Everything you always wanted to know about rings in GAP*.
(* but were afraid to ask)

Diplomarbeit zur Erlangung des akademischen Grades eines Diplom-Ingenieurs in der Studienrichtung Technische Mathematik, Stzw. Informations- und Datenverarbeitung

Angefertigt am Institut für Mathematik der Technisch-Naturwissenschaftlichen Fakultät der Johannes Kepler Universität Linz

Eingereicht bei Univ.-Prof. Dr. Günter Pilz

Jürgen Ecker

October 7, 1999

Preface

We dance around in a ring and suppose,
But the secret sits in the middle and knows.
Robert Frost (1875–1963)

There exist thousands of books on ring-theory each containing hundreds of theorems, most of them for special classes of rings. Nonetheless given a ring certain properties can not be checked by hand in a reasonable time (unless by a genius). This makes computer programs desireable that do the dirty work.

We shall dance around a little and gether the most common results in ring theory. Chapter 1 starts with a glance at universal algebra. In chapter 2 we gether results about rings. Chapters 3 and 4 are about analyzing the structure of a ring and end with the famous density theorem due to Jacobson. Chapters 5 and 6 shall make computation easier in many cases. In chapter 7 we give an overview about results for polynomial rings featuring Buchberger's Algorithm for Groebner Bases. We omit proofs if they are long and not of particular importance for the algorithms.

However, the main part is a library of GAP-functions presented in chapter 8. This is a users manual for all the functions for rings so far implemented in GAP. Note, that this is not a collection of super-fast algorithms for very special rings, but of default functions which will work for any ring using as little knowledge about the ring as possible. For those interested in curious programming styles we have added the source code of all new functions in chapter 10.

Contents

Chapter 1

Universal Algebra

Nothing will come of nothing.
William Shakespeare (1564–1616)

Algebraic structures as semigroups, groups, rings, nearrings, etc. have nice properties in common. Therefore it seems useful to define certain things generally and use the results in each of the various structures.

1.1 Algebras

Definition 1 *An* algebra $\mathcal{A}$ *is a tuple* $(A, \circ_1, \ldots, \circ_m, \star_1, \ldots, \star_n, \Omega)$*, where* A *is a set,* $\circ_1, \ldots, \circ_m$ *are binary operations on* A *and* $\star_1, \ldots, \star_n$ *are external operations on* A *with operator domain* Ω*.* $\tau := (m, n, \Omega)$ *is called the* type *of* $\mathcal{A}$*. If* $n = 0$ *we write* (m) *instead of* $(m, 0, \emptyset)$*. A subset of the class of all algebras of the same type is called a* class *of algebras.*

Definition 2 *Let* $\{\mathcal{A}_\lambda : \lambda \in \Lambda\}$ *be a family of algebras of the same type* (n, m, Ω) *and the same class, where the binary (external) operations of* $\mathcal{A}_\lambda$ *are* $\circ_1^\lambda, \ldots, \circ_n^\lambda$ $(\star_1^\lambda, \ldots, \star_m^\lambda)$*. On the cartesian product of the sets* A_i *we can define an algebra of the same class, defining as binary (external) operations the "componentwise operations":*

$$(a_\lambda)_{\lambda\in\Lambda} \circ_i (b_\lambda)_{\lambda\in\Lambda} := (a_\lambda \circ_i^\lambda b_\lambda)_{\lambda\in\Lambda}$$
$$\omega \star_j (a_\lambda)_{\lambda\in\Lambda} := (\omega \star_j^\lambda a_\lambda)_{\lambda\in\Lambda}$$

for all $i \in \{1, \ldots, n\}$*,* $j \in \{1, \ldots, m\}$*.*
This algebra is called the (direct) product *of the algebras* A_λ*, denoted* $\prod_{\lambda\in\Lambda} \mathcal{A}_\lambda$*.*

Definition 3 *Let $\mathcal{A}$, the algebra above, be of class $\mathcal{C}$. Then $B \subseteq A$ together with all operations of $\mathcal{A}$ forms an algebra $\mathcal{B}$ of the same class if B is closed under all operations of $\mathcal{A}$. $\mathcal{B}$ is called a $\mathcal{C}$-*subalgebra *of $\mathcal{A}$. We denote this $\mathcal{B} \leq_{\mathcal{C}} \mathcal{A}$ or without an index if it's clear from the context.*

Definition 4 *The set $\{(\ldots, a_\lambda, \ldots) :$ only finitely many $a_\lambda \neq 0\}$ induces a subalgebra of $\prod_{\lambda\in\Lambda} \mathcal{A}_\lambda$, called the* direct sum *of the $\mathcal{A}_\lambda$. We write $\bigoplus_{\lambda\in\Lambda} \mathcal{A}_\lambda$.*

Definition 5 *Let S be a set, $\circ$ a binary operation on S, $\star$ an external operation on S with operator domain Ω and $\mathcal{R}$ a relation in S. $\mathcal{R}$ is* compatible *with $\circ$ iff $\forall a, b, \tilde{a}, \tilde{b} \in S : (aRb) \wedge (\tilde{a}R\tilde{b}) \implies (a \circ \tilde{a})\mathcal{R}(b \circ \tilde{b})$. $\mathcal{R}$ is* compatible *with $\star$ iff $\forall a, b \in S, \lambda \in \Omega : a\mathcal{R}b \implies (\lambda \star a)\mathcal{R}(\lambda \star b)$. A* congruence *on $\mathcal{A}$ is an equivalence-relation, that is compatible with all operations on S.*

Definition 6 *Let $\mathcal{A}$ be the algebra above, $\sim$ a congruence in $\mathcal{A}$. On $A/_\sim$ we define*

$$[a]\tilde{\circ}_i[b] := [a \circ_i b] \text{ and } \lambda\tilde{\star}_j[a] := [\lambda \star_j a].$$

Then $\mathcal{A}/_\sim := (A/_\sim, \tilde{\circ}_1, \ldots, \tilde{\circ}_m, \tilde{\star}_1, \ldots, \tilde{\star}_n, \Omega)$ is an algebra of the same type, called the factor algebra *of $\mathcal{A}$ w.r.t. $\sim$. If $\mathcal{A}/_\sim$ is of the same class $\mathcal{C}$ as $\mathcal{A}$, we call $\mathcal{A}/_\sim$ a $\mathcal{C}$-factor algebra.*

Definition 7 *Let $\mathcal{R} = (R, \circ_1, \ldots, \circ_m, \star_1, \ldots, \star_n, \Omega)$, $\mathcal{S} = (S, \circ_1, \ldots, \circ_m, \star_1, \ldots, \star_n, \Omega)$ be two algebras. A mapping $h : R \to S$ is called a* homomorphism *iff for all $r_1, r_2 \in R$, for all $1 \leq i \leq m$ and for all $1 \leq j \leq n$*

$$\begin{aligned} h(r_1 \circ_i r_2) &= h(r_1) \circ_i h(r_2) \\ h(r_1 \star_j r_2) &= h(r_1) \star_j h(r_2) \end{aligned}$$

If a homomorphism h is injective, we call h a monomorphism. *If it is surjective an* epimorphism, *a bijective homomorphism is called an* isomorphism. *Homomorphisms from R into R are called* endomorphisms, *bijective ones* automorphisms. *The* Kernel *of h* $\ker(h)$ *is the relation on R defined by $a \sim_h b :\iff h(a) = h(b)$. The* Image *of h is the algebra induced by the set $h(R)$. Finally be $Hom(\mathcal{R}, \mathcal{S})$ the set of all homomorphisms from R to S and $End(\mathcal{R}) = Hom(\mathcal{R}, \mathcal{R})$.*

Remark 1 $\ker(h)$ *is always a congruence on R. $Im(h)$ is always a subalgebra of $\mathcal{S}$.*

Example 1 *Let K be a field. Then a vector space V over K forms an algebra of type $(1, 1, K)$ with coordinatewise addition and multiplication with field elements. Subalgebras are exactly the subspaces, homomorphisms correspond to linear functions, factor algebras are factor spaces.*

Theorem 1 *Let $h : R \to S$ be a homomorphism. Then*

$$A/\ker(h) \cong Im(h) \tag{1.1}$$

This means that the homomorphic images of an algebra "can be found in the algebra itself".

Example 2

- *Let $\mathcal{R}$ be an algebra, $\sim$ a congruence on R, $h : R \to R/_\sim$ defined by $h(r) := [r]$. Then $\ker(h) = \sim$. h is called the* canonical epimorphism *from R onto $R/_\sim$. On the other hand we saw, that for every homomorphism h, $\ker(h)$ is a congruence. So we will allow ourselves to speak of a congruence, when we mean the kernel of a homomorphism and vice versa.*
- *The identity mapping $1_R : R \to R$ is an isomorphism. For the corresponding congruence we have $r \sim_{1_R} s \iff r = s$. We feel free to write 1_R for the congruence as well.*

Theorem 2 *Let $\mathcal{R}$, $\mathcal{S}$ be two algebras of the same type, $Hom(\mathcal{R}, \mathcal{S}) \ni h$ onto. Then there is a 1:1 correspondance between the congruences of $\mathcal{S}$ and those of $\mathcal{R}$ containing $\ker(h)$. Inclusions are preserved.*

1.2 Moore–Systems

In chapter 6.1 we will talk much about generated subrings, ideals, etc.. We will define precisely, what we mean by that. Informally, we are looking for the smallest set that contains all the generators. It seems natural to define this as the intersection of all sets, that contain the generators. A Moore-system guarantees that the intersection is always defined.

Definition 8 *Let $A \neq \emptyset$. $\mathcal{M} \subseteq \mathcal{P}(A)$ is called a* Moore-system *on A iff $\mathcal{M} \ni A$ and $\mathcal{N} \subseteq \mathcal{M} \Longrightarrow \bigcap_{N \in \mathcal{N}} N \in \mathcal{M}$.*

Definition 9 *Let $G \subseteq A$ be a set of* generators*, $\mathcal{M}$ a Moore-system on A. Then there is exactly one smallest set*

$$< G >_{\mathcal{M}} := \bigcap_{G \leq M}^{M \in \mathcal{M}} M$$

(w.r.t. $\subseteq$) in $\mathcal{M}$. We call $< G >_{\mathcal{M}}$ the set generated *by G in $\mathcal{M}$. An $M \in \mathcal{M}$ is* finitely generated *iff there is a finite set G such that $< G >_{\mathcal{M}} = M$.*

Example 3 *Let V be a vector space over the field K, $\mathcal{M}$ the set of all subspaces of V, $v_1, \ldots, v_n$ vectors in V. Then $< \{v_1, \ldots, v_n\} > = \mathcal{L}(v_1, \ldots, v_n) = \{\sum_{i=1}^n \lambda_i v_i : \lambda_i \in \mathbf{R} \quad \forall i \in I\}$. I.e. the generated set is the linear hull of the generators.*

1.3 More about Congruences and Subdirect Products

In this section we shall consider some properties of the set of congruences of an algebra and apply these to study subdirect products.

Theorem 3 *The congruences of an algebra $\mathcal{A}$ form a Moore–system.*

Sketch of the *proof:* (cf. Jacobson [10], p.66)
Let $\{\sim_\alpha\}$ be a set of congruences. We check, that $\bigcap \sim_\alpha$ is an equivalence relation and compatible with all operations on $\mathcal{A}$. ■

Theorem 4 *Let a and b be two distinct elements of an algebra R, $D(a,b)$ the set of all congruences $\sim$, s.t. $a \not\sim b$. Then $D(a,b)$ is nonempty and contains a maximal element.*

Proof: (cf. Jacobson [10], p.67)
$1_R \in D(a,b)$, so $D(a,b) \neq \emptyset$. Let $\{\sim_\alpha\}$ be a chain[1] in $D(a,b)$. Consider $\sim := \bigcup \sim_\alpha$ and let i be a fixed element of $\{1,\dots,m\}$ and x, y, v and w elements of R, s.t. $x \sim y$ and $v \sim w$. By the definition of $\sim$ we get that there exist α and $\bar{\alpha}$, s.t. $x \sim_\alpha y$ and $v \sim_{\bar{\alpha}} w$. From the definition of a chain we see that we may assume w.l.o.g. that $v \sim_\alpha w$, either. From this we get $x \circ_i v \sim_\alpha y \circ_i w$, since $\sim_\alpha$ is a congruence, and finally $x \circ_i v \sim y \circ_i w$. Hence $\sim$ is again a congruence. Obviously $\sim \in D(a,b)$ and contains every $\sim_\alpha$. Thus every chain in $D(a,b)$ has an upper bound, so by Zorn's Lemma[2] $D(a,b)$ has a maximal element. ■

Definition 10 *We say that an algebra $\mathcal{A}$ is a* subdirect product *of the algebras $\mathcal{A}_\lambda$, iff there exists a monomorphism $h : \mathcal{A} \to \prod_{\lambda\in\Lambda} \mathcal{A}_\lambda$, s.t. every projection $\pi_\lambda \circ h$ is surjective. If one of the projections is an isomorphism we call the subdirect product* trivial *and call $\mathcal{A}$* subdirectly irreducible.

Theorem 5 *An algebra $\mathcal{A}$ with more than one element is subdirectly irreducible iff the intersection of all of the congruences $\neq 1_A$ is again $\neq 1_A$.*

Proof: (cf. Jacobson [10], p.69)
The special case, where $\mathcal{A}$ is a ring is proved in chapter 4. ■

Theorem 6 *Every algebra $\mathcal{A}$ is a subdirect product of subdirectly irreducible algebras.*

[1]see chapter A.
[2]see chapter A

Sketch of the *proof:* (cf. Jacobson [10], p.69)
W.l.o.g. we may assume, that $\mathcal{A}$ has at least 2 distinct elements a and b. Let $\sim_{a,b}$ be a maximal congruence w.r.t. having $a \not\sim_{a,b} b$. Then $\mathcal{A}/_{\sim_{a,b}}$ is subdirectly irreducible and $\bigcap \sim_{a,b} = 1_A$. Hence $\mathcal{A}$ is a subdirect product of the subdirectly irreducible algebras $\mathcal{A}/_{\sim_{a,b}}$. ■

1.4 Chain Conditions

Man is born free,
yet he is everywhere in chains.
J. J. Rousseau (1712–1778)

Nonetheless there is hope that chains have an end at least sometimes. This issue is so important that we gather the results in a separate chapter.

Definition 11 *Let K be a set, $\leq$ a partial ordering*[3] *on K. $(K, \leq)$ has the* DCC (descending chain condition) *iff any descending chain $k_1 > k_2 > \ldots$ in K stops after finitely many steps. Alternatively we may call K* artinian*.*

This definition is equivalent to the following.

Definition 12 *Every nonempty subset has at least one minimal element.*

Definition 13 *Let $K, \leq$ be as above. $(K, \leq)$ has the* ACC (ascending chain condition) *iff every ascending chain $k_1 < k_2 < \ldots$ in K stops after finitely many steps. Alternatively we may call K* noetherian*.*

Again we have an equivalent condition.

Definition 14 *Every nonempty subset has at least one maximal element.*

Surprisingly DCC and ACC have not very much in common. None of the two implies the other in general.

In chapter 5 we shall make use of the above definitions. Then many of the concepts we have studied will turn out to be equivalent if we can assume that one or the other of the chain conditions holds.

[3]for an exact definition see chapter A

Chapter 2

Rings

The choicest pleasures of life lie within the ring of moderation.
Tupper (1810–1889)

We will begin studying a class of algebras of type $(2, 0, \emptyset) = (2)$, called rings and watch out for the choicest pleasures.

2.1 Rings and Subrings

Definition 15 *A* ring *is a triple $(R, +, *)$, where R is a set and $+$ and $-$ are two binary operations on R, such that the following holds*[1]*:*

- *$(R, +)$ is an abelian group*[2]*.*
- *$(R, *)$ is a semigroup.*
- *$(r + s) * t = r * t + s * t \quad \forall r, s, t \in R$ (right distributive law).*
- *$r * (s + t) = r * s + r * t \quad \forall r, s, t \in R$ (left distributive law).*

Definition 16 *Instead of $(R, +, *)$ we often write R, if it is clear, which operations are considered. We denote the neutral element of $(R, +)$ by 0, the inverse element of $r \in r$ is denoted $-r$. For $r * s$ we write rs if it is convenient.*

[1]an abstract definition of a ring is given in chapter A
[2]for an exact definition see chapter A

Definition 17 *A ring R is* commutative *iff $*$ is commutative. If R has a neutral element, we call it* 1 *and call R a ring with identity. R is finite iff the set R is finite. R is called a* division ring *iff it is a ring with identity and for every $R \in R$ except 0, there exists an element $r^{-1} \in R$, s.t. $rr^{-1} = r^{-1}r = 1$. A commutative division ring we call a* field.

Example 4 *1. Probably the best-investigated areas nowadays are polynomials over finite fields (with pointwise addition and multiplication). They form an infinite commutative ring with identity. The ideal-structure has been extensively studied (see Buchberger [5], Winkler [24]). We shall deal with polynomial rings in chapter 7.*

2. $(End(G), +, \circ)$ *forms a ring for any abelian group* $(G, +)$. *This ring is not commutative but has an identity.*

3. *The* $n \times n$*-matrices over a finite field with usual matrix addition and multiplication form a noncommutative ring with identity.*

Definition 18 *A* subring S *of a ring* R *is a subalgebra of type* (2) *which is again a ring. We shall denote this by* $S \leq R$.

Distributivity and commutativity of addition are preserved when considering subsets, so all we have to check is, whether the subset is closed under the operations.

Theorem 7 *Let* R *be a ring,* $\emptyset \neq S \subseteq R$. *Then*

$$S \leq R \iff s_1 - s_2 \in S \wedge s_1 s_2 \in S \quad \forall s_1, s_2 \in S \tag{2.1}$$

2.2 Ideals and Factor Rings

Blessed is who carries with him an ideal,
and who obeys it.
Louis Pasteur (1822–1895)

A special class of subrings is of interest for further considerations. Namely those, that appear as kernel of a homomorphism.

Definition 19 *A subring* I *of a ring* R *is an* ideal *of* R $(S \trianglelefteq R)$ *iff*

$$i \in I, r \in R \Longrightarrow ir \in I \tag{2.1}$$
$$i \in I, r \in R \Longrightarrow ri \in I \tag{2.2}$$

If only the first (resp. second) implication is satisfied, I *is called a* right–*(resp.*left–*)*ideal.

Theorem 8 *Let* $I \leq R$. *Then*

$$I \trianglelefteq R \iff \sim_I \textit{ is a congruence} \tag{2.3}$$

where $\sim_I$ *is the equivalence-relation defined by* $r_1 \sim_I r_2 \iff r_1 - r_2 \in I$.

Remark 2 *Kernels of homomorphisms, congruences and ideals are essentially the same.*

Definition 20 *Let $I \trianglelefteq R$. $R/I := R/\sim_I$ is called* factor ring *of the R modulo the ideal I. We will write elements of R/I as $[r]$ or more often as $r + I$.*

Remark 3 *This definition makes sense because of 2.3.*

We gather some more definitions in the following

Definition 21 *A ring R is* simple*, if the only ideals are $\{0\}$ and R. An ideal I of R is* maximal *if $I \neq R$ and there is no ideal J such that $I \subset J \subset R$. An ideal I of R is* minimal *if $I \neq \{0\}$ and there is no ideal J such that $\{0\} \subset J \subset I$.*

Theorem 9 *A ring is a field iff it is a commutative simple ring with identity.*

Proof: (cf. Pilz [17], p.127)
"$\Rightarrow$": Let R be a field. By definition R is commutative and has an identity. Suppose $(0) \neq I \trianglelefteq R$, $i \in I$. For any $r \in R$ $r = r1 = ri^{-1}i \in I$ and hence $I = R$.
"$\Leftarrow$": We only have to show that for any $a \neq 0$ there exists a b s.t. $ab = ba = 1$. Consider (a). Since R is simple and $a \neq 0$, $(a) = R$ and hence there must exist a b s.t. $ab = ba = 1$. ■

Theorem 10 *Let $I \trianglelefteq R$. Then*

$$I \text{ is maximal} \iff R/I \text{ is simple.}$$

Definition 22 *Let R be a ring, $I, J \trianglelefteq R$. $I + J := \{i + j : i \in I, j \in J\}$ and $IJ := \{\sum_k^{finite} i_k j_k : i_k \in I, j_k \in J\}$ are ideals of R. For II we shall write I^2 and so on.*
(Note that $\{ij : i \in I, j \in J\}$ is not always an ideal in R, but
$< \{ij : i \in I, j \in J\} >= IJ$.)

Proof: "+": Let $a_1 = i_1 + j_1$, $a_2 = i_2 + j_2$ be in $I + J$, $r \in R$. Then

$$a_1 - a_2 = i_1 + j_1 - i_2 - j_2 = \underbrace{i_1 - i_2}_{\in I} + \underbrace{j_1 - j_2}_{\in J} \in I + J$$

and

$$a_1 a_2 = (i_1 + j_1)(i_2 + j_2) = \underbrace{i_1 i_2 + i_1 j_2 + j_1 i_2}_{\in I} + \underbrace{j_1 j_2}_{\in J} \in I + J.$$

Hence $I + J \leq R$. Finally $ra_1 = ri_1 + rj_1 \in I + J$ and the same holds for multiplication from the right side. This proves the Theorem.
"*": is done similar, but much longer. ■

Definition 23 *An ideal P of a ring R is called a* prime ideal, *if for all ideals I, J of R*

$$IJ \subseteq P \Longrightarrow I \subseteq P \vee J \subseteq P$$

R is called a prime ring *if (0) is a prime ideal.*

Definition 24 *Let $I \trianglelefteq R$. We call*

$$\sqrt{I} := \{r \in R : \exists n \in \mathbf{N} \quad r^n \in I\}$$

the radical *of I.*

Definition 25 *Let R be a ring, $r \in R$, $I \trianglelefteq R$.*

- *r is called* nilpotent *iff $\exists n \in \mathbf{N} \quad r^n = 0$, i.e. $r \in \sqrt{(0)}$.*
- *I is called* nilpotent *iff $\exists n \in \mathbf{N} \quad I^n = (0)$.*
- *I is called* nil *iff all $i \in I$ are nilpotent.*

Remark 4 *Obviously every nilpotent ideal is nil.*

2.3 Modules

For the next section about the Jacobson–Radical we need some results about an area that is very closely related to ring theory – modules. Informally a module is a vector space over a ring, which is not necessarily a field. In accordance with the usual notation for vector spaces and the definitions for external operations on algebras we define left-modules rather than right-modules (although Jacobson did it the other way round). The idea is the same and later we will use both kinds of modules. Throughout this section module will mean left-module (unless otherwise stated).

Definition 26 *A* (left-)module *${}_RM$ over the ring R (*R–module*) is an algebra of type $(1, 1, R)$ with binary operation $'+'$ and external operation $'*'$ (where, as usually we write $a * b$ as ab), s.t. $(M, +)$ is an abelian group and the following conditions hold for any $r, s \in R$, $m, n \in M$:*

1. *$(r + s)m = rm + sm$*
2. *$r(m + n) = rm + rn$*
3. *$(rs)m = r(sm)$*

Definition 27 Submodules, module homomorphisms *and* factor modules *are defined in accordance with the definitions in the chapter about algebras.*

Evidently, the results from chapter 1 are valid for modules.

Definition 28 *An R-module ${}_RM$ is* unital *iff R has an identity 1 and for all $m \in M: \quad 1m = m$.*

Examples 1 1. *For any ring R, $(R,+)$ is an R-module (denoted ${}_RR$) with the ring multiplication as external operation. The submodules of ${}_RR$ are the left ideals of R.*

2. *Vector spaces are unital modules over fields.*

Theorem 11 *Every submodule induces a congruence.*

A faithful R-module is undestroyable by R.

