid i \in \{1, \ldots, l\}\} \\ \cup \{(f_j(1_A, \ldots, 1_A), 1_B) \mid j \in \{1, \ldots, n\}\} \\ \cup \{(1_A, f_j(1_B, \ldots, 1_B)) \mid j \in \{1, \ldots, n\}\}. \end{aligned}$$

Proof. Let (a, b) be an arbitrary but fixed element of $\mathbf{A} \times \mathbf{B}$. First we want to show that $(a, 1_B) \in \langle G \rangle$:
Consider $a = t(g_1, \ldots, g_k) = t(\bar{g})$ with $t \in \mathrm{Clo}_k(A)$ and $g_1, \ldots, g_k \in G$. We prove $(a, 1_B) \in \langle G \rangle$ by induction on the length $L(t)$ of the term t:
Induction basis: $L(t) = 1$, i.e. $a = g_i$. Then $(a, 1_B)$ is clearly in $\langle G \rangle$.
Induction hypothesis: If $a = t(g_1, \ldots, g_k)$ and t is a term function with $L(t) \leq n$, then $(a, 1_B) \in \langle G \rangle$.
Induction step: Let $a = f(s_1(\bar{g}), \ldots, s_m(\bar{g}))$ where $f \in \{f_1, \ldots, f_k\}$ has arity m and $s_1, \ldots, s_m \in \mathrm{Clo}_k(A)$. Hence $t = f(s_1, \ldots, s_m)$. Assume that $L(t) = n + 1$. Then $L(s_1) \leq n, \ldots, L(s_m) \leq n$. By induction hypothesis we know that $(s_1(\bar{g}), 1_B), \ldots, (s_m(\bar{g}), 1_B) \in \langle G \rangle$. Hence

$$(a, f(\bar{1}_B)) = (f(s_1(\bar{g}), \ldots, s_m(\bar{g})), f(\bar{1}_B)) \in \langle G \rangle.$$

By the definition of G we also know that

$$(1_A, f(\bar{1}_B)) \in \langle G \rangle.$$

Due to that,

$$(a, f(\bar{1}_B)) \cdot (1_A, f(\bar{1}_B))^{-1} = (a, 1_B) \in \langle G \rangle.$$

Similarly we can show that $(1_A, b) \in \langle G \rangle$ and therefore $(a, b) \in \langle G \rangle$. Hence we showed that G generates $\mathbf{A} \times \mathbf{B}$. □

By induction on the number of factors, a more general theorem follows directly from the theorem above:

Theorem 2.8. *Let* $\mathbf{A_1}, \ldots, \mathbf{A_n}$ *be expanded groups with the same additional operations* $f_1, \ldots, f_k$*. Let* $G_1, \ldots, G_n$ *be generator sets of* $\mathbf{A_1}, \ldots, \mathbf{A_n}$*, respectively. Then* $\mathbf{A_1} \times \cdots \times \mathbf{A_n}$ *is generated by the set*

$$\begin{aligned} &\bigcup_{i=1}^{n} \{g \in \prod_{l=1}^{n} \mathbf{A_l} \mid g(i) \in G_i \wedge g(j) = 1_{A_j} \text{ for } j \neq i\} \\ \cup &\bigcup_{i=1}^{n} \bigcup_{j=1}^{k} \{g \in \prod_{l=1}^{n} \mathbf{A_l} \mid g(i) = f_j(1_{A_i}, \ldots, 1_{A_i}) \wedge g(r) = 1_{A_r} \text{ for } r \neq i\}. \end{aligned}$$

Proof. Induction base: Let $n = 2$. Then the statement holds by Theorem 2.7.
Induction hypothesis: $\mathbf{A_1} \times \cdots \times \mathbf{A_{n-1}}$ is generated by the set

$$G_{1\ldots n-1} := \bigcup_{i=1}^{n-1} \{g \in \prod_{l=1}^{n-1} \mathbf{A_l} \mid g(i) \in G_i \wedge g(j) = 1_{A_j} \text{ for } j \neq i\}$$
$$\cup \bigcup_{i=1}^{n-1} \bigcup_{j=1}^{k} \{g \in \prod_{l=1}^{n-1} \mathbf{A_l} \mid g(i) = f_j(1_{A_i}, \ldots, 1_{A_i}) \wedge g(r) = 1_{A_r} \text{ for } r \neq i\}.$$

Induction step: By induction hypothesis and Theorem 2.7 $(\mathbf{A_1} \times \cdots \times \mathbf{A_{n-1}}) \times \mathbf{A_n}$ is generated by

$$\{g \in \prod_{l=1}^{n} \mathbf{A_l} \mid (g(1), \ldots, g_{n-1}) \in G_{1\ldots n-1} \wedge g(n) = 1_{A_n}\}$$
$$\cup \{g \in \prod_{l=1}^{n} \mathbf{A_l} \mid g(n) \in G_n \wedge g(j) = 1_{A_j} \text{ for } j \neq n\}$$
$$\cup \bigcup_{j=1}^{k} \{g \in \prod_{l=1}^{n} \mathbf{A_l} \mid g(i) = f_j(1_{A_i}, \ldots, 1_{A_i}) \text{ for } i \leq n-1 \wedge g(n) = 1_{A_n}\}$$
$$\cup \bigcup_{j=1}^{k} \{g \in \prod_{l=1}^{n} \mathbf{A_l} \mid g(i) = 1_{A_i} \text{ for } i \leq n-1 \wedge g(n) = f_j(1_{A_n}, \ldots, 1_{A_n})\}.$$

The elements contained in the set

$$\bigcup_{j=1}^{k} \{g \in \prod_{l=1}^{n} \mathbf{A_l} \mid g(i) = f_j(1_{A_i}, \ldots, 1_{A_i}) \text{ for } i \leq n-1 \wedge g(n) = 1_{A_n}\}$$

can be generated by the set

$$\{g \in \prod_{l=1}^{n} \mathbf{A_l} \mid (g(1), \ldots, g_{n-1}) \in G_{1\ldots n-1} \wedge g(n) = 1_{A_n}\}.$$

Hence $\mathbf{A_1} \times \cdots \times \mathbf{A_n}$ is generated by

$$\bigcup_{i=1}^{n} \{g \in \prod_{l=1}^{n} \mathbf{A_l} \mid g(i) \in G_i \wedge g(j) = 1_{A_j} \text{ for } j \neq i\}$$
$$\cup \bigcup_{i=1}^{n} \bigcup_{j=1}^{k} \{g \in \prod_{l=1}^{n} \mathbf{A_l} \mid g(i) = f_j(1_{A_i}, \ldots, 1_{A_i}) \wedge g(r) = 1_{A_r} \text{ for } r \neq i\}.$$

□

Another interesting concept of expanded groups are ideals:

Definition 2.9. Let $\mathbf{A} = \langle A, \cdot, ^{-1}, 1, f_1, \ldots, f_n \rangle$ be an expanded group. $I \subseteq A$ is called an *ideal* of $\mathbf{A}$ if

(i) I forms a normal subgroup of $\langle A, \cdot, ^{-1}, 1 \rangle$,

(ii) for every $k \in \mathbb{N}$ and every function $f \in \{f_1, \ldots, f_l\}$ of arity k

$$\forall x \in A^k, i \in I^k \colon f(x \cdot i) \cdot (f(x))^{-1} \in I.$$

The following example shows us that not every normal subgroup is an ideal:

Example 2.10. Let S_3 denote the set of permutations on $\{1, 2, 3\}$. Define the following operations on S_3:

$$f_1(x) = x \circ (1, 3)$$
$$f_2(x) = \begin{cases} (1, 2) & \text{if } x = (1, 2, 3), \\ () & \text{otherwise.} \end{cases}$$

Then $\mathbf{A} = \{S_3, \circ, f_1, f_2\}$ is an expanded group. By some GAP computations (details later in Section 2.5) we get following results:
The normal subsets of S_3 are $\{()\}, A_3 = \{(), (1, 2, 3), (1, 3, 2)\}$ and S_3.
While S_3 and $()$ are ideals of $\mathbf{A}$, A_3 is not an ideal of $\mathbf{A}$, because $f_2(() \circ (1, 2, 3)) \circ f_2(())^{-1} = (1, 2)$ is not in A_3.

2.4 A Data Structure for Expanded Groups in GAP

We will now introduce a data structure for expanded groups in GAP. The first idea for this data structure was developed by Peter Mayr, Johannes Kepler University Linz, and Max Neunhöffer, University of St. Andrews. The author of this thesis then implemented the details. See Appendix **??** and **??** for the concrete implementation and Appendix F for the documentation in GAPDoc.

Computing with expanded groups, we want to represent all unary functions as functions taking one element and all k-ary functions, $k > 1$, as functions taking a list with k elements. For example, the ternary operation $f(x, y, z) = x \cdot y \cdot z$ will be represented as follows:

```
gap>  f := function(x) return x[1]*x[2]*x[3]; end;
function( x ) ... end
```

instead of

```
gap>  f := function(x, y, z) return x*y*z; end;
function( x, y, z ) ... end
```

To avoid problems with the abstract representation of groups, all groups should be used as permutation groups (remember the command `AsPermGroup` in the `SONATA` package). We want to explain the new data structure based on an example. Let `e` be an expanded group which is formed by the group S_4 and the two additional operations

$$f_1(x) = x \circ (1,3)$$
$$f_2(x) = \begin{cases} (1,2) & \text{if } x = (1,2,3), \\ () & \text{otherwise.} \end{cases}$$

```
gap> g := SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return x * (1,3); end;
function( x ) ... end
gap> f2 := function(x) if x = (1,2,3) then return (1,2);
> else return (); fi; end;
function( x ) ... end
```

For implementing an expanded group $\langle A, \cdot, ^{-1}, 1, f_1, \ldots, f_n \rangle$ we need to know the group $\langle A, \cdot, ^{-1}, 1, \rangle$, the additional functions $\{f_1, \ldots, f_l\}$ and the arities of all additional functions.

`ExpandedGroup` is implemented as component object, i.e. it contains some components. The usual components of an `ExpandedGroup` are the group reduct `g`, a list `arities` containing the arities of the additional functions and a list `ops` containing the additional functions.
`ExpandedGroup(g, arities, ops)` declares a new object of the type `ExpandedGroup` containing those usual components:

```
gap>   e := ExpandedGroup (g, [1,1],[f1, f2]);
<expanded group Sym( [ 1 .. 4 ] ) with 2 operations>
```

The command `!.` gives access to the single components of an expanded group:

```
gap> e!.g;
Sym( [ 1 .. 4 ] )
gap> e!.arities;
[ 1, 1 ]
gap> e!.ops;
[ function( x ) ... end, function( x ) ... end ]
```

`IsExpandedGroup(e)` returns `true` if `e` is of type `ExpandedGroup` and `false` otherwise:

```
gap> IsExpandedGroup(e);
true
gap> IsExpandedGroup(SymmetricGroup(5));
false
gap> IsExpandedGroup(10);
false
```

Note that in GAP the elements of `e` are exactly the elements of `g`. Hence they contain information about the group operation of `g` and they can be multiplied by the group operation. However the elements of `e` do not contain any information about the additional operations. Therefore we must use the command `Op(e, i, x)` to return the value of the `i`-th operation of `e` applied to the element `x`:

```
gap> (1,2) * (1,2,3);
(1,3)
gap> Op(e, 1, (1,2, 3));
(1,2)
gap> Op(e, 2, (1,2));
()
```

`ExpandedSubgroup(e, gen)` returns a new expanded group, which is a subalgebra of `e` and generated by the elements of the list `gen`. Therefore the new expanded subgroup has the same operations and arities as `e`. Note that there exists no record entry `e!.g` in the subalgebra, but an entry `e!.generators` which is equal to the input list `gen`. Thus we avoid computing the universe of the subalgebra explicitly.

```
gap> s := ExpandedSubgroup(e, [(1,2,3)]);
<expanded group with 2 operations>
gap> IsBound(s!.g);
false
gap> s!.arities;
```

```
[ 1, 2 ]
gap> s!.ops;
[ function( x ) ... end, function( x ) ... end ]
gap> s!.generators;
[ (1,2,3) ]
```

`GroupOfExpandedGroup(e)` returns `e!.g`. If `e!.g` does not exist, the universe of `e!.g` is computed by Algorithm 2, page 23 and `e!.g` is set to the result. If other commands for expanded groups need the component `e!.g` and `e!.g` does not exist, they will call `GroupOfExpandedGroup(e)` to compute `e!.g`. `GroupOfExpandedGroup(e, deg)` can be used if all additional operations have finite degree. `deg` must be a list containing the corresponding degrees of the additional operations. `GroupOfExpandedGroup(e, deg)` computes the universe by Algorithm 3, page 27:

```
gap> s := ExpandedSubgroup(e, [(1,2,3)]);
<expanded group with 2 operations>
gap> IsBound(s!.g);
false
gap> GroupOfExpandedGroup(s);
Group([ (1,2,3), (1,3) ])
gap> IsBound(s!.g);
true
gap> StructureDescription(s!.g);
"S3"
```

`x in e` returns `true` if `x` is an element of `e` and `false` otherwise:

```
gap> s := ExpandedSubgroup(e, [(1,2,3)]);
<expanded group with 2 operations>
gap> IsBound(s!.g);
false
gap> (1,2) in s;
true
gap> IsBound(s!.g);
true
gap> (1,2,4) in s;
false
```

This was an example for a command which computes the universe of the expanded group and sets `e!.g` to the result of the computation if the record entry `e!.g` does not exist.

`Size(e)` returns the order of the expanded group `e`:

```
gap> Size(e);
24
gap> Size(s);
6
```

`Identity(e)` returns the identity element of the expanded group `e`:

```
gap> Identity(e);
()
```

`Random(e)` returns a randomly chosen element of the universe of the expanded group `e`:

```
gap> Random(e);
(2,4,3)
gap> Random(e);
(1,2,3)
```

`IsGroupHomomorphismOfExpandedGroup(e, h)` returns true if `h` is a group homomorphism of the expanded group, i.e. if `h` is a group homomorphism of `e!.g` and the homomorphism property holds also for the additional operations of the expanded group. Otherwise it returns `false`.
If we already know that `h` is a group homomorphism, we can use the command `IsHomomorphismOfExpandedGroup(e, h)` which checks only the homomorphism property for the additional operations of the expanded group `e` and assumes that `h` is a group homomorphism:

```
gap> h := function(x) return x; end;
function( x ) ... end
gap> h := MappingByFunction(e!.g, e!.g, h);
MappingByFunction( Sym( [ 1 .. 4 ] ), Sym( [ 1 .. 4 ] ),
function( x ) ...  end )
gap> IsGroupHomomorphismOfExpandedGroup(e, h);
true
gap> IsGroupHomomorphism(h);
true
gap> IsHomomorphismOfExpandedGroup(e, h);
true
```

`IsReductOfExpandedGroup(e, s)` returns true if the group `s` is a reduct of the expanded group `e`, i.e. if `s` is a subgroup of `e!.g` and `s` is closed under the additional operations of `e`, and `false` otherwise:

```
gap> IsReductOfExpandedGroup(e, Group((1,2,3)) );
false
```

Note that in GAP ideals are represented as groups and not as sets like in Definition 2.9. If a group `I` is an ideal in GAP, the universe of `I` fulfills Definition 2.9.
For the examples about ideals and factor algebras we introduce another expanded group `e2` which has more ideals than the expanded group we used until now:

```
gap> f1 := function(x) return x * (1,3); end;
function( x ) ... end
gap> g:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> e2 := ExpandedGroup(G, [1], [f1]);
<expanded group Sym( [ 1 .. 4 ] ) with 1 operations>
```

`IsIdealOfExpandedGroup(e, s)` returns true if the group `s` forms an ideal of the expanded group `e` and false otherwise:

```
gap> IsIdealOfExpandedGroup(e2,
>Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]));
true
gap> IsIdealOfExpandedGroup(e2,
>Group([ (2,4), (1,4)(2,3), (1,3)(2,4) ]));
false
```

`IdealsOfExpandedGroup(e)` returns a list containing all ideals of the expanded group `e`:

```
gap> IdealsOfExpandedGroup(e2);
[ Sym( [ 1 .. 4 ] ), Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (1,4)(2,3), (1,3)(2,4) ]), Group(()) ]
```

`IdealsByGenerators(e, gen)` returns the ideal of an expanded group `e` which is generated by the elements contained in the list `gen`:

```
gap> IdealByGenerators(e2, [(1,3,2)]);
Group([ (1,3,2), (2,3,4) ])
gap> IdealByGenerators(e2, [()]);
Group(())
gap> IdealByGenerators(e2, [(2,4),(1,4)]);
Group([ (2,4), (1,2,4), (1,2,3) ])
```

`Factor(e, s)` returns the factor algebra `e/s` of an expanded group `e` and an ideal `s` of `e`. The operations of this expanded factor group are given via the natural homomorphism $\bar{f}(x + I) := f(x)$:

```
gap> f := Factor(e2, Group([ (1,4)(2,3), (1,3)(2,4) ]));
<expanded group <group of size 6 with 2 generators>
   with 1 operations>
gap> Display(f!.ops[1]);
function ( x )
    return Image(NaturalHomomorphismByNormalSubgroup(e!.g, s),
       Op( e, i,
         PreImagesRepresentative(
             NaturalHomomorphismByNormalSubgroup(
                 e!.g, s ), x ) ) );
end
gap> flist := AsSortedList(f!.g);
[ <identity> of ..., f1, f2, f1*f2, f2^2, f1*f2^2 ]
gap> Op(f, 1, flist[4]);
<identity> of ...
gap> Op(f, 1, flist[2]);
f2
gap> Size(f);
6
```

It is also possible to define direct powers of expanded groups. Then the operations will operate component-wise. For the examples we will use again the expanded group `e`:

```
gap> g:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return x * (1,3); end;
function( x ) ... end
gap> f2 := function(x) if x = (1,2,3) then return (1,2);
> else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup (g, [1,1],[f1, f2]);
<expanded group Sym( [ 1 .. 4 ] ) with 2 operations>
```

`DirectPower(e, n)` creates $\mathtt{e}^{\mathtt{n}}$, the `n`-th direct power of an expanded group `e`. The new object is of the new type `ExpandedGroupDirectPower` and contains also the record entries `g`, `ops` and `arities`, where `g` is the `n`-th direct power of `e!.g` and the operations stay the same, now acting component-wise.

