

New Algebra: groups, rings, modules and fields with carrier sets

Lawrence C. Paulson

Abstract

A development of abstract algebra in which every structure is a *carrier set* together with its operations, following the locale-based style of Ballarín's *Basic Algebra* [1], itself after Jacobson [2]. A group is a set with a composition and a unit; a ring is a set with two operations; an extension is a carrier with its own addition and multiplication. Nothing is a record, and no structure is identified with a type, so substructures, quotients and extensions are all ordinary values that can be quantified over, ordered, and exhausted by transfinite arguments.

The group theory covers subgroup lattices, normal series, the Zassenhaus and Schreier refinement theorems, the Jordan–Hölder uniqueness theorem, finite and indexed products, group actions, and solvability. The ring theory covers ideals, quotients, the Chinese remainder theorem for finite families, Krull's theorem on the existence of maximal ideals, principal ideal domains, Euclidean domains, unique factorisation, and polynomials in one and in indexed families of indeterminates. The module theory covers submodules, exact sequences, complements, free modules and their universal property, and finite-dimensional vector spaces with the Steinitz exchange lemma and a well-defined dimension.

On these rest the field theory and Galois theory: minimal polynomials, the degree of a finite extension and the tower law, transitivity of algebraicity, normality and separability, Artin's theorem, the Galois correspondence, normal closures, the primitive element theorem, and the finite fields. The cyclicity of the multiplicative group of a finite field is proved, along with the residue rings $\mathbb{Z}/n\mathbb{Z}$, their unit groups, and Euler's and Fermat's theorems.

The development is self-contained: it depends only on the Isabelle/HOL distribution, and is intended to supersede the record-based `HOL-Algebra` rather than to build upon it. Importing the theory `Algebra` makes the whole of it available.

Contents

0.1	Compatibility with Ballarín's original development	17
1	Monoids and Groups	24

1.1	Monoids of Transformations and Abstract Monoids	24
1.2	Groups of Transformations and Abstract Groups	25
1.3	The subgroup generated by a set	32
1.4	Isomorphisms. Cayley's Theorem	37
1.5	Generalized Associativity. Commutativity	46
1.6	Orbits. Cosets of a Subgroup	46
1.7	Congruences. Quotient Monoids and Groups	53
1.8	Homomorphisms	65
1.9	Correspondence Theorem and Third Isomorphism Theorem . . .	76
1.10	Dropping from elements of a partition to representatives	85
1.11	Abstract type of monoids	89
2	Partial orders, lattices and complete lattices	90
2.1	Partial orders	90
2.1.1	Bounds	91
2.1.2	Suprema and infima	91
2.2	Lattices	92
2.2.1	Lattice laws	93
2.3	Complete lattices	94
2.4	Isotone maps	96
2.5	Galois connections	96
3	The lattice of subgroups	98
3.1	Intersections of subgroups	98
3.2	The complete lattice of subgroups	99
4	Isomorphism classes of set-based monoids	100
4.1	The trivial isomorphism class	102
5	Finite normal series	103
6	Finite relative normal chains	108
6.1	Isomorphism classes of normal-series factors	108
6.2	Normal factors	109
6.3	Isomorphism classes of normal-series factors	112
7	Group Actions	112
7.1	Definition and Basic Properties	112
7.2	Stabilizer	113
7.3	Fixed Points	114

7.4	Orbits	114
7.5	The Orbit-Stabilizer Theorem	116
7.6	Connection to Transformation Groups	118
7.7	The permutation representation	119
7.8	Orbit Partition	120
8	Sylow's Theorem	124
8.1	Cyclic subgroups	128
9	The First Sylow Theorem	132
9.1	Conjugation, Center, and Centralizer	137
9.2	First Sylow Theorem	143
10	The Second Sylow Theorem	148
11	Commutators and the Derived Subgroup	166
11.1	The Derived Series and Solvable Groups	170
12	Transfer of Solvability	172
12.1	Subgroups	176
12.2	Solvability of an extension	177
13	The external direct product of two groups	178
13.1	Projections are homomorphisms; the derived series is componentwise-bounded	179
13.2	Product of solvable groups is solvable	180
14	The direct product of an indexed family of groups	181
14.1	The product locale	182
14.2	Carrier, composition and unit	182
14.3	The product is a group	183
14.4	Projections are homomorphisms	184
14.5	The product of a family of abelian groups is abelian	185
15	The Zassenhaus Lemma	185
15.1	The quotient isomorphism	191
16	The Schreier refinement theorem	194
16.1	Refining pairs of finite normal series	194
16.2	Refinement rows	198
16.3	Factor-class refinements	200

16.4	Reduced Schreier refinements	201
17	Composition series, etc.	203
17.1	Simple groups	203
17.2	Normal chains inside a simple factor	205
18	The Jordan–Hölder theorem	210
19	Maximal normal subgroups	212
19.1	The simple-quotient characterisation	213
19.2	Existence in finite groups	217
20	Composition series of finite groups	219
20.1	Existence	221
21	Solvability and composition factors	222
21.1	Auxiliary group-theoretic bridges	222
21.2	The factor criterion	225
21.3	Independence and finite-group formulations	227
22	The Symmetric and Alternating Groups	229
22.1	The symmetric group as a <i>Group</i>	229
22.2	Relation to the development’s abstract symmetric group	230
22.3	The sign homomorphism and the alternating group	230
22.4	Three-cycles	231
23	Insolvability of the Symmetric Group	232
23.1	Three-cycles as commutators	232
23.2	Persistence of three-cycles in the derived series	233
23.3	The main theorem	235
23.4	The alternating group	235
24	Generating the Symmetric Group	237
24.1	A general lemma on generated subgroups	237
24.2	Conjugation is an automorphism of the generated subgroup	237
24.3	Transpositions generate the symmetric group	239
24.4	The standard n -cycle	239
24.5	Conjugating a transposition	240
24.6	The symmetric group is generated by a transposition and a cycle	241
24.7	A transposition and a p -cycle generate S_p for prime p	243
24.8	An arbitrary cycle and transposition generate S_p	248

24.9	An order- p element of S_p is a p -cycle	248
25	Composition factors of the symmetric and alternating groups	251
25.1	A finite group has a composition series, and S_5 in particular . . .	251
25.2	The criterion is consistent with S_n being unsolvable	252
26	Subfields and Field Homomorphisms (Type-Class Setting)	253
26.1	The subfield generated by a set	253
26.2	The subfield locale	253
26.3	Field homomorphisms on a subfield	255
27	Polynomials over a Subfield; Minimal Polynomials in a Tower	257
27.1	Polynomials with coefficients in a subfield	257
27.2	Division by a monic polynomial over a subfield	259
27.3	Minimal polynomial over a subfield	260
28	Simple Algebraic Extensions are Subfields	265
29	Extending a Field Homomorphism across a Simple Algebraic Extension	267
29.1	The mapped polynomial <i>map-poly f</i> on <i>poly-over K</i>	268
29.2	Evaluation commutes with the homomorphism	269
29.3	Well-definedness of the extension	270
29.4	The extension theorem	270
30	Rings	271
30.1	Definition and Elementary Properties	271
30.2	Commutative rings, integral domains, and fields	273
30.3	Ideals, Quotient Rings	274
30.4	Homomorphisms of Rings. Basic Theorems	277
30.5	Abstract type of rings	282
30.6	Linking the locale-based field theory to Isabelle's type classes . .	284
31	Generic field automorphisms and fixed fields	286
31.1	Automorphisms are additive, multiplicative and fix the fixed field	286
32	Subfields of the Complex Numbers and their Automorphisms	294
32.1	Subfields of $\mathbb{C}$	294
32.2	Field automorphisms and the Galois group	296
32.3	Complex conjugation as an automorphism	299
32.4	The subfield generated by a set; splitting subfields	299

32.5 Faithfulness of the action on a generating set	302
33 Iterating the Iso-Extension over a Finite Set of Algebraic Elements	303
34 Transitivity of the Galois Action on the Roots	306
34.1 Rational Subfields and Irreducibility over the Rationals	310
35 The Galois Group Acts on the Roots	311
36 Embedding a Faithful Finite Action into a Concrete Symmetric Group	314
36.1 Relabelling Symmetric Groups along a Bijection	314
37 The Symmetric Group on the Roots and Non-Solvability	318
38 Finite p-groups	320
39 The subring generated by a set	324
39.1 Intersections of subrings	324
39.2 The generated subring	326
40 The characteristic of a ring; finite domains have prime characteristic	326
41 Product Operator for Commutative Monoids	329
41.1 Products over Finite Sets	329
41.2 Results for Abelian Groups	335
41.3 Transfer between two commutative monoids	336
42 The Polynomial Ring $R[X]$ (Carrier-Set Construction)	338
42.1 Carrier, coefficients, and support	338
42.2 Zero and one	339
42.3 Addition	339
42.4 Multiplication (convolution)	339
42.5 Closure under the operations	339
42.6 Distribution of multiplication over finite additive sums	341
42.7 Pointwise additive structure	342
42.8 Multiplicative unit	344
42.9 Distributivity	344
42.10A Fubini law for the additive finite sum	345
42.11 Associativity of convolution	346

42.12	Coefficients, degree, monomials, constants, and the variable . . .	347
42.13	Polynomials of bounded degree	348
42.14	The ring structure	355
42.15	Evaluation at a point	355
42.16	Evaluation along a ring homomorphism	360
43	Polynomials over a Commutative Ring (continued): evaluation is multiplicative	362
44	Polynomials over a Commutative Ring	364
45	Polynomials over an Integral Domain	365
46	Polynomial Division over a Field	366
46.1	Units of $F[X]$	366
46.2	Remainder modulo a polynomial	372
46.3	The factor theorem	375
46.4	A nonzero polynomial has at most <i>degree</i> roots	376
46.5	Low-degree irreducibility	378
46.6	Factorisation into irreducibles	379
46.7	Splitting into linear factors	380
46.8	Monic polynomials and normalisation	384
46.9	The polynomial ring over a field is a principal ideal domain . . .	385
47	Pushing a Polynomial along a Ring Homomorphism	387
48	Prime and Maximal Ideals	392
49	Principal Ideals	400
49.1	Existence of maximal ideals	400
50	Ideals of the Polynomial Ring; Kronecker's Construction	405
50.1	The size of the quotient ring	408
50.2	Divisibility and Euclid's lemma	411
50.3	Uniqueness of the irreducible factorisation	413
51	The prime fields $GF(p) = \mathbb{Z}/p\mathbb{Z}$	416
51.1	The residue rings $\mathbb{Z}/n\mathbb{Z}$	416
51.2	The residue ring as a field	418
51.3	The two-element field as a non-vacuity witness	419
51.4	The two-element field with polynomial machinery	420

51.5	The 3-element prime field with polynomial machinery	421
51.6	The four-element field $GF(4)$ via Kronecker's construction . . .	422
51.7	The eight-element field $GF(8)$ via Kronecker's construction . . .	424
51.8	The nine-element field $GF(9)$ via Kronecker's construction . . .	425
52	Divisibility in commutative rings: units, associates, irreducibles, primes	426
52.1	Divisibility and units	426
52.2	Associated elements	427
52.3	Greatest common divisors and least common multiples	428
52.4	Irreducible and prime elements	429
52.5	In an integral domain, prime elements are irreducible	430
52.6	Divisibility via principal ideals	432
52.7	Noetherian rings	433
52.8	Principal ideal domains	434
52.8.1	The ascending chain condition on principal ideals	435
52.8.2	Bezout: existence of greatest common divisors	437
52.9	Euclidean domains	439
52.10	Factorizations into irreducibles	441
52.11	Uniqueness of factorization	442
52.12	Factorial domains (unique factorization domains)	445
52.12.1	Existence of factorizations	445
52.13	Euclidean type classes as locale Euclidean domains; the integers .	448
52.14	The general type-class bridge	448
52.15	The integers	449
53	The prime subfield of a finite integral domain	449
53.1	Finite integral domains	449
54	Modules over a ring	455
54.1	Elementary closure and arithmetic	455
54.2	Submodules	456
54.3	The sum of two submodules	457
54.4	Linear combinations	461
54.5	A vector space is a module over a field	461
55	Generators, independence, bases and free modules	462
55.1	Coefficient functions	462
55.2	Membership in the span	464
55.3	Generating sets, independence, bases	468

55.4 Uniqueness of coordinates	470
55.5 Coordinates over a basis	472
56 Finite-dimensional vector spaces over a field	475
57 The Steinitz exchange lemma and dimension over an arbitrary field	481
57.1 Span basics	481
57.2 Extension of a linear combination by zero coefficients	482
57.3 Span is a subspace	482
57.4 Span closure, idempotence and transitivity	483
57.5 Vector subtraction inside a span	485
57.6 The exchange step	485
57.7 Independence as irredundancy	487
57.8 Bases span and are independent	489
57.9 The Steinitz exchange lemma	491
57.10 The cardinality bound and uniqueness of dimension	492
57.11 Existence of a basis	493
58 A finite subgroup of the multiplicative group of a field is cyclic	502
58.1 The order of a power in a finite group	502
58.2 Counting elements of each order in a cyclic group	503
58.3 Counting the Residues Coprime to a Modulus	504
58.4 The abstract group criterion for being cyclic	505
58.5 The multiplicative group of a field and the root bound	508
58.6 The polynomial root bound, via the polynomial ring	508
58.7 The main theorem	510
59 A field extension as a vector space over a subfield	511
59.1 A subfield is a locale-based field	511
59.2 The extension as a vector space	512
60 Frobenius on finite subfields	515
60.1 Cardinality of finite fields	515
61 The lattice of ideals	520
61.1 Intersections of ideals	520
61.2 The ideal generated by a set	521
61.3 The product of two ideals	522
61.4 The complete lattice of ideals	523

62	Extension and contraction of ideals along a ring homomorphism	525
62.1	Contraction: the preimage of an ideal	526
62.2	Extension: the ideal generated by the image	527
62.3	The adjunction	528
63	The direct product of an indexed family of rings	531
63.1	The product locale	531
63.2	Carrier and operations	531
63.3	The additive abelian group	532
63.4	The multiplicative monoid	534
63.5	The product is a ring	535
63.6	Projections are ring homomorphisms	535
63.7	The product of a family of commutative rings is commutative . .	536
64	The Chinese Remainder Theorem for commutative rings	537
64.1	Associative-commutative rewriting in an abelian group	537
64.2	The binary direct product of two rings	538
64.3	Sum of two ideals	540
64.4	Two comaximal ideals	543
64.5	The Chinese Remainder Theorem	546
65	The n-ary Chinese Remainder Theorem	549
65.1	Pairwise-comaximal ideals	549
65.2	The canonical projection into the indexed product	551
65.3	Kernel and surjectivity	559
65.4	The n-ary Chinese Remainder Theorem	563
66	The field of fractions of an integral domain	564
66.1	Fractions and their equivalence	564
66.2	The carrier and class equality	566
66.3	The operations on classes	567
66.3.1	Representative selection	568
66.3.2	Congruence: well-definedness under the equivalence . . .	568
66.3.3	The <i>Class</i> -formulas	570
66.3.4	Closure under the operations	572
66.4	The field of fractions	573
66.4.1	The additive abelian group of <i>Fractions</i>	573
66.4.2	The multiplicative monoid of <i>Fractions</i>	576
66.4.3	Distributivity and the ring axiom	577

66.5	The canonical embedding $R \rightarrow \text{Fractions}$	580
67	The polynomial ring over a field is a Euclidean domain	582
67.1	Units and irreducibility across the seam	584
67.2	The field of remainders modulo an irreducible polynomial	586
68	Polynomials in an Indexed Family of Indeterminates	591
68.1	Submultisets	592
68.2	Carrier and coefficients	593
68.3	Operations	593
68.4	Closure under the operations	594
68.5	Additive structure	595
68.6	Multiplicative unit	595
68.7	Distributivity	596
68.8	Associativity of the convolution	597
68.9	The indexed polynomial ring	599
68.10	Constants and indeterminates	599
68.11	Indeterminates a polynomial does not involve	600
68.12	Monomials	601
68.13	Reading a polynomial off its value at an indeterminate	603
69	Polynomials over a Commutative Ring	605
70	The Univariate Case, as a Check on the Convolution	606
71	Extensions of a Field inside the Indexed Polynomial Ring	607
71.1	Operations defined only on their carrier	608
71.2	Lifted polynomials	609
71.3	What it is to be an extension	609
71.4	The ordering by inclusion	612
71.5	Powers and evaluation transfer along the ordering	613
71.6	The base extension	615
71.7	Unions of chains	615
71.8	Evaluation is the same in a member and in the union	623
71.9	The union of a nonempty chain is an extension	623
72	Transporting a Field Structure along an Injection	625
72.1	The base extension is an extension	627
72.2	A maximal extension exists	627
72.3	A fresh index always exists	628

72.4	Transporting a field onto its image, with restricted operations . .	629
73	Working Inside a Fixed Extension	630
73.1	Every polynomial over the base field splits in a maximal extension	641
74	An Extension in which Every Base Polynomial Splits	643
75	Linking the carrier-set polynomial ring to Isabelle's polynomial type	643
75.1	Finite composites are sums and products	644
75.2	The two representations of a polynomial	644
75.3	The dictionary at a type-class field	645
75.4	The ring operations correspond	645
75.5	Constants, the indeterminate, degree and evaluation	646
75.6	Splitting into linear factors	647
75.7	The dictionary read backwards	648
76	Degree of a simple algebraic extension	649
76.1	Uniqueness of the minimal polynomial	649
76.2	The minimal polynomial as a function	650
76.3	The extension degree	651
76.4	The power basis of a simple algebraic extension	652
76.5	The tower law, for a two-step simple tower	656
77	Finite field extensions and their degree	659
77.1	Finite intermediate extensions	662
78	Finite-dimensional extensions and transitivity of algebraicity	663
78.1	A finite-dimensional extension is algebraic	664
78.2	Transitivity of algebraicity	666
78.3	Algebraicity along a radical/simple tower	667
79	Generic properties of algebraic field extensions	670
80	The tower law for finite field extensions	673
81	Extensions generated by finitely many algebraic elements	684
81.1	A subfield is a finite extension of itself	684
81.2	A simple algebraic extension is finite	684
81.3	Finite extensions compose	685
81.4	Finitely many algebraic elements generate a finite extension . . .	686

81.5	Coordinates over a finite extension	686
82	Homogeneous linear systems with more unknowns than equations	687
83	The Algebraic Closure of a Field	690
83.1	An extension of the whole type in which every polynomial splits	690
83.2	The extension as a type	692
83.3	The base field inside the extension	695
83.4	Every polynomial over the base subfield splits	696
83.5	The algebraic closure	698
83.6	The closure is algebraically closed	699
84	Module homomorphisms	704
84.1	Basic properties	704
84.2	Composition and inverses	706
84.3	The kernel	708
84.4	The image	708
84.5	Injectivity criterion	709
85	Quotient modules and the first isomorphism theorem	710
85.1	The quotient module of a submodule	710
85.1.1	The induced scaling is well-defined	711
85.1.2	The quotient scaling operation and the quotient module .	713
85.1.3	The natural projection	715
86	The first isomorphism theorem for modules	716
86.1	The kernel-quotient and the induced map	717
86.2	The image $\eta^{-1} M$ is a submodule of the target	718
86.3	The induced map is a module homomorphism, injective, surjective	719
87	The second and third isomorphism theorems for modules	721
87.1	The kernel of the projection is N/K	725
87.2	The third isomorphism theorem	726
87.3	The second isomorphism theorem	727
88	Complementary submodules and projections	733
89	Exact sequences of modules	740
90	The direct product and direct sum of a family of modules	753

90.1	The product locale	753
90.2	Carrier and operations	754
90.3	The additive abelian group of the product	755
90.4	The product is a module	756
90.5	Projections	757
90.6	The direct sum	757
91	The universal property of a free module	758
91.1	The twisted sum	759
91.2	The extension is well defined	761
91.3	The extension	762
92	The rank–nullity theorem	767
92.1	Finite-dimensional consequences	774
93	Isomorphisms of finite-dimensional vector spaces	775
94	Complements in finite-dimensional vector spaces	779
94.1	Dimension of quotient spaces	779
94.2	Grassmann’s dimension formula	780
94.3	Complements in finite-dimensional vector spaces	787
95	Simple Algebraic Extensions inside the Complex Numbers	789
95.1	Evaluation of a rational polynomial at a complex point	789
95.2	The kernel is generated by the minimal polynomial	790
95.3	The simple extension $\mathbb{Q}(a)$ and the base isomorphism	790
96	The Galois correspondence between intermediate fields and subgroups	792
96.1	The fixed field of a set of automorphisms	793
96.2	Intermediate fields and the relative Galois group	793
96.3	The adjunction	795
96.4	The Galois connection	796
97	Normal and separable complex field extensions	798
98	Fixed fields of finite normal and separable extensions	805
99	Primitive elements of finite field extensions	808
99.1	The two-generator theorem over an infinite base	809
99.2	Finite separable extensions are simple	812

100	The order of the Galois group of a finite normal separable extension	815
100.1	The minimal polynomial is irreducible over the base	815
100.2	An F -automorphism of $F(a)$ is determined by its value at a . . .	816
100.3	The order of the Galois group of a simple normal extension . . .	817
100.4	Separability in characteristic 0 : the number of conjugates is the degree	818
100.5	The headline: order of the Galois group of a simple normal extension	820
100.6	Generic finite normal and separable extensions	820
101	Field-side inverse and degree formulas for simple normal extensions	823
101.1	Algebraicity of generated splitting fields	823
102	Generic restriction of field automorphisms	827
102.1	The inductive step: solvability climbs the tower	830
102.2	Complex splitting-field extension of generic restriction	831
102.3	A normal solvable tower has a solvable Galois group	832
102.3.1	The trivial Galois group	832
102.3.2	Normal solvable towers	833
102.3.3	The assembly	833
103	Restriction along a stable intermediate field of a splitting field	834
104	Artin's lemma: a finite automorphism group bounds the degree over its fixed field	838
104.1	Artin's lemma	839
104.2	Transporting a dependence along an automorphism	840
104.3	Descending the coefficients to the fixed field	842
104.4	Dedekind's lemma: distinct automorphisms are linearly independent	845
105	Artin's degree bound: the fixed field of a finite group has small index	848
105.1	The fixed field sits below K as a tower	848
105.2	Artin's bound	848
105.3	The reverse bound: the Galois group is no larger than the degree	850
105.4	Artin's theorem	852
106	The fundamental theorem for finite Galois extensions	855

106.1	The finite correspondence	856
106.2	Degree and index formulas	860
106.3	Normal intermediate fields	862
106.4	Restriction and the quotient theorem	865
107	Radical extensions and solvability by radicals	867
107.1	Radical steps and towers	867
107.2	Solvability by radicals	868
107.3	Concatenation of radical towers	868
107.4	Base change: lifting a radical tower to a larger base	869
108	Stability of root-generated intermediate fields	870
108.1	An automorphism fixes the base field setwise	870
108.2	Root-generated fields are stable	871
108.3	A field homomorphism carries radical steps to radical steps . . .	871
108.4	A field homomorphism carries radical towers to radical towers . .	872
108.5	The top of a radical tower is a subfield containing the base . . .	873
108.6	Splicing a list of towers over a common base into one radical tower	873
108.7	A radical tower is increasing; every field lies in the top	875
109	Finite field extensions	875
110	Galois groups of finite fields	881
111	The New-Algebra Development	888


```

theory Set-Theory
  imports HOL-Library.FuncSet HOL-Library.Disjoint-Sets

begin

no-notation divide (infixl <'/'> 70)
no-notation inverse-divide (infixl <'/'> 70)
no-notation quotient (infixl <'/'> 90)

abbreviation identity  $S \equiv (\lambda x \in S. x)$ 

theorem bij-betw-iff-has-inverse:
  assumes  $\alpha \in S \rightarrow_E T$ 
  shows  $\text{bij-betw } \alpha S T \longleftrightarrow (\exists \beta \in T \rightarrow_E S. \text{compose } S \beta \alpha = \text{identity } S \wedge \text{compose } T \alpha \beta = \text{identity } T)$ 
  (is  $\longleftrightarrow (\exists \beta \in T \rightarrow_E S. ?INV \beta)$ )
proof
  assume  $\text{bij-betw } \alpha S T$ 
  then have  $?INV (\text{restrict } (\text{inv-into } S \alpha) T)$ 
  by (simp add: bij-betw-imp-surj-on compose-id-inv-into compose-inv-into-id)
  then show  $\exists \beta \in T \rightarrow_E S. ?INV \beta$ 
  by (metis <bij-betw  $\alpha S T$ > bij-betw-imp-surj-on inv-into-funcset restrict-PiE restrict-Pi-cancel)
next
  assume  $\exists \beta \in T \rightarrow_E S. ?INV \beta$ 
  then obtain  $\beta$  where  $\alpha \in S \rightarrow T \beta \in T \rightarrow S \wedge x. x \in S \implies \beta (\alpha x) = x \wedge y. y \in T \implies \alpha (\beta y) = y$ 
  by (metis Int-iff PiE-def assms compose-eq restrict-apply')
  then show  $\text{bij-betw } \alpha S T$  by (rule bij-betwI)
qed

```

0.1 Compatibility with Ballarin's original development

Maps as extensional HOL functions

p 5, ll 21–25

```

locale map =
  fixes  $\alpha$  and  $S$  and  $T$ 
  assumes graph [intro, simp]:  $\alpha \in S \rightarrow_E T$ 
begin

```

p 5, ll 21–25

```

lemma map-closed [intro, simp]:
   $a \in S \implies \alpha a \in T$ 
using graph by fast

```

p 5, ll 21–25

```

lemma map-undefined [intro]:
   $a \notin S \implies \alpha \ a = \text{undefined}$ 
using graph by fast

end

p 7, ll 7–8

locale surjective-map = map +
  assumes surjective [intro]:  $\alpha \ ' S = T$ 

p 7, ll 8–9

locale injective-map = map +
  assumes injective [intro, simp]: inj-on  $\alpha \ S$ 

locale bijjective-map = map +
  assumes bijjective [intro, simp]: bij-betw  $\alpha \ S \ T$ 

begin

p 7, ll 9–10

sublocale surjective-map by unfold-locale (simp add: bij-betw-imp-surj-on)

p 7, ll 9–10

sublocale injective-map using bij-betw-def by unfold-locale fast

p 9, ll 12–13

sublocale inverse: map restrict (inv-into  $S \ \alpha$ )  $T \ T \ S$ 
proof qed (simp add: inv-into-into surjective)

p 9, ll 12–13

lemma bij-betw-inverse: bij-betw (restrict (inv-into  $S \ \alpha$ )  $T$ )  $T \ S$ 
  by (simp add: bij-betw-inv-into)

end

locale Equivalence =
  fixes  $S$  and  $E$ 
  assumes equiv: equiv  $S \ E$ 
begin

lemma closed [intro, simp]:  $E \subseteq S \times S$ 
  by (simp add: equiv-type equiv)

lemma reflexive [intro, simp]:  $a \in S \implies (a, a) \in E$ 
  by (metis eq-equiv-class-iff2 equiv)

lemma symmetric [sym]:  $(a, b) \in E \implies (b, a) \in E$ 

```

by (*meson equivE eqv symD*)

lemma *transitive* [*trans*]: $\llbracket (a, b) \in E; (b, c) \in E \rrbracket \implies (a, c) \in E$
by (*meson equiv-def eqv transE*)

lemma *left-closed* [*intro*]:
 $(a, b) \in E \implies a \in S$
using *closed by blast*

p 11, ll 6–11

lemma *right-closed* [*intro*]:
 $(a, b) \in E \implies b \in S$
using *closed by blast*

Equivalence classes

definition *Class* $\equiv (\lambda a \in S. E^{\cdot\cdot}\{a\})$

p 11, ll 24–26

lemma *Class-closed* [*dest*]:
 $\llbracket a \in \text{Class } b; b \in S \rrbracket \implies a \in S$
unfolding *Class-def* **by** *auto*

p 11, ll 24–26

lemma *Class-closed2* [*intro, simp*]:
 $a \in S \implies \text{Class } a \subseteq S$
unfolding *Class-def* **by** *auto*

p 11, ll 24–26

lemma *Class-undefined* [*intro, simp*]:
 $a \notin S \implies \text{Class } a = \text{undefined}$
unfolding *Class-def* **by** *simp*

p 11, ll 24–26

lemma *ClassI* [*intro, simp*]:
 $(a, b) \in E \implies a \in \text{Class } b$
by (*simp add: Class-def local.symmetric right-closed*)

p 11, ll 24–26

lemma *Class-revI* [*intro, simp*]:
 $(a, b) \in E \implies b \in \text{Class } a$
by (*blast intro: symmetric*)

p 11, ll 24–26

lemma *ClassD* [*dest*]:
 $\llbracket b \in \text{Class } a; a \in S \rrbracket \implies (b, a) \in E$
by (*simp add: Class-def local.symmetric*)

p 11, ll 30–31

theorem *Class-self* [*intro, simp*]:
 $a \in S \implies a \in \text{Class } a$
unfolding *Class-def* **by** *simp*

p 11, l 31; p 12, l 1

theorem *Class-Union* [*simp*]:
 $(\bigcup_{a \in S} \text{Class } a) = S$
by *blast*

p 11, ll 2–3

theorem *Class-subset*:
 $(a, b) \in E \implies \text{Class } a \subseteq \text{Class } b$

proof
fix *a* **and** *b* **and** *c*
assume $(a, b) \in E$ **and** $c \in \text{Class } a$
then have $(c, a) \in E$ **by** *auto*
also note $\langle (a, b) \in E \rangle$
finally have $(c, b) \in E$ **by** *simp*
then show $c \in \text{Class } b$ **by** *auto*

qed

p 11, ll 3–4

theorem *Class-eq*:
 $(a, b) \in E \implies \text{Class } a = \text{Class } b$
by (*auto intro!*: *Class-subset intro: symmetric*)

p 12, ll 1–5

theorem *Class-equivalence*:
 $\llbracket a \in S; b \in S \rrbracket \implies \text{Class } a = \text{Class } b \longleftrightarrow (a, b) \in E$

proof
fix *a* **and** *b*
assume $a \in S \ b \in S \ \text{Class } a = \text{Class } b$
then have $a \in \text{Class } a$ **by** *auto*
also note $\langle \text{Class } a = \text{Class } b \rangle$
finally show $(a, b) \in E$ **by** (*fast intro: $\langle b \in S \rangle$*)
qed (*rule Class-eq*)

p 12, ll 5–7

theorem *not-disjoint-implies-equal*:
assumes $\text{Class } a \cap \text{Class } b \neq \{\}$
assumes $a \in S \ b \in S$
shows $\text{Class } a = \text{Class } b$
by (*metis ClassD Class-eq assms disjoint-iff-not-equal*)

definition *Partition* = *Class* ‘*S*

notation *Equivalence.Partition* (**infixl** $\langle ' / \rangle$ 75)

p 12, ll 7–8

lemma *Class-in-Partition* [intro, simp]:

$a \in S \implies \text{Class } a \in \text{Partition}$
unfolding *Partition-def* **by** *fast*

p 12, ll 7–8

theorem *partition*:

partition-on S Partition

proof (*intro partition-onI*)

fix $p\ q$

assume $p \in \text{Partition}\ q \in \text{Partition}\ p \neq q$

with *not-disjoint-implies-equal* **show** $\text{disjnt } p\ q$

by (*auto simp: Partition-def disjnt-iff*)

qed (*force simp add: Partition-def*)**+**

end

context *Equivalence*

begin

p 12, ll 18–20

lemma *representant-exists*: $A \in S / E \implies \exists a \in S. a \in A \wedge A = \text{Class } a$

using *Class-self Partition-def image-iff* **by** *auto*

p 12, ll 18–20

lemma *quotient-ClassE*: $A \in S / E \implies (\bigwedge a. a \in S \implies P (\text{Class } a)) \implies P A$

using *representant-exists* **by** *auto*

end

lemma *equivalenceI*:

assumes *closed*: $E \subseteq S \times S$

and *reflexive*: $\bigwedge a. a \in S \implies (a, a) \in E$

and *symmetric*: $\bigwedge a\ b. (a, b) \in E \implies (b, a) \in E$

and *transitive*: $\bigwedge a\ b\ c. [(a, b) \in E; (b, c) \in E] \implies (a, c) \in E$

shows *Equivalence S E*

using *assms Equivalence-def equiv-def refl-on-def sym-def trans-def*

by *metis*

p 12, ll 21–23

sublocale *Equivalence* $\subseteq$ *natural: surjective-map Class S S / E*

proof **qed** (*auto simp: Partition-def*)

Technical device to achieve Jacobson’s syntax; context where α is not a parameter.

p 12, ll 25–26

locale *fiber-relation-notation* = **fixes** $S :: 'a\ \text{set}$ **begin**

p 12, ll 25–26

definition *Fiber-Relation* ($\langle E'(-) \rangle$) **where** *Fiber-Relation* $\alpha = \{(x, y). x \in S \wedge y \in S \wedge \alpha x = \alpha y\}$

end

Context where classes and the induced map are defined through the fiber relation. This will be the case for monoid homomorphisms but not group homomorphisms.