Definition 29 *For a subset $N \subseteq {}_RM$ we define the* annihilator *of N as $Ann(N) := \{r \in R : rn = 0 \quad \forall n \in N\}$. ${}_RM$ is called* faithful *iff $Ann(M) = \{0\}$.*

Theorem 12 *The annihilator $I := Ann(N)$ of any R-submodule N of an R-module M is an ideal of R.*

Proof:
Let $i, j \in I$, $r \in R$, $n \in N$. $(i-j)n = in - jn = 0$, $(ij)n = i(jn) = i0 = 0$, $(ri)n = r(in) = r0 = 0$ and $(ir)n = i(rn) = in' = 0$, from this we get the result. ■

Definition 30 *Let M be an R-module. Then ${}_RM$ is* simple *iff it has no submodules except $\{0\}$ and M. A nontrivial simple module is called* irreducible.

The above theorem "induces" the following definition.

Definition 31 *An ideal $I \trianglelefteq R$ is* primitive *iff there is an irreducible R-module ${}_RM$, s.t. $I = Ann(M)$.*

Theorem 13 *Let M be an R-module. Then R/Ann(M) can be embedded into End(M) (viewed as group endomorphisms).*

Proof: (cf. Pilz [17], p.152)
$h : R \to End(M)$, $r \mapsto [h_r : M \to M,\ m \mapsto rm]$ is a homomorphism with $\ker(h) = Ann(M)$. By the homomorphism theorem we have $R/Ann(M) = R/\ker(h) \cong Im(h) = \{h_r : r \in R\} \leq End(M)$. ■

Lemma 1 *Let M and N be irreducible R-modules. Then any homomorphism $h \in Hom_R(M,N)$ is either $\mathbf{0}$ or an isomorphism. Moreover $End_R(M)$ is a division ring.*

Proof: (cf. Jacobson [10], p.118)
Let $h \in Hom_R(M,N)$. Then $\ker(h) \leq {}_RM$ and $Im(h) \leq {}_RN$. Since M and N are irreducible, $\ker(h)$ is either M or $\{0\}$ and $Im(h)$ is either N or $\{0\}$. If $h \neq \mathbf{0}$ then $\ker(h) \neq M$ and $Im(h) \neq \{0\}$. So $\ker(h) = \{0\}$ and $Im(h) = N$, i.e. h is an isomorphism. Moreover, if $N = M$, the result is that any nonzero endomorphism is an automorphism. So for $\mathbf{0} \neq h \in End_R(M)$, $h^{-1} \in End_R(M)$ and $End_R(M)$ is a division ring. ■

For the following chapter we need some more material to play with.

Definition 32 *A left ideal L is called* modular *iff there exists an $e \in L$ (called* right identity modulo L*), s.t. for all $r \in R \quad re - r \in L$.*

The following theorem gives a small overview of the results about modular left ideals.

Theorem 14 *For the following let R be a ring and L_1, L_2, L and M left ideals of R.*

1. *$L_1 \subseteq L_2$ and L_1 modular $\Longrightarrow$ L_2 modular.*
2. *L modular $\Longrightarrow$ there exists a maximal modular left ideal $M \supseteq L$.*
3. *L_1 and L_2 modular and $R = L_1 + L_2 \Longrightarrow L_1 \cap L_2$ modular.*

Proof: (cf. Pilz [17], p.154)

1. is clear from the definition.
2. Let $\mathcal{L} := \{J \lhd R : J \supseteq L, L \text{ modular}\}$. $\mathcal{L} \neq \emptyset$, since $L \in \mathcal{L}$. Every chain $\mathcal{C} = \{J_\alpha : \alpha \in A\}$ has an upper bound $J := \bigcup_{\alpha \in A} J_\alpha$. From 1. we know, that J is modular, hence it is in $\mathcal{L}$. So Zorn's Lemma guarantees the existence of a maximal element M in $\mathcal{L}$. ■

Definition 33 *Let R be a ring and for $a, b \in R$*

$$a \circ b := a + b - ab$$

Then $(R, \circ)$ is a monoid. We call the (left-/right-)invertible elements of $(R, \circ)$ (left-/right-)quasi-regular ((l/r)qr)*. A (left-/right-)ideal I is called* quasi-regular (qr) *iff all elements of I are (l/r)qr.*

We have not defined an lqr left-ideal. The next theorem explains, why.

Theorem 15 *Let L be a qr left ideal. Then all elements of L are qr.*

Proof: (cf. Pilz [17], p.159)
Let $l \in L$. l has an inverse in $(R, \circ)$, say l^-, i.e. $l^- + l - l^- l = 0$. Hence $l^- = l^- l - l \in L$. So l^- has again an inverse in $(R, \circ)$, say $(l^-)^-$. Now $l = 0 \circ l = ((l^-)^- \circ l^-) \circ l = (l^-)^- \circ (l^- \circ l) = (l^-)^- \circ 0 = (l^-)^-$. From this we get $l^- \circ l = 0 = l \circ l^-$ and l is qr. ■

Concluding the chapter we give a theorem, that relates the last definition with known things.

Theorem 16 *Every nilpotent element of a ring R is qr.*

Proof:
If $r^n = 0$ then $s := -r - r^2 - \ldots - r^{n-1}$ is the inverse of r. ■

Chapter 3

Radical-Theory

I destroy my enemy when I make him my friend.
Abraham Lincoln (1809–1865)

3.1 Jacobson Radical

Definition 34 *Let R be a ring. The intersection of all primitive ideals of R is called the* (Jacobson-)radical *of R, denoted $\mathcal{J}(R)$. If $\mathcal{J}(R) = (0)$, R is called* (Jacobson-)semi-simple.

Primitivity of ideals is not so easy to check. Hence we would prefer another definition. The following theorem gives some:

Theorem 17 *The radical $\mathcal{J}(R)$ is*

1. *the intersection of all primitive left ideals of R.*
2. *the intersection of all simple left-R-modules.*
3. *the intersection of all irreducible left-R-modules.*
4. *the intersection of all left primitive ideals.*
5. *the intersection of all maximal modular left ideals.*
6. *sum of all qr left ideals.*

Proof: (see Pilz [17], pp.157-159)

■

The same holds if we replace omit the word left or replace it by right in the above theorem.

3.2 Nil Radicals

The goal of the following definitions is to define a (1) nil ideal I of R, s.t. (2) R/I contains no nilpotent elements. One sees immediately, that the two conditions (1) and (2) work against each other. At a second glance one notices, that there is not only one ideal that satisfies these two conditions. This is the reason, why here come a lot of nil radicals. Very often (in particular, when the ring is finite), all radicals mean the same ideal, and we can choose whichever definition is convenient for our purposes.

Definition 35 *The* lower nil radical $\mathcal{P}$ *of a ring* R *is the intersection of all prime ideals of* R.

Definition 36 *The* upper nil radical $\mathcal{N}$ *of a ring* R *is the sum of all (one- or two-sided) nil ideals.*

Remark 5 *(cf. Jacobson [11])*
$\mathcal{P}$ is the smallest ideal satisfying condition (2). This is the reason for it's name. $\mathcal{N}$ is the biggest nil ideal satisfying condition (2). Again this is the reason for the name. Baer [2] gives an example of a ring R, where $\mathcal{P}(R) \neq \mathcal{N}(R)$ and such that there exist ideals $\mathcal{P}(R) \subseteq I \subseteq \mathcal{N}(R)$ which do not satisfy conditions (1) and (2). Nevertheless we will find many situations, where $\mathcal{P} = \mathcal{N}$.

Definition 37 *The* Brown-MacCoy radical $\mathcal{G}$ *of a ring* R *is the intersection of all modular maximal ideals of* R.

Theorem 18 *For any ring* R *the following holds:*

$$\mathcal{P}(R) \subseteq \mathcal{N}(R) \subseteq \mathcal{J}(R) \subseteq \mathcal{G}(R) \tag{3.1}$$

Now we are going to consider a special case, where (at least) some of the radical-definitions coincide.

Remark 6 *R commutative* $\Longrightarrow \mathcal{N}(R) = \mathcal{P}(R) = \sqrt{(O)}$.

Chapter 4

Subdirect Decomposition

Conquered, we conquer.
Plautus (254–184 B.C.)

4.1 Subdirect Decomposition of Rings

Rings can not always be decomposed into a direct product of smaller rings, which would be a nice property when we study rings. Following Plautus we do not give up and invest some work on this issue. Birkhoff's theorem shows that we can at least decompose every ring into a subdirect product.

Definition 38 *Let R and $R_i, i \in I$ be rings. Every embedding $h : R \to \prod_{i \in I} R_i$ is said to represent R as a* subdirect product *of the rings R_i iff each of the maps $\pi_i \circ h : R \to R_i$ is surjective. If one (or more) of the maps $R \to R_i$ is an isomorphism we call the subdirect product representation a* trivial *one. (Observe that, since the image of a homomorphism is a subring of the range, the above can be read as: R is isomorphic to a subring of the direct product $\prod R_i$ of the rings R_i.)*

Theorem 19 *R can be represented as a subdirect product of $R_i, i \in I$ iff for every $i \in I$ there exists a surjective ring homomorphism $\varphi_i : R \to R_i$ such that $\bigcap_{i \in I} \ker \varphi_i = 0$.*

Proof: (cf. Pilz [17], p.136)
"$\Rightarrow$": Suppose h represents R as a subdirect product. Let $\varphi_i := \pi_i \circ h$ for all $i \in I$ and $k \in \bigcap_{i \in I} \ker(\varphi_i)$. If $k \neq 0$ then $h(k) \neq 0$, since h is an isomorphism. But every component of $h(k)$ is $(\pi_i \circ h)(k) = \varphi_i(k) = 0$, which yields a contradiction.
"$\Leftarrow$": The map $h : R \to \prod_{i \in I} R_i$, $r \mapsto (\ldots, \varphi_i(r), \ldots)$ is 1:1 (since $\ker(h) = \bigcap_{i \in I} \ker(\varphi_i) = \{0\}$) an hence an isomorphism. ■

This encourages us to make the following

Definition 39 *We call a ring R* subdirectly reducible *iff there is a nontrivial representation of R as a subdirect product. Otherwise we call R* subdirectly irreducible.

From the theorem above we get

Corollary 1 *For a nonzero ring R the following statements are equivalent:*

1. *R is subdirectly irreducible.*
2. *The intersection of all nonzero ideals of R is nonzero.*
3. *R has a smallest nonzero ideal (w.r.t. $\subseteq$), called* heart *(or sometimes* little *ideal).*

Examples 2 1. *Any simple ring is subdirectly irreducible.*

2. *$\mathbf{Z}/(p^n)$, $p \in \mathbf{P}$ is subdirectly irreducible. For $n = 1$ this follows from 1.. For $n = 2$ we give an example with $p = 2$:*
   The ideals of $\mathbf{Z}/(4) \cong \mathbf{Z}_4$ are $(0) = \{0\}$, $(1) = \{0, 1, 2, 3\}$, $(2) = \{0, 2\}$, $(3) = \{0, 1, 2, 3\}$. Obviously (2) is the heart of $\mathbf{Z}_4$.
   In general, the heart of $\mathbf{Z}/(p^n)$ is $(p^{n-1})/(p^n)$.
3. *$\mathbf{Z}$ is subdirectly reducible (e.g. with increasing n the ideals (p^n) form a nonterminating falling chain with zero intersection.) In fact for any infinite sequence of primes $\{p_i\}$, the canonical map $\mathbf{Z} \to \prod \mathbf{Z}/(p_i^{n_i})$, $n_i > 0$ is a nontrivial representation of $\mathbf{Z}$ as a subdirect product of subdirectly irreducible rings. This shows that representations of a ring as subdirect products of (even subdirectly irreducible) rings are neither finite nor unique in general.*

The following theorem is a special case of Birkhoff's theorem in chapter 1. In the case of rings we give a more detailed proof.

Theorem 20 **Any** *nonzero ring R can be represented as a subdirect product of subdirectly irreducible rings.*

Proof (cf. Lam [13], p.205):
For any $0 \neq a \in R$, let M_a be an ideal maximal with respect to the property that $a \notin M_a$. (Such an ideal exists by Zorn's Lemma.) Now let I be a nonzero ideal in R/M_a. By the diamond lemma (2) I has a corresponding ideal $\bar{I}$ in R which contains M_a. Since M_a was maximal w.r.t. not containing a, $\bar{I}$ must contain a. From this we get $a + M_a \in I$ (recall that the correspondance between the ideals of R and those of R/M_a is given by the natural homomorphism $h : R \to R/M_a$, $x \mapsto x + M_a$). So $a + M_a \neq 0 + M_a$ is contained in the intersection of all nonzero ideals, hence R/M_a is subdirectly irreducible. Since $\bigcap_{a\neq 0} M_a = 0$, the canonical map $R \to \prod_{a\neq 0} R/M_a$ represents R as a subdirect product of the subdirectly irreducible rings R/M_a. ■

4.2 Consequences of Birkhoff's theorem

When anger rises, think of the consequences.
Confucius

Science is the knowledge of consequences,
and dependence of one fact upon another.
Thomas Hobbes (1588–1679)

We start with a special case, where we can replace subdirect by direct.

Theorem 21 *Let $K_1, \ldots, K_n$ be maximal ideals of a ring R, s.t. $\bigcap_{i=1}^n K_i = (0)$. Then there is a subset A of $\{1, \ldots, n\}$, s.t. R is isomorphic to the direct sum of the rings R/K_α $(\alpha \in A)$.*

Proof: (see Pilz [17], p.137)

■

Corollary 2 *A commutative ring with identity R is isomorphic to a direct sum of fields if it has finitely many maximal ideals with zero intersection.*

4.2.1 Primitive Rings

Primitive rings will turn out to be the basic objects from which we can build up complicated rings.

Definition 40 *A ring R is called* primitive *on M iff M is a faithful irreducible R-module. If such an M exists R is called* primitive.

Examples 3 *The following are primitive rings:*

1. *Simple rings with identity, division rings and fields*
   Proof: Since the annihilator of every submodule of ${}_RR$ is an ideal of R and R is simple ${}_RR \neq \{0\}$ is irreducible. Suppose $Ann({}_RR)$ contains an $r \neq 0$. Then $ar = 0$ for all $r \in R$. In particular $a = a.1 = 0$ which yields a contradiction.

2. *$Hom_D(V, V)$ for any vector space V over the division ring D*
   Proof: We first check that V is a $Hom_D(V, V)$-module straightforward. Irreducibility of V is then shown as in the first example. Finally V is faithful, because $\forall v : f(v) = 0 \Longrightarrow f = 0$.

$\blacksquare$

Commutative primitive rings can be described very exactly.

Theorem 22 *A commutative ring R is primitive iff it is a field.*

Proof: (cf. Pilz [17], p.163 or Jacobson [11], p.7)
"$\Leftarrow$": Every field is primitive (${}_RR$ is a faithful irreducible R-module).
"$\Rightarrow$": Let R be commutative and primitive. We have to show that R has an identity and is simple. R has a modular maximal left ideal L, s.t. $\{r \in R : rR \subseteq L\} = \{0\}$, which is an ideal, because R is commutative. Hence $LR \subseteq L$, so $L \subseteq \{r \in R : rR \subseteq L\} = \{0\}$. $\{0\}$ is modular, so R has an identity. ■

4.2.2 Semi-simplicity

Jacobson's idea was to measure the departure of a ring from semi-simplicity (see [11]).

Definition 41 *Let R be a ring.*

- $\mathcal{P}(R) = (0) \iff$ *R is a subdirect product of prime rings. We call R $\mathcal{P}$-*semisimple*.*
- $\mathcal{N}(R) = (0) \iff$ *R contains no nonzero nil ideal. We call R $\mathcal{N}$-*semi-simple*.*
- $\mathcal{G}(R) = (0) \iff$ *R is a subdirect product of simple rings with identity. We call R $\mathcal{G}$-*semi-simple*.*
- $\mathcal{J}(R) = (0) \iff$ *R is a subdirect product of primitive rings. We call R ($\mathcal{J}$-)*semi-simple*.*

(Subdirect products will be introduced in section 4. We shall give the proof then. For the moment the theorem shows, "that semi-simple rings can be decomposed into rings with nice properties".)

Obviously (from 3.1),
$\mathcal{G}$-semi-simplicity $\Longrightarrow$ semi-simplicity $\Longrightarrow$ $\mathcal{N}$-semi-simplicity $\Longrightarrow$ $\mathcal{P}$-semi-simplicity.

4.2.3 Density Theorems

The above results are useful when studying the problem of "interpolation at finitely many points". We shall not go into details, but present two interesting results.

Definition 42 *Let D be a division ring, V a vector space over D. Consider a subring R of the ring of "linear mappings" $Hom_D(V, V)$. R is* n-transitive *if for all linear in-Dependant $x_1, \ldots, x_n \in V$ and arbitrary $y_1, \ldots, y_n$ there exists an $h \in R:\ h(x_i) = y_i,\ i = 1, \ldots, n$. If R is n-transitive for all $n \in \mathbf{N}$, we say R is* dense *in $Hom_D(V, V)$.*

Theorem 23

1. *Let R be a ring with identity, which is primitive on M. Then R is isomorphic to a dense subring of $Hom_{\Delta}(M,M)$, where M is a vector space over the division ring $\Delta := End_R(M)$.*
2. *Conversely, every dense subring of a $Hom_D(V,V)$, where V is a vector space over the division ring D, is primitive.*

Proof: (see Pilz [17], p.164, Jacobson [10], p.199)

■

Theorem 24 *Let $V \neq \emptyset$ be a vector space over the division ring D, $R \leq Hom_D(V,V)$.*

1. *If R is 1-transitive, then R is primitive on V.*
2. *If R is 2-transitive, then $Hom_R(V,V) \cong D$ and R is dense in $Hom_D(V,V)$.*

2. says that who can interpolate at 2 points is able to interpolate at arbitrary many points.

Proof: (see Pilz [17], p.165)

■

Chapter 5

Chain Conditions

Everything rises but to fall,
And increases but to decay.
Sallust (B.C. 86-34)

As promised earlier, the theory we have enjoyed is not quite as complicated if we can assume, that a chain condition holds. We shall start with the DCC and see, what we can get.

5.1 DCC

This section will result in a theorem due to Wedderburn and Artin. **Along this section we assume that R is a left artinian ring, i.e. it's left ideals together with $\subseteq$ have the DCC.**

Theorem 25 *$\mathcal{J}(R)$ is nilpotent.*

Proof: (cf. Pilz [17], p.167, Jacobson [10], p.202)
Let $\mathcal{J}^1(R)$, $\mathcal{J}^n(R) := \mathcal{J}(R)\mathcal{J}^{n-1}(R)$. Then

$$\mathcal{J}(R) \supseteq \mathcal{J}^2(R) \supseteq \dots$$

is a descending chain, which terminates with $\mathcal{J}^n(R) =: I$. Suppose $I \neq \{0\}$. Then

$$\mathbf{L} := \{L : L \text{ left ideal of } R, L \subseteq I \text{ and } IL \neq \{0\}\} \neq \emptyset,$$

since $II = \mathcal{J}^{2n}(R) = \mathcal{J}^n = I \neq \{0\}$, so $I \in \mathbf{L}$. $\mathbf{L}$ has a minimal element L_0 in which we find an element l, s.t. $Il \neq \{0\}$. Since $I \supset L_0 \supset Il \neq \{0\}$ and L_0 was chosen minimal with this property, $Il = L_0$. Hence there is an $i \in I$, s.t. $il = l$. Let i° be the inverse of i in $(R, \circ)$[1]. Then $0 = l - il - j(l - il) = l - (j + i - ji)l =$

[1]Recall the definition of quasi-regularity in chapter 2.3.

$l-(j\circ l)l=l-0l=l$, which contradicts the fact that $Il\neq\{0\}$. ■

Theorem 26 *Let I be a one- or two-sided ideal of R. Then I is nilpotent iff it is nil.*

Proof: (cf. Pilz [17], p.167)
We have to show the implication from the right to the left side. Let I be a nil ideal. Then $I\subseteq\mathcal{N}(R)\subseteq\mathcal{J}(R)$. But $\mathcal{J}(R)$ is nilpotent, so I is either. ■

Theorem 27 *Let R be a ring with identity. Then the following are equivalent:*

1. *R is primitive.*
2. *$R\cong Hom_D(V,V)$, where V is a finite-dimensional vector space over a division ring D.*
3. *R is a simple ring with identity.*
4. *R is a prime ring.*

Proof: (see Pilz [17], p.167)

■

From this we get the following result, which makes Radical-theory much easier.

Remark 7 *For a left- or right-artinian (in particular for a finite) ring R*

$$\mathcal{P}(R)=\mathcal{N}(R)=\mathcal{J}(R)=\mathcal{G}(R)$$

Proof:
With the results of chapter 3 it is enough to show that every maximal modular ideal is prime. Suppose M is a maximal modular ideal. Then R/M is a simple ring with identity which is equivalent to being a prime ring. So M was a prime ideal. ■

Theorem 28 *R is semi-simple iff it is isomorphic to a finite direct sum of simple rings with identity.*

Proof: (see Pilz [17], p.168, Jacobson [10], p.203)

■

Lemma 2 *A finite direct sum of simple rings fulfills both chain conditions.*

Sketch of the *Proof:*
We show that every ideal of the finite direct sum of rings is a finite direct sum of an ideal of the corresponding ring. Since all the rings are simple, the number of ideals we get in this way is finite either. From this we get that both chain conditions must hold. ■

Corollary 3 *A semi-simple ring fulfills both chain conditions.*

Since the right hand side of the last theorem is independent of "left" and "right" we get

Theorem 29 *If R is semi-simple, then R is left-artinian $\iff$ R is right-artinian.*

Theorem 30 *For a ring R the following are equivalent:*

1. *R is artinian and contains no nilpotent ideals $\neq 0$.*
2. *R is semi-primitive and artinian.*
3. *R is semi-simple and artinian.*
4. *${}_RR$ is a complete reducible R-module.*
5. *R is a direct sum of a finite number of rings R_i each of which is isomorphic to the ring of linear transformations of a finite-dimensional vector space over a division ring.*

Proof: (see Jacobson [10], p.203)

■

5.2 ACC

The following result is an immediate consequence of the definition of the ACC.

Theorem 31 *Let R be a* noetherian *ring, i.e. its ideals together with $\subseteq$ have the ACC. Then every ideal of R is finitely generated.*

Proof:
Let I be an arbitrary ideal of R. Since $I \neq \emptyset$, there we can choose an element i_1 in I. If $(i_1) = I$ we are done. If not we can choose an element $i_2 \in I \setminus (i_1)$. Now we check, whether $(i_1, i_2) = I$ and so on. This yields an ascending chain $(i_1) \subset (i_1, i_2) \subset \ldots$ which must end after a finite number of steps. The last ideal in this chain is finitely generated and equal to I. ■

Chapter 6

Some results for improved algorithms

Where we cannot invent, we may at least improve.
Colton (1780–1832)

6.1 Ideal Generators

Lowliness is the base of every virtue,
And he who goes the lowest builds the safest.
P.J. Bailey (1816–1902)

A set of generators is as powerful as the whole set of elements (it describes the whole set), but usually the set of generators is by far smaller. In this section we gather results that have to do with Moore-system properties. Using generators one can work not only with finite, but also with many infinite domains (so called finitely generated domains). $(\mathbf{Z}, +, *)$ is an example of an infinite ring, which is generated by a single element (namely 1). Any ideal of $\mathbf{Z}$ can be generated by a single element.

Infinite domains are not the only application. In the case of finite domains one can often decrease the complexity of algorithms.

Theorem 32 *The ideals of a ring form a Moore-system. It makes sense to speak of the ideal generated by a set of generators. For $G = \{g_1, \dots, g_r\}$ we denote the generated ideal $< G >$ by $(g_1, \dots, g_r)$. If an ideal is generated by a singleton, we call it a* principal ideal. *If every ideal of a ring is principal, the ring is called a* principal ideal domain.

The number of generators is fairly independent of the size of a ring, i.e. a smaller ring (e.g. a subring) can have more generators. This remains true if one takes minimal numbers of generators. As an example see that $\mathbf{Q}[x]$ is a principal ideal domain (i.e. every ideal can be generated by only one element, whereas its subring $\mathbf{Z}[x]$ is not.