Moreover, there are record entries `n` and `base`, which save the exponent `n` and the expanded group `e`:

```
gap> d := DirectPower(e, 3);
<expanded group <group of size 13824 with 6 generators>
   with 2 operations>
gap> d!.g;
Group([ (1,2,3,4), (1,2), (5,6,7,8),
        (5,6), (9,10,11,12), (9,10) ])
gap> d!.base;
<expanded group Sym( [ 1 .. 4 ] ) with 2 operations>
gap> d!.n;
3
gap> d!.ops;
[ function( x ) ... end, function( x ) ... end ]
gap> d!.arities;
[ 1, 1 ]
```

`IsExpandedGroupDirectPower(d)` returns `true` if `d` is of type `ExpandedGroupDirectPower` and `false` otherwise:

```
gap> IsExpandedGroupDirectPower(d);
true
gap> IsExpandedGroupDirectPower(e);
false
```

The commands introduced for expanded groups can be used for direct powers of expanded groups, too:

```
gap> Op(d, 1, (1,2)(5,6)(11,12));
(1,2,3)(5,6,7)(9,11,12)
gap> Op(d, 2, (1,2)(5,6,7)(9,10));
(5,6)
gap> Size(d);
13824
gap> (1,4)(9,12) in d;
true
gap> (1,6)(9,12) in d;
false
gap> Random(d);
(2,4)(5,7,6)(9,11)
gap> Identity(d);
()
```

`ExpandedSubgroup(d, gen)` creates a subalgebra of `d` (a direct power of an expanded group) generated by `gen`. In contrast to the subalgebra of an expanded group `exp` defined by `ExpandedSubgroup(exp, gen)`, the subalgebra of `d` contains an additional component `parent`, which stores `d` and is necessary for using the operations component-wise by `Projection` and `Embedding`. Moreover it contains the basic expanded group `d!.base` and the exponent `n` of `d` as components:

```
gap> sub := ExpandedSubgroup(d, [(1,2,3), (5,6), (10,11)]);
<expanded group with 2 operations>
gap> sub!.parent;
Group([ (1,2,3,4), (1,2), (5,6,7,8), (5,6),
        (9,10,11,12), (9,10) ])
gap> sub!.n;
3
gap> sub!.base;
<expanded group Sym( [ 1 .. 4 ] ) with 2 operations>
gap> sub!.generators;
[ (1,2,3), (5,6), (10,11) ]
gap> sub!.arities;
[ 1, 1 ]
gap> sub!.ops;
[ function( x ) ... end, function( x ) ... end ]
gap> Op(sub, 1, (1,3,2));
(2,3)(5,7)(9,11)
gap> Op(sub, 2, (1,3,2)(5,6)(9,10,11));
(9,10)
```

`Projection(d, i)` and `Embedding (d, i)` returns the `i`-th projection and embedding of `d`, respectively:

```
gap> Image(Embedding(d, 3), (1,3));
(9,11)
gap> Image(Embedding(sub, 3), (2,3));
(10,11)
gap> Image(Projection(d, 3), (9,11));
(1,3)
```

2.5 Computing in Expanded Groups with GAP

In Example 2.10 we stated that not every normal subuniverse of S_3 is an ideal of some expansion of S_3. We can now verify this in GAP using our

newly developed data structure:

```
gap> G := SymmetricGroup(3);
Sym( [ 1 .. 3 ] )
gap> f1 := function(x) return x * (1,3); end;
function( x ) ... end
gap> f2 := function(x) if x = (1,2,3) then return (1,2);
> else return (); fi; end;
function( x ) ... end
gap>  e := ExpandedGroup(G, [1, 1], [f1, f2]);
<expanded group Sym( [ 1 .. 3 ] ) with 2 operations>
gap> normal := NormalSubgroups(G);
[ Sym( [ 1 .. 3 ] ), Group([ (1,2,3) ]), Group(()) ]
gap> ideals := IdealsOfExpandedGroup(e);
[ Sym( [ 1 .. 3 ] ), Group(()) ]
```

As a second example how to use GAP for computing in expanded groups we want to find the number of k-ary polynomial functions, $k = 1, 2$ or 3, of the expanded group introduced in Example 1.24: The group $\langle \mathbb{Z}_6, +, -, 0 \rangle$ expanded with the function

$$f : \mathbb{Z}_6 \to \mathbb{Z}_6, x \mapsto \begin{cases} 2 & \text{if } x \in 2\mathbb{Z}_6 = \{0, 2, 4\}, \\ 0 & \text{otherwise.} \end{cases}$$

We start with defining this expanded group in G. Note that the package `SONATA` must be loaded to use the command `AsPermGroup`:

```
gap> G:= AsPermGroup(CyclicGroup(6));
Group([ (1,2), (3,4,5) ])
gap> f1 := function(x) if x^3 = () then return (3,4,5);
> else return ();  fi; end;
function( x ) ... end
gap> e:= ExpandedGroup(G, [1], [f1]);
<expanded group <group of size 6 with 2 generators>
  with 1 operations>
```

The set of k-ary polynomial functions $\mathrm{Pol}_k(\mathbf{A})$ of an algebra $\mathbf{A} = \langle A, F \rangle$ is the subuniverse of $\mathbf{A}^{A^k}$ that is generated by all constants and the k projections. To compute the number of k-ary polynomial functions we only have to determine the size of this subuniverse.

We define a function `NumbPolExp(e, k)` which computes the number of `k`-ary polynomial functions of an expanded group `e`. Note that we use again the function `AsDPElement` which we defined on page 16.

```
1  NumbPolExp := function(e, k)
2  #
3     local H, d, gen, p, gens, t, i, pol;
4  #
5  gens := GeneratorsOfGroup(e!.g);
6  H := Tuples(e!.g,k);
7  d := DirectPower(e, Size(H));
8  gen:= [];
9  # projection functions
10 for i in [1..k] do
11 p:=Tuple(List(H, x -> x[i]));
12 Add(gen, AsDPElement(d,p));
13 od;
14 # constant functions
15 for i in [1..Size(gens)] do
16    t:= Tuple(List( [1..Size(H)], x->gens[i] ));
17    Add(gen, AsDPElement(d,t));
18 od;
19 pol := ExpandedSubgroup(d, gen);
20 return Size(pol);
21 end;
```

Using the function `NumbPolExp(e, k)` we can compute the number of `k`-ary polynomial functions of the expanded group `e` $= \langle \mathbb{Z}_6, +, -, 0, f \rangle$. The command `time` displays the computation time in milliseconds. For the computations we used a Intel®Core$^{\text{TM}}$i7-3770 CPU with 3.4 GHz and 32 GB RAM:

```
gap> Display(NumbPolExp(e, 1));
108
gap> time;
16
gap> Display(NumbPolExp(e, 2));
5832
gap> time;
7084
gap> NumbPolExp(e, 3);
2834352
gap> time;
303221466
```

We want to verify our computational results. The following theorem allows us to compute the number of k-ary polynomial functions of the expanded group $\langle \mathbb{Z}_6, +, -, 0, f \rangle$ by hand:

Theorem 2.11. *Let f be the function*

$$f : \mathbb{Z}_6 \to \mathbb{Z}_6, x \mapsto \begin{cases} 2 & \text{if } x \in 2\mathbb{Z}_6 = \{0, 2, 4\}, \\ 0 & \text{otherwise.} \end{cases}$$

Let $\langle \mathbb{Z}_6, +, -, 0, f \rangle$ be an expanded group and let $k \in \mathbb{N}$. For $r \in \{0,1\}^k$ define

$$d_r : \mathbb{Z}_6^k \to \mathbb{Z}_6, x \mapsto \begin{cases} 2 & \text{if } x \in r + 2\mathbb{Z}_6^k, \\ 0 & \text{otherwise.} \end{cases}$$

Then

$$\begin{aligned} \mathrm{Pol}_k(\langle \mathbb{Z}_6, +, -, 0, f \rangle) = \Big\{ g : \mathbb{Z}_6^k \to \mathbb{Z}_6 \mid & \exists a_1, \ldots, a_k \in \mathbb{Z}_6 \exists b \in \{0,1\} \, \forall r \in \{0,1\}^k \\ & \exists c_r \in \{0,1,2\} \, \forall x \in \mathbb{Z}_6^k : \\ & g(x) = \sum_{i=1}^{k} a_i x_i + b + \sum_{r \in \{0,1\}^k} c_r d_r(x) \Big\}. \end{aligned}$$

Proof. To simplify notation we define

$$\begin{aligned} G := \Big\{ g : \mathbb{Z}_6^k \to \mathbb{Z}_6 \mid & \exists a_1, \ldots, a_k \in \mathbb{Z}_6 \exists b \in \{0,1\} \, \forall r \in \{0,1\}^k \\ & \exists c_r \in \{0,1,2\} \, \forall x \in \mathbb{Z}_6^k : \\ & g(x) = \sum_{i=1}^{k} a_i x_i + b + \sum_{r \in \{0,1\}^k} c_r d_r(x) \Big\} \end{aligned}$$

and

$$\mathbf{A} := \langle \mathbb{Z}_6, +, -, 0, f \rangle.$$

First we show that $\mathrm{Pol}_k(\mathbf{A}) \subseteq G$:
Obviously every projection function $\pi_i(x) = x_i$ is contained in G.

$$\sum_{r \in \{0,1\}^k} d_r(x) = 2$$

and therefore

$$b + c \sum_{r \in \{0,1\}^k} d_r(x) = \begin{cases} 0 & \text{if } b = 0 \text{ and } c = 0, \\ 1 & \text{if } b = 1 \text{ and } c = 0, \\ 2 & \text{if } b = 0 \text{ and } c = 1, \\ 3 & \text{if } b = 1 \text{ and } c = 1, \\ 4 & \text{if } b = 0 \text{ and } c = 2, \\ 5 & \text{if } b = 1 \text{ and } c = 2. \end{cases}$$

Hence all constant functions of $\mathbb{Z}_6$ are contained in G. Let

$$g_1(x) = \sum_{i=1}^{k} a_i x_i + b + \sum_{r \in \{0,1\}^k} c_r d_r(x),$$

$$g_2(x) = \sum_{i=1}^{k} a'_i x_i + b' + \sum_{r \in \{0,1\}^k} c'_r d_r(x)$$

Then

$$g_1(x) + g_2(x) = \sum_{i=1}^{k} (a_i + a'_i) x_i + (b + b') + \sum_{r \in \{0,1\}^k} (c_r + c'_i) d_r(x).$$

Using

$$\sum_{r \in \{0,1\}^k} d_r(x) = 2$$

and computing modulo 6 as usual in $\mathbb{Z}_6$, it is easy to see that

$$\forall g_1, g_2 \in G \colon g_1 + g_2 \in G.$$

Let $g \in G$. We claim that fg is constant on the cosets of $2\mathbb{Z}_6^k$ in $\mathbb{Z}_6^k$. Assume $x \in \mathbb{Z}_6^k$, and $y \in 2\mathbb{Z}_6^k$. Then

$$\begin{aligned} fg(x+y) &= \begin{cases} 2 & \text{if } g(x+y) \in 2\mathbb{Z}_6, \\ 0 & \text{otherwise,} \end{cases} \\ &= \begin{cases} 2 & \text{if } g(x) \in 2\mathbb{Z}_6, \\ 0 & \text{otherwise,} \end{cases} \\ &= fg(x). \end{aligned}$$

Since the image of f is a subset of $2\mathbb{Z}_6^k$, it follows that

$$fg = \sum_{r \in \{0,1\}^k} c'_r d_r \text{ for some } c_r.$$

This proves our claim that fg is constant on the cosets of $2\mathbb{Z}_6^k$ in $\mathbb{Z}_6^k$.

Due to that G is closed under $+$ and f. Therefore $\mathrm{Pol}_k(\mathbf{A}) \subseteq G$.

Now we prove that $G \subseteq \mathrm{Pol}_k(\mathbf{A})$. We have to show that every function $g \in G$ can be generated by the constants and projections. This is the case if all functions d_r can be generated. We claim that

$$2^{k-1} \cdot d_{(1,\dots,1)}(x) = \sum_{S \subseteq \{1,\dots,k\}} (-1)^{|S|} f(\sum_{i \in S} x_i). \tag{2.3}$$

We know that there are exactly 2^k subsets of $\{1, ..., k\}$ and exactly half of them have an even number of elements. Therefore

$$\sum_{0 \le l \le \frac{k}{2}} \binom{k}{2l} = 2^{k-1}.$$

Let $x = (x_1, \dots, x_k)$ such that $\forall i \in \{1, \dots, k\}\colon x_i \in \{1, 3, 5\}$. Then

$$f(\sum_{i \in S} x_i) = \begin{cases} 2 & \text{if } |S| \text{ is even,} \\ 0 & \text{if } |S| \text{ is odd.} \end{cases}$$

Hence we can compute

$$\begin{aligned} \sum_{S \subseteq \{1,\dots,k\}} (-1)^{|S|} f(\sum_{i \in S} x_i) &= \sum_{S \subseteq \{1,\dots,k\}} (-1)^{|S|} \binom{k}{|S|} f(|S| \bmod 2) \\ &= 2 \sum_{0 \le l \le \frac{k}{2}} \binom{k}{2l} = 2 \cdot 2^{k-1} = 2^k. \end{aligned}$$

Now we consider $x = (x_1, \dots, x_k)$ such that $\exists i \in \{1, \dots, k\} : x_i \in \{0, 2, 4\}$. Let $S \subseteq \{1, \dots, k\}$ and $j \in S$ such that $x_j \in \{0, 2, 4\}$. Then

$$f(\sum_{i \in S} x_i) = f(\sum_{i \in S \setminus \{j\}} x_i). \tag{2.4}$$

Hence we can compute

$$\begin{aligned} &\sum_{S \subseteq \{1,\dots,k\}} (-1)^{|S|} f(\sum_{i \in S} x_i) \\ &= \sum_{S \subseteq \{1,\dots,k\} \setminus \{j\}} \Big((-1)^{|S|} f(\sum_{i \in S} x_i) + (-1)^{|S|+|1|} f(\sum_{i \in S \cup \{j\}} x_i) \Big) \\ &= \sum_{S \subseteq \{1,\dots,k\} \setminus \{j\}} \Big((-1)^{|S|} f(\sum_{i \in S} x_i) - (-1)^{|S|} f(\sum_{i \in S} x_i) \Big) = 0. \end{aligned}$$

For the penultimate equality we used (2.4). We showed that

$$\sum_{S\subseteq\{1,\dots,k\}} (-1)^{|S|} f(\sum_{i\in S} x_i) = \begin{cases} 2^k & \text{if } x \in \{1,3,5\}^k, \\ 0 & \text{if otherwise.} \end{cases}$$

This proves our claim (2.3). Since $2^{k-1} d_{(1,\dots,1)} \neq 0$, we have $d_{(1,\dots,1)} \in \mathrm{Pol}_k(\mathbf{A})$.

We get for every $r \in \{0,1\}^k$ the corresponding d_r by substituting $x_i + 1 - r_i$ for x_i:

$$d_r(x) = d_{(1,\dots,1)}(x + (1,\dots,1) - r).$$

Thus $d_r \in \mathrm{Pol}_k(\mathbf{A})$ for every $r \in \{0,1\}^k$ and consequently $G \subseteq \mathrm{Pol}_k(\mathbf{A})$. □

A lemma tells us that if a k-ary function contained in G is 0 for every $x \in \mathbb{Z}_6^k$, then all coefficients a_i, b, c_r of the function are 0:

Lemma 2.12. *Let $k \in \mathbb{N}, a_1, \dots, a_k \in \mathbb{Z}_6$ and $b \in \{0,1\}$. For each $r \in \{0,1\}^k$ let*

$$d_r : \mathbb{Z}_6^k \to \mathbb{Z}_6, x \mapsto \begin{cases} 2 & \text{if } x \in r + 2\mathbb{Z}_6^k, \\ 0 & \text{otherwise.} \end{cases}$$

Let $c_r \in \{0,1,2\}$ and let

$$g(x) = \sum_{i=1}^{k} a_i x_i + b + \sum_{r\in\{0,1\}^k} c_r d_r(x)$$

such that

$$\forall x \in \mathbb{Z}_6^k \colon g(x) = 0.$$

Then

$$\begin{aligned} \forall i \in \{1,\dots,k\} \colon a_i &= 0, \\ b &= 0, \\ \forall r \in \{0,1\}^k \colon c_r &= 0. \end{aligned}$$

Proof. From

$$0 = g(0,\dots,0) = b + c_{(0,\dots,0)} d_{(0,\dots,0)}(0,\dots,0) = b + 2c_{(0,\dots,0)}$$

follows that

$$b = -2c_{(0,\dots,0)}.$$

Due to $b \in \{0,1\}$ and $c_r \in \{0,1,2\}$ we can conclude that

$$b = 0 = c_{(0,\ldots,0)}.$$

Take an i_1 arbitrary but fixed from $\{0,\ldots,k\}$. Consider $x = (x_1,\ldots,x_k)$ such that $x_{i_1} = 2$ and for all $j \in \{0,\ldots,k\} \setminus \{i_1\} : x_j = 0$. Then

$$0 = g(x) = 2a_{i_1} + 2c_{(0,\ldots,0)} = 2a_{i_1}$$

and therefore

$$\forall i \in \{0,\ldots,k\} \colon a_i \in 3\mathbb{Z}_6. \tag{2.5}$$

Take an i_2 arbitrary but fixed from $\{0,\ldots,k\}$. Consider $x = (x_1,\ldots,x_k)$ such that $x_{i_2} = 1$ and for all $j \in \{0,\ldots,k\} \setminus \{i_2\} : x_j = 0$. Due to

$$0 = g(x) = a_{i_2} + 2c_x$$

and $c_x \in \{0,1,2\}$ we get that

$$\forall i \in \{0,\ldots,k\} \colon a_i \in 2\mathbb{Z}_6. \tag{2.6}$$

By (2.5) and (2.6) we know that

$$\forall i \in \{0,\ldots,k\} \colon a_i = 0.$$

Thus

$$\forall x \in \mathbb{Z}_6^k \colon g(x) = \sum_{r \in \{0,1\}^k} c_r d_r(x) = 0.$$

Hence

$$\forall r \in \{0,1\}^k \colon c_r = 0.$$

□

The following corollary allows us to count the number of k-ary polynomial functions:

Corollary 2.13. *Let f be the function*

$$f : \mathbb{Z}_6 \to \mathbb{Z}_6, x \mapsto \begin{cases} 2 & \text{if } x \in 2\mathbb{Z}_6 = \{0,2,4\}, \\ 0 & \text{otherwise.} \end{cases}$$

Then

$$|\mathrm{Pol}_k(\langle \mathbb{Z}_6, +, -, 0, f \rangle)| = 6^k \cdot 2 \cdot 3^{2^k}.$$

Proof. Again we denote the set

$$\Big\{g : \mathbb{Z}_6^k \to \mathbb{Z}_6 \mid \exists a_1, \ldots, a_k \in \mathbb{Z}_6 \exists b \in \{0,1\}\, \forall r \in \{0,1\}^k \\ \exists c_r \in \{0,1,2\}\, \forall x \in \mathbb{Z}_6^k : \\ g(x) = \sum_{i=1}^{k} a_i x_i + b + \sum_{r \in \{0,1\}^k} c_r d_r(x)\Big\}$$

by G. Remember the result of Theorem 2.11:

$$\mathrm{Pol}_k(\langle \mathbb{Z}_6, +, -, 0, f \rangle) = G$$

Let

$$g_1(x) = \sum_{i=1}^{k} a_i x_i + b + \sum_{r \in \{0,1\}^k} c_r d_r(x),$$

$$g_2(x) = \sum_{i=1}^{k} a'_i x_i + b' + \sum_{r \in \{0,1\}^k} c'_r d_r(x)$$

be two functions in G such that $g_1(x) = g_2(x)$ for every $x \in \mathbb{Z}_6^k$. Due to Lemma 2.12 and

$$\begin{aligned} 0 &= g_1(x) - g_2(x) \\ &= \sum_{i=1}^{k} a_i x_i + b + \sum_{r \in \{0,1\}^k} c_r d_r(x) - (\sum_{i=1}^{k} a'_i x_i + b' + \sum_{r \in \{0,1\}^k} c'_r d_r(x)) \\ &= \sum_{i=1}^{k} (a_i - a'_i) x_i + (b - b') + \sum_{r \in \{0,1\}^k} (c_r - c'_r) d_r(x)) \end{aligned}$$

we know that

$$\begin{aligned} \forall i \in \{1, \ldots, k\} : a_i - a'_i &= 0, \\ b - b' &= 0, \\ \forall r \in \{0,1\}^k : c_r - c'_r &= 0. \end{aligned}$$

This means that for every function $g \in G$ there exists exactly one unique representation given by the parameters a_i, b, c_r. Hence we get the number of k-ary polynomial functions by counting all possible combinations of parameters $a_1, \ldots, a_k \in \mathbb{Z}_6, b \in \{0,1\}$, and $c_r \in \{0,1,2\}$ for every $r \in \{0,1\}^k$. Hence

$$|\mathrm{Pol}_k(\langle \mathbb{Z}_6, +, -, 0, f \rangle)| = |\mathbb{Z}_6|^k \cdot |\{0,1\}| \cdot |\{0,1,2\}|^{|\{0,1\}|^k} = 6^k \cdot 2 \cdot 3^{2^k}.$$

□

Therefore our final results are

$$|\mathrm{Pol}_1(\langle \mathbb{Z}_6, +, -, 0, f \rangle)| = 6^1 \cdot 2 \cdot 3^{2^1} = 108,$$
$$|\mathrm{Pol}_2(\langle \mathbb{Z}_6, +, -, 0, f \rangle)| = 6^2 \cdot 2 \cdot 3^{2^2} = 5832,$$
$$|\mathrm{Pol}_3(\langle \mathbb{Z}_6, +, -, 0, f \rangle)| = 6^3 \cdot 2 \cdot 3^{2^3} = 2834352,$$

which are equal to the results computed by GAP.