Avoids infinite interpretation chain.

p 12, ll 25–26

locale *fiber-relation* = *map* **begin**

Install syntax

p 12, ll 25–26

sublocale *fiber-relation-notation* .

p 12, ll 26–27

sublocale *Equivalence* **where** $E = E(\alpha)$
unfolding *Fiber-Relation-def*
by (*force intro: equivalenceI*)

“define $\bar{\alpha}$ by $\bar{\alpha}(\bar{a}) = \alpha(a)$ ”

p 13, ll 8–9

definition *induced* = $(\lambda A \in S / E(\alpha). \text{THE } b. \exists a \in A. b = \alpha a)$

p 13, l 10

theorem *Fiber-equality*:

$\llbracket a \in S; b \in S \rrbracket \implies \text{Class } a = \text{Class } b \longleftrightarrow \alpha a = \alpha b$
unfolding *Class-equivalence* **unfolding** *Fiber-Relation-def* **by** *simp*

p 13, ll 8–9

theorem *induced-Fiber-simp* [*simp*]:

assumes [*intro, simp*]: $a \in S$
shows *induced* (*Class* a) = αa

proof –

have $\bigwedge b. \exists x \in \text{Class } a. b = \alpha x \implies b = \alpha a$
by (*metis ClassD Class-closed Class-eq Fiber-equality assms*)
then have (*THE* $b. \exists a \in \text{Class } a. b = \alpha a$) = αa
by *blast*
then show *?thesis* **unfolding** *induced-def* **by** *simp*

qed

p 13, ll 10–11

interpretation *induced: map induced $S / E(\alpha) T$*
proof (*unfold-locales, rule*)
 fix A
 assume [*intro, simp*]: $A \in S / E(\alpha)$
 obtain a **where** $a \in S$ $a \in A$
 using *representant-exists* **by** *blast*
 have (*THE* $b. \exists a \in A. b = \alpha a$) $\in T$
 by (*metis* (*no-types, lifting*) $\langle A \in \text{Partition} \rangle$ *induced-Fiber-simp*
induced-def map-closed representant-exists restrict-apply)
 then show $\text{induced } A \in T$ **unfolding** *induced-def* **by** *simp*
 qed (*simp add: induced-def*)

p 13, ll 12–13

sublocale *induced: injective-map induced $S / E(\alpha) T$*
proof
 show *inj-on induced Partition*
 unfolding *inj-on-def*
 by (*metis Fiber-equality induced-Fiber-simp representant-exists*)
 qed

p 13, ll 16–19

theorem *factorization-lemma:*
 $a \in S \implies \text{compose } S \text{ induced } \text{Class } a = \alpha a$
by (*simp add: compose-eq*)

p 13, ll 16–19

theorem *factorization* [*simp*]: $\text{compose } S \text{ induced } \text{Class} = \alpha$
by (*rule ext*) (*simp add: compose-def map-undefined*)

p 14, ll 2–4

theorem *uniqueness:*
 assumes *map*: $\beta \in S / E(\alpha) \rightarrow_E T$
 and *factorization*: $\text{compose } S \beta \text{Class} = \alpha$
 shows $\beta = \text{induced}$
proof
 fix A
 show $\beta A = \text{induced } A$
proof (*cases* $A \in S / E(\alpha)$)
 case *True*
 then obtain a **where** [*simp*]: $A = \text{Class } a$ $a \in S$
 using *representant-exists* **by** *auto*
 then have $\beta (\text{Class } a) = \alpha a$ **by** (*metis compose-eq factorization*)
 also have $\dots = \text{induced } (\text{Class } a)$ **by** *simp*
 finally show *?thesis* **by** *simp*
 qed (*simp add: induced-def PiE-arb [OF map]*)
 qed

end

hide-const (**open**) *map*

end

theory *Group-Theory*
 imports *Set-Theory*

begin

1 Monoids and Groups

1.1 Monoids of Transformations and Abstract Monoids

locale *Monoid* =
 fixes *M* **and** *composition* (**infixl** $\langle \cdot \rangle$ 70) **and** *unit* ($\langle 1 \rangle$)
 assumes *composition-closed* [*intro*, *simp*]: $\llbracket a \in M; b \in M \rrbracket \implies a \cdot b \in M$
 and *unit-closed* [*intro*, *simp*]: $1 \in M$
 and *associative* [*intro*]: $\llbracket a \in M; b \in M; c \in M \rrbracket \implies (a \cdot b) \cdot c = a \cdot (b \cdot c)$
 and *left-unit* [*intro*, *simp*]: $a \in M \implies 1 \cdot a = a$
 and *right-unit* [*intro*, *simp*]: $a \in M \implies a \cdot 1 = a$

p 29, ll 27–28

locale *Submonoid* = *Monoid* *M* ($\cdot$) 1
 for *N* **and** *M* **and** *composition* (**infixl** $\langle \cdot \rangle$ 70) **and** *unit* ($\langle 1 \rangle$) +
 assumes *subset*: $N \subseteq M$
 and *sub-composition-closed*: $\llbracket a \in N; b \in N \rrbracket \implies a \cdot b \in N$
 and *sub-unit-closed*: $1 \in N$

begin

p 29, ll 27–28

lemma *sub* [*intro*, *simp*]:
 $a \in N \implies a \in M$
 using *subset* **by** *blast*

p 29, ll 32–33

sublocale *sub*: *Monoid* *N* ($\cdot$) 1
proof **qed** (*auto simp: sub-composition-closed sub-unit-closed*)

end

p 29, ll 33–34

theorem *submonoid-transitive*:
 assumes *Submonoid* *K* *N* *composition* *unit*
 and *Submonoid* *N* *M* *composition* *unit*
 shows *Submonoid* *K* *M* *composition* *unit*
proof –
 interpret *K*: *Submonoid* *K* *N* *composition* *unit* **by** *fact*

interpret M : *Submonoid* N M *composition unit* **by** *fact*
show *?thesis* **by** *unfold-locales auto*
qed

p 28, l 23

locale *transformations* =
fixes $S :: 'a$ *set*

Monoid of all transformations

p 28, ll 23-24

sublocale *transformations* $\subseteq$ *Monoid* $S \rightarrow_E S$ *compose* S *identity* S
by *unfold-locales (auto simp: PiE-def compose-eq compose-assoc Id-compose compose-Id)*

N is a Monoid of transformations of the set S .

p 29, ll 34-36

locale *transformation-monoid* =
transformations $S +$ *Submonoid* M $S \rightarrow_E S$ *compose* S *identity* S **for** M **and** S
begin

p 29, ll 34-36

lemma *transformation-closed* [*intro, simp*]:
 $\llbracket \alpha \in M; x \in S \rrbracket \Longrightarrow \alpha x \in S$
by (*metis PiE-iff sub*)

p 29, ll 34-36

lemma *transformation-undefined* [*intro, simp*]:
 $\llbracket \alpha \in M; x \notin S \rrbracket \Longrightarrow \alpha x = \text{undefined}$
by (*metis PiE-arb sub*)

end

1.2 Groups of Transformations and Abstract Groups

context *Monoid* **begin**

Invertible elements

p 31, ll 3-5

definition *invertible* **where** *invertible* $u \equiv (\exists v \in M. u \cdot v = \mathbf{1} \wedge v \cdot u = \mathbf{1})$

p 31, ll 3-5

lemma *invertibleI* [*intro*]:
 $\llbracket u \cdot v = \mathbf{1}; v \cdot u = \mathbf{1}; v \in M \rrbracket \Longrightarrow \text{invertible } u$
unfolding *invertible-def* **by** *fast*

p 31, ll 3-5

lemma *invertibleE* [elim]:
 $\llbracket \text{invertible } u; \bigwedge v. \llbracket u \cdot v = \mathbf{1} \wedge v \cdot u = \mathbf{1}; v \in M \rrbracket \implies P \rrbracket \implies P$
unfolding *invertible-def* **by** *fast*

p 31, ll 6–7

theorem *inverse-unique*:
 $\llbracket u \cdot v' = \mathbf{1}; v \cdot u = \mathbf{1}; u \in M; v \in M; v' \in M \rrbracket \implies v = v'$
by (*metis associative left-unit right-unit*)

p 31, l 7

definition *inverse* **where** *inverse* $\equiv (\lambda u \in M. \text{THE } v. v \in M \wedge u \cdot v = \mathbf{1} \wedge v \cdot u = \mathbf{1})$

p 31, l 7

theorem *inverse-equality*:
 $\llbracket u \cdot v = \mathbf{1}; v \cdot u = \mathbf{1}; u \in M; v \in M \rrbracket \implies \text{inverse } u = v$
using *inverse-def local.inverse-unique* **by** *auto*

p 31, l 7

lemma *invertible-inverse-closed* [intro, simp]:
 $\llbracket \text{invertible } u; u \in M \rrbracket \implies \text{inverse } u \in M$
using *inverse-equality* **by** *auto*

p 31, l 7

lemma *inverse-undefined* [intro, simp]:
 $u \notin M \implies \text{inverse } u = \text{undefined}$
by (*metis (lifting) inverse-def restrict-apply*)

p 31, l 7

lemma *invertible-left-inverse* [simp]:
 $\llbracket \text{invertible } u; u \in M \rrbracket \implies \text{inverse } u \cdot u = \mathbf{1}$
using *inverse-equality* **by** *auto*

p 31, l 7

lemma *invertible-right-inverse* [simp]:
 $\llbracket \text{invertible } u; u \in M \rrbracket \implies u \cdot \text{inverse } u = \mathbf{1}$
using *inverse-equality* **by** *auto*

p 31, l 7

lemma *invertible-left-cancel* [simp]:
 $\llbracket \text{invertible } x; x \in M; y \in M; z \in M \rrbracket \implies x \cdot y = x \cdot z \longleftrightarrow y = z$
by (*metis associative invertible-def left-unit*)

p 31, l 7

lemma *invertible-right-cancel* [simp]:
 $\llbracket \text{invertible } x; x \in M; y \in M; z \in M \rrbracket \implies y \cdot x = z \cdot x \longleftrightarrow y = z$
by (*metis associative invertible-def right-unit*)

p 31, l 7

lemma *inverse-unit* [simp]: *inverse 1 = 1*
using *inverse-equality* **by** *blast*

p 31, ll 7–8

theorem *invertible-inverse-invertible* [intro, simp]:
[[*invertible u*; *u* ∈ *M*]] ⇒ *invertible (inverse u)*
using *invertible-left-inverse invertible-right-inverse* **by** *blast*

p 31, l 8

theorem *invertible-inverse-inverse* [simp]:
[[*invertible u*; *u* ∈ *M*]] ⇒ *inverse (inverse u) = u*
by (simp add: *inverse-equality*)

end

context *Submonoid* **begin**

Reasoning about *invertible* and *local.inverse* in submonoids.

p 31, l 7

lemma *submonoid-invertible* [intro, simp]:
[[*sub.invertible u*; *u* ∈ *N*]] ⇒ *invertible u*
using *invertibleI* **by** *blast*

p 31, l 7

lemma *submonoid-inverse-closed* [intro, simp]:
[[*sub.invertible u*; *u* ∈ *N*]] ⇒ *inverse u* ∈ *N*
using *inverse-equality* **by** *auto*

end

Def 1.2

p 31, ll 9–10

locale *Group* =
 Monoid G (*·*) **1** **for** *G* **and** *composition* (infixl *⟨·⟩* 70) **and** *unit* (*⟨1⟩*) +
 assumes *invertible* [simp, intro]: *u* ∈ *G* ⇒ *invertible u*

A self-contained introduction rule for groups: the monoid laws together with the existence of a two-sided inverse for every element (stated directly, without reference to the *Monoid.invertible* predicate, which would otherwise require a separate monoid interpretation).

lemma *GroupI*:
 fixes *G* **and** *composition* (infixl *⟨·⟩* 70) **and** *unit* (*⟨1⟩*)
 assumes $\bigwedge a\ b. \llbracket a \in G; b \in G \rrbracket \Longrightarrow a \cdot b \in G$
 and *1* ∈ *G*

```

    and  $\bigwedge a\ b\ c. \llbracket a \in G; b \in G; c \in G \rrbracket \implies (a \cdot b) \cdot c = a \cdot (b \cdot c)$ 
    and  $\bigwedge a. a \in G \implies \mathbf{1} \cdot a = a$ 
    and  $\bigwedge a. a \in G \implies a \cdot \mathbf{1} = a$ 
    and inv:  $\bigwedge u. u \in G \implies \exists v \in G. u \cdot v = \mathbf{1} \wedge v \cdot u = \mathbf{1}$ 
  shows Group  $G\ (\cdot)\ \mathbf{1}$ 
proof -
  interpret Monoid  $G\ (\cdot)\ \mathbf{1}$ 
  using assms(1–5) by unfold-locales blast+
  show ?thesis
proof
  fix  $u$  assume  $u \in G$ 
  then obtain  $v$  where  $v \in G\ u \cdot v = \mathbf{1}\ v \cdot u = \mathbf{1}$  using inv by blast
  then show invertible  $u$  using invertibleI by blast
qed
qed

```

p 31, ll 11–12

The first parameter H is the *subgroup* and G the ambient group, mirroring the parameter order of *Submonoid*. At all use sites one writes *subgroup* $H\ G$ (subgroup then ambient); the later locales *subgroup-of-group* and *normal-subgroup* take their parameters in the same order.

```

locale Subgroup = Submonoid  $H\ G\ (\cdot)\ \mathbf{1}$  + sub: Group  $H\ (\cdot)\ \mathbf{1}$ 
  for  $H$  and  $G$  and composition (infixl  $\langle \cdot \rangle$  70) and unit ( $\langle \mathbf{1} \rangle$ )
begin

```

Reasoning about *invertible* and *local.inverse* in subgroups.

p 31, ll 11–12

```

lemma subgroup-inverse-equality [simp]:
   $u \in H \implies \text{inverse } u = \text{sub.inverse } u$ 
  by (simp add: inverse-equality)

```

p 31, ll 11–12

```

lemma subgroup-inverse-iff [simp]:
   $\llbracket \text{invertible } x; x \in G \rrbracket \implies \text{inverse } x \in H \longleftrightarrow x \in H$ 
  using invertible-inverse-inverse sub.invertible-inverse-closed by fastforce

```

end

A subgroup is in particular a group in its own right.

```

lemma subgroup-imp-Group:
  assumes Subgroup  $H\ M\ \text{composition}\ \text{unit}$ 
  shows Group  $H\ \text{composition}\ \text{unit}$ 
  using assms Subgroup.axioms(2) by fastforce

```

p 31, ll 11–12

```

lemma subgroup-transitive [trans]:

```

```

    assumes Subgroup K H composition unit
    and Subgroup H G composition unit
    shows Subgroup K G composition unit
  proof -
    interpret K: Subgroup K H composition unit by fact
    interpret H: Subgroup H G composition unit by fact
    show ?thesis by unfold-locale auto
  qed

lemma subgroup-restrict:
  assumes I: Subgroup I G composition unit
  and H: Subgroup H G composition unit
  and I  $\subseteq$  H
  shows Subgroup I H composition unit
proof -
  interpret I: Subgroup I G composition unit by fact
  interpret H: Subgroup H G composition unit by fact
  have Submonoid I H composition unit
proof (rule Submonoid.intro[OF H.sub.Monoid-axioms], rule Submonoid-axioms.intro)
  show I  $\subseteq$  H by fact
  show  $\bigwedge a b. \llbracket a \in I; b \in I \rrbracket \implies \text{composition } a b \in I$ 
    by (rule I.sub-composition-closed)
  show unit  $\in I$  by (rule I.sub-unit-closed)
qed
then show ?thesis
  by (rule Subgroup.intro) (rule I.sub.Group-axioms)
qed

```

context Monoid begin

Jacobson states both directions, but the other one is trivial.

p 31, ll 12–15

```

theorem subgroupI:
  fixes G
  assumes subset [THEN subsetD, intro]:  $G \subseteq M$ 
  and [intro]:  $1 \in G$ 
  and [intro]:  $\bigwedge g h. \llbracket g \in G; h \in G \rrbracket \implies g \cdot h \in G$ 
  and [intro]:  $\bigwedge g. g \in G \implies \text{invertible } g$ 
  and [intro]:  $\bigwedge g. g \in G \implies \text{inverse } g \in G$ 
  shows Subgroup G M ( $\cdot$ ) 1
proof -
  interpret sub: Monoid G ( $\cdot$ ) 1 by unfold-locale auto
  show ?thesis
  proof unfold-locale
    fix u assume [intro]:  $u \in G$  show sub.invertible u
    using invertible-left-inverse invertible-right-inverse by blast
  qed auto
qed

```

p 31, l 16

definition $Units = \{u \in M. \text{invertible } u\}$

p 31, l 16

lemma *mem-UnitsI*:

$\llbracket \text{invertible } u; u \in M \rrbracket \implies u \in Units$

unfolding *Units-def* **by** *clarify*

p 31, l 16

lemma *mem-UnitsD*:

$\llbracket u \in Units \rrbracket \implies \text{invertible } u \wedge u \in M$

unfolding *Units-def* **by** *clarify*

p 31, ll 16–21

interpretation *units*: *Subgroup Units M*

proof (*rule subgroupI*)

fix *u1 u2*

assume *Units [THEN mem-UnitsD, simp]: u1 ∈ Units u2 ∈ Units*

have $(u1 \cdot u2) \cdot (\text{inverse } u2 \cdot \text{inverse } u1) = (u1 \cdot (u2 \cdot \text{inverse } u2)) \cdot \text{inverse } u1$

by (*simp add: associative del: invertible-left-inverse invertible-right-inverse*)

also have $\dots = 1$ **by** *simp*

finally have *inv1*: $(u1 \cdot u2) \cdot (\text{inverse } u2 \cdot \text{inverse } u1) = 1$ **by** *simp* — ll 16–18

have $(\text{inverse } u2 \cdot \text{inverse } u1) \cdot (u1 \cdot u2) = (\text{inverse } u2 \cdot (\text{inverse } u1 \cdot u1)) \cdot u2$

by (*simp add: associative del: invertible-left-inverse invertible-right-inverse*)

also have $\dots = 1$ **by** *simp*

finally have *inv2*: $(\text{inverse } u2 \cdot \text{inverse } u1) \cdot (u1 \cdot u2) = 1$ **by** *simp* — l 9, “and similarly”

show $u1 \cdot u2 \in Units$ **using** *inv1 inv2 invertibleI mem-UnitsI* **by** *auto*

qed (*auto simp: Units-def*)

p 31, ll 21–22

theorem *group-of-Units* [*intro, simp*]:

Group Units (·) 1

..

p 31, l 19

lemma *composition-invertible* [*simp, intro*]:

$\llbracket \text{invertible } x; \text{invertible } y; x \in M; y \in M \rrbracket \implies \text{invertible } (x \cdot y)$

using *mem-UnitsD mem-UnitsI* **by** *blast*

p 31, l 20

lemma *unit-invertible*:

invertible 1

by *fast*

Useful simplification rules

p 31, l 22

lemma *invertible-right-inverse2*:
 $\llbracket \text{invertible } u; u \in M; v \in M \rrbracket \implies u \cdot (\text{inverse } u \cdot v) = v$
by (*simp add: associative [THEN sym]*)

p 31, l 22

lemma *invertible-left-inverse2*:
 $\llbracket \text{invertible } u; u \in M; v \in M \rrbracket \implies \text{inverse } u \cdot (u \cdot v) = v$
by (*simp add: associative [THEN sym]*)

p 31, l 22

lemma *inverse-composition-commute*:
assumes [*simp*]: *invertible x invertible y x ∈ M y ∈ M*
shows *inverse (x · y) = inverse y · inverse x*
proof –
have *inverse (x · y) · (x · y) = (inverse y · inverse x) · (x · y)*
by (*simp add: invertible-left-inverse2 associative*)
then show ?thesis **by** (*simp del: invertible-left-inverse*)
qed

lemma *conjugate-composition*:
assumes *invertible g g ∈ M a ∈ M b ∈ M*
shows *inverse g · (a · b) · g =*
(inverse g · a · g) · (inverse g · b · g)
proof –
have *invg: inverse g ∈ M*
by (*rule invertible-inverse-closed[OF assms(1,2)]*)
have *ab: a · b ∈ M* **by** (*rule composition-closed[OF assms(3,4)]*)
have *bg: b · g ∈ M* **by** (*rule composition-closed[OF assms(4,2)]*)
have *ia: inverse g · a ∈ M* **by** (*rule composition-closed[OF invg assms(3)]*)
have *ib: inverse g · b ∈ M* **by** (*rule composition-closed[OF invg assms(4)]*)
have *ibg: inverse g · b · g ∈ M*
by (*rule composition-closed[OF ib assms(2)]*)
have *cancel: g · (inverse g · (b · g)) = b · g*
by (*rule invertible-right-inverse2[OF assms(1,2) bg]*)
have *left:*
inverse g · (a · b) · g = inverse g · (a · (b · g))
proof –
have *inverse g · (a · b) · g = inverse g · ((a · b) · g)*
by (*rule associative[OF invg ab assms(2)]*)
also have *... = inverse g · (a · (b · g))*
by (*simp only: associative[OF assms(3,4,2)]*)
finally show ?thesis .
qed
have *right:*
(inverse g · a · g) · (inverse g · b · g) =
inverse g · (a · (b · g))
proof –
have *(inverse g · a · g) · (inverse g · b · g) =*
(inverse g · a) · (g · (inverse g · b · g))

```

    by (rule associative[OF ia assms(2) ibg])
  also have ... = (inverse g · a) · (g · (inverse g · (b · g)))
    by (simp only: associative[OF invg assms(4,2)])
  also have ... = (inverse g · a) · (b · g)
    by (simp only: cancel)
  also have ... = inverse g · (a · (b · g))
    by (rule associative[OF invg assms(3) bg])
  finally show ?thesis .
qed
show ?thesis by (rule trans[OF left right[symmetric]])
qed

```

end

context Group begin

The whole group is a subgroup of itself.

lemma *group-self-subgroup*: Subgroup $G \ G \ (\cdot) \ 1$
 by (rule subgroupI) auto

The trivial subgroup, consisting of the unit alone.

lemma *trivial-subgroup*: Subgroup $\{1\} \ G \ (\cdot) \ 1$
 by (rule subgroupI) auto

1.3 The subgroup generated by a set

The subgroup generated by a set S of elements of a group, defined inductively as the smallest set containing $S \cap G$ and closed under the group operations. This basic construction underlies commutators, the derived series, and solvability.

inductive-set *generate* :: 'a set $\Rightarrow$ 'a set **for** $S ::$ 'a set
where
 unit: $1 \in \text{generate } S$
 | incl: $x \in S \cap G \implies x \in \text{generate } S$
 | comp: $\llbracket x \in \text{generate } S; y \in \text{generate } S \rrbracket \implies x \cdot y \in \text{generate } S$
 | inv: $x \in \text{generate } S \implies \text{inverse } x \in \text{generate } S$

definition *Gen* :: 'a set $\Rightarrow$ 'a set ($\langle \cdot \rangle$)
where $\langle S \rangle \equiv \text{generate } (S \cap G)$

Everything generated lies in the carrier.

lemma *generate-subset-carrier*: $\text{generate } S \subseteq G$
proof

```

  fix x assume x ∈ generate S
  then show x ∈ G
  proof induction
    case unit then show ?case by simp
  next
    case (incl x) then show ?case by simp

```



```

next
  case (comp x y) then show ?case by simp
next
  case (inv x) then show ?case by simp
qed
qed

```

lemma *Gen-subset-carrier*: $\langle S \rangle \subseteq G$
unfolding *Gen-def* **by** (rule *generate-subset-carrier*)

The generated set is a subgroup.

lemma *generate-is-subgroup*: *Subgroup* (*generate* S) G ($\cdot$) **1**
proof (rule *subgroupI*)
 show *generate* $S \subseteq G$ **by** (rule *generate-subset-carrier*)
next
 show **1** $\in$ *generate* S **by** (rule *generate.unit*)
next
 fix g h **assume** $g \in$ *generate* S $h \in$ *generate* S
 then show $g \cdot h \in$ *generate* S **by** (rule *generate.comp*)
next
 fix g **assume** $g \in$ *generate* S
 then have $g \in G$ **using** *generate-subset-carrier* **by** *auto*
 then show *invertible* g **by** *simp*
next
 fix g **assume** $g \in$ *generate* S
 then show *inverse* $g \in$ *generate* S **by** (rule *generate.inv*)
qed

lemma *Gen-is-subgroup*: *Subgroup* $\langle S \rangle$ G ($\cdot$) **1**
unfolding *Gen-def* **by** (rule *generate-is-subgroup*)

The generating set is included (intersected with the carrier).

lemma *generate-incl*: $S \cap G \subseteq$ *generate* S
using *generate.incl* **by** *auto*

lemma *Gen-incl*: $S \cap G \subseteq \langle S \rangle$
unfolding *Gen-def* **using** *generate.incl* **by** *auto*

generate S is the smallest subgroup containing $S \cap G$.

lemma *generate-minimal*:
assumes *Subgroup* H G ($\cdot$) **1** **and** $S \cap G \subseteq H$
shows *generate* $S \subseteq H$
proof
interpret H : *Subgroup* H G ($\cdot$) **1** **by** (rule *assms*(1))
 fix x **assume** $x \in$ *generate* S
 then show $x \in H$
proof *induction*
 case *unit* then show ?case **by** (rule *H.sub-unit-closed*)
next

```

    case (incl x) then show ?case using assms(2) by auto
  next
    case (comp x y)
    then show ?case using H.sub-composition-closed by blast
  next
    case (inv x)
    then show ?case using H.submonoid-inverse-closed H.sub.invertible by auto
qed
qed

```

lemma *Gen-minimal*:

```

  assumes Subgroup H G (·) 1 and  $S \cap G \subseteq H$ 
  shows  $\langle S \rangle \subseteq H$ 
  unfolding Gen-def using assms by (intro generate-minimal) auto

```

Monotonicity.

lemma *generate-mono*:

```

  assumes  $S \subseteq T$ 
  shows generate  $S \subseteq$  generate  $T$ 
proof (rule generate-minimal)
  show Subgroup (generate  $T$ ) G (·) 1 by (rule generate-is-subgroup)
  show  $S \cap G \subseteq$  generate  $T$  using assms generate-incl by auto
qed

```

lemma *Gen-mono*:

```

  assumes  $S \subseteq T$ 
  shows  $\langle S \rangle \subseteq \langle T \rangle$ 
  unfolding Gen-def using assms by (intro generate-mono) (auto)

```

A subgroup generates itself.

lemma *generate-subgroup-eq*:

```

  assumes Subgroup H G (·) 1 and  $H \subseteq G$ 
  shows generate  $H = H$ 
proof
  show generate  $H \subseteq H$  using assms by (intro generate-minimal) auto
  show  $H \subseteq$  generate  $H$  using assms generate-incl by auto
qed

```

lemma *Gen-subgroup-eq*:

```

  assumes Subgroup H G (·) 1
  shows  $\langle H \rangle = H$ 
proof –
  interpret H: Subgroup H G (·) 1 by (rule assms)
  have  $H \subseteq G$  by auto
  then show ?thesis unfolding Gen-def using assms
    by (simp add: Int-absorb2 generate-subgroup-eq)
qed

```

Idempotence.

lemma *Gen-idem*: $\langle\langle S \rangle\rangle = \langle S \rangle$
using *Gen-is-subgroup* **by** (rule *Gen-subgroup-eq*)

The intersection characterisation: $\langle S \rangle$ is the intersection of all subgroups containing $S \cap G$.

lemma *Gen-eq-Inter*:
 $\langle S \rangle = \bigcap \{H. \text{Subgroup } H \ G \ (\cdot) \ \mathbf{1} \wedge S \cap G \subseteq H\}$
proof
show $\langle S \rangle \subseteq \bigcap \{H. \text{Subgroup } H \ G \ (\cdot) \ \mathbf{1} \wedge S \cap G \subseteq H\}$
using *Gen-minimal* **by** *auto*
next
have $\langle S \rangle \in \{H. \text{Subgroup } H \ G \ (\cdot) \ \mathbf{1} \wedge S \cap G \subseteq H\}$
using *Gen-is-subgroup* *Gen-incl* **by** *auto*
then show $\bigcap \{H. \text{Subgroup } H \ G \ (\cdot) \ \mathbf{1} \wedge S \cap G \subseteq H\} \subseteq \langle S \rangle$
by (rule *Inter-lower*)
qed

The set generated inside a subgroup (with the restricted operation) is contained in the set generated in the ambient group. A locality property of *generate*.

lemma *generate-subgroup-subset*:
assumes $H: \text{Subgroup } H \ G \ (\cdot) \ \mathbf{1}$
shows $\text{Group.generate } H \ (\cdot) \ \mathbf{1} \ S \subseteq \text{generate } S$
proof
interpret $S: \text{Subgroup } H \ G \ (\cdot) \ \mathbf{1}$ **by** (rule *H*)
fix x **assume** $x \in \text{Group.generate } H \ (\cdot) \ \mathbf{1} \ S$
then show $x \in \text{generate } S$
proof *induction*
case *unit*
then show *?case* **by** (rule *generate.unit*)
next
case (*incl* x)
then have $x \in S \cap G$ **using** *S.sub* **by** *auto*
then show *?case* **by** (rule *generate.incl*)
next
case (*comp* $x \ y$)
then show *?case* **using** *generate.comp* **by** *blast*
next
case (*inv* x)
have $x \in H$ **using** *inv.hyps* *S.sub.generate-subset-carrier* **by** *auto*
then have $S.\text{sub.inverse } x = \text{inverse } x$ **by** (*simp add: S.subgroup-inverse-equality*)
then show *?case* **using** *inv.IH* *generate.inv* **by** *simp*
qed
qed

end

p 31, l 24

context *transformations* **begin**

p 31, ll 25–26

theorem *invertible-is-bijective*:
 assumes *dom*: $\alpha \in S \rightarrow_E S$
 shows *invertible* $\alpha \longleftrightarrow \text{bij-betw } \alpha \ S \ S$
 using *bij-betw-iff-has-inverse dom invertible-def* **by** *auto*

p 31, ll 26–27

theorem *Units-bijective*:
 $\text{Units} = \{\alpha \in S \rightarrow_E S. \text{bij-betw } \alpha \ S \ S\}$
 unfolding *Units-def* **by** (*auto simp add: invertible-is-bijective*)

p 31, ll 26–27

lemma *Units-bij-betwI* [*intro, simp*]:
 $\alpha \in \text{Units} \implies \text{bij-betw } \alpha \ S \ S$
 by (*simp add: Units-bijective*)

p 31, ll 26–27

lemma *Units-bij-betwD* [*dest, simp*]:
 $\llbracket \alpha \in S \rightarrow_E S; \text{bij-betw } \alpha \ S \ S \rrbracket \implies \alpha \in \text{Units}$
 unfolding *Units-bijective* **by** *simp*

p 31, ll 28–29

abbreviation *Sym* $\equiv \text{Units}$

p 31, ll 26–28

sublocale *symmetric*: *Group Sym compose S identity S*
 by (*fact group-of-Units*)

end

p 32, ll 18–19

locale *transformation-group* =
 transformations S + symmetric: Subgroup G Sym compose S identity S for G and S
begin

p 32, ll 18–19

lemma *transformation-group-closed* [*intro, simp*]:
 $\llbracket \alpha \in G; x \in S \rrbracket \implies \alpha \ x \in S$
 using *bij-betwE* **by** *blast*

p 32, ll 18–19

lemma *transformation-group-undefined* [*intro, simp*]:
 $\llbracket \alpha \in G; x \notin S \rrbracket \implies \alpha \ x = \text{undefined}$
 by (*metis compose-def symmetric.sub.right-unit restrict-apply*)

end

1.4 Isomorphisms. Cayley's Theorem

Def 1.3

p 37, ll 7–11

locale *Monoid-isomorphism* =
bijjective-map η M M' + *source*: *Monoid* M $(\cdot)$ $\mathbf{1}$ + *target*: *Monoid* M' $(\cdot')$ $\mathbf{1}'$
for η **and** M **and** *composition* (**infixl** $\langle \cdot \rangle$ 70) **and** *unit* ($\langle \mathbf{1} \rangle$)
and M' **and** *composition'* (**infixl** $\langle \cdot' \rangle$ 70) **and** *unit'* ($\langle \mathbf{1}' \rangle$) +
assumes *commutes-with-composition*: $\llbracket x \in M; y \in M \rrbracket \implies \eta x \cdot' \eta y = \eta (x \cdot y)$
and *commutes-with-unit*: $\eta \mathbf{1} = \mathbf{1}'$

p 37, l 10

definition *isomorphic-as-monoids* (**infixl** $\langle \cong_M \rangle$ 50)
where $\mathcal{M} \cong_M \mathcal{M}' \iff (\text{let } (M, \text{composition}, \text{unit}) = \mathcal{M}; (M', \text{composition}', \text{unit}') = \mathcal{M}' \text{ in}$
 $(\exists \eta. \text{Monoid-isomorphism } \eta \ M \ \text{composition} \ \text{unit} \ M' \ \text{composition}' \ \text{unit}'))$

p 37, ll 11–12

locale *Monoid-isomorphism'* =
bijjective-map η M M' + *source*: *Monoid* M $(\cdot)$ $\mathbf{1}$ + *target*: *Monoid* M' $(\cdot')$ $\mathbf{1}'$
for η **and** M **and** *composition* (**infixl** $\langle \cdot \rangle$ 70) **and** *unit* ($\langle \mathbf{1} \rangle$)
and M' **and** *composition'* (**infixl** $\langle \cdot' \rangle$ 70) **and** *unit'* ($\langle \mathbf{1}' \rangle$) +
assumes *commutes-with-composition*: $\llbracket x \in M; y \in M \rrbracket \implies \eta x \cdot' \eta y = \eta (x \cdot y)$

p 37, ll 11–12

sublocale *Monoid-isomorphism* $\subseteq$ *Monoid-isomorphism'*
by *unfold-locale* (use *commutes-with-composition* **in** *auto*)

Both definitions are equivalent.

p 37, ll 12–15

sublocale *Monoid-isomorphism'* $\subseteq$ *Monoid-isomorphism*

proof

```
{
  fix y assume y ∈ M'
  then obtain x where η x = y x ∈ M
    using bij-betw-imp-surj-on by blast
  then have y ·' η 1 = y using commutes-with-composition by auto
}
then show η 1 = 1'
  using bij-betwE by fastforce
qed (auto simp: commutes-with-composition)
```

context *Monoid-isomorphism*
begin

p 37, ll 30–33

theorem *inverse-monoid-isomorphism:*

Monoid-isomorphism (restrict (inv-into M η) M') $M' (\cdot)' \mathbf{1}' M (\cdot) \mathbf{1}$
proof –
 have $\bigwedge x y. \llbracket x \in M'; y \in M \rrbracket$
 $\implies \text{inv-into } M \eta x \cdot \text{inv-into } M \eta y = \text{inv-into } M \eta (x \cdot' y)$
 using *commutes-with-composition surjective* **by** *fastforce*
 with *bij-betw-inv-into bij-betw-inv-into-left commutes-with-unit target.unit-closed*
 show *?thesis*
 proof *unfold-locales qed fastforce+*
qed

end

We only need that η is symmetric.

p 37, ll 28–29

theorem *isomorphic-as-monoids-symmetric:*

$(M, \text{composition}, \text{unit}) \cong_M (M', \text{composition}', \text{unit}') \implies (M', \text{composition}', \text{unit}') \cong_M (M, \text{composition}, \text{unit})$
by (*simp add: isomorphic-as-monoids-def*) (*meson Monoid-isomorphism.inverse-monoid-isomorphism*)

Reflexivity and transitivity complete the picture: isomorphism is an equivalence relation on monoid triples, which is what lets isomorphism classes be formed as a quotient.