Sums of finitely generated ideals are again finitely generated and the generators are "nothing new".

Lemma 3 *Let $I = (a_1, \ldots, a_n)$, $J = (b_1, \ldots, b_m)$, $K = (a_1, \ldots, a_n, b_1, \ldots, b_m)$. Then*

$$I + J = K \tag{6.1}$$

Proof: "$\subseteq$": let $I + J \ni x = a + b$, $a \in I$, $b \in J$. Clearly $a \in K$ and $b \in K$, since $I, J \subseteq K$. Hence $a + b \in K$.
"$\supseteq$": by definition K is the smallest ideal containing each of $a_1, \ldots, a_n, b_1, \ldots, b_m$. $I + J$ is an ideal containing $a_1, \ldots, a_n, b_1, \ldots, b_m$, hence $K \subseteq I + J$. ■

(Recall, that the sum of two ideals is again an ideal.)

Corollary 4 *Let R be a ring, $I = (a_1, \ldots, a_n) \trianglelefteq R$. Then*

$$a \in I \iff I + (a) = I$$

The product of ideals can also be handled in a simple way.

Theorem 33 *Let R be a ring, $I = (a_1, \ldots, a_n)$, $J = (b_1, \ldots, b_m)$, $K =< \{a_i b_j\}_{\substack{i=1,\ldots,n \\ j=1,\ldots,m}} > \trianglelefteq R$. Then $IJ = K$.*

Proof: "$\subseteq$": Every $a_i b_j$ is in K.
"$\supseteq$": We have to show that any $ab = k \in K$ can be generated by the $a_i b_j$'s. This is done by structural induction and left to hardcore-algebraists. ■

Generators make the computation of radicals easier, too.

Corollary 5 *(to (6.1)) Let R be a ring, s.t. all one- or two-sided nil ideals are finitely generated. Then $\mathcal{N}(\mathcal{R})$ is generated by the set of all the generators of these nil ideals. In particular $\mathcal{N}(\mathcal{R})$ is generated by the set of nilpotent elements of R.*

6.2 Factor^{2}ring–Avoidance

Sometimes it can happen, that one has to consider not only factor rings of rings but even factor rings of such factor rings. Computation in such factor^{2}rings is much more costly than in ordinary factor rings. The following theorem gives a constructive way to avoid factor^{2}rings and use isomorphic factor rings instead.

Lemma 4 *Let R be a ring $a_1, \ldots, a_n \in R$,
$\varphi : R \to R/(a_1, \ldots, a_n) = \bar{R}$ canonical. Then*

$$\varphi((a_1, \ldots, a_n, a_{n+1})) = (\varphi(a_{n+1})) \tag{6.2}$$

Proof: By (6.1) $x \in (a_1, \ldots, a_{n+1})$ can be written in the form $x_1 + x_2$, where $x_1 \in (a_1, \ldots, a_n)$ and $x_2 \in (a_{n+1})$. $\varphi(x) = \varphi(x_1) + \varphi(x_2) = 0 + (a_1, \ldots, a_n) + x_2 + (a_1, \ldots, a_n) = x_2 + (a_1, \ldots, a_n)$.
Therefore we only have to consider $x \in (a_{n+1})$. We assume that x is of the form

$$x = za_{n+1} + ra_{n+1} + a_{n+1}s + \sum_{finite} r_i a_{n+1} s_i$$

where $z \in \mathbf{Z}$, $r, s, r_i, s_i \in R$. Hence $\varphi(x) = z\varphi(a_{n+1}) + \varphi(r)\varphi(a_{n+1}) + \varphi(a_{n+1})\varphi(s) + \sum_{finite} \varphi(r_i)\varphi(a_{n+1})\varphi(s_i) \in (\varphi(a_{n+1}))$. ■

Theorem 34 *Let R be a ring $a_1, \ldots, a_{n+1} \in R$. Then*

$$[R/(a_1, \ldots, a_n)]/(a_{n+1} + (a_1, \ldots, a_n)) \cong R/(a_1, \ldots, a_{n+1}) \tag{6.3}$$

Proof: By the diamond–lemma (2)

$$\bar{R}/\bar{I} \cong R/\varphi^{-1}(\bar{I})$$

,where $\varphi : R \to R/(a_1, \ldots, a_n) = \bar{R}$ is the canonical homomorphism. Let $I = (a_1, \ldots, a_n)$ and $\bar{I} = (\varphi(a_{n+1}))$. We have to show that

$$\varphi^{-1}((\varphi(a_{n+1}))) = (a_1, \ldots, a_{n+1})$$

From (6.2) we have $\varphi((a_1, \ldots, a_{n+1})) = (\varphi(a_{n+1}))$. So we have to show that whenever $\varphi(x) \in (\varphi(a_{n+1}))$ then $x \in (a_1, \ldots, a_{n+1})$.
Suppose $\varphi(x) \in (\varphi(a_{n+1}))$.

$$\varphi(x) \in (\varphi(a_{n+1})) = \varphi((a_1, \ldots, a_{n+1})) \Rightarrow x \in (a_1, \ldots, a_{n+1})$$

■

Corollary 6 *Let R be a ring, $I \trianglelefteq R$, $J = (a_1 + I, \ldots, a_n + I) \trianglelefteq R/I$. Then*

$$[R/I]/J \cong R/[I + (a_1, \ldots, a_n)]$$

6.3 Divide et Impera

Work divided is in that manner shortened.
Martial (43–104 A.D.)

The proof of Birkhoff's theorem is in a way constructive. One can compute the ideals M_a for all a, if the ring is finite. (Be aware that the implication "constructivity $\Longrightarrow$ efficiency" does **not** hold. Computing an ideal maximal w.r.t. not containing an element a is almost as complex as computing the lattice of ideals of the ring.) The following is an algorithm, that "imitates" Birkhoff's proof.

```
Algorithm (SubDecDir):
input: R ring
output: subdirect decomposition of R
begin
    forall ($a \in R$) do
        find an ideal $M_a$ maximal w.r.t. not containing $a$
    od
    if the intersection of all $M_a$'s is nonzero then
            return ("the ring is subdirectly irreducible")
    else
            return ($\{R/M_a\}_{a \in R}$)
    fi
end
```

Algorithm SubDecDir is partially correct, i.e. it computes an isomorphic subdirect product if it terminates. **Z** is an example of a ring, where the algorithm does not terminate, since **Z** is only isomorphic to an infinite subdirect product of subdirectly irreducible rings. If the ring ideals together with $\leq$ does not have the ACC, some of the ideals M_a might not be finitely generated, hence the algorithm would not terminate either.

There are many simple improvements. If one has found ideals with zero intersection, one can stop. In general this algorithm computes almost the whole ideal lattice of the ring. Hence all improvements one uses for computing the ideal lattice, can be used.

Nevertheless this is not a good idea, in particular, when the ring is subdirectly irreducible, one has to compute all ideals. There are better ways to do it in such situations. For this purpose we give some remarks, which are easy to prove.

Remark 8

- *Every ideal contains a principal ideal.*
- *The intersection of all ideals of a ring is nonzero iff the intersection of all principal ideals is nonzero.*
- *A ring is subdirectly irreducible iff the intersection of all principal ideals is nonempty.*
- *(6.3) shows, that if $I = (a_1, \ldots, a_n)$, the factor ring R/I can be computed by iteratively factoring R modulo the principal ideals (a_1), $(a_2 + (a_1))$, etc.*

This means, that we can check for subdirect irreducibility using principal ideals only. The only disadvantage with principal ideals is (in the reducible case), that the resulting factor rings are not necessarily subdirectly irreducible. But we can simply go on decomposing the resulting factor rings until we come to an irreducible end.

Now we describe a recursive algorithm for decomposing a ring into a subdirect product of subdirectly irreducible rings:

```
**Algorithm (SubDecRec):**
input: set S of rings
output: Irred = set of subdirectly irreducible rings in S
    Red = set of subdirect factors of the reducible rings in S
begin
    forall rings in S do
        while the intersection of the ideals computed is nonzero do
            choose a new element $a$ from the ring
            if there is none left then
                    add the ring to the set of subdirectly irreducible rings
                    exit the while loop
            fi
            compute the principal ideal $(a)$
        od
        add the factor rings of the ring modulo the principal
        ideals to the set of reducible rings
    od
    return the two sets of rings
end
```

If we apply this algorithm to the set $\{R\}$, and then to the set of reducible rings until this set is empty, we get a subdirect decomposition of R. (6.3) convinces us, that this algorithm terminates, whenever algorithm SubDecDir terminates (and possibly in other cases, too). Now we see, how often we will need the factor^{2}ring-results of section 6.2. The big advantage of the algorithm SubDecRec is that one can compute with principal ideals in smaller rings, rather than big ideals in a big ring. This is the real gain in complexity.

Chapter 7

Polynomial rings

We assume that the reader has some knowledge about polynomials. Otherwise see Pilz [17]. Throughout the chapter R shall denote a commutative ring with identity such as $\mathbf{Z}$ or $\mathbf{Q}$. Whenever we talk about multivariate polynomials $K[x_1, \ldots, x_n]$, K shall denote a field as $\mathbf{Q}$ or $GF(q)$. By $[X]$ we denote the monoid (under multiplication) of *power products* $x_1^{i_1} \ldots x_n^{i_n}$.

Definition 43 *The set of all (univariate) polynomials with coefficients from R forms a ring with the usual addition and multiplication for polynomials, called* polynomial ring *over R. We denote it $R[x]$.*

$R[x]$ is again a commutative ring with identity. Hence we can go on and introduce another variable y different from x, and so on.

Notation 1 *For the ring $(((R[x_1])[x_2])\ldots)[x_n]$ we shall write $R[x_1, \ldots, x_n]$ or $R[X]$.*

Theorem 35 *If R is Noetherian (i.e. every ascending chain of ideals stops, i.e. every ideal is finitely generated) then $R[x_1, \ldots, x_r]$ is also Noetherian.*

Proof: (see Jacobson [10], p.418) ■

Clearly, every field is Noetherian, hence $K[X]$ is always Noetherian, i.e. every ideal in $K[X]$ is generated by only finitely many polynomials.

7.1 Orderings on $R[X]$

Order is a lovely nymph, the child of Beauty and Wisdom;
her attendants are Comfort, Neatness and Activity;
her abode is the valley of happiness:
she is always to be found when sought for,
and never appears so lovely
as when contrasted with her opponent, Disorder.
Samuel Johnson (1709–1784)

For the following orderings will be of interest, which are "compatible" with the multiplication of power products.

Definition 44 *An* admissible ordering *on* $[X]$ *fulfills the conditions*

- $1 = x_1^0 \dots x_n^0 < t \quad \forall t \in [X] \setminus \{1\}$ *and*
- $s < t \Longrightarrow su < tu$ *for all* $s, t, u \in [X]$.

Examples 4
- *The* lexicographic ordering *with* $x_{\pi(1)} > \dots > x_{\pi(n)}$, π *a permutation of* $\{1, \dots, n\}$.
$x_1^{i_1} \dots x_n^{i_n} <_{lex,\pi} x_1^{j_1} \dots x_n^{j_n}$ *iff there exists a* $k \in \{1, \dots, n\}$, *s.t. for all* $1 \leq l < k \quad i_{\pi(l)} = j_{\pi(l)}$ *and* $i_{\pi(k)} < j_{\pi(k)}$.
$\pi = id$ *yields the usual lexicographic ordering* $<_l ex$*:*
$$1 < x_1 < x_1^2 < \dots < x_2 < x_1x_2 < \dots$$
- *The* graduated lexicographic ordering *w.r.t. the permutation* π *and the weight function* $\omega : \{1, \dots, n\} \to \mathbf{R}^+$*:*
$s <_{glex,\pi,\omega} t$ *iff*
$$\left(\sum_{k=1}^{n} \omega(k) i_k < \sum_{k=1}^{n} \omega(k) j_k\right) \text{ or } \left(\sum_{k=1}^{n} \omega(k) i_k = \sum_{k=1}^{n} \omega(k) j_k \text{ and } s <_{lex,\pi} t\right)$$
for $s = x_1^{i_1} \dots x_n^{i_n}$, $t = x_1^{j_1} \dots x_n^{j_n}$.
$\pi = id$ *and* $\omega = 1_{const}$ *yields the usual graduated lexicographic ordering* $<_{glex}$*:*
$$1 < x_1 < x_2 < \dots < x_n < x_1^2 < x_1x_2 < x_2^2 < \dots < x_n^2 < \dots$$
- *The* graduated reverse lexicographic ordering*:*
for s, t *as above,* $s <_{grlex} t$ *iff*
$$\deg(s) < \deg(t) \text{ or } (\deg(s) = \deg(t) \text{ and } t <_{glex,\pi,1} s),$$
where $\pi(j) = n - j + 1$*:*
$$1 < x_n < x_{n-1} < \dots < x_1 < x_n^2 < x_n x_{n-1} < \dots < x_1^2 < \dots$$

- *From the admissible orderings* $<_1$ *on* $X_1 = [x_1, \ldots, x_i]$ *and* $<_2$ *on* $X_2 = [x_{i+1}, \ldots, x_n]$ *we can define the* product ordering *w.r.t.* $i \in \{1, \ldots, n-1\}$. $s <_{prod,i,<_1,<_2} t$ *iff*

$$s_1 <_1 t_1 \text{ or } (s_1 = t_1 \text{ and } s_2 <_2 t_2),$$

for $s = s_1 s_2$, $t = t_1 t_2$ *and* $s_1, t_1 \in X_1$, $s_2, t_2 \in X_2$.

Definition 45 *Let s be a power product in $[X]$, f a nonzero polynomial in $R[X]$, F a subset of $R[X]$, $<$ an admissible ordering on $[X]$.*

- *$coeff(f, s)$ is the coefficient of s in f.*
- *$lpp(f) := \max_<\{t \in [X] : coeff(f, t) \neq 0\}$, the* leading power product *of f.*
- *$lc(f) := coeff(f, lpp(f))$, the* leading coefficient *of f.*
- *$in(f) := lc(f)lpp(f)$, the* initial *of f.*
- *$red(f) := f - in(f)$, the* reductum *of f.*
- *$lpp(F)$, $lc(F)$, $in(F)$ and $red(F)$ are defined in the obvious way.*

Admissible orderings on $[X]$ induce a partial ordering on $R[X]$:

Definition 46 *Let $f, g \in R[X]$.*

$$\begin{array}{lll} f \ll g & :\Longleftrightarrow & f = 0 \text{ and } g \neq 0 \text{ or} \\ & & f \neq 0, g \neq 0 \text{ and } lpp(f) < lpp(g) \text{ or} \\ & & f \neq 0, g \neq 0, lpp(f) = lpp(g) \text{ and } red(f) \ll reg(g). \end{array}$$

Lemma 5 *$\ll$ is a Noetherian partial ordering on $R[X]$.*

7.2 Reduction

Manifest plainness, embrace simplicity,
Reduce selfishness, have few desires.
Lao-Tzu (B.C. 600)

In order to get simple expressions, one usually transforms expressions using simple rules. We shall start with a well-known example.

Example 5 *Let $p(x) = (x + 1)(x - 1)$. We want to know, what $p(3)$ is. The answer is simple: $(3 + 1)(3 - 1)$. Usually this is not satisfying, so we start to simplify this expression: $(3+1)(3-1) \to 4*(3-1) \to 4*2 \to 8$. Here we stop. We say that 8 is in* **normal form** *(it can not be further simplified). This is not the only way to simplify the expression, we could also have computed: $(3+1)(3-1) \to 3 * (3 - 1) + 1 * (3 - 1) \to \ldots \to 8$. Each of the simplification sequences terminates, we say the simplification is* **Noetherian**. *The result is always the same, although the intermediate results are different. We call this property* **(local) confluence**. *If two expressions are equivalent (i.e. they "mean" the same thing), then once during simplification they will be simplified to the same expression. This is called the* **Church–Rosser property**.

Now we shall give a short introduction in the theory of reduction relations. Proofs will be omitted here. They can be found in almost any book about reduction relations or string rewriting (as [3], [4], [7], [23], [24]).

7.3 Reduction Relations

Definition 47 *Let M be a set, $\rightarrow$ a binary relation (i.e. as subset of $M \times M$). Then we call $\rightarrow$ a* reduction relation*. If $a \rightarrow b$ we say that a* reduces *to b. For reduction relations $\rightarrow$ and $\rightarrow'$ we define:*

- *$a \rightarrow \circ \rightarrow' b$ iff there exists c, s.t. $a \rightarrow c \rightarrow' b$.*
- *$a \leftarrow b$ iff $b \rightarrow a$. $\leftarrow$ is called the* inverse relation *of $\rightarrow$.*
- *$a \leftrightarrow b$ iff $a \rightarrow b$ or $a \leftarrow b$. $\leftrightarrow$ is called the* symmetric closure *of $\rightarrow$.*
- *$\rightarrow^i$ is the reduction relation defined by*
  $\rightarrow^0 := =$ and
  $\rightarrow^i := \rightarrow \circ \rightarrow^{i-1}$, for $i \geq 1$.
  I.e. $a \rightarrow^i b$ iff there exist $c_0, \ldots, c_i$, s.t. $a = c_0 \rightarrow c_1 \rightarrow \ldots \rightarrow c_i = b$.
- *$\rightarrow^+ := \bigcup_{i=1}^{\infty} \rightarrow^i$, the* transitive closure *of $\rightarrow$.*
- *$\rightarrow^* := \bigcup_{i=0}^{\infty} \rightarrow^i$, the* reflexive–transitive closure *of $\rightarrow$.*
- *$\leftrightarrow^*$ is called the* reflexive–transitive–symmetric *closure of $\rightarrow$.*

Definition 48 *For abbreviating our arguments we shall introduce some more notations.*

- *$x \rightarrow$ means x is* reducible*, i.e. $\exists y : \; x \rightarrow y$.*
- *$\underline{x}$ means x is* irreducible *or in* normal form*.*
- *$x \downarrow y$ means that x and y have a* common successor *z, i.e. $x \rightarrow z \leftarrow y$.*
- *$x \uparrow y$ means that x and y have a* common predecessor *z, i.e. $x \leftarrow z \rightarrow y$.*
- *x is a* normal form *of y iff $y \rightarrow^* \underline{x}$.*

Definition 49
- *$\rightarrow$ is* Noetherian *iff every reduction sequence terminates, i.e. there is no infinite sequence $x_1 \rightarrow x_2 \rightarrow \ldots$.*
- *$\rightarrow$ is* Church–Rosser *iff $a \leftrightarrow^* b$ implies $a \downarrow_* b$.*

Theorem 36 *If $\rightarrow$ is Noetherian and Church–Rosser, then the equivalence problem for $\rightarrow$ is decidable.*

Definition 50 *$\rightarrow$ is* confluent *iff $x \uparrow^* y \Longrightarrow x \downarrow_* y$.*
$\rightarrow$ is locally confluent *iff $x \uparrow y \Longrightarrow x \downarrow_* y$.*

Theorem 37 *$\rightarrow$ is Church–Rosser iff $\rightarrow$ is confluent. If $\rightarrow$ is Noetherian, then $\rightarrow$ is Church–Rosser iff $\rightarrow$ is locally confluent.*

7.4 Reduction in $K[X]$

Now, we shall say, what the reduction relation is for polynomials.

Definition 51 *Let $f, g, h \in K[X]$, $F \subseteq K[X]$. g* reduces to *h modulo f ($g \rightarrow_f h$) iff there are power products $s, t \in [X]$, s.t. $coeff(g, s) = c \neq 0$, $s = lpp(f)t$ and*

$$h = g - \frac{c}{lc(f)} t f.$$

Among others the following lemma says that this reduction is compatible with addition and multiplication of polynomials and with any admissible ordering.

Lemma 6 *Let $g, \bar{g}, h \in K[X]$, $F \subseteq K[X]$, $s \in [X]$, $a \in K^* = K \setminus \{0\}$.*

- *$g \rightarrow_F \bar{g} \Longrightarrow \bar{g} \ll g$.*
- *$\rightarrow_F$ is Noetherian.*
- *$g \rightarrow_F \bar{g} \Longrightarrow asg \rightarrow_F as\bar{g}$.*
- *$g \rightarrow_F \bar{g} \Longrightarrow g + h \downarrow_F^* \bar{g} + h$.*

The following theorem explains the usefulness of this kind of reduction.

Theorem 38 *Let $F \subseteq K[X]$, $I :=< F >$, $f, g \in K[X]$.*

$$\sim_I = \leftrightarrow_F^*$$

Together with the results from chapter 7.2, we get that equality in the ring $K[X]/ < F >$ can be decided, if $\rightarrow_F$ is Church–Rosser.

Definition 52 *$F \subseteq K[X]$ is a* Groebner basis *for $< F >$ iff $\rightarrow_F$ is Church–Rosser.*

7.5 Buchberger's Theorem

By Newman's Lemma it suffices to test for local confluence of $\rightarrow_F$ for testing, whether $\rightarrow_F$ is Church–Rosser. However, there are still infinitely many $f \uparrow_F g$. Buchberger [5] gives a method to reduce this test to finitely many f's and g's, so called critical pairs.

Definition 53 *Let $f, g \in K[X]^*$, $t = lcm(lpp(f), lpp(g))$.*

$$cp(f,g) := \left(t - \frac{1}{lc(f)} \frac{t}{lpp(f)} f, t - \frac{1}{lc(g)} \frac{t}{lpp(g)} g \right)$$

is the critical pair *of f and g. The difference between the components of $cp(f,g)$ is the* S-polynomial *$spol(f,g)$ of f and g. (The name S-polynomial is an abbreviation for "*syzygy-polynomials*"[1].)*

Theorem 39 *The following statements are equivalent:*

1. *F is a Groebner Basis.*
2. *$g_1 \downarrow_F^* g_2$ for all critical pairs (g_1, g_2) of elements of F.*
3. *$spol(f,g) \to_F 0 \quad \forall f, g \in F$.*

Proof: (see Buchberger [5]) ■

Algorithm (Buchberger's algorithm):
input: F, a finite subset of $K[X]^*$.
output: G, a Groebner Basis for the ideal $< F >$.

```
begin
    $G := F$;
    $C := \{\{g_1, g_2\} : g_1, g_2 \in G, g_1 \neq g_2\}$;
    while not all pairs in C are marked do
        choose an unmarked pair $\{g_1, g_2\}$;
        mark $\{g_1, g_2\}$;
        $h :=$ normal form of $spol(g_1, g_2) w.r.t. \to_G$;
        if $h \neq 0$ then
            $C := C \cup \{\{g, h\} : g \in G\}$;
            $G := G \cup \{h\}$;
        fi
    od
    return G;
end
```

Theorem 40 *Every ideal in $K[X]$ has a Groebner basis.*

Example 6 *Let $F = \{x^2y^2 + y - 1, x^2y + x\}$. The above algorithm produces a Groebner basis $G = \{x^2y^2 + y - 1, x^2y + x - 1, -xy + y - 1, y - 1, -x\}$.*

[1]For more about syzygies see Cox, Little, O'Shea [6].

$\{y-1, x\} \subset G$ would also be a Groebner basis (as we can verify), but is much simpler. Next, we have a look at some operations on Groebner bases that preserve the Church–Rosser property.

Theorem 41 *Let G be a Groebner basis for $I \trianglelefteq K[X]$, $g, h \in G$, $g \neq h$. Then*

$$\begin{array}{rcl} lpp(g) | lpp(h) & \Longrightarrow & G' = G \setminus \{h\} \text{ is also a Groebner basis for } I. \\ h \rightarrow_{G\setminus\{h\}} h' & \Longrightarrow & G' = (G \setminus \{h\}) \cup \{h'\} \text{ is also a Groebner basis for } I. \end{array}$$

If we transform a given Groebner basis as indicated by the last theorem, we reach a Groebner basis, which can not be transformed any more. Finally we can normalize all the polynomials (i.e. divide by the leading coefficient). Another nice property of Groebner bases is that

Theorem 42 *this basis is unique.*

7.6 Improvements on Buchberger's Algorithm

We stated that Buchberger's algorithm terminates with a unique Groebner basis. Nevertheless it can take a long time. We shall give two ideas for improvements on Buchberger's algorithm here. Both consist of reducing the set of S-polynomials, so that less polynomials have to be reduced. The following propositions give conditions under which one can omit certain S-polynomials.