Chapter 3

Subpower Intersection Problem

In this chapter we will intersect direct powers of groups. For these intersections we will first introduce a special form of generators, the so-called "strong generators". Then we will use those strong generators for developing an algorithm for solving the Subpower Intersection Problem. In the third section we will discuss the implementation of those concepts in GAP. Finally we discuss compatible functions as an application of the Subpower Intersection Problem.

3.1 Strong Generators of Groups

Furst, Hopcraft and Luks introduced in [5] strong generators for permutation groups. We will adapt this concept to direct powers of groups.

First we want to repeat and discuss cosets of groups which we introduced in Example 1.16: Let $\mathbf{H} = \langle H, \cdot, ^{-1}, 1\rangle$ be a subgroup of the group $\mathbf{G} = \langle G, \cdot, ^{-1}, 1\rangle$. Two elements $x, y \in G$ are in the same coset if $x^{-1} \cdot y \in H$. Trivially, H is a coset of H in $\mathbf{G}$. The collection of all cosets of H in $\mathbf{G}$ is denoted by G/H. Lagrange's theorem (cf. [7, p. 331]) tells us, that every coset in G/H has the same size $|H|$ and $|G| = |G/H| * |H|$, where $|G/H|$ is the number of cosets.

Moreover we need following notation: Consider two subsets A and B of a group $\langle G, \cdot, ^{-1}, 1\rangle$. By AB we denote the set $\{a \cdot b \mid a \in A \text{ and } b \in B\}$.

First we want to prove the following lemma:

Lemma 3.1. *Let* $\mathbf{G}$ *be a finite group, and let* $n \in \mathbb{N}$. *Let*

$$G = G_0 \supseteq G_1 \supseteq \cdots \supseteq G_n = \{1\}$$

be a descending chain of subuniverses. Let $T_i, i \in \{1, \ldots, n\}$, *be sets such that*

(i) $\forall i \in \{1, \ldots, n\}\colon \{1\} \subseteq T_i \subseteq G_{i-1}$,

(ii) $\forall r, s \in T_i\colon rG_i = sG_i \Rightarrow r = s$,

(iii) $\forall i, j \in \{1, \ldots, n\}\colon i \leq j \Rightarrow T_jT_i \subseteq T_1 \cdots T_n$.

Then

$$\forall i \in \{1, \ldots, n\}\colon (T_i \cdots T_n)(T_i \cdots T_n) \subseteq T_i \cdots T_n.$$

Proof. Assume $1 \leq i \leq j \leq n$ and $a \in T_j, b \in T_i$. The product $a \cdot b$ is in $T_1 \cdots T_n$ due to (iii). Hence there exist $c_1 \in T_1, \ldots, c_n \in T_n$ such that

$$ab = c_1c_2 \cdots c_n.$$

We claim that

$$T_jT_i \subseteq T_i \cdots T_n, \tag{3.1}$$

i.e.

$$ab = c_i \cdots c_n.$$

From (i) follows that $ab \in G_{i-1}$. Therefore

$$c_1 \cdots c_{i-1}G_{i-1} = G_{i-1}.$$

By induction we want to show that

$$\forall k \in \{1, \ldots, n\} \forall c_1 \in T_1, \ldots, c_k \in T_k\colon c_1 \cdots c_kG_k = G_i \Rightarrow c_1 = \ldots = c_k = 1.$$

Induction base: Let $k = 1$. Then $c_1G_1 = G_1$. From (ii) follows that $c_1 = 1$.
Induction hypothesis: Let $k \in \{1, \ldots, n\}$ and assume $\forall c_1 \in T_1, \ldots, c_{k-1} \in T_{k-1}$:

$$c_1 \cdots c_{k-1}G_{k-1} = G_{k-1} \Rightarrow c_1 = \ldots = c_{k-1} = 1 \tag{3.2}$$

Induction step: Let $c_1 \in T_1, \ldots, c_k \in T_k$ such that $c_1 \ldots c_kG_k = G_k$. Since $G_k \subseteq G_{k-1}$, we have $c_1 \ldots c_{k-1}c_kG_{k-1} = G_{k-1}$. From $c_k \in G_k$ follows $c_1 \ldots c_{k-1}G_{k-1} = G_{k-1}$. By induction hypothesis (3.2) $c_1 = \ldots = c_{k-1} = 1$. Hence $c_kG_k = G_k$ and together with (ii) follows $c_k = 1$.

From $c_1 = \ldots = c_{i-1} = 1$ and (i) follows directly our claim (3.1).

Now we show by induction

$$\forall i \in \{1, \ldots, n\}\colon (T_i \cdots T_n)(T_i \cdots T_n) \subseteq T_i \cdots T_n.$$

Induction base: Let $i = n$. Then $T_n T_n \subseteq T_n$ follows from (3.1).
Induction hypothesis: Let $i \in \{1, \ldots, n\}$ and assume

$$(T_{i+1} \ldots T_n)(T_{i+1} \ldots T_n) \subseteq T_{i+1} \ldots T_n. \tag{3.3}$$

Induction step: First we consider some $j \in \{i, \ldots, n\}$. We claim that

$$T_j(T_i \cdots T_n) \subseteq T_i \cdots T_n. \tag{3.4}$$

By (3.2)

$$(T_j T_i)(T_{i+1} \cdots T_n) \subseteq (T_i T_{i+1} \cdots T_n)(T_{i+1} \cdots T_n).$$

By (3.3)

$$T_i(T_{i+1} \cdots T_n)(T_{i+1} \cdots T_n) \subseteq T_i(T_{i+1} \cdots T_n.)$$

Hence (3.4) is proved. Now $(T_i \cdots T_n)(T_i \cdots T_n) \subseteq T_i \cdots T_n$ follows from (3.4). □

Definition 3.2. Let $\mathbf{H}$ be a finite group. Let $n \in \mathbb{N}$ and $\mathbf{G} \leq \mathbf{H}^n$. For $i \in \{0, \ldots, n\}$, let $G_i = \{a \in G \mid a_j = 1 \text{ for } j \leq i\}$ and let T_i be a transversal for G_i in G_{i-1}. Then $T_1, \ldots, T_n$ are *strong generators* for $\mathbf{G}$.

Note that every set G_i forms a subgroup of $\mathbf{G}$ and we have a descending chain of subuniverses

$$G = G_0 \supseteq G_1 \supseteq \cdots \supseteq G_n = I.$$

Lemma 3.3. *Let H be a finite group and $n \in \mathbb{N}$. Let $T_1, \ldots, T_n$ be strong generators for $\mathbf{G} \leq \mathbf{H}^n$. Then*

(i) $\forall g \in G \exists! a_1 \in T_1, \ldots, \exists! a_n \in T_n \colon g = a_1 \cdots a_n$,

(ii) $\forall i \in \{1, \ldots, n\} \colon |T_i| \leq |H|$.

Proof. For $i \in \{1, \ldots, n\}$, let $G_i = \{a \in G \mid a_j = 1 \text{ for } j \leq i\}$ and let T_i be transversal for G_i in G_{i-1}. Consider an element $g := (g_1, \ldots, g_n) \in G$. Then there exists an unique element $k := (1, k_2, \ldots, k_n) \in G$ such that $(g_1, g_2 \cdot k_2^{-1}, \ldots, g_n \cdot k_n^{-1})$ is an element of T_1. The equation

$$(g_1, \ldots, g_n) = (g_1, g_2 \cdot k_2^{-1}, \ldots, g_n \cdot k_n^{-1}) \cdot (1, k_2, \ldots, k_n)$$

shows us that $(g_1, \ldots, g_n)$ is an element of $T_1 G_1$. This representation as product of one element of T_1 and one element of G_1 is possible for every element of $\mathbf{G}$. Therefore

$$\mathbf{G} = G_0 = T_1 G_1.$$

Using that fact inductively, we get

$$\mathbf{G} = T_1 T_2 \cdots T_n G_n.$$

We know that $G_n = \{\bar{1}\}$, i.e. G_n contains only the neutral element $(1, \ldots, 1)$. Therefore we can also write

$$\mathbf{G} = T_1 T_2 \ldots T_n.$$

This tells us that any element g in $\mathbf{G}$ can be written uniquely in the form

$$g = a_1 \cdots a_n,$$

where $a_i \in T_i$. If we compute the product of the a_i's from left to right, the i-th coordinate is fixed after the computation of $a_1 \cdots a_i$ and will not change anymore.

Consider two elements $(1, \ldots, 1, a_{i+1}, \ldots, a_n)$ and $(1, \ldots, 1, b_{i+1}, \ldots, b_n)$ from G_i. They are in the same coset of G_i/G_{i+1} if

$$\begin{array}{lll} & (1, \ldots, 1, a_{i+1}, \ldots, a_n)^{-1} \cdot (1, \ldots, 1, b_{i+1}, \ldots, b_n) & \in G_{i+1} \\ \iff & (1, \ldots, 1, a_{i+1}^{-1}, \ldots, a_n^{-1}) \cdot (1, \ldots, 1, b_{i+1}, \ldots, b_n) & \in G_{i+1} \\ \iff & (1, \ldots, 1, a_{i+1}^{-1} \cdot b_{i+1}, \ldots, a_n^{-1} \cdot b_n) & \in G_{i+1} \\ \iff & a_{i+1}^{-1} \cdot b_{i+1} = 1 & \\ \iff & a_{i+1} = b_{i+1}. & \end{array}$$

This means that two elements of G_i are in the same coset of G_{i+1} in G_i, if they have the same $(i+1)$-th coordinate. There are only m different elements of $|H|$, hence there can be at most m different cosets of G_{i+1} in G_i. Therefore $|T_i| \leq m$ for every $i \in \{1, \ldots, n\}$. □

We want to construct the sets $T_1, \ldots, T_n$ such that an element g is in G if and only if g can be expressed as $a_1 \cdots a_n$ where $a_i \in T_i$. This we will call the *canonical representation* of g.
Our computations starts with the sets $T_1 = T_2 = \cdots = T_n = \{(1, \ldots, 1)\}$ where $\bar{1} = (1, \ldots, 1)$ is the neutral element of G.
The algorithm *sift*$(x, T_1, \ldots, T_n)$, which is defined below, modifies the sets $T_1, \ldots, T_n$ by inserting at most one new coset representative such that x can be written in canonical form.

Algorithm 4 *sift*$(x, T_1, \ldots, T_n)$

Input: $x \in \mathbf{G}$, sets $T_1, \ldots, T_n$ such that for every $i \in \{1, \ldots, n\}$ T_i is contained in a transversal of G_i in G_{i-1}

Output: modified $T_1, \ldots, T_n$ such that for every $i \in \{1, \ldots, n\}$ T_i is contained in a transversal of G_i in G_{i-1} and $x \in T_1 \cdots T_n$

$i \leftarrow 1$
while $i \neq n+1$ and $\exists y \in T_i$ such that $y_i = x_i$ **do**
  $i \leftarrow i + 1$
  $x \leftarrow y^{-1} \cdot x$
end while
if $i \neq n+1$ **then**
  insert x in T_i
end if
return $(T_1, \ldots, T_n)$

As an example consider the following sets:

$$T_1 = \{(1, \ldots, 1), a\},$$
$$T_2 = \cdots = T_n = \{(1, \ldots, 1)\},$$

where $a = (a_1, \ldots, a_n)$ and $a_1 \neq 1$. We want to find a canonical representation for an element $b = (a_1, b_2, \ldots, b_n)$ which has a second coordinate not equal to the second coordinate of a, i.e. $b_2 \neq a_2$, by using *sift*$(x, T_1, \ldots, T_n)$: We start with T_1 and see, that a and b have the same first coordinate. Therefore we continue with the modified element $b' = a^{-1} \cdot b = (1, a_2^{-1} \cdot b_2, \ldots, a_n^{-1} \cdot b_n)$. In T_2 exists no entry with the same second coordinate as b' yet. Therefore we insert b' in T_2. Hence we get the modified sets

$$T_1 = \{(1, \ldots, 1), a\},$$
$$T_2 = \{(1, \ldots, 1), b'\}$$
$$T_3 = \cdots = T_n = \{(1, \ldots, 1)\},$$

which contains the factors of the canonical representation of b, i.e.

$$b = a \cdot b' \cdot \bar{1} \cdot \ldots \cdot \bar{1}.$$

To find all strong generators of G we have to apply Algorithm 4 to all given generators of G and to all products of pairs of elements contained in the sets $T_1, \ldots, T_n$ until every such product can be written in canonical form with

respect to $T_1, \ldots, T_n$.

The algorithm *FindStrongGens*$(g_1, \ldots, g_k)$ constructs the transversals $T_1, \ldots, T_n$ which contain all strong generators of G by executing the three described parts:

(i) Initialize the transversals as the sets containing the neutral element of G.

(ii) Sift all generators using Algorithm 4.

(iii) Sift all products of pairs of elements in T using Algorithm 4, until every such product can be written in canonical form.

Algorithm 5 *FindStrongGens*$(g_1, \ldots, g_k)$

Input: $g_1, \ldots, g_k \in H^n$
Output: set of transversals $T_1, \ldots, T_n$ containing the strong generators of the subuniverse G of H^n generated by $g_1, \ldots, g_n$

$n \leftarrow Length(g_1)$
for $i = 1$ to n **do**
  $T_i \leftarrow \{\bar{1}\}$
end for
for $i = 1$ to k **do**
  $(T_1, \ldots, T_n) \leftarrow \text{sift}(g_i, T_1, \ldots, T_n)$
end for
while there exists $i, j \in \{1, \ldots, n\}$ and $a \in T_i$ and $b \in T_j$ such that $\text{sift}(a \cdot b, T_1, \ldots, T_n)$ has not been computed yet **do**
  $(T_1, \ldots, T_n) \leftarrow \text{sift}(a \cdot b, T_1, \ldots, T_n)$
end while
return $(T_1, \ldots, T_n)$

Theorem 3.4. *Let $k, n, m \in \mathbb{N}$ and let $\mathbf{H}$ be a finite group with $|\mathbf{H}| = m$. Let $g_1, \ldots, g_k \in \mathbf{H}^n$ and let G be the subuniverse of $\mathbf{H}^n$ which is generated by $g_1, \ldots, g_k$. Then Algorithm 5 determines strong generators $T_1, \ldots, T_n$ for G in $O(n^3m^3 + knm)$ time.*

Proof. Correctness of Algorithm 5:
Let $G := \langle g_1, \ldots, g_k \rangle$ and let $G_i := \{a \in G \mid a_1 = \cdots = a_{i-1} = 1\}$.
First claim: For every $i \in \{1, \ldots, n\}$ in every step of Algorithm 5 T_i is a subset of a transversal for G_i in G_{i-1} and $1 \in T_i$.
Proof of the first claim:

Obviously $1 \in T_i$.
Every set T_i can be changed only by calling $\text{sift}(g_i, T_1, \ldots, T_n)$ or $\text{sift}(ab, T_1, \ldots, T_n)$. Hence the element x that gets sifted is either a generator of G or generated by the current sets $T_1, \ldots, T_n \subseteq G$. In particular $x \in G$. Sift changes x by multiplying it with elements from $T_1, \ldots, T_n$. Hence in every step of sift, we still have $x \in G$. Furthermore $x \in G_i$ in step i. If a new element x is added to some old T_i by sift, then because the condition in the while-loop is not satisfied in step i. That is, there exists no $y \in T_i$ such that $y_i = x_i$. Hence the new set $T_i \cup \{x\}$ is indeed a subset of a transversal for G_i in G_{i-1}.

When Algorithm 5 finishes, its output $T_1, \ldots, T_n$ will also satisfy

(i) $g_1, \ldots, g_k \in T_1 \cdots T_n$ and

(ii) $\forall i, j \in \{1, \ldots, n\} \colon T_i T_j \subseteq T_1 \cdots T_n$.

Second claim: $G = T_1 \cdots T_n$
Proof of the second claim:
By the first claim we know that for all $i \in \{1, \ldots, n\} \colon T_i \subseteq G$. Hence

$$G \supseteq T_1 \cdots T_n.$$

Since $T_1 \cdots T_n$ forms a subgroup by Lemma 3.1 and $g_1, \ldots, g_k \in T_1 \cdots T_n$, we obtain

$$G \subseteq T_1 \cdots T_n.$$

Third claim: For every $i \in \{1, \ldots, n\}$ T_i is a transversal of G_i in G_{i-1}.
Proof of the third claim:
Let $i \in \{1, \ldots, n\}$. Consider $g \in G_{i-1}$. Then there exist $t_1 \in T_1, \ldots, t_n \in T_n$ such that $g = t_1 \cdots t_n$. We want to show that there exists a representative for g in T_i.
First we show by induction that $t_1 = \cdots = t_{i-1} = 1$. Let $1 \leq j \leq i - 1$.
We know that

$$G_j = gG_j = t_1 \cdots t_{i-1} G_j = t_1 \cdots t_j G_j.$$

Induction base: From $G_1 = t_1 G_1$ follows $t_1 = 1$ by the first claim.
Induction hypothesis: If $G_{j-1} = t_1 \cdots t_{j-1} G_{j-1}$, then $t_1 = \ldots = t_{j-1} = 1$.
Induction step: From $G_j = t_1 \cdots t_{j-1} t_j G_j$ follows $G_j = t_j G_j$ by induction hypothesis. By the first claim we obtain $t_j = 1$.
Thus $g = t_i \cdots t_n$. From that follows $gG_i = t_i G_i$. Hence g has a representative in T_i, namely t_i, and T_i is a transversal for G_i through G_{i-1}.

Therefore $T_1, \ldots, T_n$ are strong generators for G.

Complexity of Algorithm 5:
Every application of Algorithm 4 takes $O(nm)$ in the worst case because there are n transversals and every one of the transversals contains at most m elements.