lemma *isomorphic-as-monoids-refl:*

assumes *Monoid M composition unit*
shows $(M, \text{composition}, \text{unit}) \cong_M (M, \text{composition}, \text{unit})$
proof –
 interpret M : *Monoid M composition unit* **by** *fact*
 have *Monoid-isomorphism* (restrict id M) M *composition unit M composition unit*
 by *unfold-locales*
 (*auto simp: PiE-def extensional-def restrict-def bij-betw-def inj-on-def M.composition-closed*)
 then show *?thesis* **unfolding** *isomorphic-as-monoids-def* **by** *auto*
qed

lemma *isomorphic-as-monoids-transitive:*

assumes $(M, \text{composition}, \text{unit}) \cong_M (M', \text{composition}', \text{unit}')$
 and $(M', \text{composition}', \text{unit}') \cong_M (M'', \text{composition}'', \text{unit}'')$
shows $(M, \text{composition}, \text{unit}) \cong_M (M'', \text{composition}'', \text{unit}'')$
proof –
 obtain η **where** *first: Monoid-isomorphism η M composition unit M' composition' unit'*
 using *assms(1) unfolding isomorphic-as-monoids-def by auto*
 obtain ϑ **where** *second: Monoid-isomorphism ϑ M' composition' unit' M'' composition'' unit''*
 using *assms(2) unfolding isomorphic-as-monoids-def by auto*
 interpret *first: Monoid-isomorphism η M composition unit M' composition' unit'*
by *fact*

```

interpret second: Monoid-isomorphism  $\vartheta$   $M'$  composition' unit'  $M''$  composition''
unit'' by fact
have Monoid-isomorphism (restrict ( $\vartheta \circ \eta$ )  $M$ )  $M$  composition unit  $M''$  composition''
unit''
proof
  show restrict ( $\vartheta \circ \eta$ )  $M \in M \rightarrow_E M''$ 
  using first.graph second.graph by auto
  show composition'' (restrict ( $\vartheta \circ \eta$ )  $M$   $x$ ) (restrict ( $\vartheta \circ \eta$ )  $M$   $y$ )
    = restrict ( $\vartheta \circ \eta$ )  $M$  (composition  $x$   $y$ )
  if  $x \in M$   $y \in M$  for  $x$   $y$ 
  using that first.commutes-with-composition second.commutes-with-composition
    first.source.composition-closed first.map-closed by auto
  show restrict ( $\vartheta \circ \eta$ )  $M$  unit = unit''
  using first.commutes-with-unit second.commutes-with-unit first.source.unit-closed
by simp
  have bij-betw ( $\vartheta \circ \eta$ )  $M$   $M''$ 
  using first.bijective second.bijective by (rule bij-betw-trans)
  then show bij-betw (restrict ( $\vartheta \circ \eta$ )  $M$ )  $M$   $M''$ 
  by (simp add: bij-betw-restrict-eq)
qed
then show ?thesis unfolding isomorphic-as-monoids-def by auto
qed

```

Being a group is a property of the isomorphism class, not of the presentation: invertibility transports along a monoid isomorphism, because the isomorphism carries a two-sided inverse to a two-sided inverse and reaches every element of the target. This is what allows isomorphism classes to be formed once, at the level of monoids, with group-ness imposed as a constraint.

```

lemma (in Monoid-isomorphism) target-Group:
  assumes Group  $M$  ( $\cdot$ )  $\mathbf{1}$ 
  shows Group  $M'$  ( $\cdot'$ )  $\mathbf{1}'$ 
proof (rule GroupI)
  interpret source: Group  $M$  ( $\cdot$ )  $\mathbf{1}$  by (rule assms)
  fix  $u$  assume  $u: u \in M'$ 
  then obtain  $a$  where  $a: a \in M$  and  $\eta a: \eta a = u$ 
  using bijective by (metis bij-betw-imp-surj-on imageE)
  obtain  $b$  where  $b: b \in M$  and  $ab: a \cdot b = \mathbf{1}$  and  $ba: b \cdot a = \mathbf{1}$ 
  using source.invertible[OF  $a$ ] unfolding source.invertible-def by blast
  have  $\eta b \in M'$  using  $b$  by (rule map-closed)
  moreover have  $u \cdot' \eta b = \mathbf{1}'$ 
  using  $\eta a$  commutes-with-composition[OF  $a$   $b$ ]  $ab$  commutes-with-unit by simp
  moreover have  $\eta b \cdot' u = \mathbf{1}'$ 
  using  $\eta a$  commutes-with-composition[OF  $b$   $a$ ]  $ba$  commutes-with-unit by simp
  ultimately show  $\exists v \in M'. u \cdot' v = \mathbf{1}' \wedge v \cdot' u = \mathbf{1}'$  by blast
qed (use target.composition-closed target.unit-closed target.associative
  target.left-unit target.right-unit in blast)+

```

p 38, l 4

locale left-translations-of-monoid = Monoid **begin**

p 38, ll 5–7

definition *translation* ($\langle '(-)_L \rangle$) **where** *translation* = $(\lambda a \in M. \lambda x \in M. a \cdot x)$

p 38, ll 5–7

lemma *translation-map* [*intro*, *simp*]:

$a \in M \implies (a)_L \in M \rightarrow_E M$

unfolding *translation-def* **by** *simp*

p 38, ll 5–7

lemma *Translations-maps* [*intro*, *simp*]:

translation ‘ $M \subseteq M \rightarrow_E M$

by (*simp add: image-subsetI*)

p 38, ll 5–7

lemma *translation-apply*:

$\llbracket a \in M; b \in M \rrbracket \implies (a)_L b = a \cdot b$

unfolding *translation-def* **by** *auto*

p 38, ll 5–7

lemma *translation-exist*:

$f \in \text{translation} \text{ ‘ } M \implies \exists a \in M. f = (a)_L$

by *auto*

p 38, ll 5–7

lemmas *Translations-E* [*elim*] = *translation-exist* [*THEN* *bexE*]

p 38, l 10

theorem *translation-unit-eq* [*simp*]:

identity $M = (\mathbf{1})_L$

unfolding *translation-def*

by (*metis left-unit restrict-apply' restrict-ext unit-closed*)

p 38, ll 10–11

theorem *translation-composition-eq* [*simp*]:

assumes [*simp*]: $a \in M \ b \in M$

shows *compose* $M \ (a)_L \ (b)_L = (a \cdot b)_L$

by (*auto simp: associative compose-def translation-def*)

p 38, ll 7–9

sublocale *transformation*: *transformations* M .

p 38, ll 7–9

theorem *Translations-transformation-monoid*:

transformation-monoid (*translation* ‘ M) M

by *unfold-locales auto*

p 38, ll 7–9

sublocale *transformation: transformation-monoid translation* ‘ M M
by (*fact Translations-transformation-monoid*)

p 38, l 12

lemma *translation-is-map: translation* $\in M \rightarrow_E$ (*translation* ‘ M)
using *translation-def* **by** *force*

p 38, ll 12–16

theorem *translation-isomorphism [intro]:*
Monoid-isomorphism translation M $(\cdot)$ **1** (*translation* ‘ M) (*compose* M) (*identity* M)

proof

have *inj-on translation* M

proof (*rule inj-onI*)

fix a b

assume [*simp*]: $a \in M$ $b \in M$ $(a)_L = (b)_L$

have $(a)_L$ **1** $= (b)_L$ **1** **by** *simp*

then show $a = b$ **by** (*simp add: translation-def*)

qed

then show *bij-betw translation* M (*translation* ‘ M)

by (*simp add: inj-on-imp-bij-betw*)

show *translation* $\in M \rightarrow_E$ *translation* ‘ M

using *translation-is-map* **by** *auto*

qed *simp-all*

p 38, ll 12–16

sublocale *Monoid-isomorphism translation* M $(\cdot)$ **1** *translation* ‘ M *compose* M *identity* M ..

end

context *Monoid* **begin**

p 38, ll 1–2

interpretation *left-translations-of-monoid* ..

p 38, ll 1–2

theorem *cayley-monoid:*

$\exists M'$ *composition'* *unit'*. *transformation-monoid* $M' M \wedge (M, (\cdot), \mathbf{1}) \cong_M (M', \text{composition}', \text{unit}')$

by (*simp add: isomorphic-as-monoids-def*) (*fast intro: Translations-transformation-monoid*)

end

p 38, l 17

locale *left-translations-of-group* = *Group* **begin**

p 38, ll 17–18

sublocale *left-translations-of-monoid* **where** $M = G$..

p 38, ll 17–18

notation *translation* $(\cdot'(-)'_L)$

The Group of left translations is a subgroup of the symmetric Group, hence *transformation.sub.invertible*.

p 38, ll 20–22

theorem *translation-invertible* [intro, simp]:

assumes [simp]: $a \in G$

shows *transformation.sub.invertible* $(a)_L$

proof

show *compose* $G (a)_L (\text{inverse } a)_L = \text{identity } G$ **by** *simp*

next

show *compose* $G (\text{inverse } a)_L (a)_L = \text{identity } G$ **by** *simp*

qed *auto*

p 38, ll 19–20

theorem *translation-bijective* [intro, simp]:

$a \in G \implies \text{bij-betw } (a)_L G G$

by (*blast intro: transformation.invertible-is-bijective [THEN iffD1]*)

p 38, ll 18–20

theorem *Translations-transformation-group*:

transformation-group $(\text{translation } ' G) G$

proof *unfold-locales*

show $(\text{translation } ' G) \subseteq \text{transformation.Sym}$

unfolding *transformation.Units-bijective* **by** *auto*

next

fix α

assume $\alpha: \alpha \in \text{translation } ' G$

then obtain a **where** $a: a \in G$ **and** $\text{eq}: \alpha = (a)_L$..

with *translation-invertible* **show** *transformation.sub.invertible* α **by** *simp*

qed *auto*

p 38, ll 18–20

sublocale *transformation: transformation-group* *translation ' G G*

by (*fact Translations-transformation-group*)

end

context *Group* **begin**

p 38, ll 2–3

interpretation *left-translations-of-group* ..

p 38, ll 2–3

theorem *cayley-group*:
 $\exists G' \text{ composition}' \text{ unit}' . \text{ transformation-group } G' \wedge (G, (\cdot), \mathbf{1}) \cong_M (G', \text{ composition}', \text{ unit}')$
by (*simp add: isomorphic-as-monoids-def*) (*fast intro: Translations-transformation-group*)

end

Exercise 3

p 39, ll 9–10

locale *right-translations-of-group* = *Group* **begin**

p 39, ll 9–10

definition *translation* $(\langle'(-)_R\rangle)$ **where** *translation* = $(\lambda a \in G. \lambda x \in G. x \cdot a)$

p 39, ll 9–10

abbreviation *Translations* $\equiv \text{translation } \langle' G$

The isomorphism that will be established is a map different from *translation*.

p 39, ll 9–10

lemma *translation* $\in G \rightarrow_E \text{Translations}$
by (*simp add: translation-def*)

p 39, ll 9–10

lemma *translation-map* [*intro*, *simp*]:
 $a \in G \implies (a)_R \in G \rightarrow_E G$
unfolding *translation-def* **by** *simp*

p 39, ll 9–10

lemma *Translation-maps* [*intro*, *simp*]:
 $\text{Translations} \subseteq G \rightarrow_E G$
by (*simp add: image-subsetI*)

p 39, ll 9–10

lemma *translation-apply*:
 $\llbracket a \in G; b \in G \rrbracket \implies (a)_R b = b \cdot a$
unfolding *translation-def* **by** *auto*

p 39, ll 9–10

lemma *translation-exist*:
 $f \in \text{Translations} \implies \exists a \in G. f = (a)_R$
by *auto*

p 39, ll 9–10

lemmas *Translations-E* [*elim*] = *translation-exist* [*THEN* *bexE*]

p 39, ll 9–10

lemma *translation-unit-eq* [simp]:
identity $G = (\mathbf{1})_R$
unfolding *translation-def* **by** *auto*

p 39, ll 10–11

lemma *translation-composition-eq* [simp]:
assumes [simp]: $a \in G \ b \in G$
shows *compose* $G \ (a)_R \ (b)_R = (b \cdot a)_R$
unfolding *translation-def* **by** *rule (simp add: associative compose-def)*

p 39, ll 10–11

sublocale *transformation: transformations* G .

p 39, ll 10–11

lemma *Translations-transformation-monoid*:
transformation-monoid *Translations* G
by *unfold-locales auto*

p 39, ll 10–11

sublocale *transformation: transformation-monoid* *Translations* G
by (*fact* *Translations-transformation-monoid*)

p 39, ll 10–11

lemma *translation-invertible* [intro, simp]:
assumes [simp]: $a \in G$
shows *transformation.sub.invertible* $(a)_R$
proof
show *compose* $G \ (a)_R \ (\text{inverse } a)_R = \text{identity } G$ **by** *simp*
next
show *compose* $G \ (\text{inverse } a)_R \ (a)_R = \text{identity } G$ **by** *simp*
qed *auto*

p 39, ll 10–11

lemma *translation-bijective* [intro, simp]:
 $a \in G \implies \text{bij-betw } (a)_R \ G \ G$
by (*blast intro: transformation.invertible-is-bijective [THEN iffD1]*)

p 39, ll 10–11

theorem *Translations-transformation-group*:
transformation-group *Translations* G

proof *unfold-locales*

show *Translations* $\subseteq$ *transformation.Sym*
unfolding *transformation.Units-bijective* **by** *auto*

next

fix α

assume $\alpha: \alpha \in \text{Translations}$

then obtain a **where** $a: a \in G$ **and** $\text{eq}: \alpha = (a)_R$..

with *translation-invertible* **show** *transformation.sub.invertible* α **by** *simp*

qed *auto*

p 39, ll 10–11

sublocale *transformation: transformation-group Translations G*
by (*rule Translations-transformation-group*)

p 39, ll 10–11

lemma *translation-inverse-eq [simp]*:
 assumes [*simp*]: $a \in G$
 shows $\text{transformation.sub.inverse } (a)_R = (\text{inverse } a)_R$
proof (*rule transformation.sub.inverse-equality*)
 show $\text{compose } G (a)_R (\text{inverse } a)_R = \text{identity } G$ **by** *simp*
next
 show $\text{compose } G (\text{inverse } a)_R (a)_R = \text{identity } G$ **by** *simp*
qed *auto*

p 39, ll 10–11

theorem *translation-inverse-monoid-isomorphism [intro]*:
 Monoid-isomorphism $(\lambda a \in G. \text{transformation.symmetric.inverse } (a)_R) G (\cdot) \mathbf{1} \text{Translations}$
 (*compose } G) (\text{identity } G)*
 (*is Monoid-isomorphism ?inv - - - -*)

proof

note *bij-betw-compose [trans]*
 have *bij-betw inverse G G*
 by (*rule bij-betwI [where g = inverse]*) *auto*
 also have *bij-betw translation G Translations*
 by (*rule bij-betwI [where g = $\lambda \alpha \in \text{Translations}. \alpha$ 1]*) (*auto simp: translation-apply*)
 finally show *bij-betw ?inv G Translations*
 by (*simp cong: bij-betw-cong add: compose-eq del: translation-unit-eq*)
 show $?inv \mathbf{1} = \text{identity } G$
 using *transformation.symmetric.inverse-unit* **by** *auto*
 show $?inv \in G \rightarrow_E \text{Translations}$
 using *transformation.symmetric.submonoid-inverse-closed*
 transformation.symmetric.sub.invertible **by** *auto*
next
 fix x **and** y
 assume [*simp*]: $x \in G \ y \in G$
 show $\text{compose } G (?inv x) (?inv y) = (?inv (x \cdot y))$ **by** (*simp add: inverse-composition-commute del: translation-unit-eq*)
qed

p 39, ll 10–11

sublocale *Monoid-isomorphism*
 $\lambda a \in G. \text{transformation.symmetric.inverse } (a)_R) G (\cdot) \mathbf{1} \text{Translations compose } G \text{identity } G \mathbf{..}$

end

1.5 Generalized Associativity. Commutativity

p 40, l 27; p 41, ll 1-2

```

locale commutative-monoid = Monoid +
  assumes commutative:  $\llbracket x \in M; y \in M \rrbracket \implies x \cdot y = y \cdot x$ 
begin

```

The left-commutativity law: together with $\llbracket ?a \in M; ?b \in M; ?c \in M \rrbracket \implies ?a \cdot ?b \cdot ?c = ?a \cdot (?b \cdot ?c)$ and $\llbracket ?x \in M; ?y \in M \rrbracket \implies ?x \cdot ?y = ?y \cdot ?x$ it forms a confluent rewrite system for normalising products. We bundle the three as *ac* so a proof can request full associative-commutative normalisation on demand via *simp add: ac*, rather than making them global *simp* rules (which would change the simplifier's normal form throughout the development).

```

lemma left-commutative:
  assumes  $x \in M \ y \in M \ z \in M$ 
  shows  $x \cdot (y \cdot z) = y \cdot (x \cdot z)$ 
proof -
  have  $x \cdot (y \cdot z) = (x \cdot y) \cdot z$  using assms by (simp add: associative)
  also have  $\dots = (y \cdot x) \cdot z$  using assms by (simp add: commutative)
  also have  $\dots = y \cdot (x \cdot z)$  using assms by (simp add: associative)
  finally show ?thesis .
qed

```

lemmas *ac = associative commutative left-commutative*

end

p 41, l 2

```

locale Abelian-Group = Group + commutative-monoid G ( $\cdot$ ) 1

```

1.6 Orbits. Cosets of a Subgroup

context *transformation-group* **begin**

p 51, ll 18-20

```

definition Orbit-Relation
  where Orbit-Relation =  $\{(x, y). x \in S \wedge y \in S \wedge (\exists \alpha \in G. y = \alpha x)\}$ 

```

p 51, ll 18-20

```

lemma Orbit-Relation-memI [intro]:
   $\llbracket y = \alpha x; \alpha \in G; x \in S \rrbracket \implies (x, y) \in \text{Orbit-Relation}$ 
  unfolding Orbit-Relation-def by auto

```

p 51, ll 18-20

```

lemma Orbit-Relation-memE [elim]:
   $\llbracket (x, y) \in \text{Orbit-Relation}; \bigwedge \alpha. \llbracket \alpha \in G; x \in S; y = \alpha x \rrbracket \implies Q \rrbracket \implies Q$ 
  unfolding Orbit-Relation-def by auto

```

p 51, ll 20–23, 26–27

```

sublocale orbit: Equivalence S Orbit-Relation
proof (intro equivalenceI)
  show Orbit-Relation  $\subseteq S \times S$ 
    by (auto simp: Orbit-Relation-def)
next
  fix a
  assume  $a \in S$ 
  then show  $(a, a) \in \textit{Orbit-Relation}$ 
    using Orbit-Relation-memI by fastforce
next
  fix a b
  assume  $(a, b) \in \textit{Orbit-Relation}$ 
  then show  $(b, a) \in \textit{Orbit-Relation}$ 
    by (metis Orbit-Relation-memE Orbit-Relation-memI compose-eq restrict-apply'
      symmetric.sub.invertible
      symmetric.sub.invertible-def transformation-group-closed)
next
  fix a b c
  assume  $(a, b) \in \textit{Orbit-Relation}$  and  $(b, c) \in \textit{Orbit-Relation}$ 
  then have  $\exists \alpha \in G. c = \alpha a$ 
    by (metis Orbit-Relation-memE compose-eq symmetric.sub-composition-closed)
  then show  $(a, c) \in \textit{Orbit-Relation}$ 
    using  $\langle (a, b) \in \textit{Orbit-Relation} \rangle$  by blast
qed

```

p 51, ll 23–24

```

theorem orbit-equality:
   $x \in S \implies \textit{orbit.Class } x = \{\alpha x \mid \alpha. \alpha \in G\}$ 
by (simp add: orbit.Class-def) (blast intro: orbit.symmetric dest: orbit.symmetric)

```

end

context *Monoid-isomorphism* **begin**

p 52, ll 16–17

```

theorem image-subgroup:
  assumes Subgroup G M  $(\cdot) \mathbf{1}$ 
  shows Subgroup  $(\eta \text{ ' } G) M' (\cdot') \mathbf{1}'$ 
proof –
  interpret Subgroup G M  $(\cdot) \mathbf{1}$  by fact
  interpret image: Monoid  $\eta \text{ ' } G (\cdot') \mathbf{1}'$ 
    by unfold-locales (auto simp: commutes-with-composition sub.associative simp flip:
      commutes-with-unit)
  show ?thesis
  proof
    show  $\eta \text{ ' } G \subseteq M'$ 
      using bij-betw-imp-surj-on by blast
  next

```

```

fix  $u :: 'b$ 
assume  $u \in \eta \text{ ' } G$ 
then show  $image.invertible\ u$ 
  using  $commutes-with-composition\ commutes-with-unit\ sub.invertible-def\ im-$ 
 $age.invertible-def$  by  $fastforce$ 
qed  $auto$ 
qed

end

```

Technical device to achieve Jacobson's notation for *Right-Coset* and *Left-Coset*. The definitions are pulled out of *subgroup-of-group* to a context where H is not a parameter.

p 52, l 20

```

locale  $coset-notation = \text{fixes } composition\ (\text{infixl } \langle \cdot \rangle\ 70)\ \text{begin}$ 

```

Equation 23

p 52, l 20

```

definition  $Right-Coset\ (\text{infixl } \langle \cdot \rangle\ 70)\ \text{where } H \mid \cdot\ x \equiv (\lambda h. h \cdot x) \text{ ' } H$ 

```

p 53, ll 8–9

```

definition  $Left-Coset\ (\text{infixl } \langle \cdot \rangle\ 70)\ \text{where } x \mid H \equiv (\lambda h. x \cdot h) \text{ ' } H$ 

```

p 52, l 20

```

lemma  $Right-Coset-memI\ [intro]:$ 
 $h \in H \implies h \cdot x \in H \mid \cdot\ x$ 
unfolding  $Right-Coset-def$  by  $blast$ 

```

p 52, l 20

```

lemma  $Right-Coset-memE\ [elim]:$ 
 $\llbracket a \in H \mid \cdot\ x; \bigwedge h. \llbracket h \in H; a = h \cdot x \rrbracket \implies P \rrbracket \implies P$ 
unfolding  $Right-Coset-def$  by  $blast$ 

```

p 53, ll 8–9

```

lemma  $Left-Coset-memI\ [intro]:$ 
 $h \in H \implies x \cdot h \in x \mid H$ 
unfolding  $Left-Coset-def$  by  $blast$ 

```

p 53, ll 8–9

```

lemma  $Left-Coset-memE\ [elim]:$ 
 $\llbracket a \in x \mid H; \bigwedge h. \llbracket h \in H; a = x \cdot h \rrbracket \implies P \rrbracket \implies P$ 
unfolding  $Left-Coset-def$  by  $blast$ 

```

end

p 52, l 12

locale *subgroup-of-group* = *Subgroup* $H \ G \ (\cdot) \ 1$ + *coset-notation* $(\cdot) + \text{Group } G \ (\cdot) \ 1$
for H **and** G **and** *composition* (**infixl** $\langle \cdot \rangle$ 70) **and** *unit* ($\langle 1 \rangle$)
begin

p 52, ll 12–14

interpretation *left*: *left-translations-of-group* ..
interpretation *right*: *right-translations-of-group* ..

left.translation $\langle H$ denotes Jacobson's $H_L(G)$ and *left.translation* $\langle G$ denotes Jacobson's G_L .

p 52, ll 16–18

theorem *left-translations-of-subgroup-are-transformation-group* [intro]:
transformation-group (*left.translation* $\langle H$) G

proof –

have *Subgroup* (*left.translation* $\langle H$) (*left.translation* $\langle G$) (*compose* G) (*identity* G)
by (*rule* *left.image-subgroup*) *unfold-locales*
also have *Subgroup* (*left.translation* $\langle G$) *left.transformation.Sym* (*compose* G)
(*identity* G) ..

finally interpret *right-coset*: *Subgroup* *left.translation* $\langle H$ *left.transformation.Sym*
compose G *identity* G .

show ?thesis ..

qed

p 52, l 18

interpretation *transformation-group* *left.translation* $\langle H \ G$..

p 52, ll 19–20

theorem *Right-Coset-is-orbit*:

$x \in G \implies H \mid \cdot x = \text{orbit.Class } x$

using *left.translation-apply* **by** (*auto simp: orbit-equality Right-Coset-def*) (*metis imageI sub*)

p 52, ll 24–25

theorem *Right-Coset-Union*:

$(\bigcup_{x \in G} H \mid \cdot x) = G$

by (*simp add: Right-Coset-is-orbit*)

p 52, l 26

theorem *Right-Coset-bij*:

assumes G [*simp*]: $x \in G \ y \in G$

shows *bij-betw* (*inverse* $x \cdot y$) _{R} ($H \mid \cdot x$) ($H \mid \cdot y$)

proof (*rule* *bij-betw-imageI*)

show *inj-on* (*inverse* $x \cdot y$) _{R} ($H \mid \cdot x$)

unfolding *inj-on-def*

by (*smt* (*verit*) *Right-Coset-memE* *assms* *associative composition-closed in-mono* *invertible*)

invertible-inverse-closed invertible-right-inverse right.translation-apply right-unit subset)

next

show $(\text{inverse } x \cdot y)_R \text{ ‘ } (H \mid \cdot x) = H \mid \cdot y$

by *(force simp: right.translation-apply associative invertible-right-inverse2)*

qed

p 52, ll 25–26

theorem *Right-Cosets-cardinality:*

$\llbracket x \in G; y \in G \rrbracket \implies \text{card } (H \mid \cdot x) = \text{card } (H \mid \cdot y)$

by *(fast intro: bij-betw-same-card Right-Coset-bij)*

p 52, l 27

theorem *Right-Coset-unit:*

$H \mid \cdot \mathbf{1} = H$

by *(force simp add: Right-Coset-def)*

p 52, l 27

theorem *Right-Coset-cardinality:*

$x \in G \implies \text{card } (H \mid \cdot x) = \text{card } H$

using *Right-Coset-unit Right-Cosets-cardinality unit-closed* **by** *presburger*

p 52, ll 31–32

definition *index = card orbit.Partition*

Theorem 1.5

p 52, ll 33–35; p 53, ll 1–2

theorem *lagrange:*

assumes *finite G*

shows $\text{card } G = \text{card } H * \text{index}$

unfolding *index-def*

proof *(subst card-partition)*

fix $c :: 'a \text{ set}$

assume $c \in \text{orbit.Partition}$

then show $\text{card } c = \text{card } H$

using *Right-Coset-cardinality Right-Coset-is-orbit orbit.Partition-def*

by *auto*

next

fix $c1 \ c2$

assume $c1 \in \text{orbit.Partition}$ **and** $c2 \in \text{orbit.Partition}$

and $c1 \neq c2$

then show $c1 \cap c2 = \{\}$

using *orbit.not-disjoint-implies-equal*

by *(auto simp: image-iff orbit.Partition-def)*

qed *(use assms in ‹auto simp: orbit.Partition-def›)*

end

```

lemma subgroup-of-groupI:
  assumes Subgroup H G composition unit and Group G composition unit
  shows subgroup-of-group H G composition unit
proof –
  interpret H: Subgroup H G composition unit by fact
  interpret G: Group G composition unit by fact
  show ?thesis by unfold-locales
qed

```

Left cosets

```

context Subgroup begin

```

p 31, ll 11–12

```

lemma image-of-inverse [intro, simp]:
   $x \in H \implies x \in \text{inverse } 'H$ 
  by (smt (verit) imageI inverse-equality sub sub.invertible sub.invertibleE)

```

```

lemma image-of-inverse-iff:
   $x \in \text{inverse } 'H \iff x \in H$ 
  by blast

```

end

```

context Group begin

```

p 53, ll 6–7

```

lemma inverse-subgroupI:
  assumes sub: Subgroup H G (·) 1
  shows Subgroup (inverse 'H) G (·) 1
proof –
  from sub interpret Subgroup H G (·) 1 .
  interpret inv: Monoid inverse 'H (·) 1
    by unfold-locales (auto simp del: subgroup-inverse-equality)
  interpret inv: Group inverse 'H (·) 1
    by unfold-locales (force simp del: subgroup-inverse-equality)
  show ?thesis
    by unfold-locales (auto simp del: subgroup-inverse-equality)
qed

```

p 53, ll 6–7

```

lemma inverse-subgroupD:
  assumes sub: Subgroup (inverse 'H) G (·) 1
  and inv: H ⊆ Units
  shows Subgroup H G (·) 1
proof –
  from sub have Subgroup (inverse 'inverse 'H) G (·) 1 by (rule inverse-subgroupI)
  moreover from inv [THEN subsetD, simplified Units-def] have inverse 'inverse 'H = H

```

```

    by (simp cong: image-cong add: image-comp)
  ultimately show ?thesis by simp
qed

```

Intersections are the basic way of restricting subgroups and normal subgroups to a smaller ambient group.

```

lemma subgroup-intersection:
  assumes Subgroup H G (·) 1 Subgroup K G (·) 1
  shows Subgroup (H ∩ K) G (·) 1
proof -
  interpret H: Subgroup H G (·) 1 by fact
  interpret K: Subgroup K G (·) 1 by fact
  interpret HK: Monoid H ∩ K (·) 1
    by unfold-locales auto
  show ?thesis
proof
  fix u
  assume u: u ∈ H ∩ K
  have inv-H: local.inverse u ∈ H
    by (rule H.submonoid-inverse-closed) (use u in auto)
  have inv-K: local.inverse u ∈ K
    by (rule K.submonoid-inverse-closed) (use u in auto)
  have right: u · local.inverse u = 1
    using u by simp
  have left: local.inverse u · u = 1
    using u by simp
  show HK.invertible u
    unfolding HK.invertible-def
    using inv-H inv-K left right by blast
qed auto
qed

```

```

lemma commute-iff-inverse:
  assumes g ∈ G h ∈ G
  shows (g · h = h · g) ⟷ (g · h · local.inverse g = h)
proof -
  have g · h ∈ G
    using assms by simp
  then have g · h · local.inverse g · g = g · h · (local.inverse g · g)
    using assms by blast
  then show ?thesis
    using assms by auto
qed

```

end

context subgroup-of-group begin

p 53, l 6

interpretation *right-translations-of-group* ..

translation ‘ H denotes Jacobson’s $H_R(G)$ and *Translations* denotes Jacobson’s G_R .

p 53, ll 6–7

theorem *right-translations-of-subgroup-are-transformation-group* [intro]:

transformation-group (*translation* ‘ H) G

proof –

have *Subgroup* (($\lambda a \in G$. *transformation.symmetric.inverse* (a) _{R}) ‘ H) *Translations* (*compose* G) (*identity* G)

by (*rule image-subgroup*) *unfold-locales*

also have *Subgroup* *Translations* *transformation.Sym* (*compose* G) (*identity* G) ..

finally interpret *left-coset*: *Subgroup* *translation* ‘ H *transformation.Sym* *compose* G *identity* G

by (*auto intro*: *transformation.symmetric.inverse-subgroupD* *cong*: *image-cong*

simp: *image-image* *transformation.symmetric.Units-def* *simp del*: *translation-unit-eq*)

show *?thesis* ..

qed

p 53, ll 6–7

interpretation *transformation-group* *translation* ‘ H G ..