Proposition 1 *Let $G \subset K[X]$ be a set of polynomials. If*

$$lcm(lpp(f), lpp(g)) = lpp(f) \cdot lpp(g)$$

then $spol(f, g) \rightarrow_G 0$ (so it need not be tested).

Proof: (see Cox, Little, O'Shea [6], p.103)

■

In the following proposition "S is a basis of $spol(G)$" means that all S-polynomials of G can be obtained as linear combinations of the polynomials in S. I.e. we only need to consider the polynomials in S. This proposition gives a systematic way for reducing the set S to a small one.

Proposition 2 *Let $G = (g_1, \ldots, g_t)$, $S \subseteq spol(G) = \{spol(f, g) : f, g \in G\}$ a basis of $spol(G)$. If there exist distinct $g_i, g_j, g_k \in G$, s.t.*

$$lpp(g_k) \text{ divides } lcm(lpp(g_i), lpp(g_j))$$

and both $spol(g_i, g_k)$ and $spol(g_j, g_k)$ are in S, then $S \setminus \{spol(g_i, g_j)\}$ is also a basis of $spol(G)$.

Proof: (see Cox, Little, O'Shea [6], p.107)

■

These considerations lead to an improved algorithm:

Algorithm (improved Groebner basis algorithm):
input: F, a finite subset of $K[X]^*$.
output: G, a Groebner Basis for the ideal $< F >$.

```
begin
    $G := F$;
    $C := \{\{g_1, g_2\} : g_1, g_2 \in G, g_1 \neq g_2\}$;
    while not all pairs in C are marked do
        choose a pair $\{g_1, g_2\} \in \mathbf{C}$;
        if $lcm(lpp(g_1), lpp(g_2)) \neq lpp(g_1) \cdot lpp(g_2)$ and
            Criterion($g_1, g_2, \mathbf{C}$) is $false$ then
            $h :=$ normal form of $spol(g_1, g_2) w.r.t. \rightarrow_G$;
            if $h \neq 0$ then
                $C := C \cup \{\{g, h\} : g \in G\}$;
                $G := G \cup \{h\}$;
            fi
        fi
        remove $\{g_1, g_2\}$ from C;
    od
    return G;
end
```

where Criterion($g_1, g_2, \mathbf{C}$) is true iff there is $g_1, g_2 \neq f \in G$ s.t. $\{g_1, f\}$ and $\{g_2, f\}$ are not in **C** and $lpp(g_k)$ divides $lcm(lpp(g_i), lpp(g_j))$.

7.7 Applications

He that will not apply new remedies
must expect new evils.
Francis Bacon (1561–1626)

Here we gather some common applications of Groebner bases to polynomial ring theory.

Equality of ideals: For $f_1, \ldots, f_m, g_1, \ldots, g_k \in K[X]$, $I :=< f_1, \ldots, f_m >$, $J :=< g_1, \ldots, g_k >$, $I = J$ iff the corresponding unique Groebner bases are equal.

Ideal membership: Let G be a Groebner basis for $I \trianglelefteq K[X]$, $f \in K[X]$. Then

$$f \in I \iff f \rightarrow_G 0.$$

Radical membership: Let $f, f_1, \ldots, f_m \in K[X]$, $I =< f_1, \ldots, f_m >$. We wish to decide $f \in I$.
We need a little help from a great mathematician.

Theorem 43 *Let K be an algebraically closed field and $I \trianglelefteq K[X]$. Then $\sqrt{I}$ consists exactly of those polynomials in $K[X]$ which vanish on all the common roots of I.*

Proof: (see Cox, Little, O'Shea [6]) ■

So we have to check whether f vanishes on every common root of $f_1, \ldots, f_m$. This is equivalent to the fact that the system of equations $f_1 = \ldots = f_m = zf - 1 = 0$ has no solution, where z is a new variable. I.e.

$$f \in \sqrt{< f_1, \ldots, f_m >} \iff 1 \in < f_1, \ldots, f_m, zf - 1 >,$$

which can be checked as we have seen above.

Chapter 8

GAP–Manual: Rings

There will one day spring from the brain of science a machine or force so fearful in its potentialities, so absolutely terrifying, that even man, the fighter, who will dare torture and death, will be appalled, and so abandon war forever.
Thomas A. Edison (1847–1931)

Obviously that day has not come yet. Nevertheless it seemed useful to implement all the knowledge above on a computer. We chose the computer-algebra system G.A.P.

Before we start, please be aware that most of the functions work fine with finite rings, but will not necessarily do so with infinite rings. Nonetheless we sometimes left it to the user to decide, whether it makes sense to apply the functions to infinite structures, if the algorithms did not explicitly require finiteness. Following a (no longer) unwritten rule of GAP, the functions do not compute more than explicitly desired. There are nevertheless some exceptions for speeds sake.

Throughout the examples, whenever a line begins with "`gap>`" or "`>`" it is input from the user. The rest is GAP output.

8.1 Preliminaries

We start with a short introduction to the programming language GAP. We will give examples of the most basic GAP-expressions rather than exact definitions. For detailed information see [20].

8.1.1 Data Structures

Permutations: `(1,2,3,4)`, `(1,2)(3,4)` and `(1,2,3)` are permutations. As the notation suggests permutations are always represented as products of cycles.

Lists: `[1..6]`, `[1,2,4,8]`, `[a,b,c]` and `[1,2,[3,4,5],6]` are lists. From a list l we can extract the i-th component with `l[i]`.

Matrices: The matrix $\begin{pmatrix} 3 & 4 \\ 1 & 0 \end{pmatrix}$ is written as `[ [3,4], [1,0] ]` in GAP.

Records: A record is a collection of components. Each component has a unique name, which is an identifier that distinguishes this component, and a value, which is an object of arbitrary type. You can access and change the elements of a record using its name. Record literals are written by writing down the components in order between `rec(` and `)`, and separating them by commas. Each component consists of the name, the assignment operator ":=", and the value.

```
gap> rec( a := 1, b := "2" ); # a record with two components
    rec(
      a := 1,
      b := "2" )
gap> rec( a := 1, b := rec( c := 2 ) ); # rec may contain records
    rec(
      a := 1,
      b := rec(
             c := 2 ) )
```

Records usually contain elements of various types, i.e., they are usually not homogeneous like lists. In the last example `a` contains an integer, whereas `b` contains a record.

8.1.2 Operations

The usual operations as addition, multiplication, etc. can be applied to the usual objects as integers, permutations, etc. in the way known from mathematics. One advantage of GAP is that e.g. the operation "*" can have various meanings depending on which objects it is applied to. So "*" applied to integers means usual multiplication, applied to permutations it means applying the one permutation after the other, applied to words over an alphabet it means concatenating them, a.s.o. In addition we can give even more meaning to the "*"-sign. Applied to two rings, "*" shall mean computing the direct product of these rings, applied to two ideals, "*" stands computing the ideal product of these ideals. We shall use this GAP feature as often as possible, because it makes computing much easier for the mathematician, if he can use the symbols he is used to.

8.1.3 Algebraic Domains

GAP knows several different domains. All of them are represented by records containing in their components all the knowledge about them. Usually one need not know how this record looks like, because for every single information about the domain there is a function which (when called by the user) goes through the record and tries to find out if there is suitable information or a function that computes the desired information.

8.1.4 Elements of Special Classes of Fields

- Elements of the finite field $GF(q)$ of order $q = p^n$ are represented in the following way: `Z(q)` is some multiplicatively generating element of $GF(q)^{*}$[1] and `0*Z(q)` the additive neutral element. So the set of elements of the field $GF(q)$ is `{ 0*Z(q), Z(q), Z(q)^2, ..., Z(q)^(q-1) = Z(q)^0 }`. They are added and multiplied with the usual operators "+" and "*".
- Elements of the cyclotomic field $\mathbf{Q}_n = \mathbf{Q}(e_n)$ are represented in the following way: `E(n)` returns the primitive n-th root of unity $e_n = e^{\frac{2\pi i}{n}}$. Cyclotomics are usually entered as sums of roots of unity with rational coefficients.

```
gap> E(5) * E(5)^2; ( E(5) + E(5)^4 ) * E(5)^2;
    E(5)^3
    E(5)+E(5)^3
gap> ( E(5) + E(5)^4 ) * E(5);
    -E(5)-E(5)^3-E(5)^4
```

8.2 Defining a Ring

In GAP a ring is represented as a record containing the information necessary to compute with it. A ring can be defined by precising it's generating elements. GAP knows rings of square matrices over finite fields, rings of integers in subfields of cyclotomic fields, integers, gaussian integers and (univariate) polynomial rings[2] with the usual operations + and $*$. Note that matrix-rings are the only finite rings GAP knows at the moment (unless finite fields). It is not yet possible to view fields, defined in GAP as such, as rings. Neither can one use the functions applicable to fields in GAP for rings, that turn out to be fields, but are defined as rings. This problem can not yet be overcome.

[1]Note, that such an element always exists, i.e that $GF(q)^*$ is cyclic. A proof of this property can be found in Pilz [19].

[2]How polynomials are represented and handled can be read in the GAP-Online Manual [20]. Some functions especially designed for polynomials will be presented in chapter 8.7.

8.2.1 Ring Records

Any domain in GAP is stored as a record containing all the information about it in its record components. We shall list the most important components here, i.e. those which are almost always known about a ring. For the following suppose that `R` is a variable in GAP holding a ring (as a record). Then

`R.isDomain` is always `true`
`R.isRing` is always `true` (even for ideals !)
`R.isFinite` is `true` iff the ring is finite,
`R.size` contains the number of elements or the string "infinity" (if known)
`R.elements` contains the set of elements of the ring
`R.generators` contains a list of generating elements of the ring
`R.zero` contains the additive neutral element of the ring
if the component `R.name` is bound its content is printed instead of the default message.

Besides there are lots of so called "knowledge components", that hold additional information about the ring (e.g. `R.isCommutativeRing`, `R.isUniqueFactorizationRing`, `R.IsRadicalRing`, ...), which are not necessarily bound. Usually GAP adds knowledge about a ring to the record only if the user asks for it. This means that `R.isCommutativeRing` will not have any value (it will in fact not even exist) before the user asks `IsCommutativeRing(R)`. The result (`true` or `false`) will be returned and stored in `R.isCommutativeRing` automatically.

`operations` is the component containing all functions applicable to the ring (as record components). Usually this is the record `RingOps` (resp. `PolynomialRingOps` for polynomial rings, `IdealOps` for ideals) and `PolynomialIdealOps` for polynomial ring ideals).

The function `RecFields` returns a list of all record components of a record.

```
gap> RecFields(RingOps);
[ "name", "operations", "Elements", "IsFinite", "Size", "=",
  "<", "in", "IsSubset", "Intersection", "Union", "IsParent",
  "Parent", "Difference", "Representative", "Random", "Print",
  "Polynomial", "FastPolynomial", "PolynomialRing",
  "LaurentPolynomialRing", "Indeterminate",
  "InterpolatedPolynomial", "IsCommutativeRing",
  "IsIntegralRing", "IsUniqueFactorizationRing",
  "IsEuclideanRing", "Quotient", "IsUnit", "Units",
  "IsAssociated", "StandardAssociate", "Associates",
  "IsIrreducible", "IsPrime", "Factors", "EuclideanDegree",
  "EuclideanRemainder", "EuclideanQuotient",
  "QuotientRemainder", "QuotientMod", "PowerMod", "Gcd",
  "GcdRepresentation", "Lcm", "AsGroup", "AsAdditiveGroup",
  "Module", "AsRing", "FactorRing", "PartialDecomposition",
  "SubdirectDecomposition", "IsNilpotentElement",
  "Nilradical", "Nilpotents", "JacobsonRadical",
  "IsRadicalRing", "IsSimple", "IsSemisimple", "Subring",
  "LeftIdeal", "RightIdeal", "Ideal", "IsLeftIdeal",
  "IsRightIdeal", "IsIdeal" ]
```

8.2.2 Ring

`Ring` (gen)
`Ring` returns the ring generated by the elements of the list *gen* (in the sense of Moore-systems). Valid elements in *gen* are integers, gaussian integers, elements of finite fields, elements of cyclotomic fields and square matrices over finite fields. Rings are usually printed out in the form `Ring(`list of generators`)`.

```
gap> Ring(1);
Integers;
gap> m1 := [[Z(3)^0,Z(3)^0,Z(3)],[Z(3),0*Z(3),Z(3)],
[0*Z(3),Z(3),0*Z(3)]];  # first matrix
[ [ Z(3)^0, Z(3)^0, Z(3) ], [ Z(3), 0*Z(3), Z(3) ],
  [ 0*Z(3), Z(3), 0*Z(3) ] ]
gap> m2 := [[Z(3),Z(3),Z(3)^0],[Z(3),0*Z(3),Z(3)],
[Z(3)^0,0*Z(3),Z(3)]];  # second matrix
[ [ Z(3), Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), Z(3) ],
  [ Z(3)^0, 0*Z(3), Z(3) ] ]
gap> r := Ring(m1,m2);
Ring( [ [ Z(3)^0, Z(3)^0, Z(3) ], [ Z(3), 0*Z(3), Z(3) ],
  [ 0*Z(3), Z(3), 0*Z(3) ] ],
[ [ Z(3), Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), Z(3) ],
  [ Z(3)^0, 0*Z(3), Z(3) ] ] )
gap> Size(r);
2187
```

8.2.3 PolynomialRing

`PolynomialRing` (R)
`PolynomialRing` returns the polynomial ring $R[x]$ if R is a commutative ring with identity or a field[3]

```
gap> f := GF(3^2); # the Galois field with 9 elements
GF(3^2)
gap> pr := PolynomialRing(f);
PolynomialRing( GF(3^2) )
```

8.2.4 AsRing

Another possibility is to construct a ring out of an abelian group and a binary function, representing multiplication.

[3]For more details about fields see the GAP-online-manual [20].

`AsRing` (G , f)
`AsRing` returns the ring, that results from the abelian group G by adding the binary function f as a multiplication. f is tested for associativity and distributivity over the group operation and eventually an error is reported. If the group is not abelian an error is reported.

```
gap> s3 := Group((1,2),(1,2,3)); # the group generated by the two permutations
Group( (1,2), (1,2,3) )
gap> mult := function (x,y)      # mult is a function that returns the
> return ();                     # identity permutation () for all
> end;                           # arguments x and y
function ( x, y ) ... end
gap> AsRing(s3,mult);            # try the noncommutative group S3
Error, sorry, <G> is not abelian in
G.operations.AsRing( G, mult ) called from
AsRing( s3, mult ) called from
main loop
brk> quit;
gap> z8 := Group((1,2,3,4,5,6,7,8)); # this group is isomorphic to Z8
Group( (1,2,3,4,5,6,7,8) )
gap> r8 := AsRing(z8,mult);
RingfromGroup(Group( (1,2,3,4,5,6,7,8) ) , function ( x, y )
    return ();
end
gap> Size(r8);
8
gap> Elements(r8);
[ (), (1,2,3,4,5,6,7,8), (1,3,5,7)(2,4,6,8), (1,4,7,2,5,8,3,6),
  (1,5)(2,6)(3,7)(4,8), (1,6,3,8,5,2,7,4), (1,7,5,3)(2,8,6,4),
  (1,8,7,6,5,4,3,2) ]
```

Note, that the elements of the ring are different from the elements of the group. An element g of the group corresponds to the element `AsRingElement`(g,R) of the ring. Conversely `AsGroupElement`(x) returns the element x of the ring as group element.

8.2.5 AsGroupElement

`AsGroupElement` (x)
`AsGroupElement` returns the ring element x as an element of the corresponding group.

```
gap> e := Elements(r8)[2];
(1,2,3,4,5,6,7,8)
```

```
gap> e = (1,2,3,4,5,6,7,8);
false
gap> AsGroupElement(e) = (1,2,3,4,5,6,7,8);
true
```

8.2.6 AsRingElement

`AsRingElement` (x , R)
From an element x of a group `AsRingElement` computes the corresponding element in the ring R.

```
gap> (1,2,3,4,5,6,7,8) in r8;
false
gap> AsRingElement((1,2,3,4,5,6,7,8),r8);
(1,2,3,4,5,6,7,8)
gap> last in r8;
true
```

Another way to get new rings is to build them up from existing ones.

8.2.7 DirectProduct

$R_1 * R_2$ or
`DirectProduct` (L)
The first command returns the direct product of the two rings R_1 and R_2 as a ring. The second command returns the direct product of the arbitrary many rings in the list L.

8.3 Subrings and Ideals

8.3.1 Subring

A subring of a ring can be defined with the following command:

`Subring` (R , gen)
`Subring` returns the subring of the ring R generated by the elements of *gen*. This does essentially the same as Ring (gen) exept, that R is defined as a parent ring of the created ring, which is remembered in the record component "parent".

```
gap> e := [[Z(2)^0,0*Z(2),0*Z(2)],[Z(2)^0,0*Z(2),Z(2)^0],
[0*Z(2),0*Z(2),0*Z(2)]];
[ [ Z(2)^0, 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ]
gap> s := Subring(r,[e]);
Ring( [ [ Z(2)^0, 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] )
gap> Size(s);
4
```

8.3.2 IsSubring

`IsSubring` (R , S)
If *S* is a subring of *R*, `IsSubring` returns `true` and *R* is remembered in the record component `S.parent`. Otherwise the result is `false`.

```
gap> IsSubring(r,s);
true
```

8.3.3 Ideal

`Ideal` (R , gen)
`Ideal` returns the ideal of the ring R generated by the elements of the list *gen*. R is remembered in the record component isIdeal. **Note**, that an ideal is a subring of the ring and hence a ring. However, generators are ideal generators!

```
gap> r;
Ring( [ [ 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ Z(2)^0, 0*Z(2), Z(2)^0 ] ],
[ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] )
gap> Size(r);
64
gap> e := [[0*Z(2),0*Z(2),0*Z(2)],[0*Z(2),0*Z(2),Z(2)^0],
[0*Z(2),0*Z(2),0*Z(2)]];
[ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ]
gap> i := Ideal(t,[e]);
Ideal ( Ring( [ [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), Z(2)^0 ] ],
[ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ) ,
[ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] )
gap> Size(i);
4
gap> Elements(i);
[ [ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2) ] ],
  [ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2) ] ],
  [ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2) ] ],
  [ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ]
```

8.3.4 IsIdeal

`IsIdeal` (R , S)
`IsIdeal` returns `true` if S is an ideal of the ring R and `false` otherwise. If the result is `true`, R is stored in the record component `s.isIdeal`. An ideal is usually printed out in the form `Ideal(` ring, list of generators of the ideal `)`.

```
gap> IsIdeal(r,s);
false
gap> IsIdeal(r,i);
true
```

8.3.5 LeftIdeal

`LeftIdeal` (R , gen)
`LeftIdeal` returns the left ideal of the ring R generated by the elements of the list *gen*. R is remembered in the record component isLeftIdeal. **Note**, that a left ideal is a subring of the ring and hence a ring. However, generators are left ideal generators!

8.3.6 IsLeftIdeal

`IsLeftIdeal` (R , S)
`IsLeftIdeal` returns `true` if S is a left ideal of the ring R and `false` otherwise. If the result is `true`, R is stored in the record component `s.isLeftIdeal`.

8.3.7 RightIdeal

`RightIdeal` (R , gen)
`RightIdeal` returns the right ideal of the ring R generated by the elements of the list *gen*. R is remembered in the record component isRightIdeal. **Note**, that a right ideal is a subring of the ring and hence a ring. However, generators are right ideal generators!

8.3.8 IsRightIdeal

`IsRightIdeal` (R , S)
`IsRightIdeal` returns `true` if S is a right ideal of the ring R and `false` otherwise. If the result is `true` R is stored in the record component `s.isRightIdeal`.

8.3.9 Operations for (left-, right-) ideals

$i_1 + i_2$
returns the ideal $i_1 + i_2$ if i_1 and i_2 are ideals of the same ring.

$i_1 * i_2$
returns the ideal $i_1 i_2$ if i_1 and i_2 are ideals of the same ring.

8.3.10 Rad

`Rad` (I)
`Rad` returns the radical $\sqrt{I}$ of the ideal I.

```
gap> Rad(i);
Ideal ( Ring( [ [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), Z(2)^0 ] ],
[ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ) ,
[ [ 0*Z(2), 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ,
[ [ 0*Z(2), 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ,
[ [ 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ,
[ [ 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ,
[ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ Z(2)^0, 0*Z(2), Z(2)^0 ] ] ,
[ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), Z(2)^0 ] ] ,
[ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ Z(2)^0, 0*Z(2), Z(2)^0 ] ] ,
[ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), Z(2)^0 ] ] )
gap> Size(last);
64
```

8.3.11 IsModularLeftIdeal

`IsModularLeftIdeal` (L)
`IsModularLeftIdeal` returns `true` if there exists $e \in L$, s.t. for all $r \in R$ $re - r \in L$ and `false` otherwise.

8.3.12 IsMaximalIdeal

`IsMaximalIdeal` (I)
`IsMaximalIdeal` returns `true` if there is no ideal M s.t. $I \subset M \subset R$ and `false` otherwise.

```
gap> IsMaximalIdeal(i);
true
```

8.3.13 IsMaximalLeftIdeal

`IsMaximalLeftIdeal` (I)
`IsMaximalLeftIdeal` returns `true` if there is no left ideal M s.t. $I \subset M \subset R$ and `false` otherwise.

8.3.14 IsMaximalRightIdeal

`IsMaximalRightIdeal` (I)
`IsMaximalRightIdeal` returns `true` if there is no right ideal M s.t. $I \subset M \subset R$ and `false` otherwise.

8.3.15 IsPrimeIdeal

`IsPrimeIdeal` (I)
`IsPrimeIdeal` returns `true` if the ideal *I* is prime and `false` otherwise.

8.3.16 QuickIdealElements

`QuickIdealElements` (I , trivgens , mcs)
If one has additional knowledge about an ideal it may be simpler to compute all the set of elements. `QuickIdealElements` computes the set of elements of the ideal *I*. If one of the elements of *trivgens* is found (an element, which generates the whole ring (as ideal)), the set of elements of the ring is returned. The same if all the elements of the set *mcs* (maximal contained set) are among the ideal elements so far computed.

8.3.17 QuickLeftIdealElements

`QuickLeftIdealElements` (I , trivgens , mcs)
If one has additional knowledge about an ideal it may be simpler to compute all the set of elements. `QuickLeftIdealElements` computes the set of elements of the left ideal *I*. If one of the elements of *trivgens* is found (an element, which generates the whole ring (as ideal)), the set of elements of the ring is returned. The same if all the elements of the set *mcs* (maximal contained set) are among the ideal elements so far computed.

8.3.18 QuickRightIdealElements

`QuickRightIdealElements` (I , trivgens , mcs)
If one has additional knowledge about an ideal it may be simpler to compute all the set of elements. `QuickRightIdealElements` computes the set of elements of the right ideal *I*. If one of the elements of *trivgens* is found (an element, which generates the whole ring (as ideal)), the set of elements of the ring is returned. The same if all the elements of the set *mcs* (maximal contained set) are among the ideal elements so far computed.