(i) The initialization of $T_1, \ldots T_n$ takes $O(n)$ steps.

(ii) k generators are sifted. Hence the sifting of the generators takes $O(knm)$ steps.

(iii) There are at most $(nm)(nm)$ pairs of elements from the n transversals with at most m elements. Hence sifting the pairs takes $O(n^3m^3)$ steps.

Therefore the running time of the whole Algorithm 5 is $O(n^3m^3 + knm)$. □

The table containing the coset representatives of a direct power (as computed in this proof) can be used for solving the *Subpower Membership Problem* or for computing the order of G:

Corollary 3.5 (cf. [5, p.38])**.** *Let $n \in \mathbb{N}$ and let $\mathbf{H}$ be a finite group. Let $g_1, \ldots, g_k \in \mathbf{H}^n$ and let G be the subuniverse of $\mathbf{H}^n$ which is generated by $g_1, \ldots, g_k$. For $i \in \{1, \ldots, n\}$, let $G_i = \{a \in G \mid a_j = 1 \text{ for } j \leq i\}$ and T_i the set of coset representatives of G_{i-1}/G_i. Then*

$$|G| = |T_1| \cdot \ldots \cdot |T_n|.$$

Proof. The order of G is the product of the sizes of G_{i-1}/G_i for $i \in \{1, \ldots, n\}$. The size of G_{i-1}/G_i is equal to the number of coset representatives contained in T_i. □

Definition 3.6. The *Subpower Membership Problem* (cf. [9, p.1]) for an algebra $\mathbf{A}$ is the following decision problem:

Input	$\{a_1, \ldots, a_k\} \subseteq A^n, x \in A^n$
Problem	Is x in the subalgebra $\langle a_1, \ldots, a_k \rangle$ of $\mathbf{A}^n$ which is generated by $a_1, \ldots, a_k$?

Corollary 3.7 (cf. [5, p.38])**.** *Let $n \in \mathbb{N}$ and let $\mathbf{H}$ be a finite group with m elements. Let $g_1, \ldots, g_k \in \mathbf{H}^n$ and let G be the subuniverse of $\mathbf{H}^n$ which is generated by $g_1, \ldots, g_k$. Then $x \in G$ can be decided $O(n^3m^3 + knm)$.*

Proof. If we compute the transversals $T_1, \ldots, T_n$ with coset representatives of the subgroup $\langle g_1, \ldots, g_k \rangle$ of $\mathbf{H}^n$, we can check the membership simply by running $\text{sift}(b, T_1, \ldots, T_n)$. If b can be written in canonical form without adding a new element to the transversals, it is an element of the subgroup. Due to Theorem 3.4 these computations are in $O(n^3m^3 + knm)$. □

Hence the Subpower Membership Problem for finite groups is in P.

3.2 The Subpower Intersection Problem

In this section we want to find generators for the intersection of subgroups of a direct power of a group. We start with a formal definition of intersection:

Definition 3.8. Let $\mathbf{S}$ be a finite group. Let $\mathbf{G}$ and $\mathbf{H}$ be subgroups of $\mathbf{S}$. Then the *intersection* of $\mathbf{G}$ and $\mathbf{H}$, denoted by $\mathbf{G} \cap \mathbf{H}$, is the subgroup of $\mathbf{S}$ with universe $G \cap H$.

Definition 3.9. Let $\mathbf{S}$ be a finite group and $n \in \mathbb{N}$. The problem of finding generators of the intersection of two subgroups $\mathbf{G}, \mathbf{H}$ of $\mathbf{S}^n$ given by their generators is called the *Subpower Intersection Problem*:

Input	$\{g_1, \ldots, g_k\} \subseteq S^n$,
	$\{h_1, \ldots, h_l\} \subseteq S^n$
Problem	Find generators for the intersection of the two subgroups of $\mathbf{S}^n$
	that are generated by $\{g_1, \ldots, g_k\}$ and $\{h_1, \ldots, h_l\}$.

Of course, we can list all elements of the two groups by Algorithm 2.1 and use all elements contained in both group universes to form the universe of the intersection. However, in general Algorithm 2.1 cannot list all elements of a group in polynomial time in the size of the input, namely $k, l, n, |S|$. Therefore we want to develop another algorithm.

For computing the generators of $G \cap H$ we choose the chain of subgroups

$$\begin{aligned} S_0 \supseteq S_1 \supseteq \ldots \supseteq S_n = H \supseteq S_{n+1} \supseteq \ldots \\ \ldots \supseteq S_{2n} = G \cap H \supseteq \ldots \supseteq S_{3n} = \{\bar{1}\} \end{aligned}$$

where

$$\begin{aligned} S_0 &= \langle g_1, \ldots, g_k, h_1, \ldots, h_l \rangle, \\ \forall i \in \{1, \ldots, n\}\colon\ S_i &= \{s \in S_0 \mid \exists h \in H\, \forall 1 \leq j \leq i : s(j) = h(j)\}, \\ \forall i \in \{n+1, \ldots, 2n\}\colon\ S_i &= \{s \in H \mid \exists g \in G\, \forall 1 \leq j \leq i - n : s(j) = g(j)\}, \\ \forall i \in \{2n+1, \ldots, 3n\}\colon\ S_i &= \{g \in G \cap H \mid \forall 1 \leq j \leq i - 2n : g(j) = 1\}. \end{aligned}$$

Similarly to the transversals $T_1, \ldots, T_n$ in the proof of Theorem 3.4 we construct now $3n$ transversals labelled T_1 to T_{3n} such that for each i in $\{1, \ldots, 3n\}$ T_i is a transversal of S_{i-1}/S_i.
We want to construct the sets $T_1, \ldots, T_{3n}$ such that an element g is in S if and only if g can be expressed as $a_1 \cdot a_2 \cdots a_{3n}$ where $a_i \in T_i$. Again we will call this the *canonical representation* of g. Then the elements of the transversals $T_{2n+1}, \ldots, T_{3n}$ are the strong generators of $G \cap H$ and an element g is in $G \cap H$ if and only if g can be expressed as $a_{2n+1} \cdot a_{2n+2} \cdots a_{3n}$ where $a_i \in T_i$. The algorithm *sift2*$(x, T_1, \ldots, T_{3n}, stG, stH)$, which is defined below, modifies the sets $T_1, \ldots, T_{3n}$ by inserting at most one new coset representative such that x can be written in canonical form.

Algorithm 6 *sift2*$(x, n, T_1, \ldots, T_{3n}, stG, stH, stS_0)$

Input: $x \in \mathbf{S^n}$, exponent n of S^n, sets $T_1, \ldots, T_{3n}$ such that for every $i \in \{1, \ldots, 3n\}$ T_i is contained in a transversal of S_i in S_{i-1}, sets of strong generators stG, stH of the subuniverses G, H of $\mathbf{S}^n$, set of strong generators stS_0 of S_0
Output: modified sets $T_1, \ldots, T_{3n}$, such that for every $i \in \{1, \ldots, 3n\}$, T_i is contained in a transversal of S_i in S_{i-1} and $x \in T_1 \cdots T_{3n}$

$i \leftarrow 0$
while $1 \leq i \leq 3n$ and there exists a $y \in T_i$ such that $y^{-1} \cdot x \in S_i$ as given on page 57 **do**
    $i \leftarrow i + 1$
    $x \leftarrow y^{-1} \cdot x$
end while
if $i \neq 3n + 1$ **then**
    insert x in T_i
end if
return $(T_1, \ldots, T_{3n})$

Note that the strong generators of the subgroups G, H and S_0 are necessary to decide $y^{-1} \cdot x \in S_i$ for $i \in \{1, \ldots, 2n\}$. For testing the membership $y^{-1} \cdot x \in S_i$ for $i \in \{2n + 1, \ldots, 3n\}$ it is sufficient to test if $y_{i-2n} = x_{i-2n}$.

The following algorithm *StrongGensForIntersection*$(g_1, \ldots, g_k, h_1, \ldots, h_l)$ constructs the transversals $T_1, \ldots, T_{3n}$ and returns the transversals $T_{2n+1}, \ldots, T_{3n}$ which contain the strong generators of $G \cap H$ by executing the following four parts:

(i) Initialize the sets T_i containing the neutral element of G.

(ii) Compute the strong generators of G and H.

(iii) Sift all generators using Algorithm 6.

(iv) Sift all products of pairs of elements in T using Algorithm 6, until every such product can be written in canonical form.

Algorithm 7 *StrongGensForIntersection*$(g_1, \ldots, g_k, h_1, \ldots, h_l)$

Input: $g_1, \ldots, g_k, h_1, \ldots, h_l \in \mathbf{S}^n$

Output: strong generators $T_{2n+1}, \ldots, T_{3n}$ for the subgroup $\langle g_1, \ldots, g_k \rangle \cap \langle h_1, \ldots, h_l \rangle$

$n \leftarrow Length(g_1)$
for $i = 1$ to $3n$ **do**
  $T_i \leftarrow \{\bar{1}\}$
end for
$stG \leftarrow FindStrongGens(g_1, \ldots, g_k)$
$stH \leftarrow FindStrongGens(h_1, \ldots, h_l)$
$stS_0 \leftarrow FindStrongGens(g_1, \ldots, g_k, h_1, \ldots, h_l)$
for $i = 1 \to k$ **do**
  $(T_1, \ldots, T_{3n}) \leftarrow \text{sift2}(g_i, n, T_1, \ldots, T_{3n}, stG, stH, stS_0)$
end for
for $i = 1 \to l$ **do**
  $(T_1, \ldots, T_{3n}) \leftarrow \text{sift2}(h_i, n, T_1, \ldots, T_{3n}, stG, stH, stS_0)$
end for
while there exists $i, j \in \{1, \ldots, 3n\}$ and $a \in T_i$ and $b \in T_j$ such that $\text{sift2}(a \cdot b, T_1, \ldots, T_{3n}, stG, stH, stS_0)$ has not been computed yet **do**
  $(T_1, \ldots, T_{3n}) \leftarrow \text{sift2}(a \cdot b, T_1, \ldots, T_{3n}, stG, stH, stS_0)$
end while
return $(T_{2n+1}, \ldots, T_{3n})$

Theorem 3.10 (cf. [5, Theorem 6])**.** *Let $k, n, m \in \mathbb{N}$ and let $\mathbf{S}$ be a finite group with universe S and $|S| = m$. Let $g_1, \ldots, g_k \in \mathbf{S}^n$ and let G be the subuniverse of $\mathbf{S}^n$ which is generated by $g_1, \ldots, g_k$. Let $h_1, \ldots, h_l \in \mathbf{S}^n$ and let H be the subuniverse of $\mathbf{S}^n$ which is generated by $h_1, \ldots, h_l$. Then Algorithm 7 determines strong generators for $G \cap H$ in $O(n^4m^4 + (k+l)n^2m^2)$.*

Proof. Correctness of the Algorithm 7:
Let $S_0 := \langle g_1, \ldots, g_k, h_1, \ldots h_l \rangle$ and let

$$\forall i \in \{1, \ldots, n\}\colon\ S_i = \{s \in S_0 \mid \exists h \in H\, \forall 1 \leq j \leq i : s(j) = h(j)\},$$
$$\forall i \in \{n+1, \ldots, 2n\}\colon\ S_i = \{s \in H \mid \exists g \in G\, \forall 1 \leq j \leq i-n : s(j) = g(j)\},$$
$$\forall i \in \{2n+1, \ldots, 3n\}\colon\ S_i = \{g \in G \cap H \mid \forall 1 \leq j \leq i-2n : g(j) = 1\}.$$

First claim: For every $i \in \{1, \ldots, 3n\}$ in every step of Algorithm 5 T_i is a subset of a transversal for S_i in S_{i-1} and $1 \in T_i$.
Proof of the first claim:
Obviously $1 \in T_i$.
Every set T_i can be changed only by calling sift2$(g_i, T_1, \ldots, T_{3n}, stG, stH, stS_0)$ or sift2$(ab, T_1, \ldots, T_{3n}, stG, stH, stS_0)$. Hence the element x that gets sifted is either a generator of G or generated by the current sets $T_1, \ldots, T_{3n} \subseteq S_0$. In particular $x \in S_0$. Sift changes x by multiplying it with elements from $T_1, \ldots, T_{3n}$. Hence in every step of sift2, we still have $x \in S_0$. Furthermore $x \in S_i$ in step i. If a new element x is added to some old T_i by sift2, then because the condition in the while-loop is not satisfied in step i. That is, there exists no $y \in T_i$ such that $y^{-1} \cdot x \in S_i$. Hence the new set $T_i \cup \{x\}$ is indeed the subset of a transversal for S_i in S_{i-1}.

When Algorithm 7 finishes, $T_1, \ldots, T_{3n}$ will also satisfy

(i) $g_1, \ldots, g_k, h_1, \ldots, h_l \in T_1 \cdots T_{3n}$ and

(ii) $\forall i, j \in \{1, \ldots, 3n\}\colon T_i T_j \subseteq T_1 \cdots T_{3n}$.

Second claim: $S_0 = T_1 \cdots T_{3n}$
Proof of the second claim:
By the first claim we know that for all $i \in \{1, \ldots, 3n\}\colon T_i \subseteq S_0$. Hence

$$S_0 \supseteq T_1 \cdots T_{3n}.$$

Since $T_1 \cdots T_{3n}$ forms a subgroup by Lemma 3.1 and $g_1, \ldots, g_k, h_1, \ldots, h_l \in T_1 \cdots T_{3n}$, we obtain

$$S_0 \subseteq T_1 \cdots T_{3n}.$$

Third claim: For every $i \in \{1, \ldots, 3n\}$ T_i is a transversal of S_i in S_{i-1}.
Proof of the third claim:
Let $i \in \{1, \ldots, 3n\}$. Consider $g \in S_{i-1}$. Then there exist $t_1 \in T_1, \ldots,$ $t_{3n} \in T_{3n}$ such that $g = t_1 \cdots t_{3n}$. We want to show that there exists a representative for g in T_i.
First we show by induction that $t_1 = \cdots = t_{i-1} = 1$. Let $1 \leq j \leq i-1$.
We know that

$$S_j = gS_j = t_1 \cdots t_{i-1} S_j = t_1 \cdots t_j S_j.$$

Induction base: From $S_1 = t_1 S_1$ follows $t_1 = 1$ by the first claim.
Induction hypothesis: If $S_{j-1} = t_1 \cdots t_{j-1} S_{j-1}$, then $t_1 = \ldots = t_{j-1} = 1$.
Induction step: From $S_j = t_1 \cdots t_{j-1} t_j S_j$ follows $S_j = t_j S_j$ by induction hypothesis. By the first claim we obtain $t_j = 1$.

Thus $g = t_i \cdots t_n$. From that follows $gS_i = t_i S_i$. Hence g has a representative in T_i, namely t_i and T_i is a transversal for S_i through S_{i-1}.

Therefore $\forall i \in \{1, \ldots, n\}\colon S_{i-1} = T_i \cdots T_n$.

Complexity of the Algorithm 7:
Using the strong generators of G, H and S_0, answering the membership problems $y^{-1} \cdot x \in S_i$ is in $O(nm)$ for $i \in \{1, \ldots, 2n\}$, because we can use a modified version of Algorithm 4 where we treat only the first i relevant coordinates of the element. For $i \in \{2n+1, \ldots, 3n\}$, the membership test $y^{-1} \cdot x \in S_i$, which is equivalent to check $x_{i-2n} = y_{i-2n}$, is in $O(1)$. A certain T_i cannot contain more than m elements. Therefore there are $O(nm)$ membership tests in Algorithm 6. Hence every application of Algorithm 6 takes $O(n^2m^2)$ in the worst case.

(i) The initialization of $T_1, \ldots T_{3n}$ takes $O(n)$ steps.

(ii) By Theorem 3.4 computing the strong generators of G, H and S_0 is in $O(n^3m^3 + knm + lnm)$.

(iii) $k + l$ generators are sifted using Algorithm 6. Hence the sifting of the generators takes $O((k+l)n^2m^2)$ steps.

(iv) There are at most $(3nm)(3nm)$ pairs of elements from the $3n$ subsets of transversals with at most m elements. Hence sifting the pairs takes $O(n^4m^4)$ steps.

Therefore the running time of the whole Algorithm 7 is $O(n^4m^4 + (k+l)n^2m^2)$. □

3.3 Implementation in GAP

The author of this thesis implemented functions to find strong generators of subgroups of direct products and to find the generators of the intersection of two subgroups of a direct product of a group, which are given by arbitrary generators (see the text file `GeneratorsIntersectionDP.gap` in Appendix E). The GAP documentation of the main functions can be found in Appendix F. Note that we described the algorithms in Section 3.1 and 3.2 for direct powers of groups, but we can use them also for direct products of groups.

All functions explained in this section use tuples to represent direct product elements instead of using the normal GAP representation of direct products. For example, if we want to compute in $S_3 \times A_3$, we will represent the elements of that group as tuples, e.g.,

```
gap> Tuple([(1,2),(1,2,3)]);
DirectProductElement( [ (1,2), (1,2,3) ] )
```

instead of elements of

```
gap> D := DirectProduct(SymmetricGroup(3), AlternatingGroup(3));
Group([ (1,2,3), (1,2), (4,5,6) ])
```

Moreover it is not possible to use the usual representation of groups in GAP, i.e.

```
gap> GeneratorsOfGroup(SmallGroup(4,2));
[ f1, f2 ]
```

because the implemented function `CloseTower` cannot compute the product of a pair of strong generators given in this representation. Therefore we will give every group element as an element of a permutation group as described on page 15 using the package `SONATA`, i.e.

```
gap> GeneratorsOfGroup(AsPermGroup(SmallGroup(4,2)));
[ (1,2), (3,4) ]
```

In GAP we represent the strong generators as list of n sublists, i.e. for every $i \in \{1, \ldots, n\}$ the i-th sublist represents the transversal T_i. We call such a list of lists a *table of strong generators*.

The function `StrongGen(gen)` takes a list of generators `gen` of a subgroup H of a direct product of groups $G_1, \ldots, G_n$ and returns a list of lists `T` of strong generators of H (compare Algorithm 5) such that `T[i]` is a list containing the elements of T_i. The function `StrongGen(gen)` uses the auxiliary function `CloseTable(T)` which closes `T` by sifting all products of pairs of elements in `T`, until every such product can be written in canonical form and returns the modified table.

As an example we want to find the strong generators of the subgroup of the direct product $S_3 \times A_3$ generated by $((1,2),(2,3,1))$ and $((1,3),(2,3,1))$:

```
gap> Read("GeneratorsIntersectionDP.gap");
gap> T := StrongGen([Tuple([(1,2),(2,3,1)]),
>                    Tuple([(1,3),(2,3,1)])]);
[ [ DirectProductElement( [ (), () ] ),
      DirectProductElement( [ (1,2), (1,2,3) ] ),
      DirectProductElement( [ (1,3), (1,2,3) ] ),
      DirectProductElement( [ (1,3,2), (1,3,2) ] ),
      DirectProductElement( [ (1,2,3), (1,3,2) ] ),
      DirectProductElement( [ (2,3), () ] ) ],
  [ DirectProductElement( [ (), () ] ),
      DirectProductElement( [ (), (1,3,2) ] ),
      DirectProductElement( [ (), (1,2,3) ] ) ] ]
```

From this description it becomes obvious that this group is actually $S_3 \times A_3$.