Equation 23 for left cosets

p 53, ll 7–8

theorem *Left-Coset-is-orbit*:

$x \in G \implies x \cdot | H = \text{orbit.Class } x$

using *translation-apply*

by (*auto simp*: *orbit-equality* *Left-Coset-def*) (*metis imageI sub*)

end

1.7 Congruences. Quotient Monoids and Groups

Def 1.4

p 54, ll 19–22

locale *Monoid-congruence* = *Monoid* + *Equivalence* **where** $S = M +$

assumes *cong*: $\llbracket (a, a') \in E; (b, b') \in E \rrbracket \implies (a \cdot b, a' \cdot b') \in E$

begin

p 54, ll 26–28

theorem *Class-cong*:

$\llbracket \text{Class } a = \text{Class } a'; \text{Class } b = \text{Class } b'; a \in M; a' \in M; b \in M; b' \in M \rrbracket \implies$

$\text{Class } (a \cdot b) = \text{Class } (a' \cdot b')$

by (*simp add*: *Class-equivalence cong*)

p 54, ll 28–30

definition *quotient-composition* (**infixl** $\langle \cdot \rangle$ 70)
where *quotient-composition* = $(\lambda A \in M / E. \lambda B \in M / E. \text{THE } C. \exists a \in A. \exists b \in B. C = \text{Class } (a \cdot b))$

p 54, ll 28–30

theorem *Class-commutes-with-composition*:
 $\llbracket a \in M; b \in M \rrbracket \implies \text{Class } a [\cdot] \text{Class } b = \text{Class } (a \cdot b)$
by (*auto simp: quotient-composition-def intro: Class-cong [OF Class-eq Class-eq] del: equalityI*)

p 54, ll 30–31

theorem *quotient-composition-closed* [*intro, simp*]:
 $\llbracket A \in M / E; B \in M / E \rrbracket \implies A [\cdot] B \in M / E$
using *Class-commutes-with-composition Partition-def* **by** *auto*

p 54, l 32; p 55, ll 1–3

sublocale *quotient*: *Monoid* M / E ($[\cdot]$) *Class* **1**
by *unfold-locales (auto simp: Class-commutes-with-composition associative elim!: quotient-ClassE)*

end

p 55, ll 16–17

locale *group-congruence* = *Group* + *Monoid-congruence* **where** $M = G$ **begin**

p 55, ll 16–17

notation *quotient-composition* (**infixl** $\langle \cdot \rangle$ 70)

p 55, l 18

theorem *Class-right-inverse*:
 $a \in G \implies \text{Class } a [\cdot] \text{Class } (\text{inverse } a) = \text{Class } \mathbf{1}$
by (*simp add: Class-commutes-with-composition*)

p 55, l 18

theorem *Class-left-inverse*:
 $a \in G \implies \text{Class } (\text{inverse } a) [\cdot] \text{Class } a = \text{Class } \mathbf{1}$
by (*simp add: Class-commutes-with-composition*)

p 55, l 18

theorem *Class-invertible*:
 $a \in G \implies \text{quotient.invertible } (\text{Class } a)$
by (*blast intro!: Class-right-inverse Class-left-inverse*)**+**

p 55, l 18

theorem *Class-commutes-with-inverse*:
 $a \in G \implies \text{quotient.inverse } (\text{Class } a) = \text{Class } (\text{inverse } a)$
by (*rule quotient.inverse-equality (auto simp: Class-right-inverse Class-left-inverse)*)

p 55, l 17

sublocale *quotient*: *Group* G / E ($[\cdot]$) *Class* **1**
by *unfold-locales* (*metis* *Class-invertible quotient-ClassE*)

end

Def 1.5

p 55, ll 22–25

locale *normal-subgroup* =
subgroup-of-group K G ($\cdot$) **1** for K and G and composition (**infixl** $\langle \cdot \rangle$ 70) and unit
($\langle 1 \rangle$) +
assumes *normal*: $\llbracket g \in G; k \in K \rrbracket \implies \text{inverse } g \cdot k \cdot g \in K$

context *Group* **begin**

The trivial subgroup is normal in every group.

lemma *trivial-normal-subgroup*:
normal-subgroup $\{1\}$ G ($\cdot$) **1**
proof (*intro normal-subgroup.intro subgroup-of-group.intro*)
show *Subgroup* $\{1\}$ G ($\cdot$) **1** by (*rule trivial-subgroup*)
show *Group* G ($\cdot$) **1** by (*rule Group-axioms*)
show *normal-subgroup-axioms* $\{1\}$ G ($\cdot$) **1**
proof
fix g k
assume $g \in G$ $k \in \{1\}$
then show *inverse* $g \cdot k \cdot g \in \{1\}$ by *simp*
qed
qed
end

Lemmas from the proof of Thm 1.6

context *subgroup-of-group* **begin**

We use H for K .

p 56, ll 14–16

theorem *Left-equals-Right-coset-implies-normality*:
assumes [*simp*]: $\bigwedge g. g \in G \implies g \cdot H = H \cdot g$
shows *normal-subgroup* H G ($\cdot$) **1**
proof
fix g k
assume [*simp*]: $g \in G$ $k \in H$
have $k \cdot g \in g \cdot H$ by *auto*
then obtain k' where $k \cdot g = g \cdot k'$ and $k' \in H$ by *blast*
then show *inverse* $g \cdot k \cdot g \in H$ by (*simp add: associative invertible-left-inverse2*)
qed

end

Thm 1.6, first part

context *group-congruence* **begin**

Jacobson's K

p 56, l 29

definition $Normal = Class\ 1$

p 56, ll 3–6

interpretation *Subgroup Normal* $G\ (\cdot)\ 1$

unfolding *Normal-def*

proof (*rule subgroupI*)

fix $k1$ **and** $k2$

assume $K: k1 \in Class\ 1\ k2 \in Class\ 1$

then have $k1 \cdot k2 \in Class\ (k1 \cdot k2)$ **by** *blast*

also have $\dots = Class\ k1\ [\cdot]\ Class\ k2$ **using** K **by** (*auto simp add: Class-commutes-with-composition Class-closed*)

also have $\dots = Class\ 1\ [\cdot]\ Class\ 1$ **using** K **by** (*metis ClassD Class-eq unit-closed*)

also have $\dots = Class\ 1$ **by** *simp*

finally show $k1 \cdot k2 \in Class\ 1$.

next

fix k

assume $K: k \in Class\ 1$

then have $inverse\ k \in Class\ (inverse\ k)$

by *blast*

also have $\dots = quotient.inverse\ (Class\ k)$

using *Class-commutes-with-inverse* K **by** *blast*

also have $\dots = quotient.inverse\ (Class\ 1)$

by (*metis Class-eq K Equivalence.ClassD Equivalence-axioms unit-closed*)

also have $\dots = Class\ 1$ **using** *quotient.inverse-unit* **by** *blast*

finally show $inverse\ k \in Class\ 1$.

qed *auto*

Coset notation

p 56, ll 5–6

interpretation *subgroup-of-group Normal* $G\ (\cdot)\ 1\ ..$

Equation 25 for right cosets

p 55, ll 29–30; p 56, ll 6–11

theorem *Right-Coset-Class-unit:*

assumes $g: g \in G$ **shows** $Normal\ |\cdot\ g = Class\ g$

unfolding *Normal-def*

proof *auto*

fix a — ll 6–8


```

assume  $a: a \in \text{Class } g$ 
then have §:  $a \cdot \text{inverse } g \in \text{Class } 1 \implies \exists h. a = h \cdot g \wedge h \in \text{Class } 1$ 
  by (metis Class-closed associative g inverse-equality invertible invertible-def right-unit)
from  $a \ g$  have  $a \cdot \text{inverse } g \in \text{Class } (a \cdot \text{inverse } g)$  by blast
also from  $a \ g$  have  $\dots = \text{Class } a \ [\cdot] \text{Class } (\text{inverse } g)$ 
  using Class-closed Class-commutes-with-composition invertible
  invertible-inverse-closed by presburger
also from  $a \ g$  have  $\dots = \text{Class } g \ [\cdot] \text{quotient.inverse } (\text{Class } g)$ 
  by (metis Class-commutes-with-inverse Class-eq ClassD)
also from  $g$  have  $\dots = \text{Class } 1$  by simp
finally show  $a \in \text{Class } 1 \mid \cdot g$ 
  using § by (auto simp: Right-Coset-def)
next
fix  $a$  — ll 8–9
assume  $a: a \in \text{Class } 1 \mid \cdot g$ 
then obtain  $k$  where  $\text{eq}: a = k \cdot g$  and  $k: k \in \text{Class } 1$ 
  by blast
with  $g$  have  $\text{Class } a = \text{Class } k \ [\cdot] \text{Class } g$ 
  using Class-commutes-with-composition by auto
also from  $k$  have  $\dots = \text{Class } 1 \ [\cdot] \text{Class } g$ 
  by (metis ClassD Class-eq unit-closed)
also from  $g$  have  $\dots = \text{Class } g$  by simp
finally show  $a \in \text{Class } g$  using  $g \ \text{eq} \ k$  composition-closed quotient.unit-closed by
blast
qed

```

Equation 25 for left cosets

p 55, ll 29–30; p 56, ll 6–11

theorem *Left-Coset-Class-unit*:

assumes $g: g \in G$ **shows** $g \cdot \mid \text{Normal} = \text{Class } g$

unfolding *Normal-def*

proof *auto*

fix a — ll 6–8

assume $a: a \in \text{Class } g$

then have §: $\text{inverse } g \cdot a \in \text{Class } 1 \implies \exists h. a = g \cdot h \wedge h \in \text{Class } 1$

by (*metis Class-closed associative g inverse-equality invertible invertible-def right-unit*)

from $a \ g$ **have** $\text{inverse } g \cdot a \in \text{Class } (\text{inverse } g \cdot a)$ **by** *blast*

also from $a \ g$ **have** $\dots = \text{Class } (\text{inverse } g) \ [\cdot] \text{Class } a$

using *Class-closed Class-commutes-with-composition invertible*

invertible-inverse-closed **by** *presburger*

also from $a \ g$ **have** $\dots = \text{quotient.inverse } (\text{Class } g) \ [\cdot] \text{Class } g$

by (*metis ClassD Class-commutes-with-inverse Class-eq*)

also from g **have** $\dots = \text{Class } 1$ **by** *simp*

finally show $a \in g \cdot \mid \text{Class } 1$

using § **by** (*auto simp: Left-Coset-def*)

next

fix a — ll 8–9, “the same thing holds”

assume $a: a \in g \cdot \mid \text{Class } 1$

then obtain k **where** $\text{eq}: a = g \cdot k$ **and** $k: k \in \text{Class } 1$

```

    by blast
  with  $g$  have  $\text{Class } a = \text{Class } g \mid \cdot \text{Class } k$ 
    using Class-commutes-with-composition by auto
  also from  $k$  have  $\dots = \text{Class } g \mid \cdot \text{Class } 1$ 
    by (metis ClassD Class-eq unit-closed)
  also from  $g$  have  $\dots = \text{Class } g$  by simp
  finally show  $a \in \text{Class } g$  using  $g \text{ eq } k$  composition-closed quotient.unit-closed by
blast
qed

```

Thm 1.6, statement of first part

p 55, ll 28–29; p 56, ll 12–16

theorem *Class-unit-is-normal:*
normal-subgroup Normal G (·) 1

```

proof –
{
  fix  $g$ 
  assume  $g \in G$ 
  then have  $g \mid \cdot \text{Normal} = \text{Normal} \mid \cdot g$  by (simp add: Right-Coset-Class-unit
Left-Coset-Class-unit)
}
then show ?thesis by (rule Left-equals-Right-coset-implies-normality)
qed

```

sublocale *normal: normal-subgroup Normal G (·) 1*
 by (fact *Class-unit-is-normal*)

end

context *normal-subgroup* **begin**

p 56, ll 16–19

theorem *Left-equals-Right-coset:*

$g \in G \implies g \mid \cdot K = K \mid \cdot g$

proof

assume $[simp]: g \in G$

show $K \mid \cdot g \subseteq g \mid \cdot K$

proof

fix x

assume $x: x \in K \mid \cdot g$

then obtain k where $x = k \cdot g$ and $[simp]: k \in K$ by (auto simp add: *Right-Coset-def*)

then have $x = g \cdot (\text{inverse } g \cdot k \cdot g)$ by (simp add: *associative invertible-right-inverse2*)

also from *normal* have $\dots \in g \mid \cdot K$ by (auto simp add: *Left-Coset-def*)

finally show $x \in g \mid \cdot K$.

qed

next

assume $[simp]: g \in G$

show $g \mid \cdot K \subseteq K \mid \cdot g$

```

proof
  fix  $x$ 
    assume  $x: x \in g \cdot K$ 
      then obtain  $k$  where  $x = g \cdot k$  and  $[simp]: k \in K$  by (auto simp add:
Left-Coset-def)
        then have  $x = (inverse (inverse g) \cdot k \cdot inverse g) \cdot g$  by (simp add: associative
del: invertible-right-inverse)
          also from normal [where  $g = inverse g$ ] have  $\dots \in K \mid \cdot g$  by (auto simp add:
Right-Coset-def)
            finally show  $x \in K \mid \cdot g$  .
        qed
    qed

```

Thm 1.6, second part

p 55, ll 31–32; p 56, ll 20–21

definition $Congruence = \{(a, b). a \in G \wedge b \in G \wedge inverse\ a \cdot b \in K\}$

p 56, ll 21–22

interpretation *right-translations-of-group ..*

p 56, ll 21–22

interpretation *transformation-group translation* ‘ $K\ G$ **rewrites** *Orbit-Relation =* *Congruence*

proof –

```

  interpret transformation-group translation ‘ $K\ G$  ..
  show Orbit-Relation = Congruence
    unfolding Orbit-Relation-def Congruence-def
      by (force simp: invertible-left-inverse2 invertible-right-inverse2 translation-apply
simp del: restrict-apply)
    qed rule

```

p 56, ll 20–21

lemma *CongruenceI*: $\llbracket a = b \cdot k; a \in G; b \in G; k \in K \rrbracket \implies (a, b) \in Congruence$
by (*clarsimp simp: Congruence-def associative inverse-composition-commute*)

p 56, ll 20–21

lemma *CongruenceD*: $(a, b) \in Congruence \implies \exists k \in K. a = b \cdot k$
by (*drule orbit.symmetric*) (*force simp: Congruence-def invertible-right-inverse2*)

“We showed in the last section that the relation we are considering is an equivalence relation in G for any subgroup K of G . We now proceed to show that normality of K ensures that $[...] a \equiv b \pmod{K}$ is a congruence.”

p 55, ll 30–32; p 56, ll 1, 22–28

sublocale *group-congruence* **where** $E = Congruence$ **rewrites** $Normal = K$

proof –

show *group-congruence* $G\ (\cdot)\ \mathbf{1}\ Congruence$

```

proof unfold-locales
  note CongruenceI [intro] CongruenceD [dest]
  fix a g b h
  assume 1:  $(a, g) \in \text{Congruence}$  and 2:  $(b, h) \in \text{Congruence}$ 
  then have G:  $a \in G \ g \in G \ b \in G \ h \in G$  unfolding Congruence-def by clarify+
  from 1 obtain k1 where  $a = g \cdot k1$  and k1:  $k1 \in K$  by blast
  from 2 obtain k2 where  $b = h \cdot k2$  and k2:  $k2 \in K$  by blast
  from G Left-equals-Right-coset have  $K \cdot h = h \cdot K$  by blast
  with k1 obtain k3 where  $c: k1 \cdot h = h \cdot k3$  and k3:  $k3 \in K$ 
    unfolding Left-Coset-def Right-Coset-def by blast
  from G k1 k2 a b have  $a \cdot b = g \cdot k1 \cdot h \cdot k2$  by (simp add: associative)
  also from G k1 k3 c have  $\dots = g \cdot h \cdot k3 \cdot k2$  by (simp add: associative)
  also have  $\dots = (g \cdot h) \cdot (k3 \cdot k2)$  using G k2 k3 by (simp add: associative)
  finally show  $(a \cdot b, g \cdot h) \in \text{Congruence}$  using G k2 k3 by blast
qed
then interpret group-congruence where  $E = \text{Congruence}$  .
show Normal = K
  unfolding Normal-def orbit.Class-def unfolding Congruence-def
  using invertible-inverse-inverse submonoid-inverse-closed by fastforce
qed

lemma set-prod-commute:
  assumes  $S \subseteq G$ 
  shows  $(\text{case-prod}(\cdot)) \text{ ` } (S \times K) = (\text{case-prod}(\cdot)) \text{ ` } (K \times S)$ 
proof –
  have  $\exists x \in K. \exists y \in S. a \cdot b = x \cdot y$  if  $a \in S \ b \in K$  for  $a \ b$ 
    using that assms
  by (metis Left-equals-Right-coset Right-Coset-memE coset-notation.Left-Coset-memI
    subsetD)
  moreover have  $\exists x \in S. \exists y \in K. a \cdot b = x \cdot y$  if  $a \in K \ b \in S$  for  $a \ b$ 
    using that
  by (metis Left-Coset-memE Left-equals-Right-coset assms coset-notation.Right-Coset-memI
    subsetD)
  ultimately show ?thesis
    by (auto simp add: image-iff subset-iff)
qed

p 56, ll 31–32; no infix syntax because it leads to many problems

abbreviation Factor-Group
  where  $\text{Factor-Group} \equiv G / (\text{normal-subgroup.Congruence } K \ G \ (\cdot) \ 1)$ 

end

context Group begin

lemma normal-subgroup-intersection:
  assumes K: normal-subgroup  $K \ G \ (\cdot) \ 1$ 
  and H: Subgroup  $H \ G \ (\cdot) \ 1$ 
  shows normal-subgroup  $(H \cap K) \ H \ (\cdot) \ 1$ 
proof –

```

```

interpret H: Subgroup H G (·) 1 by fact
interpret K: normal-subgroup K G (·) 1 by fact
have HK-G: Subgroup (H ∩ K) G (·) 1
  by (rule subgroup-intersection[OF H K.Subgroup-axioms])
interpret HK: Subgroup H ∩ K H (·) 1
  by (rule subgroup-restrict[OF HK-G H]) auto
show ?thesis
proof
  fix h k
  assume h: h ∈ H and k: k ∈ H ∩ K
  then show H.sub.inverse h · k · h ∈ H ∩ K
    by (metis H.sub H.sub.invertible H.sub-composition-closed
      H.subgroup-inverse-equality H.submonoid-inverse-closed Int-iff K.normal)
qed
qed

end

context normal-subgroup begin

p 56, ll 28–29

theorem Class-unit-normal-subgroup: Class 1 = K
  unfolding Class-def unfolding Congruence-def
  using invertible-inverse-inverse submonoid-inverse-closed by fastforce

The quotient by a normal subgroup is trivial exactly when that subgroup is the
whole carrier. Keeping the representative argument here avoids repeating it for
each client that packages a quotient group.

lemma Factor-Group-eq-singleton-iff:
  Factor-Group = {Class 1} ↔ K = G
proof
  assume factor-group: Factor-Group = {Class 1}
  show K = G
  proof (rule equalityI)
    show K ⊆ G by (rule subset)
    show G ⊆ K
    proof
      fix x
      assume x: x ∈ G
      have class-mem: Class x ∈ Factor-Group
        by (rule Class-in-Partition[OF x])
      have class-eq: Class x = Class 1
        using class-mem factor-group by simp
      have x ∈ Class x by (rule Class-self[OF x])
      then have x ∈ Class 1 using class-eq by simp
      then show x ∈ K using Class-unit-normal-subgroup by simp
    qed
  qed
qed
next

```

```

assume  $KG: K = G$ 
show  $Factor\text{-}Group = \{Class\ 1\}$ 
proof (rule equalityI)
  show  $Factor\text{-}Group \subseteq \{Class\ 1\}$ 
  proof
    fix  $A$ 
    assume  $A: A \in Factor\text{-}Group$ 
    then obtain  $x$  where  $x: x \in G$  and  $A\text{-eq}: A = Class\ x$ 
    using representant-exists by auto
    have  $xK: x \in K$  using  $KG\ x$  by simp
    have  $class\text{-eq}: Class\ x = Class\ 1$ 
    proof (rule Class-eq)
      show  $(x, 1) \in Congruence$ 
      proof (rule CongruenceI)
        show  $x = 1 \cdot x$  by (rule left-unit[OF  $x$ , symmetric])
        show  $x \in G$  by fact
        show  $1 \in G$  by (rule unit-closed)
        show  $x \in K$  by fact
      qed
    qed
    show  $A \in \{Class\ 1\}$  using  $A\text{-eq}\ class\text{-eq}$  by simp
  qed
show  $\{Class\ 1\} \subseteq Factor\text{-}Group$ 
using Class-in-Partition[OF unit-closed] by simp
qed

```

p 56, ll 1–2; p 56, l 29

theorem *Class-is-Left-Coset*:
 $g \in G \implies Class\ g = g \cdot | K$
using *Left-Coset-Class-unit Class-unit-normal-subgroup* **by** *simp*

p 56, l 29

lemma *Left-CosetE*: $\llbracket A \in Factor\text{-}Group; \bigwedge a. a \in G \implies P\ (a \cdot | K) \rrbracket \implies P\ A$
by (*metis Class-is-Left-Coset quotient-ClassE*)

Equation 26

p 56, ll 32–34

theorem *factor-composition* [*simp*]:
 $\llbracket g \in G; h \in G \rrbracket \implies (g \cdot | K) [| (h \cdot | K) = g \cdot h \cdot | K$
using *Class-commutes-with-composition Class-is-Left-Coset* **by** *auto*

p 56, l 35

theorem *factor-unit*:
 $K = 1 \cdot | K$
using *Class-is-Left-Coset Class-unit-normal-subgroup* **by** *blast*

p 56, l 35

```

theorem factor-inverse [simp]:
   $g \in G \implies \text{quotient.inverse } (g \cdot | K) = (\text{inverse } g \cdot | K)$ 
  using Class-commutes-with-inverse Class-is-Left-Coset by auto

lemma Factor-Group-eq-Left-Cosets:
   $\text{Factor-Group} = (\lambda g. g \cdot | K) \text{ ` } G$ 
proof (rule set-eqI)
  fix  $A$ 
  show  $A \in \text{Factor-Group} \longleftrightarrow A \in (\lambda g. g \cdot | K) \text{ ` } G$ 
  proof
    assume  $A \in \text{Factor-Group}$ 
    then obtain  $g$  where  $g \in G \ A = \text{Class } g$ 
    using quotient-ClassE by (metis representant-exists)
    then show  $A \in (\lambda g. g \cdot | K) \text{ ` } G$ 
    using Class-is-Left-Coset by blast
  next
    assume  $A \in (\lambda g. g \cdot | K) \text{ ` } G$ 
    then obtain  $g$  where  $g \in G \ A = g \cdot | K$  by blast
    then have  $A = \text{Class } g$  using Class-is-Left-Coset by simp
    then show  $A \in \text{Factor-Group}$ 
    using Class-in-Partition  $\langle g \in G \rangle$  by simp
  qed
qed

lemma Factor-Group-eq-Right-Cosets:
   $\text{Factor-Group} = (\lambda g. K \cdot | g) \text{ ` } G$ 
  by (simp add: Right-Coset-Class-unit natural.surjective)

lemma Class-cardinality:
  assumes  $g \in G$ 
  shows  $\text{card } (\text{Class } g) = \text{card } K$ 
proof –
  have  $\text{Class } g = g \cdot | K$  using Class-is-Left-Coset assms by simp
  also have  $\dots = K \cdot | g$  using Left-equals-Right-coset assms by simp
  finally show ?thesis using Right-Coset-cardinality assms by simp
qed

lemma card-Factor-Group:
  assumes finite  $G$ 
  shows  $\text{card } (\text{Factor-Group}) = \text{index}$ 
proof –
  have fin-class: finite  $p$  if  $p \in \text{Partition}$  for  $p$ 
    by (metis Class-closed2 assms finite-subset representant-exists that)
  have card-class:  $\text{card } p = \text{card } K$  if  $p \in \text{Partition}$  for  $p$ 
    using Class-cardinality representant-exists that by blast
  have  $\text{card } G = (\sum p \in \text{Partition}. \text{card } p)$ 
    using product-partition [OF partition fin-class] by simp
  also have  $\dots = (\sum p \in \text{Partition}. \text{card } K)$ 
    using card-class by simp
  also have  $\dots = \text{card } K * \text{card } \text{Partition}$ 

```

```

    by simp
  finally have card G = card K * card Partition .
  moreover have card G = card K * index
    using lagrange assms by simp
  moreover have card K > 0
    by (metis Normal-def card-eq-0-iff emptyE fin-class grOI natural.map-closed sub
      sub-unit-closed)
  ultimately have card K * card Partition = card K * index by simp
  with ⟨card K > 0⟩ have card Partition = index
    using nat-mult-eq-cancel1 by simp
  then show ?thesis by simp
qed

lemma quotient-composition-PiE:
  quotient-composition ∈ Factor-Group →E Factor-Group →E Factor-Group
proof -
  have quotient-composition A B ∈ Factor-Group if A ∈ Factor-Group B ∈ Fac-
tor-Group for A B
    using that by simp
  then show ?thesis
    unfolding quotient-composition-def by (auto simp: restrict-PiE-iff)
qed

end

p 57, ll 4-5

locale subgroup-of-abelian-group = subgroup-of-group H G (·) 1 + Abelian-Group G
(·) 1
  for H and G and composition (infixl ⟨·⟩ 70) and unit (⟨1⟩)

p 57, ll 4-5

sublocale subgroup-of-abelian-group ⊆ normal-subgroup H G (·) 1
  using commutative invertible-right-inverse2 by unfold-locales auto

context subgroup-of-abelian-group begin

lemma quotient-abelian: Abelian-Group Factor-Group quotient-composition (Class 1)
proof -
  have comm: quotient-composition X Y = quotient-composition Y X
    if XF: X ∈ Factor-Group and YF: Y ∈ Factor-Group for X Y
  proof -
    obtain a where aG: a ∈ G and Xa: X = Class a
      using representant-exists XF by blast
    obtain b where bG: b ∈ G and Yb: Y = Class b
      using representant-exists YF by blast
    have X [·] Y = Class (a · b)
      using aG bG by (simp add: Xa Yb Class-commutes-with-composition)
    also have ... = Class (b · a)
      using commutative aG bG by presburger
  qed

```



```

    also have ... = Y [.] X
    using aG bG by (simp add: Xa Yb Class-commutes-with-composition)
    finally show ?thesis .
qed
show ?thesis
  unfolding Abelian-Group-def commutative-monoid-def commutative-monoid-axioms-def
  using quotient.Group-axioms quotient.Monoid-axioms comm by auto
qed

end

```

1.8 Homomorphisms

Def 1.6

p 58, l 33; p 59, ll 1–2

```

locale Monoid-homomorphism =
  map  $\eta$   $M$   $M'$  + source: Monoid  $M$  ( $\cdot$ )  $\mathbf{1}$  + target: Monoid  $M'$  ( $\cdot$ )  $\mathbf{1}'$ 
  for  $\eta$  and  $M$  and composition (infixl  $\langle \cdot \rangle$  70) and unit ( $\langle \mathbf{1} \rangle$ )
    and  $M'$  and composition' (infixl  $\langle \cdot' \rangle$  70) and unit' ( $\langle \mathbf{1}' \rangle$ ) +
  assumes commutes-with-composition:  $\llbracket x \in M; y \in M \rrbracket \implies \eta (x \cdot y) = \eta x \cdot' \eta y$ 
    and commutes-with-unit:  $\eta \mathbf{1} = \mathbf{1}'$ 
begin

```

Jacobson notes that *commutes-with-unit* is not necessary for groups, but doesn't make use of that later.

p 58, l 33; p 59, ll 1–2

```

notation source.invertible ( $\langle \text{invertible} \rightarrow [100] 100$ )
notation source.inverse ( $\langle \text{inverse} \rightarrow [100] 100$ )
notation target.invertible ( $\langle \text{invertible}' \rightarrow [100] 100$ )
notation target.inverse ( $\langle \text{inverse}' \rightarrow [100] 100$ )

```

end

p 59, ll 29–30

```

locale Monoid-epimorphism = Monoid-homomorphism + surjective-map  $\eta$   $M$   $M'$ 

```

p 59, l 30

```

locale Monoid-monomorphism = Monoid-homomorphism + injective-map  $\eta$   $M$   $M'$ 

```

p 59, ll 30–31

```

sublocale Monoid-isomorphism  $\subseteq$  Monoid-epimorphism
  by unfold-locales (auto simp: commutes-with-composition commutes-with-unit)

```

p 59, ll 30–31

```

sublocale Monoid-isomorphism  $\subseteq$  Monoid-monomorphism
  by unfold-locales (auto simp: commutes-with-composition commutes-with-unit)

```

context *Monoid-homomorphism* **begin**

p 59, ll 33–34

theorem *invertible-image-lemma*:

assumes *invertible* $a \in M$

shows $\eta \ a \cdot' \eta \ (\text{inverse } a) = \mathbf{1}'$ **and** $\eta \ (\text{inverse } a) \cdot' \eta \ a = \mathbf{1}'$

using *assms commutes-with-composition commutes-with-unit source.inverse-equality*

by *auto (metis source.invertible-inverse-closed source.invertible-left-inverse)*

p 59, l 34; p 60, l 1

theorem *invertible-target-invertible* [*intro, simp*]:

$\llbracket \text{invertible } a; a \in M \rrbracket \implies \text{invertible}' (\eta \ a)$

using *invertible-image-lemma* **by** *blast*

p 60, l 1

theorem *invertible-commutes-with-inverse*:

$\llbracket \text{invertible } a; a \in M \rrbracket \implies \eta \ (\text{inverse } a) = \text{inverse}' (\eta \ a)$

by (*metis invertible-image-lemma map-closed*

source.invertible-inverse-closed target.inverse-equality)

end

p 60, ll 32–34; p 61, l 1

sublocale *Monoid-congruence*

$\subseteq$ *natural: Monoid-homomorphism Class* $M \ (\cdot) \ \mathbf{1} \ M \ / \ E \ ([\cdot]) \ \text{Class} \ \mathbf{1}$

proof

qed (*auto simp add: Class-commutes-with-composition*)

Fundamental Theorem of Homomorphisms of Monoids

p 61, ll 5, 14–16

sublocale *Monoid-homomorphism* $\subseteq$ *image: Submonoid* $\eta \ ' \ M \ M' \ (\cdot') \ \mathbf{1}'$

by *unfold-locales (auto simp: commutes-with-composition [symmetric] commutes-with-unit [symmetric])*

p 61, l 4

locale *Monoid-homomorphism-fundamental* = *Monoid-homomorphism*

begin

p 61, ll 17–18

sublocale *fiber-relation* $\eta \ M \ M' \ ..$

notation *Fiber-Relation* $(\langle E'(-) \rangle)$

p 61, ll 6–7, 18–20

sublocale *Monoid-congruence* **where** $E = E(\eta)$

using *Class-eq*

by *unfold-locates* (rule *Class-equivalence* [THEN *iffD1*],
auto simp: left-closed right-closed commutes-with-composition Fiber-equality)

p 61, ll 7–9

induced denotes Jacobson's $\bar{\eta}$. We have the commutativity of the diagram,
 where *induced* is unique:

compose M induced Class = η

$\llbracket ?\beta \in \text{Partition} \rightarrow_E M'; \text{compose } M ?\beta \text{ Class} = \eta \rrbracket \implies ?\beta = \text{induced}$

.

p 61, l 20

notation *quotient-composition* (**infixl** $\langle [\cdot] \rangle$ 70)

p 61, ll 7–8, 22–25

sublocale *induced: Monoid-homomorphism induced M / E(η)* ($[\cdot]$) *Class* **1** *M' ($\cdot$) 1'*

proof

fix *x y*

assume *x* $\in$ *Partition* **and** *y* $\in$ *Partition*

then show *induced* (*x* $[\cdot]$ *y*) = *induced* *x* $\cdot'$ *induced* *y*

by (*metis commutes-with-composition induced-Fiber-simp*
natural.commutes-with-composition representant-exists
source.composition-closed)

next

show *induced* (*Class* **1**) = **1'**

by (*simp add: commutes-with-unit*)

qed

p 61, ll 9, 26

sublocale *natural: Monoid-epimorphism Class M* ($\cdot$) **1** *M / E(η)* ($[\cdot]$) *Class* **1** ..

p 61, ll 9, 26–27

sublocale *induced: Monoid-monomorphism induced M / E(η)* ($[\cdot]$) *Class* **1** *M' ($\cdot$) 1'*

..

end

p 62, ll 12–13

locale *group-homomorphism* =

Monoid-homomorphism η *G* ($\cdot$) **1** *G'* ($\cdot$) **1'** +

source: Group *G* ($\cdot$) **1** + *target: Group* *G'* ($\cdot$) **1'**

for η **and** *G* **and** *composition* (**infixl** $\langle \cdot \rangle$ 70) **and** *unit* ($\langle \mathbf{1} \rangle$)

and *G'* **and** *composition'* (**infixl** $\langle \cdot' \rangle$ 70) **and** *unit'* ($\langle \mathbf{1}' \rangle$)

begin

p 62, l 13

sublocale image: *Subgroup $\eta \dashv G \dashv G' (\cdot) 1'$*
using *invertible-image-lemma* **by** *unfold-locale auto*

p 62, ll 13–14

definition $Ker = \eta - \{1'\} \cap G$

p 62, ll 13–14

lemma *Ker-equality:*
 $Ker = \{a \mid a. a \in G \wedge \eta a = 1'\}$
unfolding *Ker-def* **by** *auto*

p 62, ll 13–14

lemma *Ker-closed* [*intro, simp*]:
 $a \in Ker \implies a \in G$
unfolding *Ker-def* **by** *simp*

p 62, ll 13–14

lemma *Ker-image* [*intro*]:
 $a \in Ker \implies \eta a = 1'$
unfolding *Ker-def* **by** *simp*

p 62, ll 13–14

lemma *Ker-memI* [*intro*]:
 $\llbracket \eta a = 1'; a \in G \rrbracket \implies a \in Ker$
unfolding *Ker-def* **by** *simp*

p 62, ll 15–16

sublocale *kernel:* *normal-subgroup* $Ker \ G$

proof –

interpret *kernel:* *Submonoid* $Ker \ G$
unfolding *Ker-def* **by** *unfold-locale* (*auto simp: commutes-with-composition commutes-with-unit*)
interpret *kernel:* *Subgroup* $Ker \ G$
by *unfold-locale* (*force intro: source.invertible-right-inverse simp: Ker-image invertible-commutes-with-inverse*)
show *normal-subgroup* $Ker \ G (\cdot) 1$
apply *unfold-locale*
unfolding *Ker-def*
by (*auto simp: commutes-with-composition invertible-image-lemma(2)*)
qed

p 62, ll 17–20

theorem *injective-iff-kernel-unit:*
 $inj-on \ \eta \ G \longleftrightarrow Ker = \{1\}$
proof (*rule Not-eq-iff [THEN iffD1, OF iffI]*)
assume $Ker \neq \{1\}$
then obtain b **where** $b: b \in Ker \ b \neq 1$ **by** *blast*
then have $\eta b = \eta 1$ **by** (*simp add: Ker-image*)

```

    with b show  $\neg \text{inj-on } \eta \ G$  by (meson inj-onD kernel.sub source.unit-closed)
next
  assume  $\neg \text{inj-on } \eta \ G$ 
  then obtain a b where  $a \neq b$  and ab:  $a \in G \ b \in G \ \eta \ a = \eta \ b$  by (meson inj-onI)
  then have  $\text{inverse } a \cdot b \neq 1 \ \eta \ (\text{inverse } a \cdot b) = 1'$ 
    using ab source.invertible-right-inverse2
  by force (metis ab commutes-with-composition invertible-image-lemma(2) source.invertible
source.invertible-inverse-closed)
  then have  $\text{inverse } a \cdot b \in \text{Ker}$  using Ker-memI ab by blast
  then show  $\text{Ker} \neq \{1\}$  using  $\langle \text{inverse } a \cdot b \neq 1 \rangle$  by blast
qed

end

p 62, l 24

locale group-epimorphism = group-homomorphism + Monoid-epimorphism  $\eta \ G \ (\cdot)$ 
1  $G' \ (\cdot) \ 1'$ 

p 62, l 21

locale normal-subgroup-in-kernel =
  group-homomorphism + contained: normal-subgroup  $L \ G \ (\cdot) \ 1$  for  $L +$ 
  assumes subset:  $L \subseteq \text{Ker}$ 
begin

p 62, l 21

notation contained.quotient-composition (infixl  $\langle [\cdot] \rangle$  70)

"homomorphism onto  $G/L$ "

p 62, ll 23–24

sublocale natural: group-epimorphism contained.Class  $G \ (\cdot) \ 1$  contained.Factor-Group
( $[\cdot]$ ) contained.Class  $1 \ ..$ 

p 62, ll 25–26

theorem left-coset-equality:
  assumes eq:  $a \cdot | \ L = b \cdot | \ L$  and [simp]:  $a \in G$  and  $b: b \in G$ 
  shows  $\eta \ a = \eta \ b$ 
proof –
  obtain l where l:  $b = a \cdot l \ l \in L$ 
  by (metis b contained.Class-is-Left-Coset contained.Class-self eq kernel.Left-Coset-memE)
  then have  $\eta \ a = \eta \ a \cdot l \ \eta \ l$  using Ker-image Monoid-homomorphism.commutates-with-composition
subset by auto
  also have  $\dots = \eta \ b$  by (simp add: commutes-with-composition l)
  finally show ?thesis .
qed

 $\bar{\eta}$ 

p 62, ll 26–27

```

definition *induced* = $(\lambda A \in \text{contained.Factor-Group. THE } b. \exists a \in G. a \cdot | L = A \wedge b = \eta a)$

p 62, ll 26–27

lemma *induced-closed* [intro, simp]:

assumes [simp]: $A \in \text{contained.Factor-Group}$ **shows** *induced* $A \in G'$

proof –

obtain a **where** $a: a \in G \ a \cdot | L = A$ **using** *contained.Class-is-Left-Coset contained.Partition-def* **assms** **by** *auto*

have $(\text{THE } b. \exists a \in G. a \cdot | L = A \wedge b = \eta a) \in G'$

apply *(rule theI2)*

using a **by** *(auto intro: left-coset-equality)*

then show *?thesis* **unfolding** *induced-def* **by** *simp*

qed

p 62, ll 26–27

lemma *induced-undefined* [intro, simp]:

$A \notin \text{contained.Factor-Group} \implies \text{induced } A = \text{undefined}$

unfolding *induced-def* **by** *simp*

p 62, ll 26–27

theorem *induced-left-coset-closed* [intro, simp]:

$a \in G \implies \text{induced } (a \cdot | L) \in G'$

using *contained.Class-is-Left-Coset contained.Class-in-Partition* **by** *auto*

p 62, ll 26–27

theorem *induced-left-coset-equality* [simp]:

assumes [simp]: $a \in G$ **shows** *induced* $(a \cdot | L) = \eta a$

proof –

have $(\text{THE } b. \exists a' \in G. a' \cdot | L = a \cdot | L \wedge b = \eta a') = \eta a$

by *(rule the-equality) (auto intro: left-coset-equality)*

then show *?thesis* **unfolding** *induced-def*

using *contained.Class-is-Left-Coset contained.Class-in-Partition* **by** *auto*

qed

p 62, l 27

theorem *induced-Left-Coset-commutes-with-composition* [simp]:

$\llbracket a \in G; b \in G \rrbracket \implies \text{induced } ((a \cdot | L) [:] (b \cdot | L)) = \text{induced } (a \cdot | L) \cdot' \text{induced } (b \cdot | L)$

by *(simp add: commutes-with-composition)*

p 62, ll 27–28

theorem *induced-group-homomorphism*:

group-homomorphism *induced* *contained.Factor-Group* $([:])$ *(contained.Class 1)* G' $(\cdot') \mathbf{1}'$

apply *unfold-locales*

apply *(auto elim!: contained.Left-CosetE simp: commutes-with-composition commutes-with-unit)*

using *contained.factor-unit induced-left-coset-equality* **apply** (*fastforce simp: contained.Class-unit-normal-subgroup*)
done

p 62, l 28

sublocale *induced: group-homomorphism induced contained.Factor-Group ([·]) contained.Class 1 G' (·) 1'*
by (*fact induced-group-homomorphism*)

p 62, ll 28–29

theorem *factorization-lemma: $a \in G \implies \text{compose } G \text{ induced contained.Class } a = \eta_a$*
unfolding *compose-def* **by** (*simp add: contained.Class-is-Left-Coset*)

p 62, ll 29–30

theorem *factorization [simp]: $\text{compose } G \text{ induced contained.Class} = \eta$*
by *rule (simp add: compose-def contained.Class-is-Left-Coset map-undefined)*

Jacobson does not state the uniqueness of *induced* explicitly but he uses it later, for rings, on p 107.

p 62, l 30

theorem *uniqueness:*
assumes *map: $\beta \in \text{contained.Factor-Group} \rightarrow_E G'$*
and *factorization: $\text{compose } G \beta \text{ contained.Class} = \eta$*
shows *$\beta = \text{induced}$*
proof
fix *A*
show *$\beta A = \text{induced } A$*
proof (*cases A ∈ contained.Factor-Group*)
case *True*
then obtain a where *[simp]: $A = \text{contained.Class } a \text{ } a \in G$*
by (*meson contained.representant-exists*)
then have *$\beta (\text{contained.Class } a) = \eta a$* **by** (*metis compose-eq factorization*)
also have *$\dots = \text{induced } (\text{contained.Class } a)$* **by** (*simp add: contained.Class-is-Left-Coset*)
finally show *?thesis* **by** *simp*
qed (*simp add: induced-def PiE-arb [OF map]*)
qed

p 62, l 31

theorem *induced-image:*
 $\text{induced } ' \text{ contained.Factor-Group} = \eta ' G$
by (*metis factorization contained.natural.surjective surj-compose*)

p 62, l 33

interpretation *L: normal-subgroup L Ker*
by *unfold-locals (auto simp: subset, metis kernel.sub kernel.subgroup-inverse-equality contained.normal)*

p 62, ll 31–33

theorem *induced-kernel*:

induced.Ker = Ker / L.Congruence

proof –

have *induced.Ker = { a · l | L | a. a ∈ G ∧ a ∈ Ker }*

unfolding *induced.Ker-equality*

by *simp (metis (opaque-lifting) contained.Class-is-Left-Coset Ker-image Ker-memI*

induced-left-coset-equality contained.Class-in-Partition contained.representant-exists)

also have *... = Ker / L.Congruence*

using *L.Class-is-Left-Coset L.Class-in-Partition*

by *auto (metis L.Class-is-Left-Coset L.representant-exists kernel.sub)*

finally show *?thesis* .

qed

p 62, ll 34–35

theorem *induced-inj-on*:

inj-on induced contained.Factor-Group $\longleftrightarrow$ L = Ker

proof –

have *L = Ker*

if *L.Partition = {L}*

using *that L.Class-Union L.natural.map-closed* **by** *auto*

moreover have *L.Partition $\subseteq$ {Ker} if L = Ker*

by *(metis L.ClassD L.Class-eq L.Normal-def L.representant-exists*

contained.sub-unit-closed insert-iff subsetI that)

moreover have *{Ker} $\subseteq$ L.Partition if L = Ker*

using *that L.Normal-def L.quotient.unit-closed* **by** *auto*

ultimately show *?thesis*

by *(auto simp add: induced.injective-iff-kernel-unit induced-kernel contained.Class-unit-normal-subgroup)*

qed

end

Fundamental Theorem of Homomorphisms of Groups

p 63, l 1

locale *group-homomorphism-fundamental* = *group-homomorphism* **begin**

p 63, l 1

notation *kernel.quotient-composition* (**infixl** $\langle [\cdot] \rangle$ 70)

p 63, l 1

sublocale *normal-subgroup-in-kernel* **where** *L = Ker* **by** *unfold-locales rule*

p 62, ll 36–37; p 63, l 1

induced denotes Jacobson’s $\bar{\eta}$. We have the commutativity of the diagram,
where *induced* is unique:

compose G induced kernel.Class = η

$\llbracket ?\beta \in \text{kernel.Factor-Group} \rightarrow_E G'; \text{compose } G \text{ } ?\beta \text{ kernel.Class} = \eta \rrbracket$
 $\implies ?\beta = \text{induced}$

end

p 63, l 5

locale *group-isomorphism* = *group-homomorphism* + *bijjective-map* η G G' **begin**

p 63, l 5

sublocale *Monoid-isomorphism* η G $(\cdot)$ **1** G' $(\cdot)$ **1'**
by *unfold-locale* (*auto simp: commutes-with-composition*)

p 63, l 6

lemma *inverse-group-isomorphism:*

group-isomorphism (*restrict* (*inv-into* G η) G') G' $(\cdot)$ **1'** G $(\cdot)$ **1**

proof

qed (*use commutes-with-composition commutes-with-unit bij-betw-inverse surjective*
in *auto*)

end

p 63, l 6

definition *isomorphic-as-groups* (**infixl** $\cong_G$ 50)

where $\mathcal{G} \cong_G \mathcal{G}' \longleftrightarrow (\text{let } (G, \text{composition}, \text{unit}) = \mathcal{G}; (G', \text{composition}', \text{unit}') = \mathcal{G}' \text{ in}$
 $(\exists \eta. \text{group-isomorphism } \eta \text{ } G \text{ composition unit } G' \text{ composition}' \text{ unit}'))$

p 63, l 6

lemma *isomorphic-as-groups-symmetric:*

$(G, \text{composition}, \text{unit}) \cong_G (G', \text{composition}', \text{unit}') \implies (G', \text{composition}', \text{unit}') \cong_G (G, \text{composition}, \text{unit})$

by (*simp add: isomorphic-as-groups-def*) (*meson group-isomorphism.inverse-group-isomorphism*)

lemma *isomorphic-as-groups-transitive:*

assumes $(G, \text{composition}, \text{unit}) \cong_G (G', \text{composition}', \text{unit}')$

and $(G', \text{composition}', \text{unit}') \cong_G (G'', \text{composition}'', \text{unit}'')$

shows $(G, \text{composition}, \text{unit}) \cong_G (G'', \text{composition}'', \text{unit}'')$

proof –

obtain η **where** *first:*

group-isomorphism η G *composition* *unit* G' *composition'* *unit'*

using *assms(1)* **unfolding** *isomorphic-as-groups-def* **by** *auto*

obtain ϑ **where** *second:*

group-isomorphism ϑ G' *composition'* *unit'* G'' *composition''* *unit''*

using *assms(2)* **unfolding** *isomorphic-as-groups-def* **by** *auto*

interpret *first: group-isomorphism* η G *composition* *unit* G' *composition'* *unit'* **by**
fact

interpret *second: group-isomorphism* ϑ G' *composition'* *unit'* G'' *composition''* *unit''* **by** *fact*

```

have composite:
  group-isomorphism (restrict ( $\vartheta \circ \eta$ )  $G$ )
     $G$  composition unit  $G''$  composition'' unit''
proof
  show restrict ( $\vartheta \circ \eta$ )  $G \in G \rightarrow_E G''$ 
    using first.graph second.graph by auto
  show restrict ( $\vartheta \circ \eta$ )  $G$  (composition  $x y$ ) =
    composition'' (restrict ( $\vartheta \circ \eta$ )  $G x$ )
      (restrict ( $\vartheta \circ \eta$ )  $G y$ )
    if  $x \in G y \in G$  for  $x y$ 
    using that first.commutes-with-composition second.commutes-with-composition
      first.source.composition-closed by auto
  show restrict ( $\vartheta \circ \eta$ )  $G$  unit = unit''
    using first.commutes-with-unit second.commutes-with-unit first.source.unit-closed
by simp
have bij-betw ( $\vartheta \circ \eta$ )  $G G''$ 
  using first.bijective second.bijective by (rule bij-betw-trans)
then show bij-betw (restrict ( $\vartheta \circ \eta$ )  $G$ )  $G G''$ 
  by (simp add: bij-betw-restrict-eq)
qed
show ?thesis unfolding isomorphic-as-groups-def using composite by auto
qed

```

On groups the two notions of isomorphism coincide: a bijective monoid isomorphism between groups is already a group isomorphism, since *group-homomorphism* adds nothing to *Monoid-homomorphism* beyond the two structures being groups. So isomorphism classes may be formed once at the level of monoids without losing the group-level statement.

```

lemma isomorphic-as-monoids-iff-groups:
  assumes  $G$ : Group  $G$  composition unit and  $G'$ : Group  $G'$  composition' unit'
  shows ( $G$ , composition, unit)  $\cong_M$  ( $G'$ , composition', unit')
     $\longleftrightarrow$  ( $G$ , composition, unit)  $\cong_G$  ( $G'$ , composition', unit')
proof
  assume ( $G$ , composition, unit)  $\cong_M$  ( $G'$ , composition', unit')
  then obtain  $\eta$  where  $\eta$ : Monoid-isomorphism  $\eta$   $G$  composition unit  $G'$  composition'
    unit'
    unfolding isomorphic-as-monoids-def by auto
  interpret  $\eta$ : Monoid-isomorphism  $\eta$   $G$  composition unit  $G'$  composition' unit' by
    fact
  interpret source: Group  $G$  composition unit by (rule  $G$ )
  interpret target: Group  $G'$  composition' unit' by (rule  $G'$ )
  — The Monoid-isomorphism interpretation already supplies the map and bijectivity
    components; only the homomorphism equation is left, and it differs merely in orientation.
  have group-isomorphism  $\eta$   $G$  composition unit  $G'$  composition' unit'
    by unfold-locales (simp-all add:  $\eta$ .commutes-with-composition  $\eta$ .commutes-with-unit)
  then show ( $G$ , composition, unit)  $\cong_G$  ( $G'$ , composition', unit')
    unfolding isomorphic-as-groups-def by auto
next

```

assume $(G, \text{composition}, \text{unit}) \cong_G (G', \text{composition}', \text{unit}')$
then obtain η **where** *group-isomorphism* η G *composition* unit G' *composition'* unit'
unfolding *isomorphic-as-groups-def* **by** *auto*
then interpret η : *group-isomorphism* η G *composition* unit G' *composition'* unit' .
show $(G, \text{composition}, \text{unit}) \cong_M (G', \text{composition}', \text{unit}')$
unfolding *isomorphic-as-monoids-def*
using η .*Monoid-isomorphism-axioms* **by** *auto*
qed

p 63, l 1

sublocale *group-isomorphism* $\subseteq$ *group-epimorphism* ..

p 63, l 1

locale *group-epimorphism-fundamental* = *group-homomorphism-fundamental* + *group-epimorphism*
begin

p 63, ll 1–2

interpretation *image: group-homomorphism induced kernel.Factor-Group ([·]) kernel.Class 1* $(\eta \text{ ' } G) (\cdot') \mathbf{1}'$
by (*simp add: surjective group-homomorphism-fundamental.intro induced-group-homomorphism*)

p 63, ll 1–2

sublocale *image: group-isomorphism induced kernel.Factor-Group ([·]) kernel.Class 1* $(\eta \text{ ' } G) (\cdot') \mathbf{1}'$
using *induced-group-homomorphism*
by *unfold-locale (auto simp: bij-betw-def induced-image induced-inj-on induced.commutes-with-composition)*

end

context *group-homomorphism* **begin**

p 63, ll 5–7

theorem *image-isomorphic-to-factor-group:*

$\exists K$ *composition unit. normal-subgroup* K G $(\cdot) \mathbf{1} \wedge (\eta \text{ ' } G, (\cdot'), \mathbf{1}') \cong_G (G /$
(normal-subgroup.Congruence K G $(\cdot) \mathbf{1}), \text{composition}, \text{unit})$

proof –

interpret *image: group-epimorphism-fundamental* **where** $G' = \eta \text{ ' } G$
by *unfold-locale (auto simp: commutes-with-composition)*
have *group-isomorphism image.induced kernel.Factor-Group ([·]) (kernel.Class 1)*
 $(\eta \text{ ' } G) (\cdot') \mathbf{1}'$..
then have $(\eta \text{ ' } G, (\cdot'), \mathbf{1}') \cong_G (\text{kernel.Factor-Group}, ([·]), \text{kernel.Class } \mathbf{1})$
by (*simp add: isomorphic-as-groups-def*) (*meson group-isomorphism.inverse-group-isomorphism*)
moreover have *normal-subgroup* $\text{Ker } G$ $(\cdot) \mathbf{1}$..
ultimately show *?thesis* **by** *blast*

qed

p 65

The First Isomorphism Theorem states that for any group homomorphism $\eta : G \rightarrow G'$, the quotient $G/\ker(\eta)$ is isomorphic to the image $\text{im}(\eta)$. The isomorphism is given by the induced map $gK \mapsto \eta(g)$, which is well-defined because elements in the same coset of $\ker(\eta)$ have the same image under η .

theorem *first-isomorphism*:

$(\text{kernel.Factor-Group}, \text{kernel.quotient-composition}, \text{kernel.Class } \mathbf{1}) \cong_G (\eta \cdot G, (\cdot), \mathbf{1})$

proof —

— Restrict the codomain to the image to obtain a surjection.

interpret *image: group-epimorphism-fundamental* $\eta \cdot G (\cdot) \mathbf{1} \eta \cdot G (\cdot) \mathbf{1}'$

by *unfold-locales (auto simp: commutes-with-composition)*

— The induced map is an isomorphism from $G/\ker(\eta)$ to $\text{im}(\eta)$.

have *group-isomorphism image.induced* $\text{kernel.Factor-Group } \text{kernel.quotient-composition} (\text{kernel.Class } \mathbf{1}) (\eta \cdot G) (\cdot) \mathbf{1}' ..$

then show *?thesis*

by *(auto simp: isomorphic-as-groups-def)*

qed

end

1.9 Correspondence Theorem and Third Isomorphism Theorem

p 65

The proofs in this section, in their initial form, were generated by Claude Opus 4.6 through Dominic Mulligan's Isabelle Assistant. Then simplified and tidied by LCP.

Let K be a normal subgroup of G and H a subgroup of G containing K . Then H/K is a subgroup of G/K , the map $H \mapsto H/K$ is a bijection between subgroups of G containing K and subgroups of G/K , H is normal in G if and only if H/K is normal in G/K , and $G/H \cong (G/K)/(H/K)$.

locale *normal-subgroup-in-subgroup* =

K : *normal-subgroup* $K \cdot G (\cdot) \mathbf{1} +$

H : *subgroup-of-group* $H \cdot G (\cdot) \mathbf{1}$

for K **and** H **and** G **and** *composition* (**infixl** $\langle \cdot \rangle$ 70) **and** *unit* ($\langle \mathbf{1} \rangle$) **+**

assumes $K\text{-contained}$: $K \subseteq H$

begin

interpretation $K\text{-subgroup-}H$: *Subgroup* $K \cdot H (\cdot) \mathbf{1}$

proof **qed** *(use K-contained in auto)*

K is a normal subgroup of H .

interpretation $K\text{-in-}H$: *normal-subgroup* $K \cdot H (\cdot) \mathbf{1}$

proof

show $\bigwedge g \ k. \llbracket g \in H; k \in K \rrbracket \implies H.\text{sub.inverse } g \cdot k \cdot g \in K$

by *(metis K.normal H.sub H.subgroup-inverse-equality)*

qed

notation $K.\text{quotient-composition}$ (**infixl** $\langle [\cdot] \rangle$ 70)

H/K as a subset of G/K , defined as the image of H under the canonical projection.

definition $\text{Fract-H-K} = K.\text{Class} \text{ ` } H$

One might expect the quotient H/K to be $K.\text{Factor-Group}$ with H in place of G . However, $K.\text{Factor-Group}$ always forms the congruence of K relative to G (not H), so substituting H for the carrier would give $H / K.\text{Congruence}$. Since $K.\text{Congruence}$ is an equivalence on G rather than H , the conditional defining equation of $(/)$ cannot be applied, leaving the term opaque.

The correct formalisation of H/K is $K\text{-in-}H.\text{Factor-Group}$, the partition of H under $K\text{-in-}H.\text{Congruence}$ —the congruence of K viewed as a normal subgroup of H . The proof that Fract-H-K equals $K\text{-in-}H.\text{Factor-Group}$ follows from the fact that $K.\text{Class}$ and $K\text{-in-}H.\text{Class}$ agree on elements of H : both map $h \in H$ to the left coset hK , by $K.\text{Class-is-Left-Coset}$ and $K\text{-in-}H.\text{Class-is-Left-Coset}$.

lemma $H\text{-Class-eq}$: $h \in H \implies K\text{-in-}H.\text{Class } h = K.\text{Class } h$
using $K.\text{Left-Coset-Class-unit}$ $K\text{-in-}H.\text{Left-Coset-Class-unit}$ **by** *force*

lemma $\text{Fract-H-K-is-quotient}$: $\text{Fract-H-K} = K\text{-in-}H.\text{Partition}$
by (*auto simp: Fract-H-K-def K-in-H.Partition-def H-Class-eq*)

lemma Fract-H-K-subset : $\text{Fract-H-K} \subseteq K.\text{Factor-Group}$
unfolding Fract-H-K-def **by** *blast*

lemma Class-H-closed [*intro, simp*]:
 $h \in H \implies K.\text{Class } h \in \text{Fract-H-K}$
unfolding Fract-H-K-def **by** *auto*

lemma $\text{Fract-H-K-unit-closed}$ [*intro, simp*]:
 $K.\text{Class } 1 \in \text{Fract-H-K}$
using $H.\text{sub-unit-closed}$ **by** *auto*

lemma Fract-H-K-memE [*elim*]:
 $\llbracket A \in \text{Fract-H-K}; \bigwedge h. \llbracket h \in H; A = K.\text{Class } h \rrbracket \implies P \rrbracket \implies P$
unfolding Fract-H-K-def **by** *auto*

H/K is a subgroup of G/K .

lemma $\text{Fract-H-K-into-subset-H}$: $h \in \text{Fract-H-K} \implies h \subseteq H$
using $\text{Fract-H-K-is-quotient}$ $K\text{-in-}H.\text{natural.surjective}$ **by** *blast*

theorem $\text{Fract-H-K-subgroup}$: *Subgroup* Fract-H-K $K.\text{Factor-Group}$ ($[\cdot]$) ($K.\text{Class } 1$)
proof (*rule K.quotient.subgroupI*)
show $\text{Fract-H-K} \subseteq K.\text{Factor-Group}$ **by** (*rule Fract-H-K-subset*)
next
fix A

```

assume  $A \in \text{Fract-H-K}$ 
then obtain  $a$  where  $a: a \in H \ A = K.\text{Class } a$  by auto
then show  $K.\text{quotient.invertible } A$ 
  using  $K.\text{Class.invertible } H.\text{sub}$  by auto
show  $K.\text{quotient.inverse } A \in \text{Fract-H-K}$ 
  using  $a$  by (simp add: K.Class-commutes-with-inverse)
qed (use K.Class-commutes-with-composition in fastforce)+

sublocale  $\text{Fract-H-K}: \text{subgroup-of-group } \text{Fract-H-K } K.\text{Factor-Group } ([\cdot]) \ K.\text{Class } \mathbf{1}$ 
by (simp add: Fract-H-K-subgroup K.quotient.Group-axioms subgroup-of-group-def)

 $H$  is normal in  $G$  if and only if  $H/K$  is normal in  $G/K$ .

theorem normal-iff:
   $\text{normal-subgroup } H \ G \ (\cdot) \ \mathbf{1} \longleftrightarrow \text{normal-subgroup } \text{Fract-H-K } K.\text{Factor-Group } ([\cdot])$ 
   $(K.\text{Class } \mathbf{1})$ 
proof
  assume  $\text{normal-subgroup } H \ G \ (\cdot) \ \mathbf{1}$ 
  then interpret  $H\text{-normal}: \text{normal-subgroup } H \ G \ (\cdot) \ \mathbf{1}$  .
  show  $\text{normal-subgroup } \text{Fract-H-K } K.\text{Factor-Group } ([\cdot]) \ (K.\text{Class } \mathbf{1})$ 
proof
  fix  $A \ B$ 
  assume  $A: A \in K.\text{Factor-Group}$  and  $B: B \in \text{Fract-H-K}$ 
  then obtain  $g \ h$  where  $g: g \in G \ A = K.\text{Class } g$ 
    and  $h: h \in H \ B = K.\text{Class } h$ 
    by (meson K.representant-exists Fract-H-K-memE)
  have  $K.\text{quotient.inverse } A = K.\text{Class } (K.\text{inverse } g)$ 
    using  $g \ K.\text{Class-commutes-with-inverse}$  by auto
  then have  $\text{eq}: K.\text{quotient.inverse } A \ [\cdot] \ B \ [\cdot] \ A = K.\text{Class } (K.\text{inverse } g \cdot h \cdot g)$ 
    using  $g \ h \ K.\text{Class-commutes-with-composition}$  by force
  have  $K.\text{inverse } g \cdot h \cdot g \in H$  using  $H\text{-normal.normal } g \ h$  by auto
  then show  $K.\text{quotient.inverse } A \ [\cdot] \ B \ [\cdot] \ A \in \text{Fract-H-K}$ 
    unfolding eq by auto
qed
next
assume  $\text{Fract-H-K-normal}: \text{normal-subgroup } \text{Fract-H-K } K.\text{Factor-Group } ([\cdot]) \ (K.\text{Class } \mathbf{1})$ 
then interpret  $\text{Fract-H-K-normal}: \text{normal-subgroup } \text{Fract-H-K } K.\text{Factor-Group } ([\cdot]) \ K.\text{Class } \mathbf{1}$  .
  show  $\text{normal-subgroup } H \ G \ (\cdot) \ \mathbf{1}$ 
proof
  fix  $g \ h$ 
  assume  $g: g \in G$  and  $h: h \in H$ 
  then have  $A: K.\text{Class } g \in K.\text{Factor-Group}$  and  $B: K.\text{Class } h \in \text{Fract-H-K}$  by auto
  have  $K.\text{quotient.inverse } (K.\text{Class } g) \ [\cdot] \ K.\text{Class } h \ [\cdot] \ K.\text{Class } g$ 
     $= K.\text{Class } (K.\text{inverse } g \cdot h \cdot g)$ 
    using  $K.\text{Class-commutes-with-inverse } K.\text{natural.commutates-with-composition } g \ h$ 
by auto
  with  $\text{Fract-H-K-normal.normal } [OF \ A \ B]$ 
  obtain  $h'$  where  $h': h' \in H \ K.\text{Class } (K.\text{inverse } g \cdot h \cdot g) = K.\text{Class } h'$ 

```

```

    by (metis Fract-H-K-memE)
  then have (K.inverse g · h · g, h') ∈ K.Congruence
    using g h H.sub K.Class-equivalence by auto
  then obtain k where k: k ∈ K K.inverse g · h · g = h' · k
    using K.CongruenceD by auto
  show K.inverse g · h · g ∈ H
    using k h' K-contained H.sub-composition-closed by auto
qed
qed

```

Third Isomorphism Theorem: $G/H \cong (G/K)/(H/K)$

We assume H is normal in G and construct a surjective group homomorphism from G/K to G/H whose kernel is H/K .

theorem *third-isomorphism*:

```

assumes H-normal: normal-subgroup H G (·) 1
defines CongH ≡ normal-subgroup.Congruence H G (·) 1
and CongHK ≡ normal-subgroup.Congruence Fract-H-K K.Factor-Group K.quotient-composition
(K.Class 1)
shows (G / CongH,
  Monoid-congruence.quotient-composition G (·) CongH,
  Equivalence.Class G CongH 1) ≅G
(K.Factor-Group / CongHK,
  Monoid-congruence.quotient-composition K.Factor-Group K.quotient-composition
CongHK,
  Equivalence.Class K.Factor-Group CongHK (K.Class 1))

```

proof —

```

interpret Hn: normal-subgroup H G (·) 1
by (rule H-normal)

```

— The canonical map, sending $K.Class\ g$ to $Hn.Class\ g$; the description operator is discharged by well-definedness below.

```

define  $\varphi$  where  $\varphi = (\lambda A \in K.Factor-Group. THE\ B. \exists g \in G. A = K.Class\ g \wedge B = Hn.Class\ g)$ 

```

```

have  $\varphi$ -eq:  $\varphi\ (K.Class\ g) = Hn.Class\ g$  if  $g: g \in G$  for  $g$ 

```

proof —

```

have unfold:  $\varphi\ (K.Class\ g) = (THE\ B. \exists g' \in G. K.Class\ g = K.Class\ g' \wedge B = Hn.Class\ g')$ 

```

unfolding φ -def **using** g **by** simp

— Well defined because $K \subseteq H$: elements in one K -coset lie in one H -coset.

```

have well-def:  $(THE\ B. \exists g' \in G. K.Class\ g = K.Class\ g' \wedge B = Hn.Class\ g') = Hn.Class\ g$ 

```

proof (rule the-equality)

```

fix B assume  $\exists g' \in G. K.Class\ g = K.Class\ g' \wedge B = Hn.Class\ g'$ 

```

```

with  $g$  obtain  $g' k$  where  $g': g' \in G$  and eq:  $K.Class\ g = K.Class\ g'$  and B:  $B = Hn.Class\ g'$ 

```

```

and  $(g, g') \in K.Congruence$  and  $g = g' \cdot k$  and  $k \in K$ 

```

```

by (meson K.CongruenceD K.Class-equivalence)

```

```

then have  $(g, g') \in Hn.Congruence$ 

```

```

    using g g' K-contained by (auto intro: Hn.CongruenceI)
  with B show B = Hn.Class g
    using Hn.Class-eq by presburger
  qed (use g in blast)
  show  $\varphi (K.Class g) = Hn.Class g$ 
    by (simp add: unfold well-def)
  qed

interpret gh: group-homomorphism  $\varphi$  K.Factor-Group ([·]) K.Class 1 Hn.Factor-Group
Hn.quotient-composition Hn.Class 1
proof
  show  $\varphi \in K.Partition \rightarrow_E Hn.Partition$ 
  proof (rule PiE-I)
    fix A assume A  $\in K.Partition$ 
    then show  $\varphi A \in Hn.Partition$ 
      by (metis Hn.natural.map-closed  $\varphi$ -eq K.representant-exists)
  qed (auto simp:  $\varphi$ -def)
next
fix A B assume A: A  $\in K.Partition$  and B: B  $\in K.Partition$ 
then obtain a b where a: a  $\in G$  and A-eq: A = K.Class a
and b: b  $\in G$  and B-eq: B = K.Class b
  using K.representant-exists by meson
have  $\varphi (A [·] B) = \varphi (K.Class (a \cdot b))$ 
  by (simp add: A-eq B-eq a b K.Class-commutes-with-composition)
also have  $\dots = Hn.Class (a \cdot b)$ 
  using a b by (simp add:  $\varphi$ -eq)
also have  $\dots = Hn.quotient-composition (\varphi A) (\varphi B)$ 
  using A-eq B-eq Hn.natural.commutes-with-composition  $\varphi$ -eq a b by presburger
finally show  $\varphi (A [·] B) = Hn.quotient-composition (\varphi A) (\varphi B)$  .
next
show  $\varphi (K.Class 1) = Hn.Class 1$ 
  by (simp add:  $\varphi$ -eq)
qed

have  $\varphi$ -surj:  $\varphi \text{ ' } K.Factor-Group = Hn.Factor-Group$ 
  using Hn.representant-exists  $\varphi$ -eq by blast

have  $\varphi$ -ker: gh.Ker = Fract-H-K
proof (intro equalityI subsetI)
  fix A assume A-ker: A  $\in gh.Ker$ 
  then have A  $\in K.Factor-Group$  and  $\varphi$ -A:  $\varphi A = Hn.Class 1$ 
    by (auto simp: gh.Ker-def)
  then obtain g where g: g  $\in G$  and A-eq: A = K.Class g
    by (meson K.representant-exists)
  then have (g, 1)  $\in Hn.Congruence$ 
    using g Hn.Class-equivalence  $\varphi$ -A  $\varphi$ -eq by force
  then show A  $\in Fract-H-K$ 
    using A-eq Hn.CongruenceD by blast
next
fix A assume A  $\in Fract-H-K$ 

```



```

then obtain  $h$  where  $h: h \in H$  and  $A\text{-eq}: A = K.\text{Class } h$ 
  by auto
then have  $(h, 1) \in Hn.\text{Congruence}$ 
  using  $Hn.\text{Normal-def } h$  by blast
then have  $\varphi A = Hn.\text{Class } 1$ 
  by (simp add: A-eq Hn.Class-eq  $\varphi$ -eq h)
then show  $A \in gh.\text{Ker}$ 
  using  $\langle A \in \text{Fract-}H\text{-}K \rangle$  by blast
qed
show ?thesis
  unfolding  $\text{CongHK-def CongH-def}$ 
  by (metis  $\varphi$ -ker  $\varphi$ -surj gh.first-isomorphism isomorphic-as-groups-symmetric)
qed

```

end

The product of a subgroup and a normal subgroup is again a subgroup. We keep the construction set-based, in keeping with the rest of this development, and package its closure facts in a locale so that later isomorphism theorems can use the product as an ambient group in its own right.

```

locale normal-subgroup-product =
   $K$ : normal-subgroup  $K$   $G$   $(\cdot)$  1 +
   $H$ : subgroup-of-group  $H$   $G$   $(\cdot)$  1
  for  $K$  and  $H$  and  $G$  and composition (infixl  $\langle \cdot \rangle$  70) and unit ( $\langle 1 \rangle$ )
begin

```

```

definition  $HK$  where  $HK \equiv (\text{case-prod}(\cdot)) \text{ ' } (H \times K)$ 

```

```

lemma  $HK\text{-memI}$ :
   $\llbracket h \in H; k \in K \rrbracket \implies h \cdot k \in HK$ 
  unfolding  $HK\text{-def}$  by blast

```

```

lemma  $HK\text{-memE}$ :
  assumes  $x \in HK$ 
  obtains  $h k$  where  $h \in H$   $k \in K$   $x = h \cdot k$ 
proof –
  from assms obtain  $p$  where  $p: p \in H \times K$   $x = \text{case-prod } (\cdot) p$ 
  unfolding  $HK\text{-def}$  by blast
  obtain  $h k$  where  $p = (h, k)$  by (cases p)
  then have  $h \in H$   $k \in K$   $x = h \cdot k$ 
  using  $p$  by simp-all
  then show thesis by (rule that)
qed

```

```

lemma  $HK\text{-eq-KH}$ :  $HK = (\text{case-prod}(\cdot)) \text{ ' } (K \times H)$ 
  using  $H.\text{subset } HK\text{-def } K.\text{set-prod-commute}$  by presburger

```

```

interpretation  $HK\text{-mon}$ : Monoid  $HK$ 
proof

```

```

fix a b
assume a ∈ HK and b ∈ HK
then have ∃ hb kb. h · k · (ha · ka) = hb · kb ∧ hb ∈ H ∧ kb ∈ K
  if h ∈ H k ∈ K ha ∈ H ka ∈ K for h ha k ka
proof -
  have hG: h ∈ G and haG: ha ∈ G using that H.sub by auto
  have kG: k ∈ G and kaG: ka ∈ G using that K.sub by auto
  have k · ha ∈ (case-prod(·)) ‘ (K × H)
    using that by auto
  then have k · ha ∈ HK
    using HK-eq-KH by simp
  then obtain h' k' where hk': k · ha = h' · k' h' ∈ H k' ∈ K
    by (auto simp: HK-def)
  have h · k · (ha · ka) = h · (k · ha) · ka
    by (simp add: K.associative hG haG kG kaG)
  also have ... = (h · h') · (k' · ka)
    using ⟨k · ha = h' · k'⟩ by (simp add: K.associative hG hk' kaG)
  finally have eq: h · k · (ha · ka) = (h · h') · (k' · ka) .
  then show ?thesis
    using hk' ⟨h ∈ H⟩ ⟨ka ∈ K⟩ by blast
qed
then show a · b ∈ HK
  using HK-def ⟨a ∈ HK⟩ ⟨b ∈ HK⟩ by force
qed (force simp: HK-def)+

```

interpretation HK-sub: Subgroup HK G

proof

```

fix a assume a: a ∈ HK
then show ∧ b. b ∈ HK ⇒ a · b ∈ HK
  by blast
obtain h k where hk: h ∈ H k ∈ K a = k · h
  using HK-eq-KH a by auto
define a' where a' ≡ H.sub.inverse h · K.sub.inverse k
have a' ∈ HK
  using HK-def a'-def hk by auto
have a · a' = 1
  using hk unfolding a'-def
  by (metis H.sub H.subgroup-inverse-equality K.composition-closed K.inverse-composition-commute
    K.invertible K.invertible-right-inverse K.sub K.subgroup-inverse-equality)
moreover have a' · a = 1
  unfolding ⟨a = k · h⟩ a'-def
  by (metis H.sub H.subgroup-inverse-equality K.composition-closed K.inverse-composition-commute
    K.invertible K.invertible-left-inverse K.sub K.subgroup-inverse-equality hk)
moreover have a' ∈ HK
  using hk by (auto simp: a'-def HK-def)
ultimately show Monoid.invertible HK (·) 1 a
  using HK-mon.invertibleI by blast
qed (force simp: HK-def)+