8.3.19 Subrings

`Subrings` (R)
`Subrings` returns the set of all subrings of the ring *R*.

```
gap> Subrings(r8);
[ RingfromGroup(Subgroup( Group( (1,2,3,4,5,6,7,8) ), [  ] ) ,
    function ( x, y )
        return ();
    end, RingfromGroup(Group( (1,2,3,4,5,6,7,8) ) ,
  function ( x, y ) return ();
    end, RingfromGroup(Subgroup( Group( (1,2,3,4,5,6,7,8) ),
    [ (1,5)(2,6)(3,7)(4,8), (1,3,5,7)(2,4,6,8) ] ) ,
function ( x, y ) return ();
    end, RingfromGroup(Subgroup( Group( (1,2,3,4,5,6,7,8) ),
    [ (1,5)(2,6)(3,7)(4,8) ] ) ,
function ( x, y ) return ();
    end ]
```

8.3.20 Ideals

`Ideals` (R)
`Ideals` returns the set of all ideals of the ring *R*.

```
gap> Ideals(r);
[ Ideal ( RingfromGroup(Group( (1,2,3,4,5,6,7,8) ) ,
function ( x, y ) return (); end , [] ),
  Ideal ( RingfromGroup(Group( (1,2,3,4,5,6,7,8) ) ,
function ( x, y ) return (); end , (1,2,3,4,5,6,7,8) ),
  Ideal ( RingfromGroup(Group( (1,2,3,4,5,6,7,8) ) ,
function ( x, y ) return (); end ,
(1,5)(2,6)(3,7)(4,8) , (1,3,5,7)(2,4,6,8) ),
  Ideal ( RingfromGroup(Group( (1,2,3,4,5,6,7,8) ) ,
function ( x, y ) return (); end ,
(1,5)(2,6)(3,7)(4,8) ) ]
```

8.3.21 LeftIdeals

`LeftIdeals` (R)
`LeftIdeals` returns the set of all left ideals of the ring *R*.

8.3.22 RightIdeals

`RightIdeals` (R)
`RightIdeals` returns the set of all right ideals of the ring R.

8.3.23 IsSimple

`IsSimple` (R)
`IsSimple` returns `true` if the ring R is simple and `false` otherwise.

```
gap> IsSimple(r);
false
```

8.4 Factor Rings

Given a ring R and an ideal $I \trianglelefteq R$ we might wish to compute the factor ring R/I of R modulo I. The following function does that.

8.4.1 FactorRing

R/I or
`FactorRing` (R , I)
`FactorRing` returns the factor ring of the ring R modulo the congruence $\sim_I$. **Note**, that in case the ring R was already a factor ring, the result is again a factor ring of the same ring in accordance with the results about factor^2–rings. The following example shows, how a factor ring is printed out in GAP.

```
gap> ri := r/i;
FactorRing ( Ring( [ [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), Z(2)^0 ] ],
[ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ) , Ideal ( Ring(
[ [ 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ Z(2)^0, 0*Z(2), Z(2)^0 ] ],
[ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ) ,
[ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ) )
gap> IsRing(ri);
```

```
true
gap> Size(ri);
16
gap> ri.generators;
[ FactorRingElement (
    [ [ 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ Z(2)^0, 0*Z(2), Z(2)^0 ] ] + Ideal ( Ring(
    [ [ 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ Z(2)^0, 0*Z(2), Z(2)^0 ] ],
    [ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ) ,
    [ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ) ),
  FactorRingElement ( [ [ Z(2)^0, 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), 0*Z(2) ]
     ] + Ideal ( Ring(
    [ [ 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ Z(2)^0, 0*Z(2), Z(2)^0 ] ],
    [ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ) ,
    [ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ) ) ]
gap> IsSimple(ri);
true
gap> IsMaximalIdeal(i);
true
```

8.4.2 IsFactorRing

`IsFactorRing` (F , R)
`IsFactorRing` returns `true` if F is defined as a factor ring of the ring R and `false` otherwise.

```
gap> IsFactorRing(ri,r);
true
```

Elements of a factor ring R/I can be handled with the following function

8.4.3 FactorRingElement

$x + I$ or
`FactorRingElement` (x , I)
`FactorRingElement` returns a representation for the element $x+I$ in R/I, where I must be an ideal of R and x and element of R. One can compute with these objects as usual, i.e. add and multiply them and check for equality.

8.5 Decomposition of Rings

Here one can choose from two different functions. The first computes a representation of a ring as a subdirect product of subdirectly irreducible rings. The second is a "sloppy" version of the first, as it computes a decomposition of the ring into not necessarily subdirectly irreducible factors. The second function is useful, when one studies large rings. In this case it is possible to get a "sloppy" solution within a few seconds, where the complete solution could last minutes or longer. In any case, the second function gives some insight, in particular if the ring is subdirectly irreducible, the two functions are equivalent.

8.5.1 SubdirectDecomposition

`SubdirectDecomposition` (R)[4]
`SubdirectDecomposition` returns a set of subdirectly irreducible rings R_i, s.t. R is isomorphic to a subdirect product of the R_i's and all R_i's are of the form R/I, where the I are ideals of R. If the ring R is subdirectly irreducible, the result is (as it should) the set containing only R. Since computing a decomposition of a ring can be very time consuming, the result of this function is stored in the record component `R.subdirectDecomposition` s.t. it has not to be computed twice. Evidently the dispatcher function tests, if this component is bound, before calling the function in the rings operations record.
Note, that possibly this function will not terminate or at least stop with interesting error-messages in the case of infinite rings. This problem can not be overcome.

```
gap> w := Ring([[0*Z(2),0*Z(2),Z(2)^0],[Z(2)^0,0*Z(2),0*Z(2)],
[Z(2)^0,0*Z(2),Z(2)^0]]); # we start with some ring
Ring( [ [ 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ Z(2)^0, 0*Z(2), Z(2)^0 ] ] )
gap> RingOperationTables(w); # this shall give some insight

 + |  0  1  2  3
---+------------
  0|  0  1  2  3
  1|  1  0  3  2
  2|  2  3  0  1
  3|  3  2  1  0

 * |  0  1  2  3
---+------------
  0|  0  0  0  0
  1|  0  3  1  2
```

[4]The two functions "RingOperationTables" and "Call" appearing in the following example are described in section 8.8

```
  2|  0  1  2  3
  3|  0  2  3  1
gap> w2 := w*w;;     # w2 is the direct product of w with itself
gap> Call(w2,"w2");  # this tells GAP to use w2 as an abbreviation
                     # whenever it is printed out
"w2"
gap> decw2 := SubdirectDecomposition(w2);
relax, this can take a while ...
-> Decomposition of w2              # the ring w2 is being decomposed
ideal-intersection has 12 elements   # we are looking for ideals the
ideal-intersection has 12 elements   # intersection of which contains
ideal-intersection has 4 elements    # one element only. these lines give
ideal-intersection has 1 elements    # information about the progress

3 rings left to decompose   # w2 is now decomposed into three subdirect
                            # factors, which can possibly be further
                            # decomposed
-> Decomposition of FactorRing ( w2 , Ideal ( w2 ,
[ [ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2) ] ],
  [ [ Z(2)^0, 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), Z(2)^0 ] ] ] ) )
-> Decomposition of FactorRing ( w2 , Ideal ( w2 ,
[ [ [ Z(2)^0, 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), Z(2)^0 ] ],
  [ [ 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ Z(2)^0, 0*Z(2), Z(2)^0 ] ] ] ) )
-> Decomposition of FactorRing ( w2 , Ideal ( w2 ,
[ [ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ Z(2)^0, 0*Z(2), 0*Z(2) ] ],
  [ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ] ) )
# obviously these 3 rings where irreducible.
------ d o n e -----------------------------------
[ FactorRing ( w2 , Ideal ( w2 ,
    [ [ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), 0*Z(2) ],
          [ 0*Z(2), 0*Z(2), 0*Z(2) ] ],
      [ [ Z(2)^0, 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), Z(2)^0 ],
          [ 0*Z(2), 0*Z(2), Z(2)^0 ] ] ] ) ),
  FactorRing ( w2 , Ideal ( w2 ,
    [ [ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
          [ Z(2)^0, 0*Z(2), 0*Z(2) ] ],
      [ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), 0*Z(2) ],
          [ 0*Z(2), 0*Z(2), 0*Z(2) ] ] ] ) ) ]
gap> RingOperationTables(decw2[1]);

 + |  0  1  2  3
---+------------
  0|  0  1  2  3
  1|  1  0  3  2
  2|  2  3  0  1
  3|  3  2  1  0

 * |  0  1  2  3
---+------------
  0|  0  0  0  0
  1|  0  3  1  2
  2|  0  1  2  3
  3|  0  2  3  1
gap> RingOperationTables(decw2[2]);

 + |  0  1  2  3
---+------------
  0|  0  1  2  3
  1|  1  0  3  2
  2|  2  3  0  1
  3|  3  2  1  0

 * |  0  1  2  3
---+------------
  0|  0  0  0  0
  1|  0  3  1  2
  2|  0  1  2  3
  3|  0  2  3  1
gap> decw2[1]=decw2[2]; # are the 2 factors equal ?
false
```

Clearly the decomposition of a direct product of irreducible rings yields the initial rings again, as we can see from the operation tables. Nonetheless all we get are isomorphic images of the initial rings. Since there is no function `IsIsomorphicRing` yet, we use `RingOperationTables` and our eyes.

8.5.2 PartialDecomposition

`PartialDecomposition` (R)
`PartialDecomposition` returns a set of (not necessarily subdirectly irreducible) rings R_i, s.t. R is isomorphic to a subdirect product of the R_i's.
`SubdirectDecomposition` yields a complete decomposition into subdirectly irreducible factors.

8.6 Radical Theory

As we have seen in chapter 3, radical theory for finite rings is simple. The following functions are available for finite rings only.

The following functions will not always work properly, i.e. terminate, with infinite rings. In this case a warning is printed and one can hope the best and wait.

8.6.1 IsNilpotentElement

`IsNilpotentElement` (R , x)
`IsNilpotentElement` returns `true` if x is a nilpotent element of the ring R.

```
gap> x:=[[0*Z(2),0*Z(2),0*Z(2)],[0*Z(2),0*Z(2),0*Z(2)],
[Z(2)^0,0*Z(2),0*Z(2)]];
[ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ Z(2)^0, 0*Z(2), 0*Z(2) ] ]
gap> x^2;
[ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2) ] ]
gap> IsNilpotentElement(r,x);
true
```

8.6.2 Nilpotents

`Nilpotents` (R)
`Nilpotents` returns the set of all nilpotent elements of the ring R.

```
gap> n:=Nilpotents(r);;
gap> Size(n);
16
gap> IsSubring(r,n);
false
```

8.6.3 IsNilIdeal

`IsNilIdeal` (I)
`IsNilIdeal` returns `true` if the ideal I is nil and `false` otherwise.

With the introducing remark the following two functions do exactly the same in the finite case.

8.6.4 Nilradical

`Nilradical` (R)
`Nilradical` computes the radical $\mathcal{N}(R) = \mathcal{J}(R)$.

8.6.5 JacobsonRadical

`JacobsonRadical` (R)
`JacobsonRadical` computes the radical $\mathcal{N}(R) = \mathcal{J}(R)$.

```
gap> JacobsonRadical(t);;
gap> Size(last);
64
gap> Size(JacobsonRadical(t).generators); # number of generators
3
```

8.6.6 IsJacobsonRadicalRing

`IsJacobsonRadicalRing` (R)
`IsJacobsonRadicalRing` returns `true` if $\mathcal{J}(R) = R$ and `false` otherwise.

```
gap> IsJacobsonRadicalRing(r);
true
```

8.6.7 IsSemisimple

`IsSemisimple` (R)
`IsSemisimple` returns `true` if the ring R is (Jacobson-)semisimple and `false` otherwise.

```
gap> IsSemisimple(r);
false
```

8.7 Polynomials

The functions in this section are supposed to be applied only to polynomial rings over fields and their elements. The main reason is that most of these functions try to normalize the polynomials which is not possible in all polynomial rings. Throughout this section the ordering for polynomials is lexicographic (see chapter 7).

8.7.1 Variables

`Variables` (R , names)
In GAP the (multivariate) polynomial ring $R[x_1, \ldots, x_n]$ is represented as the ring $(R[x_1, \ldots, x_{n-1}])[x_n]$, i.e. as a (univariate) polynomial ring over a polynomial ring. If we wish to compute with multivariate polynomials, we first have to define the variables. `Variables` returns a list of variables with the names specified in the list *names*.

```
gap> v := Variables(Rationals,["x","y"]);
[ x, y ]
gap> x:=v[1];
x
gap> y:=v[2];
y
gap> c := x^2+y^2-1;
y^2 + (x^2 - 1)
gap> l := x-y;
(-x^0)*y + (x)
gap> Qx := PolynomialRing(Rationals);
RationalsPolynomials
gap> Qxy := PolynomialRing(Qx);
PolynomialRing( RationalsPolynomials )
gap> c in Qx;
false
gap> c in Qxy;
true
```

We use two representations for power products: (1) as a polynomial and (2) as a list containing the exponents of the variables of the power product.

8.7.2 Lpp

`Lpp` (p)
The output of `Lpp` is a record with the components `lpp` and `lc`, where

`lpp` is the leading power product of p in representation (2) and
`lc` is the leading coefficient of p.

```
gap> c;
y^2 + (x^2 - 1)
gap> Lpp(c);
rec(
  lc := 1,
  lpp := [ 0, 2 ] )
```

In GAP the integer 1 is different from the polynomial $1 = x^0$. For technical reasons the next two functions require an argument "one", which for the polynomial ring $R[x_1, \ldots, x_n]$ has to be x_1^0. We designed the functions this way, because it seems easier to compute the element "one" once and use it several times, than to make the functions compute it each time they are applied.

8.7.3 ReducegModf

`ReducegModf` (g , f , one)
`ReducegModf` returns h, s.t. $g \rightarrow_f h$.

```
gap> one := x^0;
x^0
gap> ReducegModf(c,l,one);
(x)*y + (x^2 - 1)
```

8.7.4 NormalFormOfgModF

`NormalFormOfgModF` (g , F , one)
`NormalFormOfgModF` returns h, s.t. $g \rightarrow_F \underline{h}$.

```
gap> NormalFormOfgModF(c,[l],one);
2*x^2 - 1*y^0
```

8.7.5 GroebnerBasis

`GroebnerBasis` (F)
`GroebnerBasis` computes the reduced Groebner basis for the ideal I w.r.t. lexicographic ordering.

The following example shows how to compute the intersection between the circle $x^2 + y^2 - 1 = 0$ and the line $x - y = 0$.

```
gap> c := x^2+y^2-1;
y^2 + (x^2 - 1)
gap> l := x - y;
(-x^0)*y + (x)
gap> F := [c,l];
[ y^2 + (x^2 - 1), (-x^0)*y + (x) ]
gap> GroebnerBasis(F);
Computing critical pairs ...

# of critical pairs (S-polynomials): 1
Basis: [ y^2 + (x^2 - 1), (-x^0)*y + (x) ]

  Checking critical pair [ 1, 2 ] ...
Computing S-polynomial ...
y^2 + ((-1/2)*x)*y + ((1/2)*x^2 + (-1/2))->
      -x^2 + (1/2)*y^0
critical pairs: [ [ 1, 3 ], [ 2, 3 ] ]

# of critical pairs (S-polynomials): 2
Basis: [ y^2 + (x^2 - 1), (-x^0)*y + (x), x^2 + (-1/2)*y^0 ]

  Checking critical pair [ 2, 3 ] ...

# of critical pairs (S-polynomials): 1
Basis: [ y^2 + (x^2 - 1), (-x^0)*y + (x), x^2 + (-1/2)*y^0 ]

  Checking critical pair [ 1, 3 ] ...
Reducing [ y^2 + (x^2 - 1), (-x^0)*y + (x), x^2 + (-1/2)*y^0 ]
Reducing [ x^2 + (-1/2)*y^0, (-x^0)*y + (x) ]
[ x^2 + (-1/2)*y^0, (-x^0)*y + (x) ]
```

8.8 Miscellaneous

Besides there are some functions for rings, that may sometimes be interesting, when working with rings.

8.8.1 RingOperationTables

This function is evidently designed for finite rings only. Here finite means very small, because for large rings the output becomes unreadable.

`RingOperationTables` (R)
`RingOperationTables` returns the operation tables for addition and multiplication in the finite ring *R*. The elements of R are ordered in a GAP-internal way, the corresponding element of the ring of the element i of the operation table is `Elements(R)[i+1]`. Usually at least the additive neutral element has no. 1, but this need not necessarily be so.

```
gap> RingOperationTables(s);

 + |  0  1  2  3
---+------------
  0|  0  1  2  3
  1|  1  0  3  2
  2|  2  3  0  1
  3|  3  2  1  0

 * |  0  1  2  3
---+------------
  0|  0  0  0  0
  1|  0  0  0  0
  2|  0  0  2  2
  3|  0  0  2  2
```

8.8.2 Call

`Call` (S , "N")
If we wish to give a name to a ring, a group,. . . , which shall be used as abbreviation for this structure, GAP allows us to use the record component `name`. Clearly we do not wish to work with record components and struggle with data structures. `Call` does all the dirty work. **Note**, that `Call` works with any object if only it is a record. If not an error is reported, so try out (even with ring elements). Please note that elements which are already defined when you apply `Call` (as the elements of a ring are very often) are a copy of the element, which gets the name and therefore stay unchanged.

```
gap> r8;
RingfromGroup(Group( (1,2,3,4,5,6,7,8) ) , function ( x, y )
    return ();
end
```

```
gap> Call(z8,"Z8");
"Z8"
gap> r8;
RingfromGroup(Z8 , function ( x, y )
    return ();
end
gap> Call(r8,"Zero ring over Z8");
"Zero ring over Z8"
gap> r8;
Zero ring over Z8
```

8.9 Deja vu

The following functions for rings have already been implemented earlier. For a precise description see [20].

8.9.1 in

x `in` S
`in` returns `true` if the element x is an element of the ring or ideal S and `false` otherwise, even for the most infinite rings. If it is not (yet) possible to decide this relation, an error is reported.

8.9.2 Elements

`Elements` (S)
`Elements` (S) returns the set of all elements of the ring or ideal S, if it is finite.

8.9.3 Others

The following is a complete overview over all functions for rings so far implemented in GAP in alphabetical order. For details and instructive examples to these functions read the GAP manual [20].

- `Associates`(R,r)[5] returns the set of all elements of R associated with r,

[5]See IsAssociated.

- `DefaultRing`(gen) does essentially the same as `Ring`(gen) with some exceptions[6],
- `EuclideanDegree`(R,r) returns the Euclidean degree of r in the ring R, if R is a Euclidean ring and `false` otherwise,
- `EuclideanQuotient`(R,r,m) returns the Euclidean quotient of the ring elements r and m in the ring R,
- `EuclideanRemainder`(R,r,m) returns the Euclidean remainder of r modulo m in the Euclidean ring R,
- `Factors`(R,r) returns a factorization of r in the ring R,
- `Gcd`(R,r1,r2,…) returns the greatest common divisor of the ring elements r1, r2, ... in the Euclidean ring R,
- `GcdRepresentation`(R,r1,r2,…) returns a representation of the greatest common divisor as a linear combination of r1, r2, …,
- `IsAssociated`(R,r,s) returns `true` iff r and s are associated in the ring R (i.e. there exists a unit $u \in R : ru = s$) and `false` otherwise,
- `IsCommutativeRing` (R) returns `true` iff the ring R is commutative,
- `IsEuclideanRing` (R) returns `true` iff R is a Euclidean ring,
- `IsIntegralRing` (R) returns `true` iff R is an integral ring,
- `IsIrreducible`(R,r) returns `true` iff there is no nontrivial factorization of r in the ring R,
- `IsPrime`(R,r) returns `true` iff for all s and t if r divides st then r divides s or r divides t,
- `IsRing` (R) returns `true` iff R is a ring,
- `IsUniqueFactorizationRing` (R) returns `true` iff the ring R is a unique factorization ring,
- `IsUnit` (R,r) returns `true` iff r is a unit in the ring R (i.e. r has a multiplicative inverse in R),
- `Lcm`(R,r1,r2,…) returns the least common multiple of the ring elements r1, r2,... in the Euclidean ring R,
- `Mod`(R,r,m) is a synonym for `EuclideanRemainder` and is obsolete,
- `PolynomialRing` (R) is described above,
- `PowerMod`(R,r,e,m) returns the e-th power of the ring element r modulo the ring element m in the ring R. e must be an integer. R must be a Euclidean ring,
- `Quotient`(R,r,s) returns the quotient of the elements r and s in the ring R if it exists and `false` otherwise,

[6]See [20].

- `QuotientMod`(R,r,s,m) returns the quotient of the ring elements r and s modulo the ring element m in the Euclidean ring R,
- `QuotientRemainder`(R,r,m) returns the Euclidean quotient and the Euclidean remainder of the ring elements r and m in the ring R as pair of ring elements,
- `Ring` (gen) is described above,
- `StandardAssociate`(R,r) returns a distinguished among the set of associates of r,
- `Units` (R) returns the set of all units of the ring R,

Appendix A

Algebraic details

A.1 The Abstract Definition of a Ring

(see Allenby [1], p.84f)
For any $a, b, c \in R$:

1. $a + b = b + a$
2. $(a + b) + c = a + (b + c)$
3. $\exists z \in R \; \forall a \in R : z + a = a + z = a$
4. $\forall a \in R \; \exists a^{-} \in R : a + a^{-} = a^{-} + a = z$
5. $(ab)c = a(bc)$
6. $a(b + c) = ab + ac$ and $(a + b)c = ac + bc$
7. $ab = ba$
8. $\exists e \in R \; \forall a \in R : ea = ae = a$
9. $\forall a \in R \setminus \{z\} \; \exists a' \in R : aa' = a'a = e$
10. $ab = z \Longrightarrow a = z \vee b = z$

Every ring satisfies axioms 1-6. The other axioms describe special classes of rings.

name	7	8	9	10
ring				
commutative ring	⋆			
ring with identity		⋆		
ring with no zero divisors				⋆
commutative ring with unity	⋆	⋆		
commutative ring with no zero divisors	⋆			⋆
ring with unity and no zero divisors		⋆		⋆
integral domain	⋆	⋆		⋆
division ring		⋆	⋆	⋆
field	⋆	⋆	⋆	⋆

A.2 Ordering

Order is heaven's first law.
Pope Alexander (1688–1744)

Definition 54 *Let S be a set and $\leq$ be a subset of the cartesian product $S \times S$. $\leq$ is called a* partial ordering *on S iff the following conditions hold for any $a, b, c \in S$:*

- $a \leq a$ *(*reflexivity*)*
- $a \leq b \wedge b \leq c \Longrightarrow a \leq c$ *(*transitivity*)*
- $a \leq b \wedge b \leq a \Longrightarrow a = b$ *(*antisymmetry*)*

If for any $a, b \in S$ either $a \leq b$ or $b \leq a$, we call $\leq$ linear *or* total*. A totally ordered subset of S we call a* chain *in S.*

Definition 55 *Let S be a set, partially ordered by $\leq$, $T \subseteq S$. $u \in S$ is called an* upper bound *of T in S, if $t \leq u$ for all $t \in T$.*

Note, that such an upper bound need not exist in general.

A.3 Zorn's Lemma

How can finite grasp infinity?
John Dryden (1631–1700)

Theorem 44 *Let S be a set, $\leq$ a partial ordering on S. If for every chain in S there exists an upper bound in S, then S contains at least one maximal element.*

A.4 Group

Definition 56 *A* semigroup *S is an algebra of type $(1, 0, \emptyset)$, s.t. for all $a, b, c \in S$*

$$(ab)c = a(bc)$$

(The binary operation is denoted by concatenation of the elements.)

Definition 57 *A* monoid *M is a semigroup containing an element $e \in S$, s.t. for all $a \in M$*

$$ae = ea = a$$

Definition 58 *A* group *G is a monoid containing with every $g \in G$ an element $g^- \in G$, s.t.*

$$gg^- = g^-g = e$$

G is called abelian *iff for all $a, b \in G$*

$$ab = ba$$

Appendix B

Source Code

To make the functions for rings described above available, type `Read("ringplus")` at the beginning of your GAP-session. This will read all the files needed.