The auxiliary function `ListOfListToSet(l)` allows us to convert a list of lists to a set:

```
gap> ListOfListToSet(T);
[ DirectProductElement( [ (), () ] ),
  DirectProductElement( [ (), (1,2,3) ] ),
  DirectProductElement( [ (), (1,3,2) ] ),
  DirectProductElement( [ (2,3), () ] ),
  DirectProductElement( [ (1,2), (1,2,3) ] ),
  DirectProductElement( [ (1,2,3), (1,3,2) ] ),
  DirectProductElement( [ (1,3,2), (1,3,2) ] ),
  DirectProductElement( [ (1,3), (1,2,3) ] ) ]
```

Note the difference to the GAP command `Flat` which flattens also the tuples and forms a list instead of a set:

```
gap> Flat(T);
[ (), (), (1,2), (1,2,3), (1,3), (1,2,3), (1,3,2), (1,3,2),
  (1,2,3), (1,3,2), (2,3), (), (), (), (), (1,3,2), (), (1,2,3)
]
```

With the function `IsRepresentable(a, T)` we can check whether an element `a` can be written in a canonical representation by the strong generators contained in `T`, i.e. we can solve the Subpower Membership Problem. If such a canonical representation is possible, it returns `true` and otherwise it returns a list `[i, x]` such that `a` can be written in canonical representation if the element `x` is inserted in the `i`-th sublist of `T` (compare Algorithm 4, page 52):

```
gap> IsRepresentable(Tuple([(1,3), (1,3,2)]), T);
true
gap> IsRepresentable(Tuple([(1,3), (1,2)]), T);
[ 2, DirectProductElement( [ (), (1,2) ] ) ]
```

The function `GenIntersect(a,b)` takes two lists of generators `a` und `b` and returns the set of generators of the intersection of the two subgroups generated by `a` and `b` (compare Algorithm 7, page 58). We will intersect two direct subgroups of S_4^3. The first subgroup is generated by $((3,4),(2,3),(1,4,2))$ and $((1,3,4,2),(1,3),(1,3,4))$. The second subgroup is generated by $((1,2,3,4),(2,3),(2,4,3))$ and $((1,3,4),(1,3)(2,4),(1,3,2,4)])$.

```
gap> a := [Tuple([(3,4), (2,3), (1,4,2)]),
>     Tuple([(1,3,4,2), (1,3), (1,3,4)])];
[ DirectProductElement( [ (3,4), (2,3), (1,4,2) ] ),
  DirectProductElement( [ (1,3,4,2), (1,3), (1,3,4) ] ) ]
gap> b := [Tuple([(1,2,3,4), (2,3), (2,4,3)]),
>     Tuple([(1,3,4), (1,3)(2,4), (1,3,2,4)])];
[ DirectProductElement( [ (1,2,3,4), (2,3), (2,4,3) ] ),
  DirectProductElement( [ (1,3,4), (1,3)(2,4), (1,3,2,4) ] ) ]
gap> c := GenIntersect(a,b);
[ DirectProductElement( [ (), (), () ] ),
  DirectProductElement( [ (), (), (2,3,4) ] ),
  DirectProductElement( [ (), (), (2,4,3) ] ),
  DirectProductElement( [ (), (), (1,2)(3,4) ] ),
  DirectProductElement( [ (), (), (1,2,3) ] ),
  DirectProductElement( [ (), (), (1,2,4) ] ),
  DirectProductElement( [ (), (), (1,3,2) ] ),
  DirectProductElement( [ (), (), (1,3,4) ] ),
  DirectProductElement( [ (), (), (1,3)(2,4) ] ),
  DirectProductElement( [ (), (), (1,4,2) ] ),
  DirectProductElement( [ (), (), (1,4,3) ] ),
  DirectProductElement( [ (), (), (1,4)(2,3) ] ),
  DirectProductElement( [ (3,4), (2,3), (1,4,2) ] ),
  DirectProductElement( [ (1,2), (2,3), (1,2,3) ] ),
  DirectProductElement( [ (1,2)(3,4), (), (1,4,2) ] ),
  DirectProductElement( [ (1,3)(2,4), (), (2,3,4) ] ),
  DirectProductElement( [ (1,3,2,4), (2,3), (1,2,4) ] ),
  DirectProductElement( [ (1,4,2,3), (2,3), (1,4)(2,3) ] ),
  DirectProductElement( [ (1,4)(2,3), (), (1,4,3) ] ) ]
gap> intersection := Group(c);
<group with 19 generators>
```

```
gap> Size(Group(a));
288
gap> Size(Group(b));
1152
gap> Size(intersection);
96
gap> StructureDescription(intersection);
"A4 x D8"
```

Hence the intersection of the two subgroups contains 96 elements and is isomorphic to $A_4 \times D_8$.
`CloseTableIntersect(T, StrongH, StrongG, StrongS, n)` is an auxiliary function used to compute the generators of the intersection $\mathbf{G} \cap \mathbf{H}$ of two subgroups of a direct product. The function closes the list of lists `T` with `3n` sublist by sifting all products of pairs of elements in `T` and returns the modified `T`. As additional input strong generators of three groups are needed: `StrongH` are the strong generators of $\mathbf{H}$, `StrongG` are the strong generators of $\mathbf{G}$ and `StrongS` are the strong generators of the group generated by $\mathbf{G} \cup \mathbf{H}$.

The function `IsRepresentable(a, T, StrongH, StrongG, StrongS, n)` takes a direct product element `a` and additional `T`, `StrongH`, `StrongG`, `StrongS`, `n` as described for `CloseTableIntersect`. `IsRepresentable` checks whether an element `a` can be written in a canonical representation by the strong generators contained in `T`. If such a canonical representation is possible, it returns `true` and otherwise it returns a list `[i, x]` such that `a` can be written in canonical representation if the element `x` is inserted in the `i`-th sublist of `T` (compare Algorithm 6, page 57).

Remark 3.11. If we want to use the implemented functions for direct powers of expanded groups, the implementation of the functions `CloseTable` and `CloseTableIntersect` must be modified: Instead of sifting only the products of pairs of elements contained in `T`, every element contained in the universe of the expanded group generated by all elements contained in `T` must be sifted.

3.4 Counting Compatible Functions

A natural application of the Subpower Intersection Problem is counting the number of compatible functions on a group. To simplify notation we will consider only unary functions in this section. The following results can be generalized to k-ary functions by some small modifications.

Definition 3.12 (cf. [8, p.17])**.** Let $\mathbf{G} = \langle G, \cdot, ^{-1}, 1\rangle$ be a group and N a normal subgroup of $\mathbf{G}$. A function $f : G \to G$ is *compatible with* N if

$$\forall x, y \in G\colon xN = yN \Rightarrow f(x)N = f(y)N.$$

The function f is *compatible on* $\mathbf{G}$ if f is compatible with all normal subgroups of $\mathbf{G}$. We denote the set of all unary functions which are compatible with N by

$$\mathrm{Comp}_1(\mathbf{G}, \{\equiv_N\}) = \{f : G \to G \mid f \text{ is compatible with } N\}.$$

The unary functions which are compatible with a normal subgroup N form a subgroup of $\mathbf{G}^G$:

$$\mathrm{Comp}_1(\mathbf{G}, \{\equiv_N\}) \leq \mathbf{G}^G$$

All unary compatible functions on $\mathbf{G}$ are the intersection

$$\mathrm{Comp}_1(\mathbf{G}) = \bigcap_{N \trianglelefteq \mathbf{G}} \mathrm{Comp}_1(\mathbf{G}, \{\equiv_N\}). \tag{3.5}$$

Hence the problem of finding all compatible functions on $\mathbf{G}$ can be solved by iteratively solving the Subpower Intersection Problem. We will intersect the set of functions that are compatible with one normal subgroup with the set of functions that are compatible with another normal subgroup. Then we intersect the result with the set of functions that are compatible with a third normal subgroup and so on. Note that not every normal subgroup is necessary in the right hand side of (3.5):

Lemma 3.13. *Let* $\mathbf{G}$ *be a group. Let* N, M *be normal subgroups of* $\mathbf{G}$*. Then*

$$\mathrm{Comp}_1(\mathbf{G}, \{\equiv_N\}) \cap \mathrm{Comp}_1(\mathbf{G}, \{\equiv_M\}) \subseteq \mathrm{Comp}_1(\mathbf{G}, \{\equiv_{N\cap M}\}).$$

Proof. Consider $f \in \mathrm{Comp}_1(\mathbf{G}, \{\equiv_N\}) \cap \mathrm{Comp}_1(\mathbf{G}, \{\equiv_M\})$. Therefore

$$\forall x, y \in G\colon xN = yN \Rightarrow f(x)N = f(y)N, \tag{3.6}$$
$$\forall x, y \in G\colon xM = yM \Rightarrow f(x)M = f(y)M. \tag{3.7}$$

Let $x, y \in G$ such that $x(N \cap M) = y(N \cap M)$, which is equivalent to $y^{-1}x \in N \cap M$. Then $y^{-1}x \in N$ and $y^{-1}x \in M$, therefore we know that $xN = yN$ and $xM = yM$. From (3.6) and (3.7) follows that

$$f(x)N = f(y)N \Rightarrow f(y)^{-1}f(x) \in N,$$
$$f(x)M = f(y)M \Rightarrow f(y)^{-1}f(x) \in M.$$

Hence $f(y)^{-1}f(x) \in N \cap M$, which is equivalent to $f(x)(N \cap M) = f(y)(N \cap M)$. Therefore $f \in \mathrm{Comp}_1(\mathbf{G}, \{\equiv_{N\cap M}\})$. □

If a subgroup of a group cannot be written as the intersection of two distinct subgroups of the group, then we call it a *meet irreducible subgroup*. Due to Lemma 3.13 and the fact that all functions $f : G \to G$ are compatible with $\mathbf{G}$ and $\mathbf{I}$, we can simplify (3.5) to

$$\mathrm{Comp}_1(\mathbf{G}) = \bigcap_{N \trianglelefteq \mathbf{G}, N \text{is meet irreducible}, N \neq G, N \neq I} \mathrm{Comp}_1(\mathbf{G}, \{\equiv_N\}).$$

For solving the Subpower Intersection Problem we need to know the generators of $\mathrm{Comp}_1(\mathbf{G}, \{\equiv_N\})$:

Theorem 3.14. *Let $\mathbf{G} = \langle G, \cdot, ^{-1}, 1\rangle$ be a group and N a normal subgroup of $\mathbf{G}$. For $g, h \in G$ define*

$$d_{g,h} \colon G \to G, x \mapsto \begin{cases} h \text{ if } x = g, \\ 1 \text{ otherwise,} \end{cases}$$

$$e_{g,h} \colon G \to G, x \mapsto \begin{cases} h \text{ if } x \in gN, \\ 1 \text{ otherwise.} \end{cases}$$

Let S be the set of generators of N and let R be a transversal through the cosets of N in G. Then $\mathrm{Comp}_1(\mathbf{G}, \{\equiv_N\})$ is generated by

$$\{d_{g,h} \mid g \in G, h \in S\} \cup \{e_{g,h} \mid g \in R, h \in R\}$$

Proof. Let $f \in \mathrm{Comp}_1(\mathbf{G}, \{\equiv_N\})$. Let $g_0, \ldots, g_k$ be the elements of the transversal R. We define a function $r : G \to R, x \mapsto y$ such that $xN = yN$. Then we define a function $f' \colon G \to G$ by

$$f'(x) = f(x) \cdot (\prod_{i=0}^{k} e_{g_i, r(f(g_i))}(x))^{-1}.$$

Let $g_i \in R, x \in g_i N$. Then

$$f'(x) = f(x) \cdot r(f(g_i))^{-1}. \tag{3.8}$$

Since f is compatible with N,

$$f(x)N = f(g_i)N = r(f(g_i))N.$$

Hence $f(x) \cdot r(f(g_i))^{-1} \in N$. By (3.8) we have $f'(G) \subseteq N$. Hence $f' : G \to N$. Then

$$f'(x) = \prod_{g \in G} d_{g, f'(g)}(x)$$

and

$$f(x) = \prod_{g \in G} d_{g,f'(g)}(x) \cdot \prod_{i=0}^{k} e_{g_i,r(f(g_i))}(x).$$

Since $N = \langle S \rangle$, we obtain

$$\forall g \in G \forall n \in N \colon d_{g,n} \in \langle d_{g,s} \mid s \in S \rangle.$$

Using the fact that

$$e_{g_i,r(f(g_i))} \in \{e_{g,h} \mid g \in R, h \in R\},$$

we showed that

$$f \in \langle \{d_{g,h} \mid g \in G, h \in S\} \cup \{e_{g,h} \mid g \in R, h \in R\} \rangle.$$

□

We want to implement a function in GAP which computes the number of compatible functions on a group. For that we will use the functions defined in `GeneratorsIntersectionDP.gap` and the package `SONATA` to use the representation of groups as permutation groups.

```
1 Read("GeneratorsIntersectionDP.gap");
2 LoadPackage("sonata");
```

As mentioned at the beginning of this section, we introduced the theory for unary compatible functions, but we want to implement our GAP-program for k-ary compatible functions.

First we define a function `GeneratorsComp(G, N, k)` which creates the generators of the `k`-ary functions compatible with `N`, a normal subgroup of `G`, as given in Theorem 3.14:

```
1  GeneratorsComp := function (G, N, k)
2  #
3     local i, j, l, n, id, g, r, R, help, gens, x, gN, f,T;
4  #
5  gN := RightTransversal(G, N);
6  gens:= [];
7  n := Size(G)^k;
8  id := Identity(G);
9
10 # generators of G -> N
```

```
11 for x in GeneratorsOfGroup(N) do
12    for i in [1..n] do
13       help := List([1..n], x-> id);
14       help[i] := x;
15       Add(gens, Tuple(help));
16    od;
17 od;
18
19 # generators for shifting into other cosets
20 R := RightTransversal(G, N);
21 T := AsSortedList(Tuples(G, k));
22 f := function (x, g, r, id, k, N)
23    if ForAll([1..k], i -> g[i]^(-1) * x[i] in N) then return
          r; else return id; fi;
24 end;
25
26 for g in Tuples(R, k) do
27    for r in R do
28       help := List([1..n], x -> f(T[x], g, r, id, k, N) );
29       Add(gens, Tuple(help));
30    od;
31 od;
32
33 return gens;
34 end;
```

Now we can define a function `GenComp(G, N, k)` which computes the generators of the group of `k`-ary compatible functions on `G` by recursively solving the Subpower Intersection Problem using a list `N` containing all meet irreducible normal subgroups of `G`:

```
1 GenComp := function (G, N, k)
2 #
3    local i, gens, genDP, DP, l, n;
4 #
5    if Size(N) = 1 then gens := GeneratorsComp(G, N[1], k);
6    else
7       n := Size(N);
8       if IsOddInt(n) then l := (n+1)/2; else l := n/2; fi;
9       gens := GenIntersect(GenComp(G,N{[1..l]},k), GenComp(G,
           N{[l+1..n]}, k));
```

```
10    fi;
11    return gens;
12  end;
```

Finally we want to compute the number of unary compatible functions on the dihedral Group with 8 elements $D_8 = \langle a, b \mid a^4 = 1, b^2 = 1, b^{-1}ab = a^{-1} \rangle$. The normal subgroups of D_8 are $1, \langle a^2 \rangle, \langle a \rangle, \langle a^2, b \rangle, \langle a^2, ba \rangle, D_8$.
Since $\langle a^2 \rangle = \langle a \rangle \cap \langle a^2, b \rangle$, it follows that

$$\mathrm{Comp}_1(D_8) = \bigcap_{N \in \{\langle a \rangle, \langle a^2, b \rangle, \langle a^2, ba \rangle\}} \mathrm{Comp}_1(D_8, \{\equiv_N\}).$$

```
gap> G := AsPermGroup(DihedralGroup(8));
Group([ (1,2)(3,8)(4,6)(5,7), (1,3,4,7)(2,5,6,8),
        (1,4)(2,6)(3,7)(5,8) ])
gap> S := NormalSubgroups(G);
[ Group([ (1,2)(3,8)(4,6)(5,7), (1,3,4,7)(2,5,6,8),
        (1,4)(2,6)(3,7)(5,8) ]),
  Group([ (1,4)(2,6)(3,7)(5,8), (1,2)(3,8)(4,6)(5,7) ]),
  Group([ (1,5)(2,3)(4,8)(6,7), (1,4)(2,6)(3,7)(5,8) ]),
  Group([ (1,3,4,7)(2,5,6,8), (1,4)(2,6)(3,7)(5,8) ]),
  Group([ (1,4)(2,6)(3,7)(5,8) ]), Group(()) ]
gap> S := Filtered(S, N -> Size(N) = 4);
[ Group([ (1,4)(2,6)(3,7)(5,8), (1,2)(3,8)(4,6)(5,7) ]),
  Group([ (1,5)(2,3)(4,8)(6,7), (1,4)(2,6)(3,7)(5,8) ]),
  Group([ (1,3,4,7)(2,5,6,8), (1,4)(2,6)(3,7)(5,8) ]) ]
gap> Size(Group(GenComp(G, S, 1)));
2048
gap> time;
1464
```

Therefore there are 2048 compatible unary functions on D_8. We want to compare our result with the result of the function `CompatibleFunctionNearRing` which is part of the package `SONATA`:

```
gap> Size(CompatibleFunctionNearRing(G));
2048
gap> time;
60
```

We see that the result is the same but the function implemented in `SONATA` is much faster. `SONATA` uses a special method for computing compatible functions on nilpotent groups and therefore avoids the effort of computing intersections in general.