```

lemma HK-subgroup: Subgroup HK G (·) 1

..

lemma *H-subset-HK*: $H \subseteq HK$
unfolding *HK-def*
using *K.sub-unit-closed H.sub* **by** *force*

lemma *K-subset-HK*: $K \subseteq HK$
unfolding *HK-def*
using *H.sub-unit-closed K.sub* **by** *force*

end

locale *second-iso-theorem* = *normal-subgroup-product K H G* ($\cdot$) **1**
for *K* **and** *H* **and** *G* **and** *composition* (*infixl* $\langle \cdot \rangle$ 70) **and** *unit* ($\langle 1 \rangle$)
begin

interpretation *HIntK-Mon*: *Monoid* $H \cap K$
proof *qed auto*

interpretation *HIntKn*: *normal-subgroup* $H \cap K$ *H*
proof
fix *u* :: 'a
assume $u \in H \cap K$
then have *uH*: $u \in H$ **and** *uK*: $u \in K$ **by** *auto*
then have *inv-H*: $H.sub.inverse\ u \in H$
using *H.sub.invertible-inverse-closed* **by** *auto*
have $H.sub.inverse\ u = K.sub.inverse\ u$
by (*metis H.subgroup-inverse-equality K.subgroup-inverse-equality uH uK*)
then show *Monoid.invertible* $(H \cap K)$ ($\cdot$) **1** *u*
using *inv-H uK* **by** *force*
fix *g*
assume $g \in H$
then show $H.sub.inverse\ g \cdot u \cdot g \in H \cap K$
by (*metis H.sub H.sub.invertible H.sub-composition-closed H.subgroup-inverse-equality*
H.submonoid-inverse-closed Int-iff K.normal uH uK)
qed *auto*

Defining the quotient groups as sets of equivalence classes

definition *Fract-HK-K* $\equiv K.Class\ 'HK$

definition *Fract-H-HIntK* $\equiv HIntKn.Class\ 'H$

definition φ **where** $\varphi \equiv (\lambda A \in Fract-HK-K. THE\ B. \exists g \in HK. A = K.Class\ g \wedge B = HIntKn.Class\ g)$

theorem *second-isomorphism*:

shows $(Fract-HK-K, K.quotient-composition, K.Class\ \mathbf{1}) \cong_G$
 $(Fract-H-HIntK, HIntKn.quotient-composition, HIntKn.Class\ \mathbf{1})$

proof –

define η **where** $\eta = (\lambda h \in H. K.Class\ h)$

```

interpret gh: group-homomorphism  $\eta$   $H$   $(\cdot)$  1  $K$ .Factor-Group  $K$ .quotient-composition
K.Class 1
proof
  show  $\eta \in H \rightarrow_E K$ .Partition
    by (simp add:  $\eta$ -def)
next
  fix  $x$   $y$  assume  $xy$ :  $x \in H$   $y \in H$ 
  then show  $\eta (x \cdot y) = K$ .quotient-composition  $(\eta x)$   $(\eta y)$ 
    by (simp add:  $K$ .natural.commutes-with-composition  $\eta$ -def)
next
  show  $\eta$  1 =  $K$ .Class 1
    unfolding  $\eta$ -def using  $H$ .sub-unit-closed by simp
qed

have image-eq:  $\eta \text{ ' } H = \text{Fract-HK-K}$ 
proof (intro equalityI subsetI)
  fix  $A$  assume  $A \in \eta \text{ ' } H$ 
  then obtain  $h$  where  $h$ :  $h \in H$   $A = \eta h$  by auto
  then have  $A = K$ .Class  $h$  unfolding  $\eta$ -def by simp
  moreover have  $h \in HK$ 
    unfolding HK-def using  $h(1)$   $K$ .sub-unit-closed  $H$ .sub
    by (force intro: image-eqI[where  $x=(h, 1)$ ])
  ultimately show  $A \in \text{Fract-HK-K}$ 
    unfolding Fract-HK-K-def by auto
next
  fix  $A$  assume  $A \in \text{Fract-HK-K}$ 
  then obtain  $a$  where  $a$ :  $a \in HK$   $A = K$ .Class  $a$ 
    unfolding Fract-HK-K-def by auto
  then obtain  $h$   $k$  where  $hk$ :  $h \in H$   $k \in K$   $a = h \cdot k$ 
    unfolding HK-def by auto
  have  $K$ .Class  $a = K$ .Class  $h$ 
    using  $hk$   $H$ .sub  $K$ .sub
    by (meson  $K$ .Class-eq  $K$ .CongruenceI  $K$ .composition-closed)
  then have  $A = \eta h$  using  $a$   $hk$  unfolding  $\eta$ -def by simp
  then show  $A \in \eta \text{ ' } H$  using  $hk(1)$  by auto
qed

have ker-eq:  $gh.Ker = H \cap K$ 
proof (intro equalityI subsetI)
  fix  $h$  assume  $h \in gh.Ker$ 
  then have  $hH$ :  $h \in H$  and  $\eta h = K$ .Class 1
    using  $\langle h \in gh.Ker \rangle$   $gh.Ker$ -image by blast+
  then have  $K$ .Class  $h = K$ .Class 1
    unfolding  $\eta$ -def by simp
  then have  $(h, 1) \in K$ .Congruence
    using  $hH$   $H$ .sub  $K$ .Class-equivalence by simp
  then have  $h \in K$ 
    using  $K$ .CongruenceD by blast
  with  $hH$  show  $h \in H \cap K$  by auto
next

```

```

fix  $h$  assume  $h \in H \cap K$ 
then have  $hH$ :  $h \in H$  and  $hK$ :  $h \in K$  by auto
have  $\eta h = K.Class\ h$  unfolding  $\eta$ -def using  $hH$  by simp
also have  $\dots = K.Class\ 1$ 
  by (metis  $K.Class$ -eq  $K.CongruenceI$   $K.left$ -unit  $K.sub$   $K.unit$ -closed  $hK$ )
finally show  $h \in gh.Ker$ 
  using  $hH$  by blast
qed

— The kernel’s congruence and quotient agree with  $HIntKn$ ’s.
— Apply the first isomorphism theorem.
have ( $gh.kernel.Partition$ ,  $gh.kernel.quotient$ -composition,  $gh.kernel.Class\ 1$ )  $\cong_G$ 
  ( $\eta \text{ ‘ } H$ ,  $K.quotient$ -composition,  $K.Class\ 1$ )
  using  $gh.first$ -isomorphism by simp
then have ( $Fract$ - $H$ - $HIntK$ ,  $HIntKn.quotient$ -composition,  $HIntKn.Class\ 1$ )  $\cong_G$ 
  ( $Fract$ - $HK$ - $K$ ,  $K.quotient$ -composition,  $K.Class\ 1$ )
  unfolding image-eq  $Fract$ - $H$ - $HIntK$ -def ker-eq by (simp add:  $HIntKn.Partition$ -def)
then show ?thesis
  by (rule isomorphic-as-groups-symmetric)
qed

```

end

1.10 Dropping from elements of a partition to representatives

This prevents a change of type for inductive proofs that involve quotienting.

```

locale class-representatives =
  fixes  $A$ :: ' $a$  set and  $P$  and composition (infixl  $\langle \cdot \rangle$  70) and unit ( $\langle 1 \rangle$ )
  assumes  $P$ : partition-on  $A\ P$ 
  assumes comp:  $(\cdot) \in P \rightarrow_E P \rightarrow_E P$ 
  assumes unit:  $1 \in P$ 
begin

```

```

definition the-part :: ' $a \Rightarrow 'a$  set where
  the-part  $x \equiv THE\ X. X \in P \wedge x \in X$ 

```

```

definition rep-comp :: ' $a \Rightarrow 'a \Rightarrow 'a$  where
  rep-comp  $\equiv \lambda x \in A. \lambda y \in A. some\_elem\ ((the\_part\ x) \cdot (the\_part\ y))$ 

```

```

lemma some-elem-typing:  $X \in P \implies some\_elem\ X \in A$ 
  by (metis  $P\ UnionI$  partition-on-def some-elem-nonempty)

```

```

lemma the-part-works:
  assumes  $x \in A$  shows  $x \in the\_part\ x$  the-part  $x \in P$ 
proof —
  obtain  $X$  where  $X$ :  $x \in X\ X \in P$ 
  using  $P$  assms by (metis  $UnionE$  partition-on-def)

```

moreover have $Y = X$ **if** $x \in Y$ $Y \in P$ **for** Y
using $P\ X$
by (*metis* (*no-types*, *lifting*) *IntI disjoint-def emptyE partition-on-def that*)
ultimately have $x \in \text{the-part } x \wedge \text{the-part } x \in P$
by (*metis* (*mono-tags*, *lifting*) *theI-unique the-part-def*)
then show $x \in \text{the-part } x$ $\text{the-part } x \in P$
by *auto*
qed

lemma *typing*: $\llbracket x \in A; y \in A \rrbracket \implies \text{some-elem } (\text{the-part } x \cdot \text{the-part } y) \in A$
by (*metis some-elem-typing PiE-E comp the-part-works(2)*)

lemma *unit-typing*: $\text{some-elem } \mathbf{1} \in A$
by (*simp add: some-elem-typing unit*)

lemma *rep-comp-typing*: $\text{rep-comp} \in A \rightarrow_E (A \rightarrow_E A)$
by (*simp add: rep-comp-def typing*)

lemma *the-part-some-elem* [*simp*]:
assumes $X \in P$
shows $\text{the-part } (\text{some-elem } X) = X$
proof –
have *disjoint* P
using P *partition-on-def* **by** *blast*
moreover
have $X \neq \{\}$
using *assms* P *partition-on-def* **by** *blast*
ultimately show *?thesis*
using *assms* **by** (*simp add: pairwise-disjnt-iff some-elem-nonempty the1-equality'*
the-part-def)
qed

lemma *rep-comp*: $\llbracket X \in P; Y \in P \rrbracket \implies \text{rep-comp } (\text{some-elem } X) (\text{some-elem } Y) =$
 $\text{some-elem } (X \cdot Y)$
by (*simp add: rep-comp-def some-elem-typing*)

lemma *Monoid-class-reps*:
assumes *Monoid* P $(\cdot)$ $\mathbf{1}$
shows *Monoid* $(\text{some-elem } \cdot P)$ *rep-comp* $(\text{some-elem } \mathbf{1})$
proof –
interpret *Monoid* P
by (*simp add: assms*)
show *?thesis*
proof **qed** (*use rep-comp associative in auto*)
qed

lemma *Group-class-reps*:
assumes *Group* P $(\cdot)$ $\mathbf{1}$
shows *Group* $(\text{some-elem } \cdot P)$ *rep-comp* $(\text{some-elem } \mathbf{1})$
proof –

```

interpret Group P
  by (simp add: assms)
interpret Mr: Monoid some-elem ' P rep-comp some-elem 1
  by (simp add: Monoid-axioms Monoid-class-reps)
have  $\bigwedge x. x \in P \implies \text{Mr.invertible (some-elem } x)$ 
  by (metis Mr.invertible-def image-eqI invertible invertible-def
    rep-comp)
then show ?thesis
proof unfold-locales qed auto
qed

lemma some-elem-inj-on: inj-on some-elem P
  by (rule inj-on-inverseI [where g = the-part]) simp

lemma some-elem-bij-betw: bij-betw some-elem P (some-elem ' P)
  using some-elem-inj-on by (simp add: bij-betw-def)

lemma Monoid-iso-class-reps:
  assumes Monoid P ( $\cdot$ ) 1
  shows Monoid-isomorphism (restrict some-elem P) P ( $\cdot$ ) 1 (some-elem ' P) rep-comp
    (some-elem 1)
proof -
  interpret Monoid P
    by (simp add: assms)
  show ?thesis
    by unfold-locales
      (auto simp: Monoid-class-reps assms some-elem-bij-betw some-elem-typing rep-comp
        associative
        bij-betw-restrict-eq)
qed

lemma group-iso-class-reps:
  assumes Group P ( $\cdot$ ) 1
  shows group-isomorphism (restrict some-elem P) P ( $\cdot$ ) 1 (some-elem ' P) rep-comp
    (some-elem 1)
proof -
  interpret Monoid-isomorphism restrict some-elem P P ( $\cdot$ ) 1 some-elem ' P rep-comp
    some-elem 1
    by (rule Monoid-iso-class-reps) (simp add: assms Group.axioms)
  show ?thesis
  proof
    show  $\bigwedge u. u \in P \implies \text{invertible } u$ 
      by (simp add: Group.invertible assms)
    show  $\bigwedge u. u \in \text{some-elem ' P} \implies \text{invertible' } u$ 
      by (meson Group.invertible Group-class-reps assms)
  qed
qed

lemma abelian-group-class-reps:
  assumes Abelian-Group P ( $\cdot$ ) 1

```

```

    shows Abelian-Group (some-elem ' P) rep-comp (some-elem 1)
  proof -
    interpret Abelian-Group P (·) 1 by (rule assms)
    have comm: rep-comp (some-elem X) (some-elem Y) = rep-comp (some-elem Y)
      (some-elem X)
      if  $X \in P$   $Y \in P$  for  $X$   $Y$ 
      using that by (simp add: rep-comp commutative)
    show ?thesis
      unfolding Abelian-Group-def commutative-monoid-def commutative-monoid-axioms-def
      using Group-class-reps[OF Group-axioms] Group.axioms comm by auto
  qed

end

```

context *normal-subgroup* **begin**

```

interpretation reps: class-representatives G Factor-Group quotient-composition Class
1
proof
  show partition-on G Factor-Group
    by (rule partition)
  show quotient-composition  $\in$  Factor-Group  $\rightarrow_E$  Factor-Group  $\rightarrow_E$  Factor-Group
    by (rule quotient-composition-PiE)
  show Class 1  $\in$  Factor-Group
    by simp
qed

```

```

theorem group-isomorphism-Factor-Group-class-reps:
  group-isomorphism
    (restrict some-elem Factor-Group) Factor-Group quotient-composition (Class 1)
    (some-elem ' Factor-Group) reps.rep-comp (some-elem (Class 1))
  by (rule reps.group-iso-class-reps) (rule quotient.Group-axioms)

```

```

lemma card-class-reps:
  card (some-elem ' Factor-Group) = card Factor-Group
  using reps.some-elem-inj-on card-image by blast

```

```

lemma class-reps-subset: some-elem ' Factor-Group  $\subseteq$  G
  using reps.some-elem-typing by auto

```

```

lemma abelian-class-reps:
  assumes Abelian-Group Factor-Group quotient-composition (Class 1)
  shows Abelian-Group (some-elem ' Factor-Group) reps.rep-comp (some-elem (Class
1))
  using reps.abelian-group-class-reps[OF assms] .

```

end

1.11 Abstract type of monoids

lemma *trivial-Monoid*: *Monoid* {*undefined*} ($\lambda x y. \text{undefined}$) *undefined*
 by (auto simp: *Monoid-def*)

lemma *trivial-Monoid-invertible*:
Monoid.invertible {*undefined*} ($\lambda x y. \text{undefined}$) *undefined* *undefined*
 by (simp add: *Monoid.unit-invertible trivial-Monoid*)

lemma *trivial-Group*: *Group* {*undefined*} ($\lambda x y. \text{undefined}$) *undefined*
 by (simp add: *Group-axioms-def Group-def trivial-Monoid trivial-Monoid-invertible*)

typedef 'a *monoid* = {(*M*::'a *set*, *f*, *e*). *Monoid M f e*}
morphisms *dest-monoid monoid*
using *trivial-Monoid* **by** *blast*

declare *dest-monoid-inverse* [simp]

definition *trivial-monoid* $\equiv$ *monoid* ({*undefined*}, ($\lambda x y. \text{undefined}$), *undefined*)

definition *mcarrier* **where** *mcarrier m* $\equiv$ *fst (dest-monoid m)*

definition *mmult* **where** *mmult m* $\equiv$ *fst (snd (dest-monoid m))*

definition *munit* **where** *munit m* $\equiv$ *snd (snd (dest-monoid m))*

lemma *monoid-is-Monoid* [iff]: *Monoid (mcarrier m) (mmult m) (munit m)*
 by (metis *Product-Type.Collect-case-prodD dest-monoid mcarrier-def mem-Collect-eq munit-def mmult-def*)

lemma *mmult-assoc* [simp]:
 $\llbracket a \in \text{mcarrier } m; b \in \text{mcarrier } m; c \in \text{mcarrier } m \rrbracket \implies \text{mmult } m (\text{mmult } m a b)$
 $c = \text{mmult } m a (\text{mmult } m b c)$
 by (meson *Monoid.associative monoid-is-Monoid*)

lemma *m-left-unit*: $a \in \text{mcarrier } m \implies \text{mmult } m (\text{munit } m) a = a$
 by (meson *Monoid.left-unit monoid-is-Monoid*)

lemma *m-right-unit*: $a \in \text{mcarrier } m \implies \text{mmult } m a (\text{munit } m) = a$
 by (meson *Monoid.right-unit monoid-is-Monoid*)

lemma (in *Monoid*) *mcarrier-monoid* [simp]:
*mcarrier (monoid (M, ($\cdot$), **1**))* = *M*
 by (simp add: *mcarrier-def Monoid-axioms monoid-inverse*)

lemma (in *Monoid*) *mmult-monoid* [simp]:
*mmult (monoid (M, ($\cdot$), **1**))* = ($\cdot$)
 by (simp add: *mmult-def Monoid-axioms monoid-inverse*)

```

lemma (in Monoid) munit-monoid[simp]:
  munit (monoid (M, (·), 1)) = 1
  by (simp add: munit-def Monoid-axioms monoid-inverse)

lemma monoid-collapse [simp]: monoid (mcarrier m, mmult m, munit m) = m
  by (simp add: mcarrier-def mmult-def munit-def)

Allows reference to the current monoid space within the locale as a value

definition (in Monoid) Self ≡ monoid (M, (·), 1)

lemma (in Monoid) mcarrier-Self [simp]: mcarrier Self = M
  by (simp add: Self-def)

lemma (in Monoid) mmult-Self [simp]: mmult Self = (·)
  by (simp add: Self-def)

lemma (in Monoid) munit-Self [simp]: munit Self = 1
  by (simp add: Self-def)

end

```

2 Partial orders, lattices and complete lattices

```

theory Order-Theory
  imports Set-Theory
begin

```

This theory uses no arithmetic; suppress HOL's $+/-$ notation (as *Ring-Theory* does) so that it composes without ambiguity when imported alongside the ring/group developments, which rebind those tokens to their locale operations.

```

no-notation plus (infixl <+> 65)
no-notation minus (infixl <-> 65)
unbundle no uminus-syntax

```

Carrier-set order theory in the locale idiom of *Group-Theory*: a *Partial-Order* fixes a carrier L and an order relation $\sqsubseteq$ (a binary predicate, with the reflexive / antisymmetric / transitive axioms restricted to L). On top of this we build binary lattices (least upper and greatest lower bounds of pairs) and complete lattices (of arbitrary subsets). Unlike HOL-Algebra's record-based *weak-partial-order* over an equivalence, we fix the relation directly, which keeps the statements close to the type-class order they specialise.

2.1 Partial orders

```

locale Partial-Order =
  fixes L :: 'a set and le :: 'a ⇒ 'a ⇒ bool (infix <⊆> 50)
  assumes reflexive [intro, simp]: a ∈ L ⇒ a ⊆ a

```

and antisymmetric: $\llbracket a \in L; b \in L; a \sqsubseteq b; b \sqsubseteq a \rrbracket \implies a = b$
and transitive [trans]: $\llbracket a \in L; b \in L; c \in L; a \sqsubseteq b; b \sqsubseteq c \rrbracket \implies a \sqsubseteq c$
begin

2.1.1 Bounds

x is an upper bound of A (a subset of the carrier) if it dominates every element of A .

definition *upper-bound* :: 'a $\Rightarrow$ 'a set $\Rightarrow$ bool
where *upper-bound* x $A \longleftrightarrow x \in L \wedge (\forall a \in A. a \sqsubseteq x)$

definition *lower-bound* :: 'a $\Rightarrow$ 'a set $\Rightarrow$ bool
where *lower-bound* x $A \longleftrightarrow x \in L \wedge (\forall a \in A. x \sqsubseteq a)$

lemma *upper-boundI*:
 $\llbracket x \in L; \bigwedge a. a \in A \implies a \sqsubseteq x \rrbracket \implies \text{upper-bound } x \ A$
unfolding *upper-bound-def* **by** *blast*

lemma *upper-boundD*: $\llbracket \text{upper-bound } x \ A; a \in A \rrbracket \implies a \sqsubseteq x$
unfolding *upper-bound-def* **by** *blast*

lemma *upper-bound-closed*: $\text{upper-bound } x \ A \implies x \in L$
unfolding *upper-bound-def* **by** *blast*

lemma *lower-boundI*:
 $\llbracket x \in L; \bigwedge a. a \in A \implies x \sqsubseteq a \rrbracket \implies \text{lower-bound } x \ A$
unfolding *lower-bound-def* **by** *blast*

lemma *lower-boundD*: $\llbracket \text{lower-bound } x \ A; a \in A \rrbracket \implies x \sqsubseteq a$
unfolding *lower-bound-def* **by** *blast*

lemma *lower-bound-closed*: $\text{lower-bound } x \ A \implies x \in L$
unfolding *lower-bound-def* **by** *blast*

2.1.2 Suprema and infima

s is a least upper bound (supremum) of A : an upper bound below every upper bound.

definition *is-sup* :: 'a $\Rightarrow$ 'a set $\Rightarrow$ bool
where *is-sup* s $A \longleftrightarrow \text{upper-bound } s \ A \wedge (\forall x. \text{upper-bound } x \ A \longrightarrow s \sqsubseteq x)$

definition *is-inf* :: 'a $\Rightarrow$ 'a set $\Rightarrow$ bool
where *is-inf* i $A \longleftrightarrow \text{lower-bound } i \ A \wedge (\forall x. \text{lower-bound } x \ A \longrightarrow x \sqsubseteq i)$

lemma *is-supI*:
 $\llbracket \text{upper-bound } s \ A; \bigwedge x. \text{upper-bound } x \ A \implies s \sqsubseteq x \rrbracket \implies \text{is-sup } s \ A$
unfolding *is-sup-def* **by** *blast*

lemma *is-sup-upper*: $\text{is-sup } s \ A \implies \text{upper-bound } s \ A$

unfolding *is-sup-def* **by** *blast*

lemma *is-sup-least*: $\llbracket \text{is-sup } s \ A; \text{upper-bound } x \ A \rrbracket \implies s \sqsubseteq x$
unfolding *is-sup-def* **by** *blast*

lemma *is-infI*:

$\llbracket \text{lower-bound } i \ A; \bigwedge x. \text{lower-bound } x \ A \implies x \sqsubseteq i \rrbracket \implies \text{is-inf } i \ A$
unfolding *is-inf-def* **by** *blast*

lemma *is-inf-lower*: $\text{is-inf } i \ A \implies \text{lower-bound } i \ A$
unfolding *is-inf-def* **by** *blast*

lemma *is-inf-greatest*: $\llbracket \text{is-inf } i \ A; \text{lower-bound } x \ A \rrbracket \implies x \sqsubseteq i$
unfolding *is-inf-def* **by** *blast*

Suprema and infima are unique when they exist (antisymmetry).

lemma *is-sup-unique*:

assumes *is-sup* $s \ A$ **and** *is-sup* $s' \ A$
shows $s = s'$

proof (*rule antisymmetric*)

show $s \in L$ **using** *assms*(1) *is-sup-upper upper-bound-closed* **by** *blast*
show $s' \in L$ **using** *assms*(2) *is-sup-upper upper-bound-closed* **by** *blast*
show $s \sqsubseteq s'$ **using** *assms* **by** (*blast dest: is-sup-least is-sup-upper*)
show $s' \sqsubseteq s$ **using** *assms* **by** (*blast dest: is-sup-least is-sup-upper*)

qed

lemma *is-inf-unique*:

assumes *is-inf* $i \ A$ **and** *is-inf* $i' \ A$
shows $i = i'$

proof (*rule antisymmetric*)

show $i \in L$ **using** *assms*(1) *is-inf-lower lower-bound-closed* **by** *blast*
show $i' \in L$ **using** *assms*(2) *is-inf-lower lower-bound-closed* **by** *blast*
show $i \sqsubseteq i'$ **using** *assms* **by** (*blast dest: is-inf-greatest is-inf-lower*)
show $i' \sqsubseteq i$ **using** *assms* **by** (*blast dest: is-inf-greatest is-inf-lower*)

qed

end

2.2 Lattices

A lattice is a partial order in which every pair of elements has a supremum (join) and an infimum (meet).

locale *Lattice* = *Partial-Order* +

assumes *ex-sup*: $\llbracket a \in L; b \in L \rrbracket \implies \exists s. \text{is-sup } s \ \{a, b\}$
and *ex-inf*: $\llbracket a \in L; b \in L \rrbracket \implies \exists i. \text{is-inf } i \ \{a, b\}$

begin

definition *join* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\sqcup$ 65)
where $a \sqcup b = (\text{THE } s. \text{is-sup } s \ \{a, b\})$

definition *meet* :: 'a $\Rightarrow$ 'a $\Rightarrow$ 'a (**infixl** $\sqcap$ 70)
where $a \sqcap b = (\text{THE } i. \text{is-inf } i \{a, b\})$

lemma *is-sup-join*:

assumes $a \in L \ b \in L$ **shows** *is-sup* $(a \sqcup b) \{a, b\}$
unfolding *join-def* **by** (*rule theI'*) (*use ex-sup[OF assms] is-sup-unique in blast*)

lemma *is-inf-meet*:

assumes $a \in L \ b \in L$ **shows** *is-inf* $(a \sqcap b) \{a, b\}$
unfolding *meet-def* **by** (*rule theI'*) (*use ex-inf[OF assms] is-inf-unique in blast*)

lemma *join-closed* [*intro, simp*]: $\llbracket a \in L; b \in L \rrbracket \Longrightarrow a \sqcup b \in L$
using *upper-bound-closed[OF is-sup-upper[OF is-sup-join]]* .

lemma *meet-closed* [*intro, simp*]: $\llbracket a \in L; b \in L \rrbracket \Longrightarrow a \sqcap b \in L$
using *lower-bound-closed[OF is-inf-lower[OF is-inf-meet]]* .

The join dominates its arguments and is least among common upper bounds.

lemma *join-upper1*: $\llbracket a \in L; b \in L \rrbracket \Longrightarrow a \sqsubseteq a \sqcup b$
using *is-sup-join is-sup-upper* **by** (*blast dest: upper-boundD*)

lemma *join-upper2*: $\llbracket a \in L; b \in L \rrbracket \Longrightarrow b \sqsubseteq a \sqcup b$
using *is-sup-join is-sup-upper* **by** (*blast dest: upper-boundD*)

lemma *join-least*: $\llbracket a \in L; b \in L; a \sqsubseteq c; b \sqsubseteq c; c \in L \rrbracket \Longrightarrow a \sqcup b \sqsubseteq c$
using *is-sup-join* **by** (*blast intro: is-sup-least upper-boundI*)

lemma *meet-lower1*: $\llbracket a \in L; b \in L \rrbracket \Longrightarrow a \sqcap b \sqsubseteq a$
using *is-inf-meet is-inf-lower* **by** (*blast dest: lower-boundD*)

lemma *meet-lower2*: $\llbracket a \in L; b \in L \rrbracket \Longrightarrow a \sqcap b \sqsubseteq b$
using *is-inf-meet is-inf-lower* **by** (*blast dest: lower-boundD*)

lemma *meet-greatest*: $\llbracket a \in L; b \in L; c \sqsubseteq a; c \sqsubseteq b; c \in L \rrbracket \Longrightarrow c \sqsubseteq a \sqcap b$
using *is-inf-meet* **by** (*blast intro: is-inf-greatest lower-boundI*)

2.2.1 Lattice laws

lemma *join-commutative*:

assumes $a \in L \ b \in L$ **shows** $a \sqcup b = b \sqcup a$

proof (*rule is-sup-unique[OF is-sup-join[OF assms]]*)

show *is-sup* $(b \sqcup a) \{a, b\}$ **using** *is-sup-join[OF assms(2,1)]* **by** (*simp add: insert-commute*)

qed

lemma *meet-commutative*:

assumes $a \in L \ b \in L$ **shows** $a \sqcap b = b \sqcap a$

proof (*rule is-inf-unique[OF is-inf-meet[OF assms]]*)

show *is-inf* ($b \sqcap a$) { a, b } **using** *is-inf-meet*[*OF* *assms*(2,1)] **by** (*simp* *add*: *insert-commute*)
qed

lemma *join-idempotent* [*simp*]: $a \in L \implies a \sqcup a = a$
by (*rule antisymmetric*) (*auto simp*: *join-upper1 join-least*)

lemma *meet-idempotent* [*simp*]: $a \in L \implies a \sqcap a = a$
by (*rule antisymmetric*) (*auto simp*: *meet-lower1 meet-greatest*)

Absorption laws.

lemma *join-absorb*: $\llbracket a \in L; b \in L \rrbracket \implies a \sqcup (a \sqcap b) = a$
by (*rule antisymmetric*) (*auto simp*: *join-upper1 join-least meet-lower1*)

lemma *meet-absorb*: $\llbracket a \in L; b \in L \rrbracket \implies a \sqcap (a \sqcup b) = a$
by (*rule antisymmetric*) (*auto simp*: *meet-lower1 meet-greatest join-upper1*)

The order is recovered from the join (equivalently the meet).

lemma *le-iff-join*:
assumes $a: a \in L$ **and** $b: b \in L$ **shows** $(a \sqsubseteq b) = (a \sqcup b = b)$
proof
assume $ab: a \sqsubseteq b$
show $a \sqcup b = b$
proof (*rule antisymmetric*)
show $a \sqcup b \in L$ **using** $a\ b$ **by** *simp*
show $b \in L$ **by** (*rule b*)
show $a \sqcup b \sqsubseteq b$ **using** *join-least*[*OF* $a\ b\ ab$] b **by** *simp*
show $b \sqsubseteq a \sqcup b$ **by** (*rule join-upper2*[*OF* $a\ b$])
qed
next
assume $a \sqcup b = b$
then show $a \sqsubseteq b$ **using** *join-upper1*[*OF* $a\ b$] **by** *simp*
qed
end

2.3 Complete lattices

A complete lattice: every subset of the carrier has a supremum and an infimum.

locale *Complete-Lattice* = *Partial-Order* +
assumes *ex-Sup*: $A \subseteq L \implies \exists s. \text{is-sup } s\ A$
and *ex-Inf*: $A \subseteq L \implies \exists i. \text{is-inf } i\ A$
begin

definition *Sup* :: ' a set $\Rightarrow$ ' a ($\sqcup$)'
where $\sqcup\ A = (\text{THE } s. \text{is-sup } s\ A)$

definition *Inf* :: ' a set $\Rightarrow$ ' a ($\sqcap$)'

where $\sqcap A = (THE\ i.\ is-inf\ i\ A)$

lemma *is-sup-Sup*: $A \subseteq L \implies is-sup (\sqcup A) A$
unfolding *Sup-def* **by** (rule *theI'*) (use *ex-Sup is-sup-unique* **in** *blast*)

lemma *is-inf-Inf*: $A \subseteq L \implies is-inf (\sqcap A) A$
unfolding *Inf-def* **by** (rule *theI'*) (use *ex-Inf is-inf-unique* **in** *blast*)

lemma *Sup-closed* [*intro, simp*]: $A \subseteq L \implies \sqcup A \in L$
using *is-sup-Sup is-sup-upper upper-bound-closed* **by** *blast*

lemma *Inf-closed* [*intro, simp*]: $A \subseteq L \implies \sqcap A \in L$
using *is-inf-Inf is-inf-lower lower-bound-closed* **by** *blast*

lemma *Sup-upper*: $\llbracket A \subseteq L; a \in A \rrbracket \implies a \sqsubseteq \sqcup A$
using *is-sup-Sup is-sup-upper* **by** (blast dest: *upper-boundD*)

lemma *Sup-least*: $\llbracket A \subseteq L; upper-bound\ x\ A \rrbracket \implies \sqcup A \sqsubseteq x$
using *is-sup-Sup* **by** (blast dest: *is-sup-least*)

lemma *Inf-lower*: $\llbracket A \subseteq L; a \in A \rrbracket \implies \sqcap A \sqsubseteq a$
using *is-inf-Inf is-inf-lower* **by** (blast dest: *lower-boundD*)

lemma *Inf-greatest*: $\llbracket A \subseteq L; lower-bound\ x\ A \rrbracket \implies x \sqsubseteq \sqcap A$
using *is-inf-Inf* **by** (blast dest: *is-inf-greatest*)

Top and bottom: the sup and inf of the whole carrier.

definition *top* :: 'a ($\langle \top \rangle$) **where** $\top = \sqcup L$

definition *bottom* :: 'a ($\langle \bot \rangle$) **where** $\bot = \sqcap L$

lemma *top-closed* [*intro, simp*]: $\top \in L$ **unfolding** *top-def* **by** *simp*

lemma *bottom-closed* [*intro, simp*]: $\bot \in L$ **unfolding** *bottom-def* **by** *simp*

lemma *top-greatest*: $a \in L \implies a \sqsubseteq \top$
unfolding *top-def* **using** *Sup-upper* **by** *blast*

lemma *bottom-least*: $a \in L \implies \bot \sqsubseteq a$
unfolding *bottom-def* **using** *Inf-lower* **by** *blast*

A complete lattice is a lattice (binary sup/inf are the two-element cases).

sublocale *Lattice*

proof

fix *a b* **assume** $a \in L\ b \in L$

then have $\{a, b\} \subseteq L$ **by** *simp*

then show $\exists s.\ is-sup\ s\ \{a, b\}$ **and** $\exists i.\ is-inf\ i\ \{a, b\}$

using *ex-Sup ex-Inf* **by** *blast+*

qed

end

2.4 Isotone maps

An isotone (order-preserving, monotone) map between two partial orders.