B.1 The Source File `ringplus`

```
################################################################################
##                                                                            ##
##                      some useful functions for rings                       ##
##                                                                            ##
##                                 27.9.1995                                  ##
##                                                                            ##
##                               Ecker Juergen                                ##
##                                                                            ##
##                    Johannes Kepler Universitaet Linz                       ##
##                                                                            ##
################################################################################

# file: <ringplus>

Read("dispatch.l");
Print("Reading ...\n");
Print("Polynomials\n");
Read("poly.l");
Print("Factorrings\n");
Read("factor.l");
Print("Subdirect Decomposition\n");
Read("subdec.l");
```

```
Print("Radical Theory\n");
Read("radical.l");
Print("Miscellaneous\n");
Read("misc.l");
Print("Subrings and Ideals\n");
Read("ideals.l");
```

B.2 The Source File `dispatch.l`

```
####### Dispatcher functions ##################################################
#
# file: <dispatch.l>

###############################################################################
##
#F  Subring( <R>, <gen> ) ... generate subring of R generated by elms of gen
##
##  Dispatcher function for computing subrings
##

Subring := function ( R , L )
  if not IsRing(R) then
        Error ("first argument has to be a ring");
  elif not IsList(L) then
        Error ("second argument has to be a list");
  elif IsBound(R.operations.Subring) then
        return R.operations.Subring(R,L);
  fi;
end;

###############################################################################
##
#F  IsSubring ( <R>, <S> ) ... check if S is a subring of R
##
##  Dispatcher function

IsSubring := function ( R , S )
  if not IsRing (R) then
        Error ("first argument has to be a ring");
  elif not IsRing (S) then
        return ForAll(S, s-> ForAll(S, t-> s-t in S and s*t in S));
  fi;
  if IsBound (S.parents) then                   # record component exists
        if ( R in S.parents ) then              # component holds the ring
                return ( true );
        elif S.operations.IsSubset(R,S) then  # does not, but should
                AddSet(S.parents,R);
                return (true);
        fi;
  else                                          # record comp. does not exist
        if S.operations.IsSubset (R,S) then
                S.parents := Set([R]);          # create component
                return (true);
        fi;
  fi;
end;

################################################################################
##
#F  LeftIdeal( <R>, <gen> ) ... compute left ideal generated by the elms of gen
##
##  Dispatcher function for computing left ideals
##

LeftIdeal := function ( R, gen )
  return R.operations.LeftIdeal (R,gen);
end;

################################################################################
##
#F  RightIdeal( <R>, <gen> ) ... compute right ideal gen. by the elms of gen
##
##  Dispatcher function for computing left ideals
##

RightIdeal := function ( R, gen )
  return R.operations.RightIdeal (R,gen);
end;

################################################################################
##
#F  Ideal( <R>, <gen> ) ... compute ideal generated by the elms of gen
##
##  Dispatcher function for computing ideals
##

Ideal := function ( R, gen )
  return R.operations.Ideal (R,gen);
end;

################################################################################
##
#F  IsLeftIdeal ( <R>, <S> ) ... check if S is a left ideal in R
##
##  Dispatcher function

IsLeftIdeal := function ( R , S )
```

```
  if not IsSubring(R,S) then
        return false;
  fi;
  # the subring is allready known to be an ideal
  if ( IsBound(S.isLeftIdeal) and (R=S.isLeftIdeal) ) then
        return (true);
  elif not IsBound(R.operations.IsLeftIdeal) then
        Error("can not check, missing function in operations record of <R>");
  else
        if R.operations.IsLeftIdeal(R,S) then
                S.isLeftIdeal := R;
                return true;
        else
                return false;
        fi;
  fi;
end;

################################################################################
##
#F  IsRightIdeal ( <R>, <S> ) ... check if S is a right ideal in R
##
##  Dispatcher function

IsRightIdeal := function ( R , S )
  if not IsSubring(R,S) then
        return false;
  fi;
  # the subring is allready known to be an ideal
  if ( IsBound(S.isRightIdeal) and (R=S.isRightIdeal) ) then
        return (true);
  elif not IsBound(R.operations.IsRightIdeal) then
        Error("can not check, missing function in operations record of <R>");
  else
        if R.operations.IsRightIdeal(R,S) then
                S.isRightIdeal := R;
                return true;
        else
                return false;
        fi;
  fi;
end;

################################################################################
##
#F  IsIdeal ( <R>, <S> ) ... check if S is an ideal in R
##
## Dispatcher function

IsIdeal := function ( R , S )
```

```
  if not IsSubring(R,S) then
        return false;
  fi;
  # the subring is allready known to be an ideal
  if ( IsBound(S.isIdeal) and (R=S.isIdeal) ) then
        return (true);
  elif not IsBound(R.operations.IsIdeal) then
        Error("can not check, missing function in operations record of <R>");
  else
        if R.operations.IsIdeal(R,S) then
                S.isIdeal := R;
                return true;
        else
                return false;
        fi;
  fi;
end;

################################################################################
##
#F  FactorRing ( <R>, <I> ) ... computes the factor ring of R modulo I
##
##  Dispatcher function for factor rings
##

FactorRing := function ( R , I )
  if not IsRing (R) or not IsRing (I) then
        Error ("usage: FactorRing ( ring , ideal ) or ring / ideal");
  elif not IsIdeal(R,I) then
        Error ("sorry, <I> is not an ideal in <R>");
  fi;

  return R.operations.FactorRing(R,I);
end;

################################################################################
##
#F  SubdirectDecomposition( <R> ) ... computes a subdirect decomposition of R
##
##  Dispatcher function

SubdirectDecomposition := function ( R )
  if not IsRing (R) then
        Error ("sorry, <R> is not a ring");
  fi;
  if not IsBound(R.subdirectDecomposition) then
        R.subdirectDecomposition := R.operations.SubdirectDecomposition(R);
  fi;

  return R.subdirectDecomposition;
```

```
end;

################################################################################
##
#F  PartialDecomposition( <R> ) ... computes a subdirect decomposition of R
##
##  Dispatcher function

PartialDecomposition := function ( R )
  if not IsRing (R) then
        Error ("sorry, <R> is not a ring");
  fi;
  return R.operations.PartialDecomposition(R);
end;

################################################################################
##
#F  IsNilpotentElement ( <R>, <x> ) ... checks if x is nilpotent
##
##  Dispatcher function
##

IsNilpotentElement := function ( R , x )
  if not IsFinite(R) then
        Print("Warning, <R> is infinite, computation may not terminate\n");
  fi;
  if IsBound(R.nilpotents) then
        return x in R.nilpotents;
  else
        return R.operations.IsNilpotentElement(R,x);
  fi;

end;

################################################################################
##
#F  IsNilIdeal ( <I> ) ... checks if I is nil
##
##  Dispatcher function
##

IsNilIdeal := function ( I )
  if not( IsBound(I.isIdeal) or IsBound(I.isLeftIdeal)
     or IsBound(I.isRightIdeal) ) then
        Error("<I> is not an ideal");
  fi;
  if not IsBound(I.isNilIdeal) then
        I.isNilIdeal := I.operations.IsNilIdeal(I);
  fi;
  return I.isNilIdeal;
end;

#################################################################################
##
#F  Nilradical( <R> ) ... compute the nil radical of R
##
##  Dispatcher function

Nilradical := function ( R )
  if not IsRing(R) then
        Error("usage: Nilradical ( ring )");
  fi;
  if not IsBound(R.nilradical) then
        R.nilradical := R.operations.Nilradical(R);
  fi;

  return R.nilradical;
end;

#################################################################################
##
#F  Nilpotents ( <R> ) ... compute set of nilpotent elements of R
##
##  Dispatcher function
##

Nilpotents := function ( R )
  if not IsRing(R) then
        Error("<R> is not a ring");
  fi;
  if not IsBound(R.nilpotents) then
        R.nilpotents := R.operations.Nilpotents(R);
  fi;

  return R.nilpotents;
end;

#################################################################################
##
#F  JacobsonRadical( <R> ) ... compute the Jacobson radical of R
##
##  Dispatcher function

JacobsonRadical := function ( R )
  if not IsRing(R) then
        Error("usage: JacobsonRadical ( ring )");
  fi;
  if not IsBound(R.jacobsonRadical) then
        R.jacobsonRadical := R.operations.JacobsonRadical(R);
  fi;

  return R.jacobsonRadical;
end;

################################################################################
##
#F  IsRadicalRing ( <R> ) ... checks if the Jacobson radical is the whole ring
##
## Dispatcher function
##

IsRadicalRing := function ( R )
  if not IsRing(R) then
        Error("<R> is not a ring");
  fi;
  if not IsBound(R.isRadicalRing) then
        R.isRadicalRing := R.operations.IsRadicalRing(R);
  fi;

  return R.isRadicalRing;
end;

################################################################################
##
#F  Rad ( <I> ) ... compute the radical of the ideal I
##
##  Dispatcher function
##

Rad := function ( I )
  if not(IsBound(I.isIdeal)) then
        Error("<I> is supposed to be an ideal");
  fi;
  if not IsBound(I.rad) then
        I.rad := I.operations.Rad(I);
  fi;

  return I.rad;
end;

################################################################################
##
#F  IsSimple ( <R> ) ... checks if R is simple
##
##  Dispatcher function
##

IsSimple := function ( R )
```

```
  if not IsRing (R) then
        Error("usage: IsSimple ( ring )");
  fi;
  if not IsBound(R.isSimple) then
        R.isSimple := R.operations.IsSimple(R);
  fi;

  return R.isSimple;
end;

################################################################################
##
#F  IsSemisimple ( <R> ) ... check if R is semisimple
##
##  Dispatcher function
##

IsSemisimple := function (R)
  if not IsRing (R) then
        Error("usage: IsSemisimple ( ring )");
  fi;

  if not IsBound(R.isSemisimple) then
        R.isSemisimple := R.operations.IsSemisimple(R);
  fi;

  return R.isSemisimple;
end;

################################################################################
##
#F  IsModularLeftIdeal ( <L> ) ... check if L is a modular left ideal
##
##  Dispatcher function
##

IsModularLeftIdeal := function ( L )
  if not IsBound(L.isLeftIdeal) then
        Error("<L> is not a left ideal");
  fi;
  if not IsBound(L.isModularLeftIdeal) then
        L.isModularLeftIdeal := L.operations.IsModularLeftIdeal(L);
  fi;

  return L.isModularLeftIdeal;
end;

################################################################################
##
#F  IsMaximalLeftIdeal ( <I> ) ... checks if I is a maximal left ideal
```

```
##
##  Dispatcher function
##

IsMaximalLeftIdeal := function ( I )
  if not IsBound(I.isLeftIdeal) then
        Error("<I> is not a left ideal");
  fi;
  if not IsBound(I.isMaximalLeftIdeal) then
        I.isMaximalLeftIdeal := I.operations.IsMaximalLeftIdeal(I);
  fi;

  return I.isMaximalLeftIdeal;
end;

################################################################################
##
#F  IsMaximalRightIdeal ( <I> ) ... checks if I is a maximal right ideal
##
##  Dispatcher function
##

IsMaximalRightIdeal := function ( I )
  if not IsBound(I.isRightIdeal) then
        Error("<I> is not a right ideal");
  fi;
  if not IsBound(I.isMaximalRightIdeal) then
        I.isMaximalRightIdeal := I.operations.IsMaximalRightIdeal(I);
  fi;

  return I.isMaximalRightIdeal;
end;

################################################################################
##
#F  IsMaximalIdeal ( <I> ) ... checks if I is a maximal ideal
##
##  Dispatcher function
##

IsMaximalIdeal := function ( I )
  if not IsBound(I.isIdeal) then
        Error("<I> is not an ideal");
  fi;
  if not IsBound(I.isMaximalIdeal) then
        I.isMaximalIdeal := I.operations.IsMaximalIdeal(I);
  fi;

  return I.isMaximalIdeal;
end;
```

```
#############################################################################
##
#F  IsPrimeIdeal ( <I> ) ... check if I is prime
##
##  Dispatcher function
##

IsPrimeIdeal := function ( I )
  if not IsBound(I.isIdeal) then
        Error("<I> is not an ideal");
  fi;
  if not IsBound(I.isPrimeIdeal) then
        I.isPrimeIdeal := I.operations.IsPrimeIdeal(I);
  fi;

  return I.isPrimeIdeal;
end;

#############################################################################
##
#F  Subrings ( <R> ) ... compute a list of all subrings
##
##  Dispatcher function
##

Subrings := function ( R )
  if not IsRing(R) then
        Error("<R> is not a ring");
  fi;
  if not IsBound(R.subrings) then
        R.subrings := R.operations.Subrings(R);
  fi;

  return R.subrings;
end;

#############################################################################
##
#F  Ideals ( <R> ) ... compute a list of all ideals
##
##  Dispatcher function
##

Ideals := function ( R )
  if not IsRing(R) then
        Error("<R> is not a ring");
  fi;
  if not IsBound(R.ideals) then
        R.ideals := R.operations.Ideals(R);
  fi;

  return R.ideals;
end;

#############################################################################
##
#F  LeftIdeals ( <R> ) ... compute a list of all left ideals
##
##  Dispatcher function
##

LeftIdeals := function ( R )
  if not IsRing(R) then
        Error("<R> is not a ring");
  fi;
  if not IsBound(R.leftIdeals) then
        R.leftIdeals := R.operations.LeftIdeals(R);
  fi;

  return R.leftIdeals;
end;

#############################################################################
##
#F  RightIdeals ( <R> ) ... compute a list of all right ideals
##
##  Dispatcher function
##

RightIdeals := function ( R )
  if not IsRing(R) then
        Error("<R> is not a ring");
  fi;
  if not IsBound(R.rightIdeals) then
        R.rightIdeals := R.operations.RightIdeals(R);
  fi;

  return R.rightIdeals;
end;

#############################################################################
##
#F  Subgroups ( <G> ) ... compute a list of all subgroups
##
##  Dispatcher function
##

Subgroups := function ( G )
  if not IsGroup(G) then
```

```
        Error("<G> is not a group");
  fi;
  if not IsBound(G.subgroups) then
        G.subgroups := G.operations.Subgroups(G);
  fi;

  return G.subgroups;
end;

################################################################################
##
#F  AsRing ( <G> , <mult> ) ... make a ring out of a group
##                              and a multiplication function
##
##  Dispatcher function
##

AsRing := function ( G , mult )
  if not IsGroup(G) then
        Error("<G> is not a group");
  fi;

  return G.operations.AsRing(G,mult);

end;

################################################################################
##
#F  AsGroupElement ( <x> ) ... make a group element out of a
##                             ring element

AsGroupElement := function ( x )
  if not IsRec(x) or not IsBound(x.value) then
        Error("sorry, this is not possible");
  fi;

  return x.value;
end;

################################################################################
##
#F  RingElement ( <x> , <fct> )      makes a ring element out of
##                               a group element

RingElement := function ( x , fct )
  local elm;            # the element

  elm := rec(
        value := x,
        operations := rec(
                \+ := function ( x , y )
                        local z;
                        z := Copy(x);
                        z.value := x.value * y.value;
                        return z;
                end,

                \- := function ( x , y )
                        local z;
                        z := Copy(x);
                        z.value := x.value * y.value^-1;
                        return z;
                end,

                \* := function ( x , y )
                        local z;
                        z := Copy(x);
                        z.value := fct(x.value,y.value);
                        return z;
                end,
                \< := function ( x , y )
                        if IsBound(x.value) and IsBound(y.value) then
                                return ( x.value < y.value );
                        else
                                return IsBound(y.value);
                        fi;
                end,
                \= := function ( x , y )
                        if IsBound(x.value) and IsBound(y.value) then
                                return ( x.value = y.value );
                        else
                                return false;
                        fi;
                end,
                Print := function ( el )
                        Print (el.value);
                end
        )
  );
  return elm;
end;

################################################################################
##
#F  AsRingElement ( <x> , <R> ) ... make a ring element out of a
##                                      group element

AsRingElement := function ( x , R )
  if not IsRing(R) or not IsBound(R.group) then
        Error("usage: AsRingElement( element , ring )");
  elif not (x in R.group) then
        Error("sorry, <x> is not in the group corresponding to <R>");
  fi;

  return RingElement(x,R.mult);
end;

################################################################################
##
#F  Call ( <struct> , <name> ) ... give a name to a structure
##

Call := function ( struct , name )
  if not IsRec(struct) then
        Error("sorry, <struct> has to be a record");
  fi;

  struct.name := name;
  struct.operations.Print := function(st) Print(st.name); end;

  return name;
end;
```

B.3 The Source File `factor.l`

```
######## F a c t o r r i n g s ################################################
#
# file: <factor.l>

################################################################################
#
# basic operations for factor ring elements

### Operations for factor ring elements ###
#

FactorRingElOps := rec (

  name := "FactorRingElOps",

  \* := function (a,b)
        local c;
        if a.ideal = b.ideal then
                c := Copy (a);
                c.element := a.element * b.element;
                return c;
        else
```

```
                return ( false );
        fi;
  end,

  \+ := function (a,b)
        local c;
        if a.ideal = b.ideal then
                c := Copy (a);
                c.element := a.element + b.element;
                return c;
        else
                return ( false );
        fi;
  end,

  \- := function (a,b)
        local c;
        if a.ideal = b.ideal then
                c := Copy (a);
                c.element := a.element - b.element;
                return c;
        else
                return ( false );
        fi;
  end,

  \= := function (a,b)
        if a.ideal = b.ideal then
                return ( (a.element - b.element) in Elements(a.ideal) );
        else
                return ( false );
        fi;
  end,

  Print := function (a)
        Print ("FactorRingElement ( ",a.element," + ",a.ideal," )");
  end
  );

################################################################################
#
# FactorRingElement ( x , I )   generates the record representating
#                               the coset x + I

FactorRingElement := function ( x , I )

  local fariel;

  if not (IsBound(I.isIdeal)) then
        Error("<I> is not an ideal.");
  elif not ( x in I.isIdeal ) then
        Error("<x> is not in the parent ring.");
  fi;

  fariel := rec (
        element := x,
        ideal := I,
        operations := FactorRingElOps
  );

  return fariel;
end;

#############################################################################
##
#F  RingOps./ ( R , I ) ... generates the factor ring of the ring
##                              R modulo the ideal I

RingOps.\/ := function ( R , I )
  return FactorRing(R,I);
end;

#############################################################################
##
#F  RingOps.FactorRing ( R , I ) ... generates the factor ring of the
##                                      ring R modulo the ideal I

RingOps.FactorRing := function ( R , I )
  local fac,    # the factor ring
        i,      # counter
        gens;   # set of generators

  if IsBound(R.isFactorRing) then
        gens := List(I.generators,x->x.element);
        gens := Union(gens,R.ideal.generators);
        return FactorRing (R.ring,Ideal(R.ring,gens));
  else
        fac := rec (
                isDomain := true,
                isRing := true,
                isFactorRing := true,
                operations := Copy (RingOps),
                zero := FactorRingElement (R.zero,I),
                ideal := I,
                ring := R
        );
        fac.operations.name := "FactorRingOps";

        # generate factor ring  generators
        i := 1;
        fac.generators := [];
        while i<=Length(R.generators) do
                Add ( fac.generators , FactorRingElement(R.generators[i],I) );
                i := i+1;
        od;

        # modify print function for factor rings
        fac.operations.Print := function ( F )
                Print ("FactorRing ( ",F.ring," , ",F.ideal," )");
        end;

        return fac;
  fi;
end;

################################################################################
##
#F  IsFactorRing ( R )    checks whether R is a factor ring of some ring

IsFactorRing := function ( R )

  return IsBound(R.isFactorRing) and R.isFactorRing ;

end;
```

B.4 The Source File subdec.l

```
####### S u b d i r e c t  D e c o m p o s i t i o n s #######################
#
# file: <subdec.l>

################################################################################
##
#F  RingOps.PartialDecomposition ( R ) ... returns a list of subdirect factors
##                                        of R which are possibly not subdirectly
##                                        irreducible

RingOps.PartialDecomposition := function ( R )
  local F,              # list of factors
        T,              # set to choose ideal generators from
        a,              # ideal generator
        K,              # ideal
        Intsctn,        # intersection of all ideals
        trivgen;        # set of trivial generators (generating the whole ring)

  F := [];
  trivgen := [];
  Intsctn := Copy(Elements(R));
  T := Difference(Intsctn,[R.zero]);
  Print("-> Decomposition of ",R,"\n");

  # find principle ideals with trivial intersection
  while Intsctn <> [R.zero] and T <> [] do
        # generate nontrivial principle ideal
        repeat
                a := Random (T);
                K := Ideal (R,[a]);
                RemoveSet (T,a);
                K.elements:=K.operations.QuickIdealElements(K,trivgen,Intsctn);
                K.size := Size(K.elements);
                AddSet(trivgen,a);
        until Size(K) < Size(R) or T=[];
        # compute intersection of all ideals
        if T<>[] then
                IntersectSet (Intsctn,K.elements);
                AddSet (F,FactorRing(R,K));
                Print("ideal-intersection has ",Size(Intsctn)," elements\n");
        fi;
  od;
  # R is irreducible ?
  if Intsctn <> [R.zero] then
        return [R];
  else
        return F;
  fi;
end;

#############################################################################
##
#F  RingOps.SubdirectDecomposition ( R ) ... returns a list of subdirectly
##                                     irreducible subdirect factors of R

RingOps.SubdirectDecomposition := function ( R )
  local SetDecomposition,       # function
        Decomp,                 # decomposition of R
        RedIrred;               # [1] set of possibly reducible factors
                                # [2] set of irreducible factors

  if not IsRing (R) then
        Error ("sorry, <R> is not a ring");
  fi;

  ###################################
  #
  # given a list of rings SetDecomposition returns the set of subdirectly
  # irreducible rings therein and a set of decompositions of the others
  #
  SetDecomposition := function ( L )
        local   PartDec,        # set of possibly reducible factors
                F,              # runs through L
                Irred,          # set of irreducible factors
                Red;            # set of possibly reducible factors

          Irred := [];
          Red := [];

          for F in L do
                PartDec := PartialDecomposition (F);
                if PartDec=[F] then
                        AddSet (Irred,F);
                else
                        UniteSet (Red,PartDec);
                fi;
          od;

        return [Red,Irred];
  end;

  # start with R
  Print("relax, this can take a while ...\n");
  RedIrred := SetDecomposition ([R]);
  Decomp := RedIrred[2];
  Print (Size(Decomp),"\n");

  # decompose reducible factors
  while RedIrred[1]<>[] do
        Print (Size(RedIrred[2]),"rings left to decompose\n");
        RedIrred := SetDecomposition(RedIrred[1]);
        UniteSet (Decomp,RedIrred[2]);
  od;

  Print("------ d o n e -----------------------------------\n");
  return Decomp;
end;
```