Appendix A

Computations in the quaternion group Q_8

See page 17 to 20 in Section 1.3.

```
1  gap> G:= SmallGroup(8,4);
2  <pc group of size 8 with 3 generators>
3  gap> s := Size(G);
4  8
5  gap> gens := GeneratorsOfGroup(G);
6  [ f1, f2, f3 ]
7  gap> t := AsSortedList(G);
8  [ <identity> of ..., f1, f2, f3, f1*f2, f1*f3, f2*f3, f1*f2*f3
      ]
9  gap> H := Tuples(G, 2);;
10 gap> gen := [];
11 [ ]
12 # p1 represents the projection on first coordinate
13 gap> p1:= Tuple(List(H, x->x[1]));;
14 # p2 represents the projection on second coordinate
15 gap> p2:= Tuple(List(H, x->x[2]));;
16 gap> Add(gen, p1);
17 gap> Add(gen, p2);
18 gap> for i in [1..Size(gens)] do c := Tuple(List([1..Size(H)],
      x-> gens[i])); Add(gen, c); od;
19 # binary polinomials are generated by projections and constants
20 gap> pol := AsSortedList(Group(gen));;
21 # FilterGroupProperties.gap contains the filters to check the
      group properties
```

```
gap> Read("FilterGroupProperties.gap");
gap> Size(pol);
4096
# filter: f(x,1) = f(1,x) = x for all x
gap> pol := FilterNeutral(pol, t, s);;
gap> Size(pol);
2
# filter: for all x there exists a y such that f(x,y) =
   f(y, x) = 1 for all x
gap> pol := FilterInverse(pol, t, s, G);;
gap> Size(pol);
2
# filter: f(x, f(y,z)) = f(f(x,y), z) for all x, y, z
gap> pol := FilterAssociative(pol, t, s);;
gap> Size(pol);
2
gap> pol[1] = Tuple(List(H, x -> x[1]*x[2]));
true
gap> pol[2] = Tuple(List(H, x -> x[2]*x[1]));
true
# filter: f(x,y) = f(y,x) for all x, y
gap> pol := FilterCommutative(pol, t, s);
[ ]
gap> Size(pol);
0
# Thus only 2 binary polynomial operations on Q_8 are neutral
    with respect to the identity and are associative, namely
    the group operation x[1]*x[2] and its opposite x[2]*x[1].
```

Appendix B

FindDegree.gap

```
1  #=============================================================
2  # function FindDegree
3  # Input: Group G, function f, arity of the function,
       upperbound to stop
4  # Output: degree of function or false if degree is greater
       than upperbound
5
6  findDegree := function (G, f, arity, upperbound)
7
8  #
9     local d, check, k, id, j, s, x, i, xi, help, degree;
10 #
11
12 if not IsAbelian(G) then Error("Group is not abelian"); fi;
13
14 id := Identity(G);
15
16 for d in [0..upperbound] do
17
18    degree := true;
19    if arity = 1 then
20       for x in EnumeratorOfTuples(G, d+1 ) do
21          check := f(id);
22          for j in [1.. d+1] do
23             help := id;
24             for s in IteratorOfCombinations([1..d+1], j) do
25                xi := id;
```

```
26              for i in [1..j] do xi := xi * x[s[i]]; od;
27              help := help * f(xi);
28
29           od;
30           if IsOddInt(j) then help := help^(-1); fi;
31           check := check * help;
32        od;
33
34        if check <> id then degree := false; break; fi;
35     od;
36  else
37     for x in EnumeratorOfTuples(Tuples(G, arity), d+1 ) do
38        check := f(List([1..arity], x-> id));
39        for j in [1.. d+1] do
40           help := id;
41           for s in IteratorOfCombinations([1..d+1], j) do
42              xi := List([1..arity], z -> id);
43              for i in [1..j] do
44                 for k in [1..Length(xi)] do xi[k] := xi[k] *
                       x[s[i]][k]; od;
45              od;
46              help := help * f(xi);
47
48           od;
49           if IsOddInt(j) then help := help^(-1); fi;
50           check := check * help;
51        od;
52        if check <> id then degree := false; break; fi;
53     od;
54
55  fi;
56  if degree = true then return d; fi;
57 od;
58
59 return false;
60 end;
```

Appendix C

ExpGroup.gd

```
1
2  BindGlobal("ExpandedGroupsFamily",
       NewFamily("ExpandedGroupsFamily"));
3
4  DeclareCategory( "IsExpandedGroup", IsGroup );
5  DeclareRepresentation( "IsExpandedGroupStdRep",
6            IsExpandedGroup and IsComponentObjectRep and
                IsAttributeStoringRep,
7            [] );
8  DeclareFilter( "IsExpandedGroupDirectPower" );
9
10 ExpandedGroup := function(g,arities,ops)
11   local e,type;
12   e := rec( g := g, arities := arities, ops := ops );
13   type := NewType(ExpandedGroupsFamily, IsExpandedGroupStdRep);
14   return Objectify(type,e);
15 end;
16
17 DeclareOperation( "Op", [IsExpandedGroup, IsPosInt, IsObject]);
18
19 DeclareOperation( "ExpandedSubgroup", [IsExpandedGroup,
       IsList]);
20
21 DeclareOperation( "GroupOfExpandedGroup",
       [IsExpandedGroupStdRep]);
22
23 DeclareOperation( "DirectPower", [IsExpandedGroup, IsPosInt] );
```

```
24
25 DeclareOperation( "IsReductOfExpandedGroup",
       [IsExpandedGroupStdRep, IsGroup]);
26
27 DeclareOperation( "IsIdealOfExpandedGroup",
       [IsExpandedGroupStdRep, IsGroup]);
28
29 DeclareOperation( "IdealsOfExpandedGroup",
       [IsExpandedGroupStdRep]);
30
31 DeclareOperation( "Factor", [IsExpandedGroupStdRep, IsGroup]);
32
33 DeclareOperation( "IsHomomorphismOfExpandedGroup",
       [IsExpandedGroupStdRep, IsMapping]);
34
35 DeclareOperation( "IsGroupHomomorphismOfExpandedGroup",
       [IsExpandedGroupStdRep, IsMapping]);
36
37 DeclareOperation( "GroupOfExpandedGroup",
       [IsExpandedGroupStdRep, IsList]);
38
39 DeclareOperation( "IdealByGenerators", [IsExpandedGroupStdRep,
       IsList]);
```

Appendix D

ExpGroup.gi

```
1  InstallMethod( ViewString, "for an expanded group in std rep",
2    [ IsExpandedGroupStdRep ],
3    function( e )
4      if IsBound( e!.g ) then
5        return Concatenation("<expanded group
            ",ViewString(e!.g)," with ",
6                         String(Length(e!.ops))," operations>");
7      else
8        return Concatenation("<expanded group with ",
9                         String(Length(e!.ops))," operations>");
10     fi;
11   end );
12 #=============================================================
13 InstallMethod( Op, "for a expanded group, a positive int and
       an object",
14   [ IsExpandedGroupStdRep, IsPosInt, IsObject ],
15   function( e, nr, x )
16     return e!.ops[nr](x);
17   end );
18
19 #=============================================================
20 InstallMethod( ExpandedSubgroup, "by generators",
21   [ IsExpandedGroupStdRep, IsList ],
22   function( e, gens )
23     local es, type;
24       if not IsBound(e!.g) then GroupOfExpandedGroup(e); fi;
25     if not ForAll( gens, x -> x in e!.g ) then
```

```
26      Error( "generators have to be elements of the parent" );
27    fi;
28
29    es := rec( arities := e!.arities, ops := e!.ops,
         generators := gens );
30    type := NewType(ExpandedGroupsFamily,
         IsExpandedGroupStdRep);
31    return Objectify(type,es);
32  end );
33 #=============================================================
34 InstallMethod( GroupOfExpandedGroup, "finds the group
     generated by the generators with the group multiplication
     and the operations",
35   [ IsExpandedGroupStdRep],
36   function( e )
37    local H, x, foundnew, i, new;
38    if IsBound( e!.g ) then
39      return e!.g;
40    fi;
41
42    H := Group(e!.generators);
43    foundnew := true;
44    while foundnew do
45      foundnew := false;
46      for i in [1..Length(e!.ops)] do
47         # for unary functions
48            if e!.arities[i] = 1 then
49               for x in H do
50               new := Op(e, i, x);
51                if not( new in H ) then
52               foundnew := true;
53             break;
54             fi;
55            od;
56               if foundnew then break; fi;
57            # for k-ary functions
58        else
59
60               for x in EnumeratorOfTuples( H, e!.arities[i] )
                    do
61                new := Op(e, i, x);
```

```
62                  if not( new in H ) then
63                     foundnew := true;
64                     break;
65              fi;
66                  od;
67                  if foundnew then break; fi;
68          fi;
69       od;
70       if foundnew then
71            H := ClosureGroup( H, new );
72       fi;
73     od;
74       e!.g := H;
75     return H;
76
77
78   end );
79
80 #=================================================================
81 InstallMethod( DirectPower, "for an expanded group and a pos
      int",
82   [ IsExpandedGroupStdRep, IsPosInt ],
83   function( e, n )
84     local d,en,l,type;
85     if n = 1 then return e; fi;
86       if not IsBound(e!.g) then GroupOfExpandedGroup(e); fi;
87     l := ListWithIdenticalEntries(n,e!.g);
88     d := DirectProduct(l);
89     en := rec( g := d, base := e, n := n, ops := e!.ops,
           arities := e!.arities );
90     type := NewType(ExpandedGroupsFamily,
91                     IsExpandedGroupStdRep and
                           IsExpandedGroupDirectPower);
92     return Objectify(type,en);
93   end );
94 #=================================================================
95 InstallMethod( Op,
96   "for a power of an expanded group, a positive int and an
        object",
97   [ IsExpandedGroupStdRep and IsExpandedGroupDirectPower,
        IsPosInt, IsObject ],
```

```
 98   function( e, nr, x )
 99     local l,xx, d;
100       if not IsBound(e!.g) then d := e!.parent; else d :=
             e!.g; fi;
101       if IsList(x)
102          then l := List([1..e!.n],i-> List([1..Size(x)], j ->
                Image(Projection(d,i), x[j])));
103          else l := List([1..e!.n],i-> Image(Projection(d,i),
                x));
104       fi;
105     l := List(l,y->e!.ops[nr](y));
106     xx := Product([1..e!.n],i->Image(Embedding(d,i), l[i]));
107     return xx;
108   end );
109 #=====================================================================
110 InstallMethod( ExpandedSubgroup, "by generators",
111   [ IsExpandedGroupStdRep and IsExpandedGroupDirectPower,
         IsList ],
112   function( d, gens )
113     local es, type;
114       if not IsBound(d!.g) then GroupOfExpandedGroup(d); fi;
115     if not ForAll( gens, x -> x in d!.g ) then
116       Error( "generators have to be elements of the parent" );
117     fi;
118     es := rec( base := d!.base, n := d!.n, parent := d!.g,
            arities := d!.arities, ops := d!.ops, generators :=
            gens );
119     type := NewType(ExpandedGroupsFamily,
            IsExpandedGroupStdRep and IsExpandedGroupDirectPower);
120     return Objectify(type,es);
121   end );
122
123 #=====================================================================
124 InstallMethod( Size, "for an expanded group",
125   [ IsExpandedGroupStdRep ],
126   function( e )
127       if not IsBound(e!.g) then return
             Size(GroupOfExpandedGroup(e)); fi;
128     return Size(e!.g);
129   end );
130 #=====================================================================
```

```
131 InstallMethod( Identity, "for an expanded group",
132   [ IsExpandedGroupStdRep ], SUM_FLAGS,
133   function( e )
134       if not IsBound(e!.g) then return
              Identity(e!.generators[1]); fi;
135     return Identity(e!.g);
136   end );
137 #=============================================================
138 InstallMethod( \in, "for an expanded group",
139   [ IsObject, IsExpandedGroupStdRep ], SUM_FLAGS,
140 # to beat the library method for "wrong family"
141   function( x, e )
142       if not IsBound(e!.g) then return x in
              GroupOfExpandedGroup(e); fi;
143     return x in e!.g;
144   end );
145 #=============================================================
146 InstallMethod( Random, "for an expanded group",
147   [ IsExpandedGroupStdRep ],
148   function( e )
149       if not IsBound(e!.g) then return
              Random(GroupOfExpandedGroup(e)); fi;
150     return Random(e!.g);
151   end );
152
153 #=============================================================
154 InstallMethod( IsReductOfExpandedGroup, "for an expanded group
      and a group",
155   [ IsExpandedGroupStdRep, IsGroup ],
156   function( e, s )
157       #
158       local closed, x, i;
159       #
160       closed := true;
161      for i in [1..Length(e!.ops)] do
162         # for unary functions
163            if e!.arities[i] = 1 then
164               for x in s do
165               if not( e!.ops[i](x) in s ) then
166                  closed := false;
167               break;
```

```
168               fi;
169             od;
170             if closed = false then break; fi;
171             # for k-ary functions
172         else
173               for x in EnumeratorOfTuples( s, e!.arities[i] )
                      do
174                   if not( e!.ops[i](x) in s ) then
175                     closed := false;
176                     break;
177              fi;
178                   od;
179               if closed = false then break; fi;
180             fi;
181         od;
182       return closed;
183   end );
184
185 #=============================================================
186 InstallMethod(Embedding, "for a power of an expanded group and
       a positive integer",
187   [ IsExpandedGroupStdRep and IsExpandedGroupDirectPower,
         IsPosInt ],
188   function( e, i )
189 #
190    local d;
191 #
192       if not IsBound(e!.g) then d := e!.parent; else d :=
             e!.g; fi;
193     return Embedding(d, i);
194   end );
195
196 #=============================================================
197 InstallMethod(Projection, "for a power of an expanded group
       and a positive integer",
198   [ IsExpandedGroupStdRep and IsExpandedGroupDirectPower,
         IsPosInt ],
199   function( e, i )
200 #
201    local d;
202 #
```

```
203       if not IsBound(e!.g) then d := e!.parent; else d :=
             e!.g; fi;
204      return Projection(e!.g, i);
205    end );
206
207  #=============================================================
208  InstallMethod( IsIdealOfExpandedGroup, "for an expanded group
        and a group",
209    [ IsExpandedGroupStdRep, IsGroup ],
210    function( e, s )
211       #
212       local closed, x, i, a, ax;
213       #
214       if not IsBound(e!.g) then GroupOfExpandedGroup(e); fi;
215       closed := true;
216       if not (IsNormal(e!.g, s) and IsSubgroup(e!.g, s)) then
             return false; fi;
217      for i in [1..Length(e!.ops)] do
218          # for unary functions
219             if e!.arities[i] = 1 then
220                for x in s do
221                   for a in e!.g do
222                   if not( e!.ops[i](a*x)* (e!.ops[i](a)^(-1))
                         in s ) then
223                      return false;
224                   fi;
225                   od;
226             od;
227             if closed = false then break; fi;
228  # for k-ary functions
229         else
230               for x in EnumeratorOfTuples( s, e!.arities[i] )
                     do
231                   for a in EnumeratorOfTuples( e!.g,
                         e!.arities[i] ) do
232                      ax := List([1..Length(a)], k ->
                            a[k]*x[k]);
233                   if not( e!.ops[i](ax)* (e!.ops[i](a)^(-1))
                         in s ) then
234                      return false;
235                   fi;
```

```
236                     od;
237                  od;
238               fi;
239         od;
240      return true;
241   end );
242
243 #=============================================================
244 InstallMethod( IdealsOfExpandedGroup, "for an expanded group",
245   [ IsExpandedGroupStdRep ],
246   function( e )
247      #
248      local l;
249      #
250      if not IsBound(e!.g) then GroupOfExpandedGroup(e); fi;
251      l := NormalSubgroups(e!.g);
252      l := Filtered(l, x -> IsIdealOfExpandedGroup(e, x));
253      return l;
254   end );
255
256 #=============================================================
257 DefFunc := function(e, s, i)
258            if e!.arities[i] = 1 then
259               return function (x)
260                              return
261                                 Image(NaturalHomomorphismByNormalSubgroup(e!.g,
                                      s),Op(e, i,
                                      PreImagesRepresentative(NaturalHomomorphismByNormal
                                      s), x)));
262                              end;
263           else
264               return function (x)
265                          return
                                 Image(NaturalHomomorphismByNormalSubgroup(e!.g,
                                 s), Op(e, i,
                                 List([1..e!.arities[i]], k ->
                                 PreImagesRepresentative(NaturalHomomorphismByNormalSubgr
                                 s), x[k])));
266                       end;
267            fi;
268 end;
```

```
269
270 InstallMethod( Factor, "for an expanded group and an ideal",
271   [ IsExpandedGroupStdRep, IsGroup ],
272   function( e, s )
273       #
274       local i, j, fac, func, g;
275       #
276       if not IsBound(e!.g) then GroupOfExpandedGroup(e); fi;
277       func := List([1..Length(e!.ops)], i -> DefFunc(e, s, i));
278       return ExpandedGroup(FactorGroup(e!.g, s), e!.arities,
              func);
279   end );
280
281
282 #=========================================================================
283 InstallMethod( IsHomomorphismOfExpandedGroup, "for an expanded
      group and a function",
284   [ IsExpandedGroupStdRep, IsMapping ],
285   function( e, h )
286       #
287       local x, i;
288       #
289       if not IsBound(e!.g) then GroupOfExpandedGroup(e); fi;
290       for i in [1..Length(e!.ops)] do
291          # for unary functions
292             if e!.arities[i] = 1 then
293                for x in e!.g do
294                   if not( Image(h,e!.ops[i](x)) =
                          e!.ops[i](Image(h, x)) ) then
295                   return false;
296                fi;
297             od;
298             # for k-ary functions
299          else
300                for x in EnumeratorOfTuples( e!.g, e!.arities[i]
                     ) do
301                   if not( Image(h, e!.ops[i](x)) =
                          e!.ops[i](List([1..e!.arities[i]], k ->
                          Image(h,x[k])))) then
302                   return false;
303                fi;
```

```
304            od;
305            fi;
306         od;
307
308
309      return true;
310   end );
311 #=====================================================================
312 InstallMethod( IsGroupHomomorphismOfExpandedGroup, "for an
       expanded group and a function",
313   [ IsExpandedGroupStdRep, IsMapping ],
314   function( e, h )
315      #
316      local x, i;
317      #
318      if not IsGroupHomomorphism(h) then return false; fi;
319      if not IsBound(e!.g) then GroupOfExpandedGroup(e); fi;
320      for i in [1..Length(e!.ops)] do
321         # for unary functions
322            if e!.arities[i] = 1 then
323               for x in e!.g do
324                  if not( Image(h,e!.ops[i](x)) =
                           e!.ops[i](Image(h, x)) ) then
325                  return false;
326               fi;
327            od;
328            # for k-ary functions
329         else
330               for x in EnumeratorOfTuples( e!.g, e!.arities[i]
                     ) do
331                  if not( Image(h, e!.ops[i](x)) =
                           e!.ops[i](List([1..e!.arities[i]], k ->
                           Image(h,x[k])))) then
332                  return false;
333               fi;
334            od;
335            fi;
336         od;
337
338
339      return true;
```

```
340   end );
341
342
343  #=============================================================
344  InstallMethod( GroupOfExpandedGroup, "finds the group
        generated by the generators with the group multiplication
        and the operations with finite degree",
345    [ IsExpandedGroupStdRep, IsList],
346    function( e , deg)
347      local H, x, foundnew, i, j, k, new, gen, check, l, t, tp,
           id;
348      if IsBound( e!.g ) then
349        return e!.g;
350      fi;
351      id := Identity(e);
352      H := Group(e!.generators);
353        foundnew := true;
354      while foundnew do
355           gen := GeneratorsOfGroup(H);
356           l := [];
357           for i in [1..Maximum(deg)] do
358              t := Tuples(gen, i);
359              tp := List([1..Length(t)], x -> id);
360              for j in [1..Length(t)] do
361                 for k in [1..Length(t[j])] do
362                    tp[j] := tp[j]*t[j][k];
363                 od;
364              od;
365              l[i] := tp;
366           od;
367        foundnew := false;
368        for i in [1..Length(e!.ops)] do
369
370              check := Set(Flat(l{[1..deg[i]]}));
371
372           # for unary functions
373              if e!.arities[i] = 1 then
374
375                 for x in check do
376                 new := Op(e, i, x);
377                  if not( new in H ) then
```

```
378               foundnew := true;
379              break;
380              fi;
381             od;
382               if foundnew then break; fi;
383
384             # for k-ary functions
385         else
386
387               for x in EnumeratorOfTuples( check,
                      e!.arities[i] ) do
388                  new := Op(e, i, x);
389                  if not( new in H ) then
390                     foundnew := true;
391                     break;
392              fi;
393                  od;
394                  if foundnew then break; fi;
395          fi;
396       od;
397       if foundnew then
398            H := ClosureGroup( H, new );
399       fi;
400     od;
401       e!.g := H;
402     return H;
403
404
405   end );
406 #=============================================================
407 InstallMethod( IdealByGenerators, "finds the ideal generated
       by the generators",
408   [ IsExpandedGroupStdRep, IsList ],
409   function( e, gens )
410 #
411     local N, x, foundnew, i, new, xn, n, a;
412 #
413       if not IsBound(e!.g) then GroupOfExpandedGroup(e); fi;
414       N := Group(gens);
415     foundnew := true;
416     while foundnew do
```

```
417       foundnew := false;
418 #check if N is closed under (x,a)|-> a^(-1)*x*a for all a in
       e!.g and x in N
419          for x in N do
420             for a in e!.g do
421               new := a^(-1)*x*a;
422                 if not( new in N ) then
423                  foundnew := true;
424                 break;
425             fi;
426             od;
427             if foundnew then break; fi;
428          od;
429
430 #check if N is closed under (x,n)|->f(xn)*f(x)^(-1) for all n
       in N and x in e!.g
431          if not foundnew then
432          for i in [1..Length(e!.ops)] do
433             # for unary functions
434                if e!.arities[i] = 1 then
435                   for n in N do
436                      for x in e!.g do
437                      new := Op(e, i, x*n)*(Op(e, i, x)^(-1));
438                       if not( new in N ) then
439                       foundnew := true;
440                       break;
441                    fi;
442                      od;
443                      if foundnew then break; fi;
444                od;
445                   if foundnew then break; fi;
446                  # for k-ary functions
447               else
448
449                   for n in EnumeratorOfTuples( N, e!.arities[i]
                          ) do
450                      for x in EnumeratorOfTuples( e!.g,
                             e!.arities[i] ) do
451                         xn := List([1..Length(x)], k ->
                                x[k]*n[k]);
452                      new := Op(e, i, xn)*(Op(e, i, x)^(-1));
```

```
453                     if not( new in N ) then
454                          foundnew := true;
455                     break;
456                fi;
457                        if foundnew then break; fi;
458                 od;
459                  od;
460                  if foundnew then break; fi;
461            fi;
462         od;
463           fi;
464       if foundnew then
465            N := ClosureGroup( N, new );
466       fi;
467     od;
468     return N;
469
470
471   end );
```