definition *isotone* :: 'a set $\Rightarrow$ ('a $\Rightarrow$ 'a $\Rightarrow$ bool) $\Rightarrow$ 'b set $\Rightarrow$ ('b $\Rightarrow$ 'b $\Rightarrow$ bool) $\Rightarrow$ ('a $\Rightarrow$ 'b) $\Rightarrow$ bool

where *isotone* *LX leX LY leY f* $\longleftrightarrow$
 $f \in LX \rightarrow LY \wedge (\forall x \in LX. \forall y \in LX. leX\ x\ y \longrightarrow leY\ (f\ x)\ (f\ y))$

lemma *isotoneI*:

$\llbracket f \in LX \rightarrow LY; \bigwedge x\ y. \llbracket x \in LX; y \in LX; leX\ x\ y \rrbracket \Longrightarrow leY\ (f\ x)\ (f\ y) \rrbracket$
 $\Longrightarrow isotone\ LX\ leX\ LY\ leY\ f$

unfolding *isotone-def* **by** *blast*

lemma *isotone-closed*: $\llbracket isotone\ LX\ leX\ LY\ leY\ f; x \in LX \rrbracket \Longrightarrow f\ x \in LY$

unfolding *isotone-def* **by** *blast*

lemma *isotone-le*: $\llbracket isotone\ LX\ leX\ LY\ leY\ f; x \in LX; y \in LX; leX\ x\ y \rrbracket \Longrightarrow leY\ (f\ x)\ (f\ y)$

unfolding *isotone-def* **by** *blast*

2.5 Galois connections

A (monotone) Galois connection between two partial orders X and Y : a lower adjoint $f : X \rightarrow Y$ and an upper adjoint $g : Y \rightarrow X$ linked by the defining adjunction $f\ x \sqsubseteq_Y y \longleftrightarrow x \sqsubseteq_X g\ y$. This is the abstract backbone of correspondences such as the Galois correspondence between subgroups and intermediate fields.

locale *Galois-Connection* =

X: *Partial-Order* *LX leX* + *Y*: *Partial-Order* *LY leY*

for *LX* **and** *leX* (**infix** $\langle \sqsubseteq_X \rangle$ 50) **and** *LY* **and** *leY* (**infix** $\langle \sqsubseteq_Y \rangle$ 50) +

fixes $f :: 'a \Rightarrow 'b$ **and** $g :: 'b \Rightarrow 'a$

assumes *f-closed*: $f \in LX \rightarrow LY$

and *g-closed*: $g \in LY \rightarrow LX$

and *galois*: $\llbracket x \in LX; y \in LY \rrbracket \Longrightarrow f\ x \sqsubseteq_Y y \longleftrightarrow x \sqsubseteq_X g\ y$

begin

lemma *f-in*: $x \in LX \Longrightarrow f\ x \in LY$ **using** *f-closed* **by** *auto*

lemma *g-in*: $y \in LY \Longrightarrow g\ y \in LX$ **using** *g-closed* **by** *auto*

Unit and counit of the adjunction: $x \sqsubseteq g\ (f\ x)$ and $f\ (g\ y) \sqsubseteq y$.

lemma *unit*: $x \in LX \Longrightarrow x \sqsubseteq_X g\ (f\ x)$

using *galois*[*of* $x\ f\ x$] *f-in* **by** *simp*

lemma *counit*: $y \in LY \Longrightarrow f\ (g\ y) \sqsubseteq_Y y$

using *galois*[*of* $g\ y\ y$] *g-in* **by** *simp*

Both adjoints are isotone.

lemma *lower-isotone*: *isotone* *LX leX LY leY f*

proof (*rule isotoneI[OF f-closed]*)
fix $x\ x'$ **assume** $x: x \in LX$ **and** $x': x' \in LX$ **and** $le: x \sqsubseteq_X x'$
 $\text{--- } x \sqsubseteq x' \sqsubseteq g(f x'), \text{ so by the adjunction } f x \sqsubseteq f x'.$
have $x \sqsubseteq_X g(f x')$
using $le\ unit[OF\ x']\ x\ x'\ g\text{-in}\ f\text{-in}$ **by** (*blast intro: X.transitive*)
then show $f x \sqsubseteq_Y f x'$ **using** $galois[OF\ x\ f\text{-in}[OF\ x']]$ **by** *simp*
qed

lemma *upper-isotone: isotone LY leY LX leX g*
proof (*rule isotoneI[OF g-closed]*)
fix $y\ y'$ **assume** $y: y \in LY$ **and** $y': y' \in LY$ **and** $le: y \sqsubseteq_Y y'$
 $\text{--- } f(g y) \sqsubseteq y \sqsubseteq y', \text{ so by the adjunction } g y \sqsubseteq g y'.$
have $f(g y) \sqsubseteq_Y y'$
using $le\ counit[OF\ y]\ y\ y'\ f\text{-in}\ g\text{-in}$ **by** (*blast intro: Y.transitive*)
then show $g y \sqsubseteq_X g y'$ **using** $galois[OF\ g\text{-in}[OF\ y]\ y']$ **by** *simp*
qed

The composite $g \circ f$ is a closure operator on X : expansive ($?x \in LX \implies ?x \sqsubseteq_X g(f ?x)$), isotone, and idempotent. Dually $f \circ g$ is a kernel (interior) operator on Y . The two *semi-inverse* identities below — $f(g(f x)) = f x$ and $g(f(g y)) = g y$ — underlie idempotence.

lemma *lower-semi-inverse: $x \in LX \implies f(g(f x)) = f x$*
proof (*rule Y.antisymmetric*)
assume $x: x \in LX$
show $f(g(f x)) \in LY$ **using** $x\ f\text{-in}\ g\text{-in}$ **by** *blast*
show $f x \in LY$ **using** $x\ f\text{-in}$ **by** *blast*
show $f(g(f x)) \sqsubseteq_Y f x$ **using** $counit[OF\ f\text{-in}[OF\ x]]$.
 $\text{--- } f x \sqsubseteq f(g(f x)) \text{ by applying the lower adjoint's isotonicity to the unit } x \sqsubseteq g(f x).$
have $gfx: g(f x) \in LX$ **using** $x\ f\text{-in}\ g\text{-in}$ **by** *blast*
show $f x \sqsubseteq_Y f(g(f x))$
by (*rule isotone-le[OF lower-isotone x gfx unit[OF x]]*)
qed

lemma *upper-semi-inverse: $y \in LY \implies g(f(g y)) = g y$*
proof (*rule X.antisymmetric*)
assume $y: y \in LY$
show $g(f(g y)) \in LX$ **using** $y\ g\text{-in}\ f\text{-in}$ **by** *blast*
show $g y \in LX$ **using** $y\ g\text{-in}$ **by** *blast*
show $g y \sqsubseteq_X g(f(g y))$ **using** $unit[OF\ g\text{-in}[OF\ y]]$.
have $fgy: f(g y) \in LY$ **using** $y\ g\text{-in}\ f\text{-in}$ **by** *blast*
show $g(f(g y)) \sqsubseteq_X g y$
by (*rule isotone-le[OF upper-isotone fgy y counit[OF y]]*)
qed

Idempotence of the closure operator $g \circ f$ on X .

lemma *closure-idempotent: $x \in LX \implies g(f(g(f x))) = g(f x)$*
using *upper-semi-inverse[OF f-in]* **by** *simp*

Idempotence of the kernel operator $f \circ g$ on Y .

```

lemma kernel-idempotent:  $y \in LY \implies f (g (f (g y))) = f (g y)$ 
  using lower-semi-inverse[OF g-in] by simp

```

```

end

```

```

end

```

3 The lattice of subgroups

```

theory Subgroup-Lattice
  imports Group-Theory Order-Theory
begin

```

The subgroups of a group G , ordered by inclusion, form a complete lattice: the infimum of a family of subgroups is their intersection (taken together with G itself, so the empty family has infimum G), and the supremum is the subgroup generated by their union. This is the first concrete consumer of the abstract order theory of *Order-Theory*.

```

context Group
begin

```

3.1 Intersections of subgroups

The intersection of a nonempty family of subgroups is a subgroup.

```

lemma subgroup-Inter:
  assumes ne:  $\mathcal{A} \neq \{\}$  and sub:  $\bigwedge H. H \in \mathcal{A} \implies \text{Subgroup } H \ G \ (\cdot) \ 1$ 
  shows  $\text{Subgroup } (\bigcap \mathcal{A}) \ G \ (\cdot) \ 1$ 
proof –
  obtain H0 where H0:  $H0 \in \mathcal{A}$  using ne by blast
  interpret H0:  $\text{Subgroup } H0 \ G \ (\cdot) \ 1$  by (rule sub[OF H0])
  show ?thesis
  proof (rule subgroupI)
    show  $\bigcap \mathcal{A} \subseteq G$  using H0 H0.subset by blast
    show  $1 \in \bigcap \mathcal{A}$ 
      using Subgroup-def Submonoid.sub-unit-closed sub by fastforce
  next
    fix g h assume g:  $g \in \bigcap \mathcal{A}$  and h:  $h \in \bigcap \mathcal{A}$ 
    show  $g \cdot h \in \bigcap \mathcal{A}$ 
      using Subgroup-def Submonoid.sub-composition-closed g h sub by fastforce
  next
    fix g assume g:  $g \in \bigcap \mathcal{A}$ 
    then show invertible g using H0.sub H0 by blast
    show inverse g  $\in \bigcap \mathcal{A}$ 
      using H0 H0.sub g invertible sub by (meson Inter-iff Subgroup.subgroup-inverse-iff)
  qed
qed

```

3.2 The complete lattice of subgroups

Carrier: the set of all subgroups of G ; order: inclusion.

definition *subgroups* :: 'a set set
where *subgroups* = $\{H. \text{Subgroup } H \ G \ (\cdot) \ 1\}$

lemma *subgroups-iff* [*simp*]: $H \in \text{subgroups} \longleftrightarrow \text{Subgroup } H \ G \ (\cdot) \ 1$
by (*simp add: subgroups-def*)

Inclusion is a partial order on the subgroups.

interpretation *subgroup-order*: *Partial-Order subgroups* ($\subseteq$)
by *unfold-locales auto*

Every family of subgroups has an infimum (its intersection with G) and a supremum (the subgroup generated by its union), so the subgroups form a complete lattice.

interpretation *subgroup-lattice*: *Complete-Lattice subgroups* ($\subseteq$)

proof (*unfold-locales*)

fix $\mathcal{A} :: 'a \text{ set set}$ **assume** $\mathcal{A}: \mathcal{A} \subseteq \text{subgroups}$

— Supremum: the subgroup generated by the union.

have *s-sub*: $\text{Subgroup } \langle \bigcup \mathcal{A} \rangle \ G \ (\cdot) \ 1$ **by** (*rule Gen-is-subgroup*)

have *subgroup-order.is-sup* $\langle \bigcup \mathcal{A} \rangle \ \mathcal{A}$

proof (*intro subgroup-order.upper-boundI subgroup-order.is-supI*)

show $\langle \bigcup \mathcal{A} \rangle \in \text{subgroups}$ **using** *s-sub* **by** *simp*

fix H **assume** $H: H \in \mathcal{A}$

then interpret $H: \text{Subgroup } H \ G \ (\cdot) \ 1$ **using** $\mathcal{A}$ **by** *auto*

show $H \subseteq \langle \bigcup \mathcal{A} \rangle$ **using** *Gen-incl H H.subset* **by** *blast*

next

fix A **assume** *subgroup-order.upper-bound* $A \ \mathcal{A}$

then have *xsub*: $\text{Subgroup } A \ G \ (\cdot) \ 1$ **and** *dom*: $\bigwedge H. H \in \mathcal{A} \implies H \subseteq A$

by (*auto dest: subgroup-order.upper-boundD simp: subgroup-order.upper-bound-def*)

then show $\langle \bigcup \mathcal{A} \rangle \subseteq A$ **using** *xsub* **by** (*intro Gen-minimal*) *auto*

qed

then show $\exists s. \text{subgroup-order.is-sup } s \ \mathcal{A} ..$

next

fix $\mathcal{A} :: 'a \text{ set set}$ **assume** $\mathcal{A}: \mathcal{A} \subseteq \text{subgroups}$

— Infimum: the intersection of $\mathcal{A}$ together with G (so the empty family gives G).

let $?i = \bigcap (\text{insert } G \ \mathcal{A})$

have *all-sub*: $\bigwedge H. H \in \text{insert } G \ \mathcal{A} \implies \text{Subgroup } H \ G \ (\cdot) \ 1$

using $\mathcal{A}$ *group-self-subgroup* **by** *auto*

have *i-sub*: $\text{Subgroup } ?i \ G \ (\cdot) \ 1$ **by** (*rule subgroup-Inter*) (*use all-sub in auto*)

have *subgroup-order.is-inf* $?i \ \mathcal{A}$

proof (*intro subgroup-order.is-infI*)

show *subgroup-order.lower-bound* $?i \ \mathcal{A}$

using *i-sub subgroup-order.lower-boundI* **by** *fastforce*

next

fix x **assume** *subgroup-order.lower-bound* $x \ \mathcal{A}$

then have *xsub*: $\text{Subgroup } x \ G \ (\cdot) \ 1$ **and** *dom*: $\bigwedge H. H \in \mathcal{A} \implies x \subseteq H$

by (*auto dest: subgroup-order.lower-boundD simp: subgroup-order.lower-bound-def*)

```

interpret  $x$ : Subgroup  $x$   $G$   $(\cdot)$  1 by (rule  $xsub$ )
show  $x \subseteq ?i$  using  $dom\ x.subset$  by  $blast$ 
qed
then show  $\exists i. subgroup\_order.is\_inf\ i\ A$  ..
qed

```

The lattice operations are the familiar ones: meet is intersection, join is the subgroup generated by the union, top is G and bottom is the trivial subgroup $\{1\}$.

```

lemma  $subgroup\_top$ :  $subgroup\_lattice.top = G$ 
proof –
  have  $subgroup\_order.is\_sup\ G\ subgroups$ 
proof (intro  $subgroup\_order.upper\_boundI\ subgroup\_order.is\_supI$ )
  show  $G \in subgroups$  by (simp add:  $group\_self\_subgroup$ )
  fix  $H$  assume  $H \in subgroups$ 
  then show  $H \subseteq G$ 
    using  $Gen\_subgroup\_eq\ Gen\_subset\_carrier$  by  $fastforce$ 
next
  fix  $A$  assume  $subgroup\_order.upper\_bound\ A\ subgroups$ 
  then show  $G \subseteq A$  by (auto simp:  $subgroup\_order.upper\_bound\_def\ group\_self\_subgroup$ )
qed
then show  $?thesis$ 
  unfolding  $subgroup\_lattice.top\_def$ 
  using  $subgroup\_order.is\_sup\_unique\ subgroup\_lattice.is\_sup\_Sup[OF\ subset\_refl]$  by
 $blast$ 
qed

end

end
theory  $Monoid-Iso-Classes$ 
  imports  $Group-Theory$ 
begin

```

4 Isomorphism classes of set-based monoids

NewAlgebra represents a monoid by its carrier, composition, and unit rather than by a record, and *'a monoid* is the type-level wrapper for a valid such triple. Quotienting it by isomorphism gives isomorphism classes.

Classes are formed once, at the level of monoids, rather than separately for each stronger structure. Nothing is lost for groups: *Monoid-isomorphism.target-Group* makes being a group a property of the isomorphism class, and *isomorphic-as-monoids-iff-groups* shows the two notions of isomorphism agree on groups. Where group-ness is needed it is imposed as a constraint, as in *monoid-iso-class-of-monoid-eq-iff-groups* and *monoid-iso-class-eq-trivial-iff* below.

```

definition  $monoid\_iso\_rel$  ::  $'a\ monoid \Rightarrow 'a\ monoid \Rightarrow bool$ 
  where

```

monoid-iso-rel $m\ n \equiv$
 $(mcarrier\ m, mmult\ m, munit\ m) \cong_M (mcarrier\ n, mmult\ n, munit\ n)$

Isomorphism is an equivalence relation on the wrapped monoids, so the quotient below records exactly the invariant needed by Jordan–Hölder.

quotient-type *'a monoid-iso-class* = *'a monoid / monoid-iso-rel*
morphisms *Rep-monoid-iso* *Abs-monoid-iso*
proof (*rule equivpI*)
show *reflp monoid-iso-rel*
by (*rule reflpI*)
(simp add: monoid-iso-rel-def isomorphic-as-monoids-refl [OF monoid-is-Monoid])
show *symp monoid-iso-rel*
by (*rule sympI*) *(simp add: monoid-iso-rel-def isomorphic-as-monoids-symmetric)*
show *transp monoid-iso-rel*
by (*rule transpI*)
(metis monoid-iso-rel-def isomorphic-as-monoids-transitive)
qed

definition *monoid-iso-class-of* :: *'a monoid* $\Rightarrow$ *'a monoid-iso-class*
where *monoid-iso-class-of* $m = Abs-monoid-iso\ m$

lemma *monoid-iso-class-of-eq-iff*:
monoid-iso-class-of $m = monoid-iso-class-of\ n \longleftrightarrow monoid-iso-rel\ m\ n$
unfolding *monoid-iso-class-of-def*
using *monoid-iso-class.abs-eq-iff* [*of m n*] **by** *simp*

lemma *monoid-iso-class-of-eq-iff-isomorphic*:
monoid-iso-class-of $m = monoid-iso-class-of\ n \longleftrightarrow$
 $(mcarrier\ m, mmult\ m, munit\ m) \cong_M (mcarrier\ n, mmult\ n, munit\ n)$
unfolding *monoid-iso-class-of-eq-iff monoid-iso-rel-def* **by** *simp*

The bridge clients use: comparing classes of monoids built from explicit triples is comparing the triples up to isomorphism.

lemma *monoid-iso-class-of-monoid-eq-iff*:
fixes $T\ U :: 'a\ set \times ('a \Rightarrow 'a \Rightarrow 'a) \times 'a$
assumes $T: Monoid\ (fst\ T)\ (fst\ (snd\ T))\ (snd\ (snd\ T))$
and $U: Monoid\ (fst\ U)\ (fst\ (snd\ U))\ (snd\ (snd\ U))$
shows $monoid-iso-class-of\ (monoid\ T) = monoid-iso-class-of\ (monoid\ U) \longleftrightarrow T \cong_M U$
proof –
obtain $M\ composition\ unit$ **where** $T-eq: T = (M, composition, unit)$ **by** (*cases T*)
simp
obtain $M'\ composition'\ unit'$ **where** $U-eq: U = (M', composition', unit')$ **by** (*cases U*) *simp*
interpret $T: Monoid\ M\ composition\ unit$ **using** T **by** (*simp add: T-eq*)
interpret $U: Monoid\ M'\ composition'\ unit'$ **using** U **by** (*simp add: U-eq*)
show *?thesis*
unfolding *monoid-iso-class-of-eq-iff-isomorphic T-eq U-eq* **by** *simp*
qed

For groups the same statement holds with group isomorphism, because the two notions agree there. This is the form the Jordan–Hölder development needs.

lemma *monoid-iso-class-of-monoid-eq-iff-groups*:
fixes $T\ U :: 'a\ set \times ('a \Rightarrow 'a \Rightarrow 'a) \times 'a$
assumes T : *Group* ($\text{fst } T$) ($\text{fst } (\text{snd } T)$) ($\text{snd } (\text{snd } T)$)
and U : *Group* ($\text{fst } U$) ($\text{fst } (\text{snd } U)$) ($\text{snd } (\text{snd } U)$)
shows $\text{monoid-iso-class-of } (\text{monoid } T) = \text{monoid-iso-class-of } (\text{monoid } U) \longleftrightarrow T \cong_G U$
proof –
obtain M *composition unit* **where** $T\text{-eq}$: $T = (M, \text{composition}, \text{unit})$ **by** (*cases* T) *simp*
obtain M' *composition' unit'* **where** $U\text{-eq}$: $U = (M', \text{composition}', \text{unit}')$ **by** (*cases* U) *simp*
have $T\text{-group}$: *Group* M *composition unit* **using** T **by** (*simp add*: $T\text{-eq}$)
have $U\text{-group}$: *Group* M' *composition' unit'* **using** U **by** (*simp add*: $U\text{-eq}$)
have $\text{monoid-iso-class-of } (\text{monoid } T) = \text{monoid-iso-class-of } (\text{monoid } U) \longleftrightarrow T \cong_M U$
by (*simp add*: *Group.axioms*(1) $T\ U$ *monoid-iso-class-of-monoid-eq-iff*)
also have $T \cong_M U \longleftrightarrow T \cong_G U$
unfolding $T\text{-eq } U\text{-eq}$ **by** (*rule isomorphic-as-monoids-iff-groups* [*OF* $T\text{-group } U\text{-group}$])
finally show *?thesis* .
qed

4.1 The trivial isomorphism class

A distinguished class for the one-element monoid lets later developments discard repetition factors without choosing a representative from each quotient. *trivial-monoid* is the canonical representative; every one-element group is isomorphic to it.

definition *trivial-monoid-iso-class* :: *'a monoid-iso-class*
where *trivial-monoid-iso-class* = *monoid-iso-class-of trivial-monoid*

lemma *monoid-iso-class-eq-trivial-iff*:
assumes G : *Group* G *composition unit*
shows $\text{monoid-iso-class-of } (\text{monoid } (G, \text{composition}, \text{unit})) = \text{trivial-monoid-iso-class} \longleftrightarrow G = \{\text{unit}\}$
proof –
interpret G : *Group* G *composition unit* **by** *fact*
have *trivial-group*:
 $\text{Group } (\text{fst } (\{\text{undefined} :: 'a\}, (\lambda x\ y. \text{undefined}), \text{undefined}))$
 $(\text{fst } (\text{snd } (\{\text{undefined} :: 'a\}, (\lambda x\ y. \text{undefined}), \text{undefined})))$
 $(\text{snd } (\text{snd } (\{\text{undefined} :: 'a\}, (\lambda x\ y. \text{undefined}), \text{undefined})))$
by (*simp add*: *trivial-Group*)
have *class-iff*:
 $\text{monoid-iso-class-of } (\text{monoid } (G, \text{composition}, \text{unit})) = \text{trivial-monoid-iso-class} \longleftrightarrow$
 $(G, \text{composition}, \text{unit}) \cong_G (\{\text{undefined} :: 'a\}, (\lambda x\ y. \text{undefined}), \text{undefined})$

```

    by (simp add: assms monoid-iso-class-of-monoid-eq-iff-groups trivial-Group trivial-monoid-def
        trivial-monoid-iso-class-def)
  also have ...  $\longleftrightarrow$   $G = \{\text{unit}\}$ 
proof
  assume iso:  $(G, \text{composition}, \text{unit}) \cong_G (\{\text{undefined} :: 'a\}, (\lambda x y. \text{undefined}), \text{undefined})$ 
  then obtain  $\eta$  where
     $\eta$ : group-isomorphism  $\eta$   $G$  composition unit  $\{\text{undefined} :: 'a\}$   $(\lambda(-::'a) (-::'a)). \text{undefined}$ 
  by (simp add: isomorphic-as-groups-def) blast
  interpret  $\eta$ : group-isomorphism  $\eta$   $G$  composition unit
     $\{\text{undefined} :: 'a\}$   $\lambda(-::'a) (-::'a). \text{undefined}$ 
  show  $G = \{\text{unit}\}$ 
    using  $\eta$ .injective-iff-kernel-unit  $\eta$ .map-closed by blast
next
  assume trivial:  $G = \{\text{unit}\}$ 
  have iso:
    group-isomorphism  $(\lambda-. \text{undefined})$   $G$  composition unit
       $\{\text{undefined} :: 'a\}$   $(\lambda(-::'a) (-::'a). \text{undefined})$ 
  using  $G$ .Group-axioms trivial-Group trivial trivial-Monoid-invertible
  by unfold-locales (auto simp: PiE-def extensional-def)
  show  $(G, \text{composition}, \text{unit}) \cong_G (\{\text{undefined} :: 'a\}, (\lambda x y. \text{undefined}), \text{undefined})$ 
    unfolding isomorphic-as-groups-def using iso by auto
qed
finally show ?thesis .
qed

end
theory Normal-Series
  imports Group-Theory
begin

```

5 Finite normal series

The canonical one-step series has the trivial subgroup at index zero and the full carrier thereafter. Values beyond its declared length are immaterial.

definition *one-step-normal-series-term* ::

$'a \text{ set} \Rightarrow 'a \Rightarrow \text{nat} \Rightarrow 'a \text{ set}$

where

one-step-normal-series-term G unit $i = (\text{if } i = 0 \text{ then } \{\text{unit}\} \text{ else } G)$

A finite normal series is a finite chain of subgroups beginning at the trivial subgroup and ending at the ambient group, with each term normal in the next one. The chain is indexed by the natural numbers from 0 through n ; repeated terms are permitted, so strictness can be added separately when defining composition series.

The definition is deliberately stated with carrier sets and the existing *normal-subgroup* locale. In particular, it does not introduce a second representation of quotient groups or subgroup chains.

```

locale normal-series =
  G: Group G (·) 1
  for G and composition (infixl ⟨·⟩ 70) and unit (⟨1⟩) +
  fixes H :: nat ⇒ 'a set and n :: nat
  assumes bottom: H 0 = {1}
  and top: H n = G
  and term-subgroup:  $\bigwedge i. i \leq n \implies \text{Subgroup } (H\ i) \ G\ (\cdot)\ 1$ 
  and normal-step:
     $\bigwedge i. i < n \implies \text{normal-subgroup } (H\ i) \ (H\ (\text{Suc } i))\ (\cdot)\ 1$ 
begin

```

The factor at index i is packaged as the carrier, operation, and unit of the quotient of $H\ (\text{Suc } i)$ by $H\ i$. The definition is total in the index; the series assumptions provide the group interpretation whenever $i < n$.

```

definition series-factor ::
  nat ⇒ ('a set set × ('a set ⇒ 'a set ⇒ 'a set) × 'a set)
where
  series-factor i =
    (normal-subgroup.Factor-Group (H i) (H (Suc i)) (·) 1,
     Monoid-congruence.quotient-composition
       (H (Suc i)) (·)
       (normal-subgroup.Congruence (H i) (H (Suc i)) (·) 1),
     Equivalence.Class (H (Suc i))
       (normal-subgroup.Congruence (H i) (H (Suc i)) (·) 1) 1)

```

The finite factor sequence is ordered by increasing index.

```

definition series-factors ::
  ('a set set × ('a set ⇒ 'a set ⇒ 'a set) × 'a set) list
where
  series-factors = List.map series-factor [0.. $n$ ]

```

```

lemma length-series-factors:
  length series-factors = n
unfolding series-factors-def by simp

```

```

lemma series-factors-prefix:
  assumes k:  $k \leq n$ 
  shows take k series-factors = List.map series-factor [0.. $k$ ]
  using k unfolding series-factors-def by (simp add: take-map)

```

```

lemma series-factors-nth:
  assumes i:  $i < n$ 
  shows series-factors ! i = series-factor i
  using i unfolding series-factors-def by simp

```



```

lemma series-factor-group:
  assumes  $i: i < n$ 
  shows  $\text{Group } (\text{fst } (\text{series-factor } i))$ 
     $(\text{fst } (\text{snd } (\text{series-factor } i))) (\text{snd } (\text{snd } (\text{series-factor } i)))$ 
proof –
  interpret  $\text{step: normal-subgroup } H \ i \ H \ (\text{Suc } i) \ (\cdot) \ 1$ 
    by  $(\text{rule normal-step}[OF \ i])$ 
  show ?thesis
    unfolding series-factor-def
    using step.quotient.Group-axioms by simp
qed

```

The adjacent terms form a subgroup and are ordered by inclusion.

```

lemma normal-step-subgroup:
  assumes  $i: i < n$ 
  shows  $\text{Subgroup } (H \ i) \ (H \ (\text{Suc } i)) \ (\cdot) \ 1$ 
proof –
  interpret  $\text{step: normal-subgroup } H \ i \ H \ (\text{Suc } i) \ (\cdot) \ 1$ 
    by  $(\text{rule normal-step}[OF \ i])$ 
  show ?thesis by  $(\text{rule step.Subgroup-axioms})$ 
qed

```

```

lemma normal-step-subset:
  assumes  $i: i < n$ 
  shows  $H \ i \subseteq H \ (\text{Suc } i)$ 
proof –
  interpret  $\text{step: normal-subgroup } H \ i \ H \ (\text{Suc } i) \ (\cdot) \ 1$ 
    by  $(\text{rule normal-step}[OF \ i])$ 
  show ?thesis by  $(\text{rule step.subset})$ 
qed

```

```

lemma series-term-subset:
  assumes  $i: i \leq n$ 
  shows  $H \ i \subseteq G$ 
proof –
  interpret  $T: \text{Subgroup } H \ i \ G \ (\cdot) \ 1$ 
    by  $(\text{rule term-subgroup}[OF \ i])$ 
  show ?thesis by  $(\text{rule T.subset})$ 
qed

```

Inclusion propagates through the entire finite chain.

```

lemma normal-series-mono:
  fixes  $i \ j :: \text{nat}$ 
  assumes  $ij: i \leq j$  and  $jn: j \leq n$ 
  shows  $H \ i \subseteq H \ j$ 
  using  $ij$ 
proof  $(\text{induction rule: inc-induct})$ 
  case base
  show  $H \ j \subseteq H \ j$  by simp

```

```

next
case (step k)
have k-lt-n:  $k < n$ 
proof (rule less-le-trans)
  show  $k < j$  by (rule step(2))
  show  $j \leq n$  by fact
qed
have step-k:  $H k \subseteq H (Suc k)$ 
  by (rule normal-step-subset[OF k-lt-n])
have ih:  $H (Suc k) \subseteq H j$ 
  by (rule step(3))
show ?case by (rule subset-trans[OF step-k ih])
qed

```

```

lemma series-bottom-subset:
  assumes  $i: i \leq n$ 
  shows  $H 0 \subseteq H i$ 
  by (rule normal-series-mono[OF zero-le i])

```

Every term in a prefix is a subgroup of the prefix's top term.

```

lemma series-prefix-subgroup:
  assumes  $k: k \leq n$  and  $i: i \leq k$ 
  shows Subgroup (H i) (H k) ( $\cdot$ ) 1
proof -
  have i-n:  $i \leq n$  by (rule le-trans[OF i k])
  have i-subset-k:  $H i \subseteq H k$  by (rule normal-series-mono[OF i k])
  show ?thesis
  proof (rule subgroup-restrict)
    show Subgroup (H i)  $G (\cdot)$  1 by (rule term-subgroup[OF i-n])
    show Subgroup (H k)  $G (\cdot)$  1 by (rule term-subgroup[OF k])
    show  $H i \subseteq H k$  by fact
  qed
qed

```

The terms up to any index form a normal series for that term.

```

lemma prefix-normal-series:
  assumes  $k: k \leq n$ 
  shows normal-series (H k) ( $\cdot$ ) 1 H k
proof -
  have Hk-subgroup: Subgroup (H k)  $G (\cdot)$  1
    by (rule term-subgroup[OF k])
  have Hk-group: Group (H k) ( $\cdot$ ) 1
    by (rule subgroup-imp-Group[OF Hk-subgroup])
  show ?thesis
  proof (intro normal-series.intro)
    show Group (H k) ( $\cdot$ ) 1 by fact
  next
    show normal-series-axioms (H k) ( $\cdot$ ) 1 H k
    proof (intro normal-series-axioms.intro)

```

```

    show  $H\ 0 = \{1\}$  by (rule bottom)
  next
    show  $H\ k = H\ k$  by simp
  next
    show  $\bigwedge i. i \leq k \implies \text{Subgroup } (H\ i) (H\ k) (\cdot) \mathbf{1}$ 
    proof -
      fix  $i$ 
      assume  $i: i \leq k$ 
      show  $\text{Subgroup } (H\ i) (H\ k) (\cdot) \mathbf{1}$ 
        by (rule series-prefix-subgroup[OF  $k\ i$ ])
      qed
    next
      show  $\bigwedge i. i < k \implies \text{normal-subgroup } (H\ i) (H\ (\text{Suc } i)) (\cdot) \mathbf{1}$ 
      proof -
        fix  $i$ 
        assume  $i: i < k$ 
        have  $i-n: i < n$  by (rule less-le-trans[OF  $i\ k$ ])
        show  $\text{normal-subgroup } (H\ i) (H\ (\text{Suc } i)) (\cdot) \mathbf{1}$ 
          by (rule normal-step[OF  $i-n$ ])
        qed
      qed
    qed
  qed

lemma zero-length-implies-trivial:
  assumes  $n = 0$ 
  shows  $G = \{1\}$ 
proof -
  have  $G = H\ 0$ 
    using top assms by simp
  also have  $\dots = \{1\}$ 
    by (rule bottom)
  finally show  $G = \{1\}$  .
qed

end

context Group begin

```

Every group has the canonical one-step normal series from its trivial subgroup to its full carrier. For the trivial group the two terms coincide, as repetitions are permitted in a normal series.

```

lemma one-step-normal-series:
  normal-series  $G\ (\cdot) \mathbf{1}$  (one-step-normal-series-term  $G\ \mathbf{1}$ ) 1
proof (intro normal-series.intro)
  show  $\text{Group } G\ (\cdot) \mathbf{1}$  by (rule Group-axioms)
  show normal-series-axioms  $G\ (\cdot) \mathbf{1}$ 
    (one-step-normal-series-term  $G\ \mathbf{1}$ ) 1
  proof (rule normal-series-axioms.intro)

```

```

show one-step-normal-series-term G 1 0 = {1}
  by (simp add: one-step-normal-series-term-def)
show one-step-normal-series-term G 1 1 = G
  by (simp add: one-step-normal-series-term-def)
show  $\bigwedge i. i \leq 1 \implies$ 
  Subgroup (one-step-normal-series-term G 1 i) G (·) 1
  by (auto simp: one-step-normal-series-term-def le-Suc-eq
    trivial-subgroup group-self-subgroup)
show  $\bigwedge i. i < 1 \implies$ 
  normal-subgroup (one-step-normal-series-term G 1 i)
    (one-step-normal-series-term G 1 (Suc i)) (·) 1
  using trivial-normal-subgroup
  by (simp add: one-step-normal-series-term-def)
qed
qed

end

end

```

6 Finite relative normal chains

```

theory Normal-Chain
  imports Normal-Series Monoid-Iso-Classes HOL-Library.Multiset
begin

```

6.1 Isomorphism classes of normal-series factors

The factor API stores quotient groups as the same carrier–operation–unit triples used throughout *New-Algebra*. We turn the valid factors of an arbitrary normal series into isomorphism classes and then into a multiset. When used with a composition series, these are the corresponding composition-factor classes.

```

context normal-series
begin

```

```

definition series-factor-class :: nat  $\Rightarrow$  'a set monoid-iso-class
  where
    series-factor-class i = monoid-iso-class-of (monoid (series-factor i))