B.5 The Source File `radical.l`

```
######  Radical Theory ########################################################
#
# file: <radical.l>

###############################################################################
##
```

```
#F  RingOps.IsNilpotentElement ( <R> , <x> ) ... checks whether x is a
##                               nilpotent element of the ring R

RingOps.IsNilpotentElement := function ( R , x )
  local xi,            # x^i
        powers;        # set of all x^j, j<=i

  xi := x;
  powers := [];
  while not xi in powers and xi<>R.zero do
        AddSet(powers,xi);
        xi := xi * x;
  od;
  return (xi=R.zero);

end;

################################################################################
##
#V  IdealOps

IdealOps := Copy(RingOps);
IdealOps.name := "IdealOps";

# function that computes all elements of an ideal
IdealOps.IdealElements := function ( S )
  local elms, set, elm, new, rgen, R, i, j, ISize, RSize, small, LRgen, one;

  if IsBound(S.isFinite) and not S.isFinite then
        Error ("<S> must be finite to compute the elements");
  elif not IsBound(S.generators) then
        Error ("sorry, generators of <S> are not known");
  fi;

  R := S.isIdeal;
  RSize := Size(R);
  LRgen := Length(R.generators);

  if not IsBound(R.one) then
        elms := Copy(S.generators);
        Add (elms,S.zero);
        set := Set(elms);
        ISize := Size(set);
        i := 1;
        small := true;
        while i <= Length(elms) and small do
                elm := elms[i];
                i := i+1;
                j := 1;
                while j <= LRgen and small do
```

```
                        rgen := R.generators[j];
                        j := j+1;
                        new := rgen * elm;
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        new := elm * rgen;
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        small := 2*ISize <= RSize;
                od;
                j := 1;
                while j < i and small do
                        new := elm + elms[j];
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        small := 2*ISize <= RSize;
                        j := j+1;
                od;
        od;
        if not small then
                set := Elements(R);
        fi;
else
        one := R.one;
        elms := Copy(S.generators);
        Add (elms,S.zero);
        set := Set(elms);
        ISize := Size(set);
        i := 1;
        small := true;
        while i <= Length(elms) and small do
                elm := elms[i];
                i := i+1;
                j := 1;
                while j <= LRgen and small do
                        rgen := R.generators[j];
                        j := j+1;
                        new := rgen * elm;
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        small := new <> one;
                        new := elm * rgen;
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        small := small and (new <> one) and (2*ISize <= RSize);
                od;
                j := 1;
                while j < i and small do
                        new := elm + elms[j];
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        small := (new <> one) and (2*ISize <= RSize);
                        j := j+1;
                od;
        od;
        if not small then
                set := Elements(R);
        fi;
  fi;
  return set;
end;

IdealOps.PrintIdeal := function ( I )
  local i;
  if IsBound (I.name) then
        Print (I.name);
  else
        Print ("Ideal ( ",I.isIdeal," , ");
        if I.generators = [] then
                Print("[] )");
        else
                for i in [1 .. Length(I.generators) - 1] do
                        Print (I.generators[i]," , ");
                od;
                Print (I.generators[Length(I.generators)]," )");
        fi;
  fi;
end;

IdealOps.SumOfIdeals := function ( x , I )

  if not IsBound (I.isIdeal) then
```

```
        Error("usage: x + I     or: I1 + I2");
  fi;

  if not IsRec(x) or not IsBound(x.isIdeal) then
        return FactorRingElement(x,I);
  fi;

  if x.isIdeal <> I.isIdeal then
        Error("sorry, these are not ideals of a common ring");
  fi;

  return Ideal(x.isIdeal,Union(x.generators,I.generators));
end;

IdealOps.ProductOfIdeals := function ( I1 , I2 )

  if I1.isIdeal <> I2.isIdeal then
        Error("sorry, these are not ideals of a common ring");
  fi;

  return Ideal(I1.isIdeal,Set(List(Cartesian(I1.generators,I2.generators),
                        function(x,y) return x*y; end)));
end;

IdealOps.SumOfLeftIdeals := function ( I1 , I2 )

  if I1.isLeftIdeal <> I2.isLeftIdeal then
        Error("sorry, these are not ideals of a common ring");
  fi;

  return LeftIdeal(I1.isLeftIdeal,Union(I1.generators,I2.generators));
end;

IdealOps.ProductOfLeftIdeals := function ( I1 , I2 )

  if I1.isLeftIdeal <> I2.isLeftIdeal then
        Error("sorry, these are not ideals of a common ring");
  fi;

  return LeftIdeal(I1.isLeftIdeal,
                Set(List(Cartesian(I1.generators,I2.generators),
                        function(x,y) return x*y; end)));
end;

IdealOps.SumOfRightIdeals := function ( I1 , I2 )

  if I1.isRightIdeal <> I2.isRightIdeal then
        Error("sorry, these are not ideals of a common ring");
  fi;
  return RightIdeal(I1.isRightIdeal,Union(I1.generators,I2.generators));
end;

IdealOps.ProductOfRightIdeals := function ( I1 , I2 )

  if I1.isRightIdeal <> I2.isRightIdeal then
        Error("sorry, these are not ideals of a common ring");
  fi;

  return RightIdeal(I1.isRightIdeal,
                Set(List(Cartesian(I1.generators,I2.generators),
                        function(x,y) return x*y; end)));
end;

# function that computes all elements of a left ideal
IdealOps.LeftIdealElements := function ( S )
  local elms, set, elm, new, rgen, R, i, j, RSize, ISize, small, LRgen, one;

  if IsBound(S.isFinite) and not S.isFinite then
        Error ("<S> must be finite to compute the elements");
  elif not IsBound(S.generators) then
        Error ("sorry, generators of <S> are not known");
  fi;

  R := S.isLeftIdeal;
  RSize := Size(R);
  LRgen := Length(R.generators);

  if not IsBound(R.one) then
        elms := Copy(S.generators);
        Add (elms,S.zero);
        set := Set(elms);
        ISize := Size(set);
        i := 1;
        small := true;
        while i <= Length(elms) and small do
                elm := elms[i];
                i := i+1;
                j := 1;
                while j <= LRgen and small do
                        rgen := R.generators[j];
                        j := j+1;
                        new := rgen * elm;
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        small := 2*ISize <= RSize;
                od;
                j := 1;
                while j < i and small do
                        new := elm + elms[j];
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        new := elm * elms[j];
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        small := 2*ISize <= RSize;
                        j := j+1;
                od;
        od;
        if not small then
                set := Elements(R);
        fi;
  else
        one := R.one;
        elms := Copy(S.generators);
        Add (elms,S.zero);
        set := Set(elms);
        ISize := Size(set);
        i := 1;
        small := true;
        while i <= Length(elms) and small do
                elm := elms[i];
                i := i+1;
                j := 1;
                while j <= LRgen and small do
                        rgen := R.generators[j];
                        j := j+1;
                        new := rgen * elm;
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        small := (new <> one) and (2*ISize <= RSize);
                od;
                j := 1;
                while j < i and small do
                        new := elm + elms[j];
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        small := new <> one;
                        new := elm * elms[j];
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        small := small and (new <> one) and (2*ISize <= RSize);
                        j := j+1;
                od;
        od;
        if not small then
                set := Elements(R);
        fi;
  fi;
  return set;
end;

IdealOps.PrintLeftIdeal := function ( I )
  local i;
  if IsBound (I.name) then
        Print(I.name);
  else
        Print ("LeftIdeal ( ",I.isLeftIdeal," , ");
        for i in [1 .. Length(I.generators) - 1] do
                Print (I.generators[i]," , ");
        od;
        Print (I.generators[Length(I.generators)]," )");
  fi;
end;

# function that computes all elements of a right ideal
IdealOps.RightIdealElements := function ( S )
  local elms, set, elm, new, rgen, R, i, j, RSize, ISize, small, LRgen, one;

  if IsBound(S.isFinite) and not S.isFinite then
        Error ("<S> must be finite to compute the elements");
  elif not IsBound(S.generators) then
        Error ("sorry, generators of <S> are not known");
  fi;

  R := S.isRightIdeal;
  RSize := Size(R);
  LRgen := Length(R.generators);

  if not IsBound(R.one) then
        elms := Copy(S.generators);
        Add (elms,S.zero);
        set := Set(elms);
        ISize := Size(set);
        i := 1;
        small := true;
        while i <= Length(elms) and small do
                elm := elms[i];
                i := i+1;
                j := 1;
                while j <= LRgen and small do
                        rgen := R.generators[j];
                        j := j+1;
                        new := elm * rgen;
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        small := 2*ISize <= RSize;
                od;
                j := 1;
                while j < i and small do
                        new := elm + elms[j];
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        new := elms[j] * elm;
                        if not new in set then
                                Add(elms,new);
                                AddSet(set,new);
                                ISize := ISize + 1;
                        fi;
                        small := 2*ISize <= RSize;
                        j := j+1;
                od;
        od;
        if not small then
                set := Elements(R);
        fi;
  else
        one := R.one;
        elms := Copy(S.generators);
        Add (elms,S.zero);
        set := Set(elms);
        ISize := Size(set);
        i := 1;
        small := true;
        while i <= Length(elms) and small do
                elm := elms[i];
        i := i+1;
        j := 1;
        while j <= LRgen and small do
            rgen := R.generators[j];
            j := j+1;
            new := elm * rgen;
            if not new in set then
                Add(elms,new);
                AddSet(set,new);
                ISize := ISize + 1;
            fi;
            small := (new <> one) and (2*ISize <= RSize);
        od;
        j := 1;
        while j < i and small do
            new := elm + elms[j];
            if not new in set then
                Add(elms,new);
                AddSet(set,new);
                ISize := ISize + 1;
            fi;
            small := new <> one;
            new := elms[j] * elm;
            if not new in set then
                Add(elms,new);
                AddSet(set,new);
                ISize := ISize + 1;
            fi;
            small := small and (new <> one) and (2*ISize <= RSize);
            j := j+1;
        od;
    od;
    if not small then
        set := Elements(R);
    fi;
 fi;
 return set;
end;

IdealOps.PrintRightIdeal := function ( I )
 local i;
 if IsBound (I.name) then
    Print(I.name);
 else
    Print ("RightIdeal ( ",I.isRightIdeal," , ");
    for i in [1 .. Length(I.generators) - 1] do
        Print (I.generators[i]," , ");
    od;
    Print (I.generators[Length(I.generators)]," )");
 fi;
```

```
end;

###############################################################################
##
#F  IdealOps.QuickLeftIdealElements ( S , trivgenset , mcs )
##                             compute all elements of a left ideal S
##                             returns the whole ring if it finds
##                             an element of trivgenset or contains
##                             the set mcs (maximal contained set)

IdealOps.QuickLeftIdealElements := function ( S , trivgenset , mcs )
  local elms,           # list of elements computed so far
        set,            # set of elements computed so far
        elm,            # element already computed
        new,            # element being tested
        rgen,           # generator of the ring
        R,              # the parent ring
        i, j,           # counters
        ISize, RSize,   # size of I and R
        small,          # flag (I does not contain set or ...)
        LRgen,          # number of generators the ring
        maxcontset;

  if IsBound(S.isFinite) and not S.isFinite then
        Error ("<S> must be finite to compute the elements");
  elif not IsBound(S.generators) then
        Error ("sorry, generators of <S> are not known");
  fi;

  R := S.isLeftIdeal;
  RSize := Size(R);
  LRgen := Length(R.generators);

  if IsBound (R.one) then
        AddSet(trivgenset,R.one);
  fi;

  elms := Copy(S.generators);
  Add (elms,S.zero);
  set := Set(elms);
  ISize := Size(set);
  maxcontset := Difference (mcs,set);
  i := 1;
  small := true;
  while i <= Length(elms) and small do
        elm := elms[i];
        i := i+1;
        j := 1;
        while j <= LRgen and small do
                rgen := R.generators[j];
                j := j+1;
                new := rgen * elm;
                if not new in set then
                        Add(elms,new);
                        AddSet(set,new);
                        RemoveSet(maxcontset,new);
                        ISize := ISize + 1;
                fi;
                small := not new in trivgenset
                         and (2*ISize <= RSize)
                         and Length(maxcontset)>0;
        od;
        j := 1;
        while j < i and small do
                new := elm + elms[j];
                if not new in set then
                        Add(elms,new);
                        AddSet(set,new);
                        RemoveSet(maxcontset,new);
                        ISize := ISize + 1;
                fi;
                small := not new in trivgenset
                         and (2*ISize <= RSize)
                         and Length(maxcontset)>0;
                j := j+1;
        od;
  od;
  if not small then
        set := Elements(R);
  fi;

  return set;
end;

#############################################################################
##
#F  IdealOps.QuickRightIdealElements ( S , trivgenset , mcs )
##                              computes all elements of a right ideal S
##                              returns the whole ring if it finds
##                              an element of trivgenset or contains
##                              the set mcs (maximal contained set)

IdealOps.QuickRightIdealElements := function ( S , trivgenset , mcs )
  local elms,           # list of elements computed so far
        set,            # set of elements computed so far
        elm,            # element already computed
        new,            # element being tested
        rgen,           # generator of the ring
        R,              # the parent ring
        i, j,           # counters
```

```
      ISize, RSize,   # size of I and R
      small,          # flag (I does not contain set or ...)
      LRgen,          # number of generators the ring
      maxcontset;

if IsBound(S.isFinite) and not S.isFinite then
      Error ("<S> must be finite to compute the elements");
elif not IsBound(S.generators) then
      Error ("sorry, generators of <S> are not known");
fi;

R := S.isRightIdeal;
RSize := Size(R);
LRgen := Length(R.generators);

if IsBound (R.one) then
      AddSet(trivgenset,R.one);
fi;

elms := Copy(S.generators);
Add (elms,S.zero);
set := Set(elms);
ISize := Size(set);
maxcontset := Difference (mcs,set);
i := 1;
small := true;
while i <= Length(elms) and small do
      elm := elms[i];
      i := i+1;
      j := 1;
      while j <= LRgen and small do
              rgen := R.generators[j];
              j := j+1;
              new := elm * rgen;
              if not new in set then
                      Add(elms,new);
                      AddSet(set,new);
                      RemoveSet(maxcontset,new);
                      ISize := ISize + 1;
              fi;
              small := not new in trivgenset
                       and (2*ISize <= RSize)
                       and Length(maxcontset)>0;
      od;
      j := 1;
      while j < i and small do
              new := elm + elms[j];
              if not new in set then
                      Add(elms,new);
                      AddSet(set,new);
```

```
                        RemoveSet(maxcontset,new);
                        ISize := ISize + 1;
                fi;
                small := not new in trivgenset
                         and (2*ISize <= RSize)
                         and Length(maxcontset)>0;
                j := j+1;
        od;
  od;
  if not small then
        set := Elements(R);
  fi;

  return set;
end;

################################################################################
##
#F  IdealOps.QuickIdealElements ( S , trivgenset , mcs )
##                              computes all elements of an ideal S
##                              returns the whole ring if it finds
##                              an element of trivgenset or contains
##                              the set mcs (maximal contained set)

IdealOps.QuickIdealElements := function ( S , trivgenset , mcs )
  local elms,           # list of elements computed so far
        set,            # set of elements computed so far
        elm,            # element already computed
        new,            # element being tested
        rgen,           # generator of the ring
        R,              # the parent ring
        i, j,           # counters
        ISize, RSize,   # size of I and R
        small,          # flag (I does not contain set or ...)
        LRgen,          # number of generators the ring
        maxcontset;

  if IsBound(S.isFinite) and not S.isFinite then
        Error ("<S> must be finite to compute the elements");
  elif not IsBound(S.generators) then
        Error ("sorry, generators of <S> are not known");
  fi;

  if IsCommutativeRing(S.isIdeal) then
        return S.operations.QuickLeftIdealElements(S,trivgenset,mcs);
  fi;

  R := S.isIdeal;
  RSize := Size(R);
  LRgen := Length(R.generators);
```

```
if IsBound (R.one) then
      AddSet(trivgenset,R.one);
fi;

elms := Copy(S.generators);
Add (elms,S.zero);
set := Set(elms);
ISize := Size(set);
maxcontset := Difference (mcs,set);
i := 1;
small := true;
while i <= Length(elms) and small do
      elm := elms[i];
      i := i+1;
      j := 1;
      while j <= LRgen and small do
              rgen := R.generators[j];
              j := j+1;
              new := rgen * elm;
              if not new in set then
                      Add(elms,new);
                      AddSet(set,new);
                      RemoveSet(maxcontset,new);
                      ISize := ISize + 1;
              fi;
              small := not new in trivgenset;
              new := elm * rgen;
              if not new in set then
                      Add(elms,new);
                      AddSet(set,new);
                      RemoveSet(maxcontset,new);
                      ISize := ISize + 1;
              fi;
              small := small
                       and not new in trivgenset
                       and (2*ISize <= RSize)
                       and Length(maxcontset)>0;
      od;
      j := 1;
      while j < i and small do
              new := elm + elms[j];
              if not new in set then
                      Add(elms,new);
                      AddSet(set,new);
                      RemoveSet(maxcontset,new);
                      ISize := ISize + 1;
              fi;
              small := not new in trivgenset
                       and (2*ISize <= RSize)
```

```
                        and Length(maxcontset)>0;
                j := j+1;
        od;
  od;
  if not small then
        set := Elements(R);
  fi;

  return set;
end;

################################################################################
##
#F  IdealOps.IsNilIdeal ( <I> ) ... checks whether all elements of the
##                       ideal I are nilpotent in I

IdealOps.IsNilIdeal := function ( I )

  return ForAll(Elements(I),x->IsNilpotentElement(I,x));
end;

################################################################################
##
## PolynomialIdealOps

PolynomialIdealOps := Copy(PolynomialRingOps);
### functions for polynomial ideals ###
PolynomialIdealOps.name := "PolynomialIdealOps";

PolynomialIdealOps.PrintIdeal := function ( I )
  local i;              # counter

  if IsBound (I.name) then
        Print (I.name);
  else
        Print ("Ideal ( ",I.isIdeal," , ");
        for i in [1 .. Length(I.generators) - 1] do
                Print (I.generators[i]," , ");
        od;
        Print (I.generators[Length(I.generators)]," )");
  fi;
end;

PolynomialIdealOps.\+ := function ( I1 , I2 )

  if Indeterminate(I1.isIdeal) <> Indeterminate(I2.isIdeal) then
        Error("sorry, these are not ideals of a common ring");
  fi;

  return Ideal(I1.isIdeal,Union(I1.generators,I2.generators));
```

```
end;

PolynomialIdealOps.\* := function ( I1 , I2 )

  if Indeterminate(I1.isIdeal) <> Indeterminate(I2.isIdeal) then
        Error("sorry, these are not ideals of a common ring");
  fi;

  return Ideal(I1.isIdeal,Set(List(Cartesian(I1.generators,I2.generators),
                                    function(x,y) return x*y; end)));
end;

PolynomialIdealOps.\= := function ( I1 , I2 )

  return ( Indeterminate(I1.isIdeal) = Indeterminate(I2.isIdeal) )
     and ( I1.generators = I2.generators );
end;

################################################################################
##
#F  PolynomialRingOps.Nilradical ( <R> ) ... compute the
##                              nilradical of the ring R
##

PolynomialRingOps.Nilradical := function (R)
  if IsField(R.baseRing) then
        return [R.zero];
  else
        return Nilradical(R.baseRing);
  fi;
end;

################################################################################
##
#F  RingOps.Nilradical ( <R> ) ... computes the ideal N(R) containing all
##                       nilpotent elements of R

RingOps.Nilradical := function ( R )
  local powers,         # set of candidates
        r,ri,           # candidates
        T,              # remaining elements to test
        gens;           # generators of the ideal

  if not IsFinite(R) then
        Error("sorry, <R> is infinite");
  fi;

  gens := [];
  T := Difference(Elements(R),[R.zero]);
  for r in T do
```

```
        powers := [];
        ri := r;
        while not(ri in powers) and (ri <> R.zero) do
                AddSet(powers,ri);
                ri := ri * r;
        od;
        if ri = R.zero then
                AddSet(gens,r);
        fi;
        SubtractSet(T,powers);
  od;
  if gens=[] then
        gens := [R.zero];
  fi;
  return Ideal(R,gens);

end;

################################################################################
##
#F  RingOps.Nilpotents ( <R> ) ... computes the set of nilpotent
##                     elements of the ring R
##

RingOps.Nilpotents := function ( R )
  local powers,         # set of candidates
        r,ri,           # candidates
        T,              # remaining elements to test
        nilpots;        # set of nilpotent elements

  if not IsFinite(R) then
        Error("sorry, <R> is infinite");
  fi;

  nilpots := [R.zero];
  T := Difference(Elements(R),[R.zero]);
  while T<>[] do
        r := T[1];
        powers := [];
        ri := r;
        while not( ri in powers or ri in nilpots ) do
                AddSet(powers,ri);
                ri := ri * r;
        od;
        if ri in nilpots then
                nilpots := Union(nilpots,powers);
        fi;
        SubtractSet(T,powers);
  od;
  return nilpots;
end;

###############################################################################
##
#F  RingOps.JacobsonRadical ( <R> ) computes the Jacobson
##                              radical of the ring R

RingOps.JacobsonRadical := function ( R )
  if IsFinite(R) then
        return Nilradical(R);
  else
        Error("sorry,
        the Jacobson-Radical can not be computed for infinite rings, yet");
  fi;
end;

###############################################################################
##
#F  RingOps.IsRadicalRing ( <R> ) ... checks if R is radical ring

RingOps.IsRadicalRing := function ( R )

        return JacobsonRadical(R)=R;

end;

###############################################################################
##
#F  IdealOps.Rad ( I ) ... computes the radical Rad(I) of the ideal I

IdealOps.Rad := function ( I )
  local r,ri,R,
        IEl,            # elements of I
        candidates,
        geners,         # generators of Rad(I)
        T;

  if not IsFinite(I) then
        Error("sorry, <I> is infinite");
  fi;

  R := I.isIdeal;
  IEl := Elements(I);
  T := Difference(Elements(R),[R.zero]);
  geners := [];

  for r in T do
        ri := r;
        candidates := [];
        while not (ri in IEl or ri in candidates) do
                AddSet(candidates,ri);
                ri := ri * r;
        od;
        if ri in IEl then
                AddSet(geners,r);
        fi;
        SubtractSet(T,candidates);
  od;
  if geners=[] then
        return Ideal(R,[R.zero]);
  else
        return Ideal(R,geners);
  fi;

end;
```