Appendix E

GeneratorsIntersectionDP.gap

```
1  # GeneratorsIntersectionDP.gap
2  #=========================================================================
3  # Stephan Zweckinger, 2013
4  #=========================================================================
5  # contains the following functions:
6  #
7  # auxiliary functions
8  #  ListOfListToSet
9  # functions for computing strong generators of subgroups of a
       direct product of groups
10 #  IsRepresentable
11 #  CloseTable
12 #  StrongGen
13 # functions for computing strong generators of intersection of
       two subgroups of a direct product of groups
14 #  IsRepresentableProjection
15 #  IsRepresentableIntersect
16 #  CloseTableIntersect
17 #  GenIntersect
18 #=========================================================================
19 # the transversals are displayed as lists, hence the set of
       all transversals is a list of list
20 # we will call this list of list a table
21 #=========================================================================
22
23
24 #=========================================================================
```

```
25 # function ListOfListToSet
26 # Input: List of lists
27 # Output: Set
28 # Description: takes a list of list and returns the set of
       elements contained in the inner lists
29 ListOfListToSet := function (T)
30 #
31    local R, i;
32 #
33 R:=[];
34 for i in [1..Length(T)] do
35    Append(R, T[i]);
36 od;
37 return Set(R);
38 end;
39
40 #=========================================================================
41 # function IsRepresentable
42 # Input: direct product element a, table of strong generators T
43 # Output: true or List [i,x]
44 # Description: true, if a is representable by T, i.e. a =
       t_1*t_2*...*t_n where t_i is element of T_i, otherwise
       [i,x], where i is the position, x has to be inserted in T,
       such that a is representable by T
45
46 IsRepresentable := function (a,T)
47 #
48    local i, j, k, found, b;
49 #
50 b:= ShallowCopy(a);
51 for i in [1..Length(a)] do
52    j := 1;
53    found := false;
54    while found = false and j <= Length(T[i]) do
55       if b[i] = T[i][j][i] then
56          for k in [1..Length(a)] do b[k] :=
                T[i][j][k]^(-1)*b[k]; od;
57          found := true;
58          else j := j+1;
59       fi;
60
```

```
61    od;
62    if found = false then return [i,Tuple(b)]; fi;
63 od;
64 return true;
65 end;
66
67
68 #=====================================================================
69 # function CloseTable
70 # Input: table of strong generators
71 # Output: closed table of strong generators
72 # Description: checks, whether the table is closed and inserts
       elements to the table, if necessary, to close the table
73 CloseTable := function (T)
74 #
75    local R, rep, L, i, j, k, l, m, n, newelem;
76 #
77 R := ShallowCopy(T);
78
79 n:= Length(R[1][1]);
80 L:= [];
81 for j in [1..Length(R)] do
82    for i in [j..Length(R)] do
83       for k in [1..Length(R[j])] do
84          for l in [1..Length(R[i])] do
85             newelem:=[];
86             for m in [1..n] do newelem[m] :=
                  R[i][l][m]*R[j][k][m]; od;
87             Add(L,Tuple(newelem));
88          od;
89
90       od;
91    od;
92 od;
93
94
95
96
97
98 while Length(L) > 0 do
99
```

```
100    rep:= IsRepresentable(L[1],R);
101    if rep = true
102       then Remove(L, 1) ;
103       else
104          for i in [1..rep[1]] do
105             for j in [1..Length(R[i])] do
106                newelem:=[];
107                for m in [1..n] do newelem[m] :=
                      rep[2][m]*R[i][j][m]; od;
108                Add(L,Tuple(newelem));
109             od;
110          od;
111          for i in [rep[1]..Length(R)] do
112             for j in [1..Length(R[i])] do
113                newelem:=[];
114                for m in [1..n] do newelem[m] :=
                      R[i][j][m]*rep[2][m]; od;
115                Add(L,Tuple(newelem));
116             od;
117          od;
118          newelem:=[];
119          for m in [1..n] do newelem[m] := rep[2][m]*rep[2][m];
                od;
120          Add(L,Tuple(newelem));
121          Add(R[rep[1]], rep[2]);
122
123    fi;
124 od;
125 return R;
126 end;
127
128
129 #=========================================================================
130 # function StrongGen
131 # Input: Set of generators of a subgroup of a direct product
       of a group
132 # Output: Table of strong generators
133 # Description: returns a table of strong generators belonging
       to a direct product generated by the input
134 StrongGen := function (gen)
135 #
```

```
136    local T, G, rep, id, i;
137  #
138
139  id := Identity(gen[1]);
140  T:= List(gen[1], x-> [id]);
141
142  for i in [1..Length(gen)] do
143     rep:= IsRepresentable(gen[i],T);
144     if rep <> true then Add(T[rep[1]], rep[2]); fi;
145  od;
146
147  T:=CloseTable(T);
148
149  return T;
150  end;
151
152
153  #=========================================================================
154  # function IsRepresentableProjection
155  # Input: direct product element a, index of projection p <=
        Length(a), table of strong generators T
156  # Output: true or false
157  # Description: checks if the first p coordinates of a are
        representable by T in the first p coordinates
158  IsRepresentableProjection := function (a,p,T)
159  #
160     local i, j, k, found, b;
161  #
162  b:= ShallowCopy(a);
163  for i in [1..p] do
164     j := 1;
165     found := false;
166     while found = false and j <= Length(T[i]) do
167        if b[i] = T[i][j][i] then
168           for k in [1..Length(a)] do b[k] :=
                 T[i][j][k]^(-1)*b[k]; od;
169           found := true;
170           else j := j+1;
171        fi;
172
173     od;
```

```
174    if found = false then break; fi;
175  od;
176  return found;
177  end;
178
179
180  #=====================================================================
181  # function IsRepresentableIntersect
182  # Input: direct product element a, table of strong generators
       T, table of strong generators of H, G and S= Group(H UNION
       G) StrongH, StrongG, StrongS, length of direct product n
183  # Output: true or List [i,x]
184  # Description: returns true if a is representable by T,i.e. a
       = t_1*t_2*...*t_3*n, otherwise returns a list [position of
       necessary insertion, element to insert] with information
       how to modify T such that a can be represented by T
185
186  IsRepresentableIntersect := function (a, T, StrongH, StrongG,
       StrongS, n)
187
188    local i, j,k, b, S, help, help2, addbool;
189  #
190  S:= ShallowCopy(T);
191  help := a;
192  if not IsRepresentable(a, StrongS) then Error( " a cannot be
       generated by the strong generators of S");fi;
193  for i in [1..3*n] do
194    addbool := true;
195    if i <= n then
196      for j in [1..Length(S[i])] do
197        help2:=[];
198        for k in [1..Length(a)] do help2[k] :=
             S[i][j][k]^(-1)*help[k]; od;
199        if IsRepresentableProjection(help2, i, StrongH)
200          then help := help2; addbool := false; break;
201        fi;
202      od;
203    elif i <= 2*n then
204      for j in [1..Length(S[i])] do
205        help2:=[];
```

```
206         for k in [1..Length(a)] do help2[k] :=
               S[i][j][k]^(-1)*help[k]; od;
207         if IsRepresentableProjection(help2, i-n, StrongG)
208            then help := help2; addbool := false; break;
209         fi;
210      od;
211
212   else
213      for j in [1..Length(S[i])] do
214         if help[i-2*n]= S[i][j][i-2*n]
215            then
216               for k in [1..Length(a)] do help[k] :=
                     S[i][j][k]^(-1)*help[k]; od;
217               addbool := false;
218               break;
219
220   fi;
221      od;
222
223
224   fi;
225
226   if addbool = true then return [i,Tuple(help)]; fi;
227
228
229 od;
230 return true;
231 end;
232
233 #=========================================================================
234 # function CloseTableIntersect
235 # Input: table of strong generators T, table of strong
       generators of H, G and S=Group(H UNION G) StrongH,
       StrongG, StrongS, length of direct product n
236 # Output: closed table of strong generators
237 # Description: closes intersection table
238
239 CloseTableIntersect := function (T, StrongH, StrongG, StrongS,
       n)
240 #
241    local R, rep, L, i, j, k, l, m, newelem;
```

```
#
R := ShallowCopy(T);
L:= [];
for j in [1..Length(R)] do
   for i in [j..Length(R)] do
      for k in [1..Length(R[j])] do
         for l in [1..Length(R[i])] do
            newelem:=[];
            for m in [1..n] do newelem[m] :=
               R[i][l][m]*R[j][k][m]; od;
            Add(L,Tuple(newelem));
         od;

      od;
   od;
od;




while Length(L) > 0 do

   rep:= IsRepresentableIntersect(L[1],R, StrongH, StrongG,
      StrongS, n);
   if rep = true
      then Remove(L, 1) ;
      else
         for i in [1..rep[1]] do
            for j in [1..Length(R[i])] do
               newelem:=[];
               for m in [1..n] do newelem[m] :=
                  rep[2][m]*R[i][j][m]; od;
               Add(L,Tuple(newelem));
            od;
         od;
         for i in [rep[1]..Length(R)] do
            for j in [1..Length(R[i])] do
               newelem:=[];
               for m in [1..n] do newelem[m] :=
                  R[i][j][m]*rep[2][m]; od;
```

```
279                 Add(L,Tuple(newelem));
280              od;
281           od;
282           newelem:=[];
283           for m in [1..n] do newelem[m] := rep[2][m]*rep[2][m];
                 od;
284           Add(L,Tuple(newelem));
285           Add(R[rep[1]], rep[2]);
286
287      fi;
288   od;
289   return R;
290   end;
291
292
293
294   #=====================================================================
295   # function GenIntersect
296   # Input: two sets of generators a,b of direct products
297   # Output: set of generators
298   # Description: computes a set of strong generators of the
          intersection of the two subgroups of a direct product of a
          group generated by a and b
299
300   GenIntersect := function (a,b)
301   #
302      local genS, S, id, T, StrongH, StrongG, StrongS, rep, i, n,
             lS;
303   #
304   genS := Concatenation(a,b);
305   id:= Identity(genS[1]);
306   n:= Length(a[1]);
307   T:= List([1..3*n], x-> [id]);
308
309   lS := Length(genS);
310   StrongH := StrongGen(a);
311   StrongG := StrongGen(b);
312   StrongS := StrongGen(genS);
313
314
315   # add necessary elements for all generators of G and H to table
```

```
316 for i in [1..lS] do
317    rep := IsRepresentableIntersect(genS[i], T, StrongH,
           StrongG, StrongS, n);
318    if rep <> true then Add(T[rep[1]], rep[2]); fi;
319 od;
320
321 # close table
322 T:= CloseTableIntersect(T, StrongH, StrongG, StrongS, n);
323
324 return ListOfListToSet(List([2*n+1..3*n],x->T[x]));
325 end;
```

Appendix F

Documentation ExpGroup

The following document describes the functions for finding the degree of a function, the functions for expanded groups and the functions for strong generators, the Subpower Membership Problem and the Subpower Intersection Problem. For creating this document GAPDoc, a document format for GAP functions, was used.

Computing in Groups and Expanded Groups

Version 1

Stephan Zweckinger

Stephan Zweckinger Email: stephan.zweckinger@gmx.de

Abstract

This documentation describes a function for finding the degree of a function, a package for handling expanded groups and functions for solving the Subpower Intersection Problem and the Subpower Membership Problem for groups. The package and all functions were developed in the frame of the master thesis "Computing in Direct Powers of Expanded Groups" by Stephan Zweckinger. More informations about the implementation of the described functions and the theory behind can be found in this master thesis.

Copyright

© 2013 Stephan Zweckinger
Distributed under GNU General Public License (Version 3 or at your option any later version)

Acknowledgements

Thanks to Max Neunhöffer (St. Andrews) for help with designing the datastructure for expanded groups in GAP and to Peter Mayr (Linz) for supporting the whole development in the frame of my master thesis.

Contents

Chapter 1

Expanded Groups

1.1 Functions with Finite Degree

Let $\mathbf{G} = \langle G, \cdot, ^{-1}, 1 \rangle$ be a group and let f be a function from G to G. We define the degree of f as the smallest $d \in \mathbb{N}_0$ such that for all $x_1, \ldots, x_n \in G$:

$$f(\prod_{i=1}^{n} x_i) \text{ is a product of functions } f(\prod_{i \in S} x_i) \text{ and } f(\prod_{i \in S} x_i)^{-1}$$

for $S \subseteq \{1, \ldots, n\}, |S| \leq d$. If no such d exists, we say that the degree of the function is infinite.

If $\mathbf{G}$ is an abelian group, the degree of f is the smallest $d \in \mathbb{N}_0$ such that

$$\prod_{S \subseteq \{1,\ldots,d+1\}} (f(\prod_{i \in S} x_i))^{(-1)^{|S|}} = 1$$

for all $x_1, \ldots, x_{d+1} \in G$.

1.1.1 FindDegree

▷ `FindDegree(G, f, arity, max)` (function)

This function returns the degree of the function `f` with arity `arity` on the abelian group `G`, if the degree is less or equal `max`. Otherwise it returns `false`. If `G` is not abelian it returns an error message.

Example

```
gap> f1 := function(x) return x[1]*x[2]; end;
function( x ) ... end
gap> f2 := function(x) return Identity(x); end;
function( x ) ... end
gap> FindDegree(CyclicGroup(5), f1, 2, 10);
1
gap> FindDegree(CyclicGroup(5), f2, 1, 10);
0
gap> IsAbelian(SymmetricGroup(5));
false
gap> FindDegree(SymmetricGroup(5), f1, 2, 3);
Error, Group is not abelian called from
<function "FindDegree">( <arguments> )
 called from read-eval loop at line 4 of *stdin*
```

```
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
```

1.2 A Package for Expanded Groups

Let $\mathbf{G} = \langle G, \cdot, ^{-1}, 1 \rangle$ be a group. Let $f_1, \ldots, f_l$ be additional operations of arbitrary finite arities on G. Then the algebra $\mathbf{E} = \langle G, \cdot, ^{-1}, 1, f_1, \ldots, f_l \rangle$ is called an expanded group. The package ExpGroup provides functions for computing with expanded groups and direct powers of expanded groups. For using the package all unary functions must be represented as functions taking one element and all k-ary functions, $k > 1$, as functions taking a list with k elements. For example, the ternary operation $f(x,y,z) = x \cdot y \cdot z$ is represented as follows:

Example

```
gap>  f := function(x) return x[1]*x[2]*x[3]; end;
function( x ) ... end
```

To avoid problems with the abstract representation of groups, all groups must be used as permutation groups. Therefore the package SONATA, which provides a command `AsPermGroup` for converting a group to a permutation group, must be loaded.