B.6 The Source File `misc.l`

```
######  Miscellaneous  #########################################################
#
# file: <misc.l>

###############################################################################
##
##  XPrint ( <i> )  prints i rightbound if i has at most 2 digits
##

XPrint := function ( i )
        if i<10 then Print("  ",i);
        else Print(" ",i);
        fi;
end;

###############################################################################
##
##  RingOperationTables ( <R> )     prints the operation tables for
##                                 the ring R

RingOperationTables := function ( R )
  local REl,    # set of elements of R
        r,s,    # elements of R
        n,      # size of the ring
        i;      # counter

  if not IsRing(R) then
        Error("<R> is supposed to be a ring");
  elif not IsFinite(R) then
        Error("the operation table for an infinite ring is big");
  fi;

  REl := Elements(R);
  n := Size(R);

  Print("\n + |");
  for i in [0..n-1] do XPrint(i); od; Print("\n");
  Print("---+");
  for i in [0..n-1] do Print("---"); od; Print("\n");

  for r in REl do
        XPrint(Position(REl,r)-1); Print("|");
        for s in REl do
                XPrint(Position(REl,r+s)-1);
        od;
        Print("\n");
  od;

  Print("\n * |");
  for i in [0..n-1] do XPrint(i); od; Print("\n");
  Print("---+");

  for i in [0..n-1] do Print("---"); od; Print("\n");

  for r in REl do
        XPrint(Position(REl,r)-1); Print("|");
        for s in REl do
                XPrint(Position(REl,r*s)-1);
        od;
        Print("\n");
  od;

end;

###############################################################################
##
#F  RingOps.IsSimple ( <R> ) ... checks, whether the ring <R> is simple

RingOps.IsSimple := function ( R )
  if not IsFinite(R) then
        Error("sorry, <R> is not finite");
  fi;
  return ForAll(Elements(R), x-> x=R.zero or Size(Ideal(R,[x]))=Size(R) );
end;

###############################################################################
##
#F  RingOps.IsSemisimple ( R ) ... checks whether the ring <R> is
```

```
##                                     (Jacobson-)semisimple.

RingOps.IsSemisimple := function ( R )
        return Size(JacobsonRadical(R))=1;
end;

################################################################################
#
# ProdEl ( L )  returns a representation for the
#               element in the direct product of the
#               ring

ProdEl := function ( L )
  local elm;            # the record representing the element

  elm := rec();
  elm.element := L;     # .element holds the value
  elm.operations := rec(
        \+ := function ( x , y )
                return ProdEl(List([1..Length(x.element)],
                                    u->x.element[u]+y.element[u]));
        end,
        \- := function ( x , y )
                return ProdEl(List([1..Length(x.element)],
                                    u->x.element[u]-y.element[u]));
        end,
        \* := function ( x , y )
                return ProdEl(List([1..Length(x.element)],
                                    u->x.element[u]*y.element[u]));
        end,
        \= := function ( x , y )
                return ForAll([1..Length(x.element)],
                                    u->x.element[u]=y.element[u]);
        end,

        \< := function ( x , y )                # lexicographic ordering

          return x.element < y.element;         # lists can be compared in GAP
        end,

        Print := function ( x )
                Print(x.element);
        end
  );

  return elm;
end;

################################################################################
##
```

```
#F  RingOps.\* ( <R1> , <R2> ) ... compute the direct
##                              product of two rings
##

RingOps.\* := function ( R1 , R2 )

  return DirectProduct([R1,R2]);
end;

###############################################################################
##
#F  RingOps.DirectProduct ( list ) ... compute the
##                               direct product of the rings

RingOps.DirectProduct := function ( l )
  local prod, i, rgen, gen;

  if not ForAll(l,IsRing) then
        Error("usage: DirectProduct( list of rings )     or R1 * R2");
  fi;

  prod := rec(
        isDomain := true,
        isRing   := true,
        isFinite := ForAll(l,IsFinite),
        operations := Copy(RingOps),
        zero := ProdEl(List(l,x->x.zero))
  );

  # build up set of generators for the direct product
  prod.generators := [];
  for i in [1..Length(l)] do
        for rgen in l[i].generators do
                gen:= Copy(prod.zero);
                gen.element[i] := rgen;
                AddSet(prod.generators,gen);
        od;
  od;

  prod.operations.Size := function ( R )
        if R.isFinite then
                return Product(List(l,Size));
        else
                return "infinity";
        fi;
  end;

  return prod;
end;
```

```
#############################################################################
##
#F  IdealOps.IsModularLeftIdeal ( <L> ) ... checks, whether in L
##                              there is a right identity modulo L

IdealOps.IsModularLeftIdeal := function ( L )
  local REl;    # elements of the ring

  if not IsFinite(L.isLeftIdeal) then
        Error("sorry, can not decide modularity in infinite rings");
  fi;

  REl := Elements(L.isLeftIdeal);

  return ForAny(Elements(L),e->ForAll(REl,r->r*e-r in L));

end;


#############################################################################
##
#F  IdealOps.IsMaximalLeftIdeal ( <I> ) ... checks whether the left
##                                              ideal I is maximal

IdealOps.IsMaximalLeftIdeal := function ( I )
  local R,      # the ring
        K,      # bigger ideal
        KEl,    # elements of K
        t,      # element of R that is added to the ideal generators
        totest, # set of possible generators
        tested; # set of elements already tested

  R := I.isLeftIdeal;
  totest := Difference(Elements(R),Elements(I));
  tested := [];

  for t in totest do
        K := I + LeftIdeal(R,[t]);
        # ["dummy"] is used when we do not require this feature
        # (see QuickLeftIdealElements)
        KEl := K.operations.QuickLeftIdealElements(K,tested,["dummy"]);
        if Size(KEl) < Size(R) then
                return false;
        fi;
        AddSet(tested,t);
  od;

  return true;

end;
```

```
#############################################################################
##
#F  IdealOps.IsMaximalRightIdeal ( <I> ) ... checks whether the right
##                                             ideal I is maximal

IdealOps.IsMaximalRightIdeal := function ( I )
  local R,      # the ring
        K,      # bigger ideal
        KEl,    # elements of K
        t,      # element of R that is added to the ideal generators
        totest, # set of possible generators
        tested; # set of elements already tested

  R := I.isRightIdeal;
  totest := Difference(Elements(R),Elements(I));
  tested := [];

  for t in totest do
        K := I + RightIdeal(R,[t]);
        # ["dummy"] is used when we do not require this feature
        # (see QuickRightIdealElements)
        KEl := K.operations.QuickRightIdealElements(K,tested,["dummy"]);
        if Size(KEl) < Size(R) then
                return false;
        fi;
        AddSet(tested,t);
  od;

  return true;

end;

#############################################################################
##
#F  IdealOps.IsMaximalIdeal ( <I> ) ... checks whether the ideal I is maximal

IdealOps.IsMaximalIdeal := function ( I )
  local R,      # the ring
        K,      # bigger ideal
        KEl,    # elements of K
        t,      # element of R that is added to the ideal generators
        totest, # set of possible generators
        tested; # set of elements already tested

  if IsCommutativeRing(I.isIdeal) then
        return IsMaximalLeftIdeal(I);
  fi;

  R := I.isIdeal;
  totest := Difference(Elements(R),Elements(I));
  tested := [];

  for t in totest do
        K := I + Ideal(R,[t]);
        # ["dummy"] is used when we do not require this feature
        # (see QuickIdealElements)
        KEl := K.operations.QuickIdealElements(K,tested,["dummy"]);
        if Size(KEl) < Size(R) then
                return false;
        fi;
        AddSet(tested,t);
  od;

  return true;

end;

################################################################################
##
#F  IdealOps.IsPrimeIdeal ( I ) ... checks whether the ideal I is prime

IsPrimeIdeal := function ( I )

  Error("sorry, this function is not yet prepared\n");
end;


################################################################################
##
#F  GroupOps.Subgroups ( <G> ) ... compute a list of all subgroups
##

GroupOps.Subgroups := function ( G )

  return List(Lattice(G).classes, x -> x.representative);
end;
```

B.7 The Source File ideals.l

```
######  Ideals  ################################################################
#
# file: <ideals.l>

####### S u b r i n g s ########################################################
#
```

```
#############################################################################
##
#F  RingOps.Subring( <R>, <S> ) ... constructs the subring of the ring R
##                                     generated by the set S and defines R
##                                     as a superring of S

RingOps.Subring := function ( R , L )
  local S;

  S := Copy (R);

  S.generators := L;
  S.parents := [R];
  Unbind(S.size);
  Unbind(S.isFinite);
  Unbind(S.isCommutative);
  Unbind(S.elements);

  return S;
end;

########## I d e a l s #####################################################
#

#############################################################################
##
#F  RingOps.LeftIdeal (R,S) computes the left ideal of the ring R generated by
##                             the subset S

RingOps.LeftIdeal := function ( R , S )

  local L,                     # generated ideal
        G,                     # generators of the ideal
        s,r;                   # elements of S and R resp.

  if IsFinite(R) and (not IsSubset(R,S)) and ForAny(S,x->not (x in R)) then
        Error ("generators do not form a subset of the ring");
  fi;

  if IsList(S) then
        L := rec (
                isDomain := true,
                isRing := true,
                parents := Set([R]),
                isLeftIdeal := R,
                generators := S,
                zero := R.zero,
                operations := Copy(IdealOps)
        );
        L.operations.name := "LeftIdealOps";
        if IsFinite(R) then
                L.isFinite := true;
        fi;
        L.operations.Elements := L.operations.LeftIdealElements;
        L.operations.Print := L.operations.PrintLeftIdeal;
        L.operations.\+ := L.operations.SumOfLeftIdeals;
        L.operations.\* := L.operations.ProductOfLeftIdeals;

        return (L);
  else
        S.isLeftIdeal := R;
        S.operations := Copy (IdealOps);
        S.operations.name := "LeftIdealOps";
        S.operations.Elements := S.operations.LeftIdealElements;
        S.operations.Print := S.operations.PrintLeftIdeal;
        S.operations.\+ := S.operations.SumOfLeftIdeals;
        S.operations.\* := S.operations.ProductOfLeftIdeals;

        return (S);
  fi;

end;

###############################################################################
##
#F  RingOps.RightIdeal ( <R>, <S> ) ... computes the right ideal of the ring R
##                                       generated by the subset S

RingOps.RightIdeal := function ( R , S )

  local I,                      # generated ideal
        G,                      # generators of the ideal
        s,r;                    # elements of S and R resp.

  if IsFinite(R) and (not IsSubset(R,S)) and ForAny(S,x->not (x in R)) then
        Error ("generators do not form a subset of the ring");
  fi;

  if IsList(S) then
        I := rec (
                isDomain := true,
                isRing := true,
                parents := Set([R]),
                isRightIdeal := R,
                generators := S,
                zero := R.zero,
                operations := Copy(IdealOps)
        );
        I.operations.name := "RightIdealOps";
        if IsFinite(R) then
                I.isFinite := true;
        fi;
        I.operations.Elements := I.operations.RightIdealElements;
        I.operations.Print := I.operations.PrintRightIdeal;
        I.operations.\+ := I.operations.SumOfRightIdeals;
        I.operations.\* := I.operations.ProductOfRightIdeals;
        return (I);
  else
        S.isRightIdeal := R;

        S.operations := Copy (IdealOps);
        S.operations.name := "RightIdealOps";
        S.operations.Elements := S.operations.RightIdealElements;
        S.operations.Print := S.operations.PrintRightIdeal;
        S.operations.\+ := S.operations.SumOfRightIdeals;
        S.operations.\* := S.operations.ProductOfRightIdeals;

        return (S);
  fi;

end;
################################################################################
##
#V  PolynomialIdealOps
##

PolynomialIdealOps := Copy(PolynomialRingOps);

### functions for polynomial ideals ###

PolynomialIdealOps.Print := function ( I )
  local i;                # counter

  if IsBound (I.name) then
        Print (I.name);
  else
        Print ("Ideal ( ",I.isIdeal," , ");
        for i in [1 .. Length(I.generators) - 1] do
                Print (I.generators[i]," , ");
        od;
        Print (I.generators[Length(I.generators)]," )");
  fi;
end;

PolynomialIdealOps.\+ := function ( I1 , I2 )

  if Indeterminate(I1.isIdeal) <> Indeterminate(I2.isIdeal) then
        Error("sorry, these are not ideals of a common ring");
  fi;

  return Ideal(I1.isIdeal,Union(I1.generators,I2.generators));
end;

PolynomialIdealOps.\* := function ( I1 , I2 )

  if Indeterminate(I1.isIdeal) <> Indeterminate(I2.isIdeal) then
        Error("sorry, these are not ideals of a common ring");
  fi;

  return Ideal(I1.isIdeal,Set(List(Cartesian(I1.generators,I2.generators),
                                      function(x,y) return x*y; end)));
end;

PolynomialIdealOps.Elements := function ( I )
  Error("Polynomial ideals are infinite");
end;

PolynomialIdealOps.\= := function ( ideal1 , ideal2 )
  if ideal1.isIdeal.indeterminate <> ideal2.isIdeal.indeterminate then
        return false;
  else
        return ideal1.generators = ideal2.generators;
  fi;
end;

PolynomialIdealOps.\in := function ( poly , ideal )
  local p,one;
  if not IsPolynomial(poly) then
        return false;
  elif not poly in ideal.isIdeal then
        return false;
  fi;

  p := ideal.generators[1];
  while IsPolynomial(LeadingCoefficient(p)) do
        p:=LeadingCoefficient(p);
  od;
  one := Indeterminate(p.baseRing);

  return ( NormalFormOfgModF(poly,ideal.generators,one)
                = Indeterminate(poly.baseRing) );

end;

PolynomialIdealOps.IsIdeal := function ( R , S )
  if not IsBound(S.isIdeal) then
        Error("can not check in an infinte ring");
  fi;
  return R.indeterminate = S.isIdeal.indeterminate;
end;

PolynomialIdealOps.IsLeftIdeal := function ( R , S )
  return S.operations.IsIdeal(R,S);
end;

PolynomialIdealOps.IsRightIdeal := function ( R , S )
  return S.operations.IsIdeal(R,S);
end;

###############################################################################
##
#F  PolynomialRingOps.Ideal ( R , L ) ... computes the ideal generated by the
##                                        list L in the polynomial ring R

PolynomialRingOps.Ideal := function ( R , L )
  local I;                     # the generated ideal

  if ForAny(L,x->not(IsPolynomial(x))) then
        Error("sorry, not all generators are polynomials");
  elif ForAny(L,x->Indeterminate(x.baseRing)<>Indeterminate(R.baseRing)) then
        Error("sorry, not all generators are polynomials in <R>");
  fi;


### creating the ideal ###

  Print("Relax and have a coffee,
              now computing a Groebner Basis for the ideal ...\n");

  I := rec (
        isDomain := true,
        isRing := true,
        isFinite := false,
        size := "infinity",
        parents := Set([R]),
        isIdeal := R,
        generators := GroebnerBasis(L),
        zero := R.zero,
        operations := PolynomialIdealOps
  );
        return (I);
end;

###############################################################################
##
#F  PolynomialRingOps.LeftIdeal ( <R> , <poly> )
##
```

```
PolynomialRingOps.LeftIdeal := function ( R , poly )

  Print("this is a commutative ring, computing a two-sided ideal \n");
  return R.operations.Ideal(R,poly);
end;

#############################################################################
##
#F  PolynomialRingOps.RightIdeal ( <R> , <poly> )
##

PolynomialRingOps.RightIdeal := function ( R , poly )

  Print("this is a commutative ring, computing a two-sided ideal \n");
  return R.operations.Ideal(R,poly);
end;

#############################################################################
#
# RingOps.Ideal (R,S) computes the ideal of the ring R gen. by the subset S

RingOps.Ideal := function ( R , S )

  local I,              # generated ideal
        G,              # generators of the ideal
        s,r,q;          # elements of S and R resp.

  if IsFinite(R) and (not IsSubset(R,S)) and ForAny(S,x->not (x in R)) then
        Error ("generators do not form a subset of the ring");
  fi;

  if IsCommutativeRing(R) then
        I := LeftIdeal(R,S);
        I.isIdeal := R;
        I.operations.Print := I.operations.PrintIdeal;
        I.operations.\+ := I.operations.SumOfIdeals;
        I.operations.\* := I.operations.ProductOfIdeals;
        return I;
  fi;

  if IsList(S) then
        I := rec (
                isDomain := true,
                isRing := true,
                parents := Set([R]),
                isIdeal := R,
                generators := S,
                zero := R.zero,
                operations := Copy(IdealOps)
        );

        I.operations.name := "IdealOps";
        I.operations.\+ := I.operations.SumOfIdeals;
        I.operations.\* := I.operations.ProductOfIdeals;
        if IsFinite(R) then
                I.isFinite := true;
        fi;
        I.operations.Elements := I.operations.IdealElements;
        I.operations.Print := I.operations.PrintIdeal;

        return (I);
  else
        S.isIdeal := R;

        S.operations := Copy (IdealOps);
        S.operations.name := "IdealOps";
        S.operations.Elements := S.operations.IdealElements;
        S.operations.Print := S.operations.PrintIdeal;
        S.operations.\+ := S.operations.SumOfIdeals;
        S.operations.\* := S.operations.ProductOfIdeals;
        return (S);
  fi;

end;

###############################################################################
##
#F  RingOps.IsLeftIdeal (R,L) checks whether L is a left ideal in R

RingOps.IsLeftIdeal := function ( arg )

  local l,r,    # ideal/ring elements
        L,R;    # set of elements of L and R-L

  if IsFinite(arg[1]) then
        L := Elements(arg[2]);
        R := Difference(Elements(arg[1]),Elements(arg[2]));
        for l in L do
                for r in R do
                        if not (r*l in L) then
                                return (false);
                        fi;
                od;
        od;
        return (true);
 else
        Error("sorry, can only check for finite rings");
 fi;
```

```
end;

###############################################################################
##
#F  RingOps.IsRightIdeal (R,I) ... checks whether I is a right ideal in R

RingOps.IsRightIdeal := function ( arg )

  local i,r,    # ideal/ring elements
        I,R;    # elements of I and R-I

  if IsFinite(arg[1]) then
        I := Elements(arg[2]);
        R := Difference(Elements(arg[1]),Elements(arg[2]));
        for i in I do
                for r in R do
                        if not (i*r in I) then
                                return (false);
                        fi;
                od;
        od;
        return (true);
 else
        Error("sorry, can only check for finite rings");
 fi;

end;

###############################################################################
##
#F  RingOps.IsIdeal (R,I) ... checks whether I is an ideal in R

RingOps.IsIdeal := function ( arg )

  local i,r,    # ideal/ring elements
        I,R;    # elements of I and R-I

  # in the commutative case leftideals are the same as ideals
  if IsCommutativeRing (arg[1]) then
        return IsLeftIdeal(arg[1],arg[2]);
  fi;

  if IsFinite(arg[1]) then
        I := Elements(arg[2]);
        R := Difference(Elements(arg[1]),Elements(arg[2]));
        for i in I do
                for r in R do
                        if not( (i*r in I) and (r*i in I) ) then
                                return (false);
                        fi;
                od;
        od;

        return (true);
 else
        Error("sorry, can only check for finite rings");
 fi;

end;

GroupRingOps := Copy(RingOps);
GroupRingOps.name := "GroupRingOps";

################################################################################
##
#F  GroupRingOps.Elements ( <R> ) ... compute the elements of
##                                a group ring

GroupRingOps.Elements := function ( R )
  local elms;
  elms := List(Elements(R.group),x->RingElement(x,R.mult));
  R.elements := elms;

  return elms;
end;

################################################################################
##
#F  GroupRingOps.Print ( <R> ) ... print function for group rings
##

GroupRingOps.Print := function ( R )
  if IsBound(R.name) then
        Print(R.name);
  else
        Print("RingfromGroup(",R.group," , ", R.mult);
  fi;
end;

################################################################################
##
#F  GroupRingOps.Subrings ( <R> ) ... compute a list of all subrings
##

GroupRingOps.Subrings := function ( R )
  local subg, subgs, elements, goodies;
  goodies := [];
  subgs := Subgroups(R.group);
  for subg in subgs do
        elements := List(Elements(subg), x-> RingElement(x,R.mult));
        if IsSubring(R,elements) then
                AddSet(goodies,AsRing(subg,R.mult));
        fi;
  od;
  return goodies;
end;

#############################################################################
##
#F  GroupRingOps.Ideals ( <R> ) ... compute a list of all ideals
##

GroupRingOps.Ideals := function ( R )
  return List(Filtered(Subrings(R), x-> IsIdeal(R,x)), y-> Ideal(R,y));
end;

#############################################################################
##
#F  GroupRingOps.LeftIdeals ( <R> ) ... compute a list of all left ideals
##

GroupRingOps.LeftIdeals := function ( R )
  return List(Filtered(Subrings(R), x-> IsLeftIdeal(R,x)), y-> LeftIdeal(R,y));
end;

#############################################################################
##
#F  GroupRingOps.RightIdeals ( <R> ) ... compute a list of all right ideals
##

GroupRingOps.RightIdeals := function ( R )
  return List(Filtered(Subrings(R), x-> IsRightIdeal(R,x)),
                                    y-> RightIdeal(R,y));
end;

#############################################################################
##
#F  GroupOps.AsRing ( <G> , <fct> )     generates a ring with group
##                               multiplication as addition
##                               and fct as multiplication

GroupOps.AsRing := function ( G , mult )

  local GEl,            # set of elements of the group
        x, y, z,        # elements of the group
        ring;           # the resulting ring

  if not IsGroup(G) then
        Error("usage: RingfromGroup ( ring , multiplication-function )");
  fi;
  # tests for non-compatibility of the multiplication with the group

  if not IsAbelian(G) then
        Error("sorry, <G> is not abelian");
  fi;
  if IsBound(G.isFinite) and IsFinite(G) then
        GEl := Elements(G);
        for x in GEl do
          for y in GEl do
                if not mult(x,y) in GEl then
                        Error("sorry, <fct> is not a binary operation on <G>");
                fi;
          od;
        od;
        for x in GEl do
          for y in GEl do
            for z in GEl do
                if mult(mult(x,y),z) <> mult(x,mult(y,z)) then
                                Error("sorry, <fct> is not associative");
                elif mult(x*y,z)<>mult(x,z)*mult(y,z) then
                                Error("sorry, <fct> is not right-distributive");
                elif mult(x,y*z)<>mult(x,y)*mult(x,z) then
                                Error("sorry, <fct> is not left-distributive");
                fi;
            od;
          od;
        od;
  fi;

  ring := rec (
        isDomain := true,
        isRing := true,
        isFinite := true,
        operations := Copy(GroupRingOps),
        zero := RingElement(G.identity , mult),
        size := Size(G),
        generators := List(G.generators,x->RingElement(x,mult)),
        group := G,
        mult := mult
  );

  return ring;

end;
```

Bibliography

Books give not wisdom where there none was before,
but where some is, there reading makes it more.
John Harrington (1561–1612)

[1] ALLENBY, R.B.J.T., "Rings, Fields and Groups", London, 1983

[2] BAER, R., "Rings with duals", Amer. J. Math., v.65 (1943), 569-584

[3] BOOK, R.V., OTTO, F., "String-Rewriting Systems", Springer-Verlag, New York, 1993

[4] BUCHBERGER, B., LOOS, "Algebraic Simplification", Computing, Suppl. 4 (1982), Springer-Verlag, Wien, 1982, 11-43

[5] BUCHBERGER, B., "An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal", Ph.D. Thesis, Math. Inst., Univ. of Innsbruck, 1965, and Aequationes Math. 4/3 (1970), 374-383

[6] COX, D., LITTLE, J., O'SHEA, D., "Ideals, Varieties, and Algorithms", Springer-Verlag, New York, 1992

[7] DERSHOWITZ, N., JOUANNAUD, J.-P., "Rewrite Systems", Handbook of Theoretical Computer Science, vol. B, Elsevier, Utrecht, 1990, 243-320

[8] DIVINSKI, N., "Rings and Radicals", University of Toronto Press, Toronto, 1965

[9] JACOBSON, N., "Basic Algebra", I-III, Freeman, San Francisco, 1974

[10] JACOBSON, N., "Basic Algebra", vol. II, Freeman, San Francisco, 1974

[11] JACOBSON, N., "Structure of Rings", Amer. Math. Soc., Rhode Island, 1968

[12] KNUTH, D.E., BENDIX, P.B., "Simple Word Problems in Universal Algebras", Computational Problems in Abstract Algebras, J. Leech, Ed., Pergammon Press, 1970, 263-297

[13] LAM, T.Y., "A First Course in Noncommutative Rings", Springer-Verlag, New York, 1991

[14] LIDL, R., PILZ, G., "Angewandte abstrakte Algebra", B.I.-Wissenschaftsverlag, Zürich, 1982

[15] NÖBAUER, C., ”On Implementing Semigroups and Nearrings in GAP”, Diploma Thesis, Linz, 1995

[16] PETERSON, G.E., STICKEL, M.E., ”Complete Sets of Reductions for Some Equational Theories”, Journal of the ACM, vol. 28, (2) 1981, 233-264

[17] PILZ, Günter, ”Algebra–Ein Reiseführer durch die schönsten Gebiete”, Universitätsverlag Rudolf Trauner, Linz 1984

[18] PILZ, Günter, ”Near-Rings”, North-Holland, Amsterdam, 1983

[19] PILZ, Günter, ”Endliche Strukturen”, Universitätsverlag Rudolf Trauner, Linz, 1988

[20] SCHÖNERT, M. et al., ”GAP–Groups, Algorithms and Programming”, Lehrstuhl D für Mathematik, RWTH Aachen, 1994

[21] VAN DER WAERDEN, B., ”Algebra”, I, II, Springer-Verlag, New York, 1966/67

[22] WEISS, P., ”Lineare Algebra und Analytische Geometrie”, Universitätsverlag Rudolf Trauner, Linz, 1987

[23] WIDI, Marcel, ”The Knuth-Bendix Completion Procedure and Other Algorithms Implemented in G.A.P.”, Diploma Thesis, Linz, 1994

[24] WINKLER, F., ”Introduction to Computer Algebra (Lecture Notes)”, RISC-Linz Report Series No. 94-13, Linz, 1994

[25] WINKLER, F. et al., ”An Algorithm for Constructing Canonical Bases (Groebner Bases) for Polynomial Ideals”

Credits

Everything you always wanted to know about rings in GAP*.
(* but were afraid to ask)
was written and conceived by
Jürgen Ecker

producer
Günter Pilz

lots of ideas
Günter Pilz and Erhard Aichinger

first aid on the machines
Reinhard Gutjahr, Marcel Widi and Tim Boykett

advice on GAP-compatible software-design
Christof Nöbauer

literature on rewrite systems
Eugen Ardeleanu, Erhard Aichinger and Marcel Widi

talks about rings
Peter Stadelmeier

a first look at this document and some corrections
Nikolaus Ecker

time and patience
Gudrun Fuß

under-undergraduate training
Maximilian Kalchmair

computers
John von Neumann

typeset
LaTeX

software
GAP

music
Tom Waits and Frank Zappa

I'd wish to express my thankfulness to the following people:

Tom, Karin, Rocky, Silke, Roman and many more of that kind
and to my parents for trying to understand what I'm doing.