Example

```
gap> LoadPackage("sonata");;
gap> AsPermGroup(DihedralGroup(8));
Group([ (1,2)(3,8)(4,6)(5,7), (1,3,4,7)(2,5,6,8), (1,4)(2,6)(3,7)(5,8) ])
```

The package consists of following functions:

1.2.1 ExpandedGroups

▷ `ExpandedGroups(`*g, arities, ops*`)` (function)

This function generates a component object of type `ExpandedGroup` which represents an expanded group and consists of the group *g* and additional operations contained in the list *ops* with corresponding arities contained in the list *arities*. The command `!.` gives access to the single components.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup(G, [1,1,2], [f1, f2, f3]);
<expanded group Sym( [ 1 .. 4 ] ) with 3 operations>
gap> e!.g;
Sym( [ 1 .. 4 ] )
gap> e!. arities;
```

```
[ 1, 1, 2 ]
gap> e!.ops;
[ function( x ) ... end, function( x ) ... end, function( x ) ... end ]
```

1.2.2 IsExpandedGroup

▷ IsExpandedGroup(*e*) (function)

This function returns true if *e* is of type ExpandedGroup and false otherwise.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup(G, [1,1,2], [f1, f2, f3]);
<expanded group Sym( [ 1 .. 4 ] ) with 3 operations>
gap> IsExpandedGroup(e);
true
gap> IsExpandedGroup(SymmetricGroup(5));
false
```

1.2.3 DirectPower

▷ DirectPower(*e*, *n*) (function)

This function returns a component object which is both of type ExpandedGroup and of type ExpandedGroupDirectPower and represents the *n*-th direct power of the expanded group *e*. It consists of the record entries g storing the direct power group, ops and arities storing the componentwise defined operations and their arities and *base*, *n* storing the base expanded group and the exponent.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup(G, [1,1,2], [f1, f2, f3]);
<expanded group Sym( [ 1 .. 4 ] ) with 3 operations>
gap> d := DirectPower(e, 3);
```

```
<expanded group <group of size 13824 with 6 generators> with 3 operations>
gap> d!.g;
Group([ (1,2,3,4), (1,2), (5,6,7,8), (5,6), (9,10,11,12), (9,10) ])
gap> d!.base;
<expanded group Sym( [ 1 .. 4 ] ) with 3 operations>
gap> d!.n;
3
gap> d!.ops;
[ function( x ) ... end, function( x ) ... end, function( x ) ... end ]
gap> d!.arities;
[ 1, 1, 2 ]
```

1.2.4 IsExpandedGroupDirectPower

▷ `IsExpandedGroupDirectPower(e)` (function)

This function returns `true` if *e* is of type `ExpandedGroupDirectPower` and `false` otherwise.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup(G, [1,1,2], [f1, f2, f3]);
<expanded group Sym( [ 1 .. 4 ] ) with 3 operations>
gap> d := DirectPower(e, 3);
<expanded group <group of size 13824 with 6 generators> with 3 operations>
gap> IsExpandedGroupDirectPower(d);
true
gap> IsExpandedGroupDirectPower(e);
false
```

1.2.5 Embedding

▷ `Embedding(d, n)` (function)

This function returns the embedding in the *n*-th component of the direct power of an expanded group *d*.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
```

```
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup(G, [1,1,2], [f1, f2, f3]);
<expanded group Sym( [ 1 .. 4 ] ) with 3 operations>
gap> d := DirectPower(e, 3);
<expanded group <group of size 13824 with 6 generators> with 3 operations>
gap> Image(Embedding(d, 3), (1,3));
(9,11)
```

1.2.6 Projection

▷ `Projection(`*d*`, `*n*`)` (function)

This function returns the projection of the direct power of an expanded group *d* onto its *n*-th component.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup(G, [1,1,2], [f1, f2, f3]);
<expanded group Sym( [ 1 .. 4 ] ) with 3 operations>
gap> d := DirectPower(e, 3);
<expanded group <group of size 13824 with 6 generators> with 3 operations>
gap> Image(Projection(d, 3), (9,11));
(1,3)
```

1.2.7 ExpandedSubgroup

▷ `ExpandedSubgroup(`*e*`, `*gen*`)` (function)

This function returns a new expanded group, which is a subalgebra of *e* and generated by the elements of the list *gen*. Therefore the new expanded subgroup has the same operations and arities as *e*. Note that there exists no record entry g in the newly generated object, but an entry `generators` which is equal to the input list *gen*. If *e* is of type `ExpandedGroupDirectPower`, then the new expanded subgroup contains a record `parent` which stores *e*. Note that direct powers which are constructed by `DirectPower` do not have a record `parent`.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
```

```
> else return (); fi; end;
function( x ) ... end
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup (G, [1,1],[f1, f2]);
<expanded group Sym( [ 1 .. 4 ] ) with 2 operations>
gap> s := ExpandedSubgroup(e, [(1,2,3)]);
<expanded group with 3 operations>
gap> IsBound(s!.g);
false
gap> s!.arities;
[ 1, 1, 2 ]
gap> s!.ops;
[ function( x ) ... end, function( x ) ... end, function( x ) ... end ]
gap> s!.generators;
[ (1,2,3) ]
gap> d := DirectPower(e, 3);
<expanded group <group of size 13824 with 6 generators> with 3 operations>
gap> sd := ExpandedSubgroup(d, [(1,2,3)(5,6)]);
<expanded group with 3 operations>
gap> sd!.parent;
Group([ (1,2,3,4), (1,2), (5,6,7,8), (5,6), (9,10,11,12), (9,10) ])
```

1.2.8 GroupOfExpandedGroup

▷ `GroupOfExpandedGroup(`*e[, deg]*`)` (function)

This function returns the group contained in the expanded group *e*. If `e!.g` is not set, `e!.g` is computed. A faster algorithm is used if all expanded group operations have finite degree and those degrees are given by an additional argument list *deg*.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup (G, [1,1],[f1, f2]);
<expanded group Sym( [ 1 .. 4 ] ) with 2 operations>
gap> s := ExpandedSubgroup(e, [(1,2,3)]);
<expanded group with 3 operations>
gap> IsBound(s!.g);
false
gap> GroupOfExpandedGroup(s);
Group([ (1,2,3), (1,3) ])
gap> IsBound(s!.g);
```

```
true
```

1.2.9 \in

▷ `\in(x, e)` (function)

This function returns `true` if x is an element of e and `false` otherwise.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup (G, [1,1],[f1, f2]);
<expanded group Sym( [ 1 .. 4 ] ) with 2 operations>
gap> (1,2,3) in e;
true
gap> (1,5,3) in e;
false
```

1.2.10 Size

▷ `Size(e)` (function)

This function returns the order of the expanded group e, that is, the size of `e!.g`.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup (G, [1,1],[f1, f2]);
<expanded group Sym( [ 1 .. 4 ] ) with 2 operations>
gap> s := ExpandedSubgroup(e, [(1,2,3)]);
<expanded group with 3 operations>
gap> Size(e);
24
gap> Size(s);
6
```

1.2.11 Identity

▷ `Identity(e)` (function)

This function returns the identity element of the expanded group `e`.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup (G, [1,1],[f1, f2]);
<expanded group Sym( [ 1 .. 4 ] ) with 2 operations>
gap> Identity(e);
()
```

1.2.12 Random

▷ `Random(e)` (function)

This function returns a randomly chosen element of the universe of the expanded group `e`.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup (G, [1,1],[f1, f2]);
<expanded group Sym( [ 1 .. 4 ] ) with 2 operations>
gap> Random(e);
(2,4,3)
```

1.2.13 IsGroupHomomorphismOfExpandedGroup

▷ `IsGroupHomomorphismOfExpandedGroup(e, h)` (function)

This function returns `true` if `h` is a group homomorphism of the expanded group `e`, i.e. if `h` is a group homomorphism of `e!.g` and the homomorphism property holds also for the additional operations of the expanded group. Otherwise it returns `false`.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup (G, [1,1],[f1, f2]);
<expanded group Sym( [ 1 .. 4 ] ) with 2 operations>
gap> h := function(x) return (1,2,3)^(-1)*x*(1,2,3); end;
function( x ) ... end
gap> h := MappingByFunction(e!.g, e!.g, h);
MappingByFunction( Sym( [ 1 .. 4 ] ), Sym( [ 1 .. 4 ] ), function( x ) ... en\
d )
gap> IsGroupHomomorphismOfExpandedGroup(e, h);
false
```

1.2.14 IsHomomorphismOfExpandedGroup

▷ `IsHomomorphismOfExpandedGroup(`*e*`, `*h*`)` (function)

This function returns `true` if the homomorphism property of *h* holds for the additional operations of the expanded group *e*. Otherwise it returns `false`.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup (G, [1,1],[f1, f2]);
<expanded group Sym( [ 1 .. 4 ] ) with 2 operations>
gap> h := function(x) return (1,2,3)^(-1)*x*(1,2,3); end;
function( x ) ... end
gap> h := MappingByFunction(e!.g, e!.g, h);
MappingByFunction( Sym( [ 1 .. 4 ] ), Sym( [ 1 .. 4 ] ), function( x ) ... en\
d )
gap> IsHomomorphismOfExpandedGroup(e, h);
false
```

1.2.15 IsReductOfExpandedSubgroup

▷ `IsReductOfExpandedSubgroup(e, s)` (function)

This function returns `true` if the group `s` is a reduct of the expanded group `e`, that is, if `s` is closed under all operations of `e`. Otherwise it returns `false`.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f1 := function(x) return (1,2)^(-1)*x*(1,2); end;
function( x ) ... end
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup (G, [1,1],[f1, f2]);
<expanded group Sym( [ 1 .. 4 ] ) with 2 operations>
gap> IsReductOfExpandedGroup(e, Group((1,2,3)) );
false
```

1.2.16 IsIdealOfExpandedGroup

▷ `IsIdealOfExpandedGroup(e, s)` (function)

This function returns `true` if the group `s` forms an ideal of the expanded group `e`. Otherwise it returns `false`. A set I is an ideal of an expanded group $\mathbf{E} = \langle G, \cdot, ^{-1}, 1, f_1, \ldots, f_l \rangle$ if I is a normal subgroup of $\langle G, \cdot, ^{-1}, 1 \rangle$ and $\forall f \in \{f_1, \ldots, f_l\} \forall x \in A^k, i \in I^k : f(x \cdot i) \cdot (f(x))^{-1} \in I$ where k is the arity of f. In GAP, ideals are given as groups with universe I instead of a set I.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup(G, [1], [f2]);
<expanded group Sym( [ 1 .. 4 ] ) with 1 operations>
gap> NormalSubgroups(G);
[ Sym( [ 1 .. 4 ] ), Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (1,4)(2,3), (1,3)(2,4) ]), Group(()) ]
gap> IsIdealOfExpandedGroup(e, Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]));
false;
```

1.2.17 IdealsOfExpandedGroup

▷ `IdealsOfExpandedGroup(e)` (function)

This function returns a list containing all ideals of the expanded group `e`. A set I is an ideal of an expanded group $\mathbf{E} = \langle G, \cdot, ^{-1}, 1, f_1, \ldots, f_l \rangle$ if I is a normal subgroup of $\langle G, \cdot, ^{-1}, 1 \rangle$ and $\forall f \in$

$\{f_1,\ldots,f_l\}\forall x\in A^k, i\in I^k : f(x\cdot i)\cdot(f(x))^{-1}\in I$ where k is the arity of f. In GAP, ideals are given as groups with universe I instead of a set I.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup(G, [1], [f2]);
<expanded group Sym( [ 1 .. 4 ] ) with 1 operations>
gap> IdealsOfExpandedGroup(e);
[ Sym( [ 1 .. 4 ] ), Group(()) ]
gap> NormalSubgroups(G);
[ Sym( [ 1 .. 4 ] ), Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (1,4)(2,3), (1,3)(2,4) ]), Group(()) ]
```

1.2.18 IdealByGenerators

▷ `IdealByGenerators(e, gen)` (function)

This function returns the ideal of the expanded group `e` that is generated by the elements contained in the list `gen`. A set I is an ideal of an expanded group $\mathbf{E}=\langle G,\cdot,^{-1},1,f_1,\ldots,f_l\rangle$ if I is a normal subgroup of $\langle G,\cdot,^{-1},1\rangle$ and $\forall f\in\{f_1,\ldots,f_l\}\forall x\in A^k, i\in I^k : f(x\cdot i)\cdot(f(x))^{-1}\in I$ where k is the arity of f. In GAP, ideals are given as groups with universe I instead of a set I.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> f2 := function(x) if x = () then return (1,2, 3)^(-1)*x*(1,2);
> else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup(G, [1], [f2]);
<expanded group Sym( [ 1 .. 4 ] ) with 1 operations>
gap> IsIdealOfExpandedGroup(e, Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]));
false
gap> IdealByGenerators(e, [(2,4),(1,4)]);
Group([ (2,4), (1,2,4), (1,2,3) ])
gap> Group([ (2,4), (1,2,4), (1,2,3) ]) = e!.g;
true
```

1.2.19 Factor

▷ `Factor(e, s)` (function)

This function returns the factor algebra `e/s` of an expanded group `e` and an ideal `s` of `e`. A set I is an ideal of an expanded group $\mathbf{E}=\langle G,\cdot,^{-1},1,f_1,\ldots,f_l\rangle$ if I is a normal subgroup of $\langle G,\cdot,^{-1},1\rangle$ and $\forall f\in\{f_1,\ldots,f_l\}\forall x\in A^k, i\in I^k : f(x\cdot i)\cdot(f(x))^{-1}\in I$ where k is the arity of f. In GAP, ideals are given as groups with universe I instead of a set I.

Example

```
gap> G:= SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
```

```
gap> f3 := function(x) if x[1] in AlternatingGroup(4) and x[2] in
> AlternatingGroup(4) then return x[1]*x[2]; else return (); fi; end;
function( x ) ... end
gap> e := ExpandedGroup(G, [2], [f3]);
<expanded group Sym( [ 1 .. 4 ] ) with 1 operations>
gap> Factor(e, Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]));
<expanded group <group of size 2 with 1 generators> with 1 operations>
```

Chapter 2

Strong Generators of Groups and their Applications

Note that all elements of direct powers in this chapter must be represented as a tuple of permutations.

Example

```
gap> Tuple([(1,2),(1,2,3)]);
DirectProductElement( [ (1,2), (1,2,3) ] )
```

2.1 Strong Generators

Let $G \leq A_1 \times A_2 \times \ldots \times A_n$ be a subgroup of a direct product. For $i \in \{1,\ldots,n\}$, let $G_i = \{a \in G \mid a_j = 1$ for $j \leq i$ and $a_j \in A_j$ for $j > i\}$ and T_i the set of coset representatives of G_{i-1}/G_i. The elements of $T_1,\ldots,T_n$ are the strong generators of G. Every element $g \in G$ can be written uniquely in the form $g = g_1 \cdot g_2 \cdot \ldots \cdot g_n$, where $g_i \in T_i$.

2.1.1 StrongGen

▷ `StrongGen(l)` (function)

This function takes a list `l` which contains tuples of permutations and returns the strong generators of the group generated by the elements of `l` as a list of the transversals $(T_1,\ldots,T_n)$.

Example

```
gap> StrongGen([Tuple([(1,2),(2,3,1)]), Tuple([(1,3),(2,3,1)])]);
[ [ DirectProductElement( [ (), () ] ),
      DirectProductElement( [ (1,2), (1,2,3) ] ),
      DirectProductElement( [ (1,3), (1,2,3) ] ),
      DirectProductElement( [ (1,3,2), (1,3,2) ] ),
      DirectProductElement( [ (1,2,3), (1,3,2) ] ),
      DirectProductElement( [ (2,3), () ] ) ],
  [ DirectProductElement( [ (), () ] ),
      DirectProductElement( [ (), (1,3,2) ] ),
      DirectProductElement( [ (), (1,2,3) ] ) ] ]
```

2.2 The Subpower Membership Problem

The Subpower Memberhip Problem is the question whether a given element is contained in a subgroup of a direct product given by generators.

2.2.1 IsRepresentable

▷ `IsRepresentable(e, strong)` (function)

This function returns `true` if `e` is an element of the group generated by the strong generators *strong* given as list of transversals. Otherwise it returns `[n, s]` such that adding the element `s` to the `n`-th transversal gives the strong generators of the group generated by *strong* and `e`.

Example

```
gap> T := StrongGen([Tuple([(1,2),(2,3,1)]), Tuple([(1,3),(2,3,1)])]);
[ [ DirectProductElement( [ (), () ] ),
      DirectProductElement( [ (1,2), (1,2,3) ] ),
      DirectProductElement( [ (1,3), (1,2,3) ] ),
      DirectProductElement( [ (1,3,2), (1,3,2) ] ),
      DirectProductElement( [ (1,2,3), (1,3,2) ] ),
      DirectProductElement( [ (2,3), () ] ) ],
  [ DirectProductElement( [ (), () ] ),
      DirectProductElement( [ (), (1,3,2) ] ),
      DirectProductElement( [ (), (1,2,3) ] ) ] ]
gap> IsRepresentable(Tuple([(1,3), (1,3,2)]), T);
true
gap> IsRepresentable(Tuple([(1,3), (1,2)]), T);
[ 2, DirectProductElement( [ (), (1,3) ] ) ]
```

2.3 The Subpower Intersection Problem

The Subpower Intersection Problem is the problem of finding generators for the intersection of two subgroups of a direct product that are given by their generators.

2.3.1 GenIntersect

▷ `GenIntersect(a, b)` (function)

This function returns a set of generators of the intersection of two groups generated by the elements of the lists `a` and `b`.

Example

```
gap> a := [Tuple([(3,4), (2,3), (1,4,2)]),
>     Tuple([(1,3,4,2), (1,3), (1,3,4)])];
[ DirectProductElement( [ (3,4), (2,3), (1,4,2) ] ),
  DirectProductElement( [ (1,3,4,2), (1,3), (1,3,4) ] ) ]
gap> b := [Tuple([(1,2,3,4), (2,3), (2,4,3)]),
>     Tuple([(1,3,4), (1,3)(2,4), (1,3,2,4)])];
[ DirectProductElement( [ (1,2,3,4), (2,3), (2,4,3) ] ),
  DirectProductElement( [ (1,3,4), (1,3)(2,4), (1,3,2,4) ] ) ]
gap> c := GenIntersect(a,b);
[ DirectProductElement( [ (), (), () ] ),
```

```
    DirectProductElement( [ (), (), (2,3,4) ] ),
    DirectProductElement( [ (), (), (2,4,3) ] ),
    DirectProductElement( [ (), (), (1,2)(3,4) ] ),
    DirectProductElement( [ (), (), (1,2,3) ] ),
    DirectProductElement( [ (), (), (1,2,4) ] ),
    DirectProductElement( [ (), (), (1,3,2) ] ),
    DirectProductElement( [ (), (), (1,3,4) ] ),
    DirectProductElement( [ (), (), (1,3)(2,4) ] ),
    DirectProductElement( [ (), (), (1,4,2) ] ),
    DirectProductElement( [ (), (), (1,4,3) ] ),
    DirectProductElement( [ (), (), (1,4)(2,3) ] ),
    DirectProductElement( [ (3,4), (2,3), (1,4,2) ] ),
    DirectProductElement( [ (1,2), (2,3), (1,2,3) ] ),
    DirectProductElement( [ (1,2)(3,4), (), (1,4,2) ] ),
    DirectProductElement( [ (1,3)(2,4), (), (2,3,4) ] ),
    DirectProductElement( [ (1,3,2,4), (2,3), (1,2,4) ] ),
    DirectProductElement( [ (1,4,2,3), (2,3), (1,4)(2,3) ] ),
    DirectProductElement( [ (1,4)(2,3), (), (1,4,3) ] ) ]
```

Index

Bibliography

[1] E. Aichinger, F. Binder, J. Ecker, P. Mayr, and C. Nöbauer, *SONATA - system of near-rings and their applications, GAP package, version 2.6*, 2012, (`http://www.algebra.uni-linz.ac.at/Sonata/`).

[2] E. Aichinger and P. Mayr, *Polynomial clones on groups of order pq*, Acta Math. Hungar. **114** (2007), no. 3, 267–285. MR 2296547 (2007k:08004)

[3] Erhard Aichinger, *The near-ring of congruence-preserving functions on an expanded group*, J. Pure Appl. Algebra **205** (2006), no. 1, 74–93. MR 2193192 (2006i:16068)

[4] Stanley Burris and H. P. Sankappanavar, *A course in universal algebra*, Graduate Texts in Mathematics, vol. 78, Springer-Verlag, New York, 1981. MR 648287 (83k:08001)

[5] Merrick Furst, John Hopcroft, and Eugene Luks, *Polynomial-time algorithms for permutation groups*, 21st Annual Symposium on Foundations of Computer Science (Syracuse, N.Y., 1980), IEEE, New York, 1980, pp. 36–41. MR 596045 (81m:20002)

[6] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.5.6*, 2012.

[7] I. Martin Isaacs, *Finite group theory*, Graduate Studies in Mathematics, vol. 92, American Mathematical Society, Providence, RI, 2008. MR 2426855 (2009e:20029)

[8] Kalle Kaarli and Alden F. Pixley, *Polynomial completeness in algebraic systems*, Chapman & Hall/CRC, Boca Raton, FL, 2001. MR 1888967 (2003a:08001)

[9] Peter Mayr, *The subpower membership problem for Mal'cev algebras*, IJAC **22** (2012), no. 7.

[10] Ralph N. McKenzie, George F. McNulty, and Walter F. Taylor, *Algebras, lattices, varieties. Vol. I*, The Wadsworth & Brooks/Cole Mathematics Series, Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, CA, 1987. MR 883644 (88e:08001)

[11] Derek J. S. Robinson, *A course in the theory of groups*, Graduate Texts in Mathematics, vol. 80, Springer-Verlag, New York, 1996.

[12] Harold N. Ward, *Combinatorial polarization*, Discrete Math. **26** (1979), no. 2, 185–197. MR 535244 (80m:05012)