```

The class sequence is ordered by increasing factor index.

```

definition series-factor-classes :: 'a set monoid-iso-class list
  where
    series-factor-classes = List.map series-factor-class [0.. $n$ ]

```

```

definition series-factor-multiset :: 'a set monoid-iso-class multiset
  where
    series-factor-multiset = mset series-factor-classes

```

```

lemma length-series-factor-classes:
  length series-factor-classes = n
  unfolding series-factor-classes-def by simp

lemma series-factor-classes-prefix:
  assumes k:  $k \leq n$ 
  shows take k series-factor-classes = List.map series-factor-class [0..<k]
  using k unfolding series-factor-classes-def by (simp add: take-map)

lemma series-factor-classes-nth:
  assumes i:  $i < n$ 
  shows series-factor-classes ! i = series-factor-class i
  using i unfolding series-factor-classes-def by simp

lemma size-series-factor-multiset:
  size series-factor-multiset = n
  unfolding series-factor-multiset-def using length-series-factor-classes by simp

lemma series-factor-class-eq-iff:
  assumes i:  $i < n$  and j:  $j < n$ 
  shows series-factor-class i = series-factor-class j  $\longleftrightarrow$ 
    series-factor i  $\cong_G$  series-factor j
  unfolding series-factor-class-def
  by (rule monoid-iso-class-of-monoid-eq-iff-groups[OF
    series-factor-group[OF i] series-factor-group[OF j]])

end

```

6.2 Normal factors

A refinement of one step in a normal series starts at an intermediate subgroup rather than at the trivial subgroup. The following locale records exactly the structure shared by such relative chains: finitely many subgroups of one ambient group, with each term normal in its successor.

```

definition normal-factor ::
  'a set  $\Rightarrow$  'a set  $\Rightarrow$  ('a  $\Rightarrow$  'a  $\Rightarrow$  'a)  $\Rightarrow$  'a  $\Rightarrow$ 
    ('a set set  $\times$  ('a set  $\Rightarrow$  'a set  $\Rightarrow$  'a set)  $\times$  'a set)
  where
    normal-factor K H composition unit  $\equiv$ 
      (normal-subgroup.Factor-Group K H composition unit,
        Monoid-congruence.quotient-composition H composition
          (normal-subgroup.Congruence K H composition unit),
        Equivalence.Class H
          (normal-subgroup.Congruence K H composition unit) unit)

definition normal-factor-class ::
  'a set  $\Rightarrow$  'a set  $\Rightarrow$  ('a  $\Rightarrow$  'a  $\Rightarrow$  'a)  $\Rightarrow$  'a  $\Rightarrow$ 
    'a set monoid-iso-class
  where

```

normal-factor-class $K\ H\ composition\ unit =$
monoid-iso-class-of (*monoid* (*normal-factor* $K\ H\ composition\ unit$))

A normal factor is a group whenever its denominator is normal in its numerator. Equality of factor classes can therefore be discharged by an ordinary group isomorphism between the corresponding quotient triples.

lemma *normal-factor-group*:

assumes *normal-subgroup* $K\ H\ composition\ unit$
shows *Group* (*fst* (*normal-factor* $K\ H\ composition\ unit$))
(*fst* (*snd* (*normal-factor* $K\ H\ composition\ unit$)))
(*snd* (*snd* (*normal-factor* $K\ H\ composition\ unit$))))

proof –

interpret N : *normal-subgroup* $K\ H\ composition\ unit$ **by** *fact*
show *?thesis*
unfolding *normal-factor-def* **using** N .*quotient.Group-axioms* **by** *simp*
qed

lemma *normal-factor-class-eq-iff*:

assumes KH : *normal-subgroup* $K\ H\ composition\ unit$
and LM : *normal-subgroup* $L\ M\ composition\ unit$
shows *normal-factor-class* $K\ H\ composition\ unit =$
normal-factor-class $L\ M\ composition\ unit \longleftrightarrow$
normal-factor $K\ H\ composition\ unit \cong_G$ *normal-factor* $L\ M\ composition\ unit$
unfolding *normal-factor-class-def*
by (*rule monoid-iso-class-of-monoid-eq-iff-groups*[*OF*
normal-factor-group[*OF* KH] *normal-factor-group*[*OF* LM]])

A normal factor is trivial precisely when its denominator and numerator coincide. The quotient-level argument is proved once in *Factor-Group-eq-singleton-iff*; the carrier lemma below exposes it through the packaged factor triple, independently of the representative chosen for the trivial isomorphism class.

lemma *normal-factor-carrier-eq-singleton-iff*:

assumes N : *normal-subgroup* $K\ H\ composition\ unit$
shows *fst* (*normal-factor* $K\ H\ composition\ unit$) =
{*snd* (*snd* (*normal-factor* $K\ H\ composition\ unit$)))} $\longleftrightarrow K = H$
by (*simp add: assms normal-factor-def normal-subgroup.Factor-Group-eq-singleton-iff*)

lemma *normal-factor-class-eq-trivial-iff*:

assumes N : *normal-subgroup* $K\ H\ composition\ unit$
shows *normal-factor-class* $K\ H\ composition\ unit =$ *trivial-monoid-iso-class* $\longleftrightarrow K$
 $= H$
unfolding *normal-factor-class-def*
using *assms*
by (*metis monoid-iso-class-eq-trivial-iff normal-factor-carrier-eq-singleton-iff*
normal-factor-group split-pairs)

locale *normal-chain* =

G : *Group* $G\ (\cdot)\ 1$
for G **and** *composition* (**infixl** $\langle \cdot \rangle\ 70$) **and** *unit* ($\langle 1 \rangle$) +

```

fixes  $C :: \text{nat} \Rightarrow 'a \text{ set}$  and  $n :: \text{nat}$ 
assumes term-subgroup:
   $\bigwedge i. i \leq n \implies \text{Subgroup } (C \ i) \ G \ (\cdot) \ 1$ 
and normal-step:
   $\bigwedge i. i < n \implies \text{normal-subgroup } (C \ i) \ (C \ (\text{Suc } i)) \ (\cdot) \ 1$ 
begin

lemma step-subset:
  assumes  $i < n$ 
  shows  $C \ i \subseteq C \ (\text{Suc } i)$ 
proof –
  interpret step: normal-subgroup  $C \ i \ C \ (\text{Suc } i) \ (\cdot) \ 1$ 
  by (rule normal-step[OF assms])
  show ?thesis by (rule step.subset)
qed

lemma chain-mono:
  fixes  $i \ j :: \text{nat}$ 
  assumes ij:  $i \leq j$  and jn:  $j \leq n$ 
  shows  $C \ i \subseteq C \ j$ 
  using ij
proof (induction rule: inc-induct)
  case (step k)
  then show ?case
    using jn order-less-le-trans step-subset by blast
qed auto

definition chain-factor-class ::  $\text{nat} \Rightarrow 'a \text{ set monoid-iso-class}$ 
where
  chain-factor-class  $i \equiv$ 
    normal-factor-class  $(C \ i) \ (C \ (\text{Suc } i)) \ (\cdot) \ 1$ 

lemma prefix-normal-chain:
  assumes  $k \leq n$ 
  shows normal-chain  $G \ (\cdot) \ 1 \ C \ k$ 
proof intro-locales
  show normal-chain-axioms  $G \ (\cdot) \ 1 \ C \ k$ 
    by (simp add: assms le-trans normal-chain-axioms-def normal-step order-less-le-trans
      term-subgroup)
qed

end

sublocale normal-series  $\subseteq$  chain: normal-chain  $G \ (\cdot) \ 1 \ H \ n$ 
proof intro-locales
  show normal-chain-axioms  $G \ (\cdot) \ 1 \ H \ n$ 
    by (simp add: normal-chain-axioms-def normal-step term-subgroup)
qed

```

6.3 Isomorphism classes of normal-series factors

We turn the valid factors of an arbitrary normal series into isomorphism classes and then into a multiset. When used with a composition series, these are the corresponding composition-factor classes.

```

context normal-series
begin

lemma series-factor-eq-normal-factor:
  series-factor i = normal-factor (H i) (H (Suc i)) (·) 1
  unfolding series-factor-def normal-factor-def by rule

lemma series-factor-class-eq-normal-factor-class:
  series-factor-class i =
    normal-factor-class (H i) (H (Suc i)) (·) 1
  unfolding series-factor-class-def normal-factor-class-def
  by (simp add: series-factor-eq-normal-factor)

end

end
theory Group-Operations
  imports Group-Theory HOL-Computational-Algebra.Primes
begin

```

7 Group Actions

This theory formalises group actions with respect to the locale-based group infrastructure of *Group-Theory*, following the same development as *GroupAction.thy* (which uses the record-based groups of *HOL-Algebra*).

A group action of a group $(G, \cdot, 1)$ on a set S is given by a group homomorphism $\varphi : G \rightarrow \text{Sym}(S)$, where $\text{Sym}(S)$ is the symmetric group on S (i.e. the group of bijections $S \rightarrow S$).

7.1 Definition and Basic Properties

```

locale Group-Action =
  grp: Group G (·) 1 + transformations S
  for G and composition (infixl ⟨·⟩ 70) and unit (⟨1⟩) and φ and S +
  assumes hom-closed:  $g \in G \implies \varphi\ g \in \text{Sym}$ 
    and hom-compose:  $\llbracket g \in G; h \in G \rrbracket \implies \varphi\ (g \cdot h) = \text{compose } S\ (\varphi\ g)\ (\varphi\ h)$ 
    and hom-unit:  $\varphi\ \mathbf{1} = \text{identity } S$ 

begin

sublocale coset-notation .

```

The action of **1** has no effect.


```

lemma one-is-id [simp]:
  assumes  $s \in S$ 
  shows  $\varphi \ 1 \ s = s$ 
  using assms hom-unit by simp

```

The action maps S to itself.

```

lemma action-closed [intro, simp]:
  assumes  $g \in G \ s \in S$ 
  shows  $\varphi \ g \ s \in S$ 
  using assms hom-closed Units-bij-betwI by (meson bij-betwE)

```

The image of any group element is a bijection on S .

```

lemma images-are-bij:
  assumes  $g \in G$ 
  shows bij-betw ( $\varphi \ g$ )  $S \ S$ 
  using assms hom-closed by auto

```

The action respects group multiplication.

```

lemma action-mult:
  assumes  $g \in G \ h \in G \ s \in S$ 
  shows  $\varphi \ g \ (\varphi \ h \ s) = \varphi \ (g \cdot h) \ s$ 
  using assms hom-compose by (simp add: compose-eq)

```

The action of $\text{grp.inverse } g$ reverts the action of g .

```

lemma group-inv-rel:
  assumes  $g \in G \ t \in S \ \varphi \ g \ t = s$ 
  shows  $\varphi \ (\text{grp.inverse } g) \ s = t$ 
  using action-mult assms by force

```

```

lemma action-inverse:
  assumes  $g \in G \ s \in S$ 
  shows  $\varphi \ (\text{grp.inverse } g) \ (\varphi \ g \ s) = s$ 
  using assms action-closed group-inv-rel by blast

```

7.2 Stabilizer

The stabilizer of a point $s \in S$ is the set of group elements that fix s under the action.

```

definition stabilizer
  where stabilizer  $s = \{g \in G. \ \varphi \ g \ s = s\}$ 

```

The stabilizer of any point in S is a subgroup of G .

```

theorem stabilizer-subgroup:
  assumes  $s \in S$ 
  shows Subgroup (stabilizer  $s$ )  $G \ (\cdot) \ 1$ 
proof (rule grp.subgroupI)
  fix  $g \ h$ 

```

```

assume  $g \in \text{stabilizer } s \ h \in \text{stabilizer } s$ 
then show  $g \cdot h \in \text{stabilizer } s$ 
  using action-mult assms stabilizer-def by force
qed (auto simp: assms group-inv-rel stabilizer-def)

```

7.3 Fixed Points

The fixed points of a group action are those elements of S that are fixed by every group element. Equivalently, a point is fixed if and only if its stabilizer is the whole group.

```

definition fixed-points
  where  $\text{fixed-points} = \{s \in S. G \subseteq \text{stabilizer } s\}$ 

```

```

lemma fixed-points-subset:  $\text{fixed-points} \subseteq S$ 
  unfolding fixed-points-def by auto

```

```

lemma fixed-point-char:
  assumes  $s \in S$ 
  shows  $s \in \text{fixed-points} \longleftrightarrow (\forall g \in G. \varphi g s = s)$ 
  unfolding fixed-points-def stabilizer-def using assms by auto

```

```

lemma fixed-pointI:
  assumes  $s \in S \ \bigwedge g. g \in G \implies \varphi g s = s$ 
  shows  $s \in \text{fixed-points}$ 
  using assms fixed-point-char by auto

```

```

lemma fixed-pointD:
  assumes  $s \in \text{fixed-points} \ g \in G$ 
  shows  $\varphi g s = s$ 
  using assms unfolding fixed-points-def stabilizer-def by auto

```

A point is fixed if and only if its stabilizer is the whole group.

```

lemma fixed-point-stabilizer:
  assumes  $s \in S$ 
  shows  $s \in \text{fixed-points} \longleftrightarrow \text{stabilizer } s = G$ 
  using assms fixed-points-def stabilizer-def by auto

```

7.4 Orbits

The orbit of a point $s \in S$ under the group action is the set of all images of s under group elements.

```

definition orbit
  where  $\text{orbit } s \equiv \{\varphi g s \mid g. g \in G\}$ 

```

```

lemma orbit-eq:  $\text{orbit } s = (\lambda g. \varphi g s) ` G$ 
  by (auto simp: orbit-def)

```

```

lemma orbit-subset:

```

```

assumes  $s \in S$ 
shows  $\text{orbit } s \subseteq S$ 
unfolding orbit-def using assms by auto

```

```

lemma orbit-self:
  assumes  $s \in S$ 
  shows  $s \in \text{orbit } s$ 
  unfolding orbit-def using assms grp.unit-closed by force

```

```

lemma orbit-mem-iff:
  assumes  $s \in S$ 
  shows  $t \in \text{orbit } s \longleftrightarrow (\exists g \in G. \varphi g s = t)$ 
  unfolding orbit-def by auto

```

```

lemma orbit-memI:
  assumes  $g \in G$ 
  shows  $\varphi g s \in \text{orbit } s$ 
  unfolding orbit-def using assms by auto

```

```

lemma orbit-memE:
  assumes  $t \in \text{orbit } s$ 
  obtains  $g$  where  $g \in G \varphi g s = t$ 
  using assms unfolding orbit-def by auto

```

```

lemma orbit-nonempty:  $\text{orbit } s \neq \{\}$ 
  using orbit-memI by auto

```

A point is fixed precisely when its orbit is the singleton containing it.

```

lemma fixed-point-iff-orbit-singleton:
  assumes  $s \in S$ 
  shows  $s \in \text{fixed-points} \longleftrightarrow \text{orbit } s = \{s\}$ 
proof
  assume fixed:  $s \in \text{fixed-points}$ 
  show  $\text{orbit } s = \{s\}$ 
  proof (rule set-eqI)
    fix  $t$ 
    show  $t \in \text{orbit } s \longleftrightarrow t \in \{s\}$ 
    proof
      assume  $t \in \text{orbit } s$ 
      then obtain  $g$  where  $g \in G \varphi g s = t$  by (rule orbit-memE)
      then show  $t \in \{s\}$  using fixed fixed-pointD by simp
    next
      assume  $t \in \{s\}$ 
      then show  $t \in \text{orbit } s$  using assms orbit-self by simp
    qed
  qed
next
  assume orbit:  $\text{orbit } s = \{s\}$ 
  show  $s \in \text{fixed-points}$ 
  proof (rule fixed-pointI [OF assms])

```

```

    fix g assume g ∈ G
    then have  $\varphi g s \in \text{orbit } s$  by (rule orbit-memI)
    then show  $\varphi g s = s$  by (simp add: orbit)
  qed
qed

lemma card-orbit-gt0I: finite G  $\implies$  card (orbit s) > 0
  by (metis card-gt-0-iff finite-imageI orbit-eq orbit-nonempty)

```

7.5 The Orbit-Stabilizer Theorem

Two group elements give the same image under the action if and only if they lie in the same right coset of the stabilizer.

```

lemma same-orbit-iff-same-coset:
  assumes s ∈ S g ∈ G h ∈ G
  shows  $\varphi g s = \varphi h s \iff \text{grp.inverse } g \cdot h \in \text{stabilizer } s$ 
proof
  assume  $\varphi g s = \varphi h s$ 
  then show  $\text{grp.inverse } g \cdot h \in \text{stabilizer } s$ 
    using assms Group-Action.action-mult Group-Action-axioms action-inverse
    by (fastforce simp: invertible-inverse-closed stabilizer-def)
next
  assume mem:  $\text{grp.inverse } g \cdot h \in \text{stabilizer } s$ 
  then have  $\varphi g (\varphi (\text{grp.inverse } g) (\varphi h s)) = \varphi g s$ 
    using assms by (simp add: action-mult stabilizer-def)
  then show  $\varphi g s = \varphi h s$ 
    using assms by (simp add: action-mult grp.invertible-right-inverse2)
qed

```

The orbit-stabilizer theorem. The map $\lambda g. \varphi g s$ from G to $\text{orbit } s$ is surjective, and its fibers are exactly the left cosets of the stabilizer. Hence the number of distinct images equals the index of the stabilizer, and Lagrange's theorem gives the cardinality form.

Every left coset of the stabilizer has the same cardinality as the stabilizer itself.

```

lemma coset-fiber:
  assumes s: s ∈ S and g: g ∈ G
  shows  $\{h \in G. \varphi h s = \varphi g s\} = g \cdot | \text{stabilizer } s$ 
proof (rule set-eqI)
  fix h
  show  $(h \in \{h \in G. \varphi h s = \varphi g s\}) = (h \in g \cdot | \text{stabilizer } s)$ 
proof
  assume h:  $h \in \{h \in G. \varphi h s = \varphi g s\}$ 
  then have hG:  $h \in G$  and eq:  $\varphi h s = \varphi g s$  by auto
  from same-orbit-iff-same-coset [OF s g hG] eq
  have  $\text{grp.inverse } g \cdot h \in \text{stabilizer } s$  by simp
  then show  $h \in g \cdot | \text{stabilizer } s$ 
    by (simp add: g grp.invertible-right-inverse2 hG rev-image-eqI Left-Coset-def)
next

```

```

assume  $h: h \in g \cdot | \text{stabilizer } s$ 
then obtain  $t$  where  $t: t \in \text{stabilizer } s$  and  $ht: h = g \cdot t$ 
  unfolding Left-Coset-def by auto
from  $t$  have  $tG: t \in G$  and  $ts: \varphi \ t \ s = s$ 
  unfolding stabilizer-def by auto
have  $hG: h \in G$  using  $ht \ g \ tG$  by auto
then show  $h \in \{h \in G. \varphi \ h \ s = \varphi \ g \ s\}$ 
  using action-mult  $g \ ht \ s \ tG \ ts$  by fastforce
qed
qed

```

The orbit-stabilizer theorem (bijection form). The map that sends each orbit element to its fiber under $\lambda g. \varphi \ g \ s$ is a bijection from *orbit* s to the set of left cosets of the stabilizer.

```

theorem orbit-stabilizer-bij:
  assumes  $s \in S$ 
  shows bij-betw  $(\lambda t. \{g \in G. \varphi \ g \ s = t\}) \ (\text{orbit } s) \ ((\lambda g. g \cdot | \text{stabilizer } s) \text{ `` } G)$ 
  using coset-fiber assms
  by (auto simp: bij-betw-def inj-on-def orbit-def)

```

The orbit-stabilizer theorem (cardinality form).

```

theorem orbit-stabilizer-card:
  assumes  $s: s \in S$  and  $\text{fin: finite } G$ 
  shows  $\text{card } G = \text{card } (\text{orbit } s) * \text{card } (\text{stabilizer } s)$ 
proof –
  let  $?f = \lambda g. \varphi \ g \ s$ 
  have surj:  $?f \text{ `` } G = \text{orbit } s$ 
    unfolding orbit-def by auto
  have fibers:  $\bigwedge g. g \in G \implies \{h \in G. ?f \ h = ?f \ g\} = g \cdot | \text{stabilizer } s$ 
    using coset-fiber [OF s] by auto
  have fiber-card:  $\text{card } \{h \in G. ?f \ h = ?f \ g\} = \text{card } (\text{stabilizer } s)$ 
    if  $g: g \in G$  for  $g$ 
  proof –
    have  $\S$ : bij-betw  $((\cdot) \ g) \ (\text{stabilizer } s) \ (g \cdot | \text{stabilizer } s)$ 
      using that by (auto simp: bij-betw-def inj-on-def stabilizer-def)
    have  $\text{card } \{h \in G. ?f \ h = ?f \ g\} = \text{card } (g \cdot | \text{stabilizer } s)$ 
      using fibers [OF that] by simp
    also have  $\dots = \text{card } (\text{stabilizer } s)$ 
      using  $\S$  by (simp add: bij-betw-same-card)
    finally show ?thesis
      by presburger
  qed
  have decomp:  $G = (\bigcup t \in \text{orbit } s. \{g \in G. ?f \ g = t\})$ 
    using surj by auto
  have disj:  $\bigwedge t1 \ t2. \llbracket t1 \in \text{orbit } s; t2 \in \text{orbit } s; t1 \neq t2 \rrbracket$ 
     $\implies \{g \in G. ?f \ g = t1\} \cap \{g \in G. ?f \ g = t2\} = \{\}$ 
    by auto
  have fin-orbit: finite  $(\text{orbit } s)$ 
    using fin surj by (metis finite-imageI)

```

```

have card  $G = \text{card } (\bigcup t \in \text{orbit } s. \{g \in G. ?f g = t\})$ 
  using decomp by simp
also have  $\dots = (\sum t \in \text{orbit } s. \text{card } \{g \in G. ?f g = t\})$ 
  using fin fin-orbit disj by (intro card-UN-disjoint) auto
also have  $\dots = (\sum t \in \text{orbit } s. \text{card } (\text{stabilizer } s))$ 
  using fiber-card
  by (intro sum.cong; force simp: orbit-def)
also have  $\dots = \text{card } (\text{orbit } s) * \text{card } (\text{stabilizer } s)$ 
  by simp
finally show ?thesis .
qed

```

7.6 Connection to Transformation Groups

The image $\varphi \text{ ` } G$ is a subgroup of the symmetric group on S , and therefore forms a transformation group. This connects the orbit defined here with the orbit equivalence relation of *transformation-group*.

```

lemma image-is-subgroup: Subgroup ( $\varphi \text{ ` } G$ ) Sym (compose  $S$ ) (identity  $S$ )
proof (rule symmetric.subgroupI)
  show  $\varphi \text{ ` } G \subseteq \text{Sym}$ 
    using hom-closed by auto
next
  show identity  $S \in \varphi \text{ ` } G$ 
    using hom-unit grp.unit-closed by (metis image-eqI)
next
  fix  $g h$ 
  assume  $g \in \varphi \text{ ` } G \ h \in \varphi \text{ ` } G$ 
  then obtain  $a b$  where  $a \in G \ g = \varphi a \ b \in G \ h = \varphi b$  by auto
  then show compose  $S \ g \ h \in \varphi \text{ ` } G$ 
    using hom-compose by (metis grp.composition-closed image-eqI)
next
  fix  $g$ 
  assume  $g \in \varphi \text{ ` } G$ 
  then obtain  $a$  where  $a \in G \ g = \varphi a$  by auto
  show  $\bigwedge g. g \in \varphi \text{ ` } G \implies \text{symmetric.invertible } g$ 
    using hom-closed by blast
  have inv-eq:  $\text{inverse } (\varphi a) = \varphi (\text{grp.inverse } a)$ 
  proof (intro inverse-equality)
    show compose  $S \ (\varphi a) \ (\varphi (\text{grp.inverse } a)) = \text{identity } S$ 
      compose  $S \ (\varphi (\text{grp.inverse } a)) \ (\varphi a) = \text{identity } S$ 
    using  $\langle a \in G \rangle$  hom-compose [symmetric] hom-unit by auto
  qed (use  $\langle a \in G \rangle$  hom-closed Units-bijective in blast) +
  have symmetric.inverse  $g \in S \rightarrow_E S$ 
    by (metis a hom-closed mem-UnitsD symmetric.invertible symmetric.invertible-inverse-closed)
  then have  $\text{inverse } (\varphi a) = \text{symmetric.inverse } g$ 
    using  $\langle a \in G \rangle \langle g = \varphi a \rangle$  hom-closed
    by (metis (no-types, lifting) inverse-equality mem-UnitsD symmetric.invertible
      symmetric.invertible-left-inverse symmetric.invertible-right-inverse)
  then show symmetric.inverse  $g \in \varphi \text{ ` } G$ 

```

by (metis $\langle a \in G \rangle$ grp.invertible grp.invertible-inverse-closed image-iff inv-eq)
qed

sublocale tg: transformation-group $\varphi \text{ ' } G \ S$
using image-is-subgroup transformation-group-def **by** blast

lemma orbit-eq-Class:
assumes $s \in S$
shows $\text{orbit } s = \text{tg.orbit.Class } s$
using assms tg.orbit-equality **by** (auto simp: orbit-def)

7.7 The permutation representation

A group action is, by definition, a homomorphism into the symmetric group on S . We make this explicit, identify its kernel with the set of elements acting trivially on every point, and deduce that a *faithful* action embeds the group as a subgroup of the symmetric group: the permutation representation. This is what turns an action on a finite set into a concrete subgroup of permutations (e.g. a Galois group acting on the roots of a polynomial becomes a subgroup of Sym of the root set).

The action, restricted to G for extensionality, is a group homomorphism into the symmetric group.

interpretation action-hom:
group-homomorphism restrict $\varphi \ G \ G \ (\cdot) \ 1 \ Sym \ compose \ S \ identity \ S$
proof
show restrict $\varphi \ G \in G \rightarrow_E \ Sym$ **using** hom-closed **by** auto
next
fix $x \ y$ **assume** $x \in G \ y \in G$
then show restrict $\varphi \ G \ (x \cdot y) = compose \ S \ (\text{restrict } \varphi \ G \ x) \ (\text{restrict } \varphi \ G \ y)$
by (simp add: hom-compose)
next
show restrict $\varphi \ G \ 1 = identity \ S$ **by** (simp add: hom-unit)
qed

On S , the image of a group element is the identity exactly when it fixes every point.

lemma phi-eq-identity-iff:
assumes $g \in G$
shows $\varphi \ g = identity \ S \longleftrightarrow (\forall s \in S. \ \varphi \ g \ s = s)$
using assms **by** auto

The kernel of the permutation representation is the set of group elements that fix every point of S , i.e. the intersection of all the stabilizers.

lemma action-kernel-eq:
group-homomorphism.Ker (restrict $\varphi \ G \ G \ (identity \ S) = \{g \in G. \ \forall s \in S. \ \varphi \ g \ s = s\}$
proof –

```

show ?thesis
  using action-hom.Ker-image by (auto simp: phi-eq-identity-iff)
qed

```

An action is *faithful* if only the unit acts trivially on all of S .

```

definition faithful
  where faithful  $\longleftrightarrow (\forall g \in G. (\forall s \in S. \varphi g s = s) \longrightarrow g = \mathbf{1})$ 

```

lemma faithful-iff-kernel-trivial:

```

  faithful  $\longleftrightarrow$  group-homomorphism.Ker (restrict  $\varphi$   $G$ )  $G$  (identity  $S$ ) =  $\{\mathbf{1}\}$ 

```

proof –

```

  have ker: action-hom.Ker =  $\{g \in G. \forall s \in S. \varphi g s = s\}$  by (rule action-kernel-eq)

```

```

  have one-in:  $\mathbf{1} \in$  action-hom.Ker unfolding ker by simp

```

```

  show ?thesis

```

proof

```

  assume faithful

```

```

  then have action-hom.Ker  $\subseteq \{\mathbf{1}\}$  unfolding ker faithful-def by blast

```

```

  with one-in show action-hom.Ker =  $\{\mathbf{1}\}$  by blast

```

next

```

  assume action-hom.Ker =  $\{\mathbf{1}\}$ 

```

```

  then show faithful unfolding faithful-def using ker by blast

```

qed

qed

A faithful action is injective.

lemma faithful-inj:

```

  assumes faithful

```

```

  shows inj-on (restrict  $\varphi$   $G$ )  $G$ 

```

```

  using action-hom.injective-iff-kernel-unit assms faithful-iff-kernel-trivial

```

```

  by force

```

The permutation representation of a faithful action is a group isomorphism onto its image, exhibiting G as a subgroup of the symmetric group on S .

theorem faithful-action-embeds:

```

  assumes faithful

```

```

  shows group-isomorphism (restrict  $\varphi$   $G$ )  $G$  ( $\cdot$ )  $\mathbf{1}$  ( $\varphi \text{ ` } G$ ) (compose  $S$ ) (identity  $S$ )

```

proof

```

  show bij-betw (restrict  $\varphi$   $G$ )  $G$  ( $\varphi \text{ ` } G$ )

```

```

    using assms bij-betw-def faithful-inj by auto

```

```

qed (auto simp: hom-closed hom-unit hom-compose)

```

7.8 Orbit Partition

The orbits of a group action partition S . Using the connection to *transformation-group*, these facts follow from the general theory of equivalence relations.

If t is in the orbit of s , then s is in the orbit of t .

lemma orbit-symmetric:


```

assumes  $s \in S \ t \in \text{orbit } s$ 
shows  $s \in \text{orbit } t$ 
proof –
  from assms have  $t \in \text{tg.orbit.Class } s$ 
    by (simp add: orbit-eq-Class)
  then have  $(t, s) \in \text{tg.Orbit-Relation}$ 
    using assms(1) by auto
  then have  $(s, t) \in \text{tg.Orbit-Relation}$ 
    by (rule tg.orbit.symmetric)
  then show  $s \in \text{orbit } t$ 
    using orbit-eq-Class by blast
qed

```

If t is in the orbit of s , the two orbits are equal.

```

lemma orbit-subset-mem:
  assumes  $s \in S \ t \in \text{orbit } s$ 
  shows  $\text{orbit } t \subseteq \text{orbit } s$ 
  using action-mult assms orbit-mem-iff by fastforce

```

```

lemma orbit-eq-mem:
  assumes  $s \in S \ t \in \text{orbit } s$ 
  shows  $\text{orbit } t = \text{orbit } s$ 
  by (meson antisym assms orbit-subset-mem orbit-subset orbit-symmetric subsetD)

```

Two orbits are either equal or disjoint.

```

lemma orbit-disjoint:
  assumes  $s \in S \ t \in S \ \text{orbit } s \neq \text{orbit } t$ 
  shows  $\text{orbit } s \cap \text{orbit } t = \{\}$ 
  by (metis Int-emptyI assms orbit-eq-mem)

```

The set of all orbits.

```

definition orbits :: 'b set set
  where orbits = orbit ' S

```

```

lemma orbits-coverI:
  assumes  $s \in S$ 
  shows  $\text{orbit } s \in \text{orbits}$ 
  using assms unfolding orbits-def by auto

```

```

lemma orbits-coverE:
  assumes  $X \in \text{orbits}$ 
  obtains  $s$  where  $s \in S \ X = \text{orbit } s$ 
  using assms unfolding orbits-def by auto

```

The orbits cover S .

```

lemma orbits-cover:  $\bigcup \text{orbits} = S$ 
  by (simp add: orbit-eq-Class orbits-def)

```

The orbits are pairwise disjoint.

lemma *orbits-disjoint*: *disjoint orbits*
by (*simp add: disjnt-def orbit-disjoint orbits-def pairwise-imageI*)

The nontrivial orbits are the orbits containing more than one point.

definition *nontrivial-orbits* :: 'b set set
where *nontrivial-orbits* = $\{X \in \text{orbits}. \neg \text{is-singleton } X\}$

lemma *nontrivial-orbits-subset*: *nontrivial-orbits $\subseteq$ orbits*
by (*auto simp: nontrivial-orbits-def*)

The nontrivial orbits cover exactly the points that are not fixed.

lemma *nontrivial-orbits-cover*:

$\bigcup \text{nontrivial-orbits} = S - \text{fixed-points}$

proof (*rule set-eqI*)

fix x

show $x \in \bigcup \text{nontrivial-orbits} \longleftrightarrow x \in S - \text{fixed-points}$

proof

assume $x \in \bigcup \text{nontrivial-orbits}$

then obtain X **where** $X: X \in \text{nontrivial-orbits}$ $x \in X$ **by** *auto*

then obtain s **where** $s: s \in S$ $X = \text{orbit } s$

using *nontrivial-orbits-def orbits-coverE* **by** *auto*

have $xS: x \in S$ **using** X s *orbit-subset* **by** *auto*

have $\text{orbit } x = X$ **using** *orbit-eq-mem* [*OF* $s(1)$] $X(2)$ $s(2)$ **by** *simp*

moreover have $\neg \text{is-singleton } X$ **using** X **by** (*simp add: nontrivial-orbits-def*)

ultimately have $x \notin \text{fixed-points}$

using *fixed-point-iff-orbit-singleton* [*OF* xS] **by** (*auto simp: is-singleton-def*)

with xS **show** $x \in S - \text{fixed-points}$ **by** *simp*

next

assume $x: x \in S - \text{fixed-points}$

have $xS: x \in S$ **using** x **by** *simp*

have $\text{orbit } x \neq \{x\}$

using *fixed-point-iff-orbit-singleton* [*of* x] x **by** *simp*

then have $\neg \text{is-singleton } (\text{orbit } x)$

using *orbit-self* [*OF* xS] **by** (*auto simp: is-singleton-def*)

then have $\text{orbit } x \in \text{nontrivial-orbits}$

using x *orbits-coverI* **by** (*simp add: nontrivial-orbits-def*)

then show $x \in \bigcup \text{nontrivial-orbits}$ **using** x *orbit-self* **by** *auto*

qed

qed

Cardinality of S as a sum over orbits.

lemma *card-S-sum-orbits*:

assumes *finite S*

shows $\text{card } S = (\sum X \in \text{orbits}. \text{card } X)$

proof –

have $\text{card } (\bigcup \text{orbits}) = (\sum X \in \text{orbits}. \text{card } X)$

proof (*rule card-Union-disjoint*)

show *disjoint orbits* **by** (*rule orbits-disjoint*)

show $\bigwedge A. A \in \text{orbits} \implies \text{finite } A$

```

    by (metis Union-upper assms orbits-cover rev-finite-subset)
  qed
  then show ?thesis by (simp add: orbits-cover)
qed

```

Splitting the orbit partition into fixed points and nontrivial orbits gives the fixed-point form of the class equation.

```

lemma fixed-point-class-equation:
  assumes fin: finite S
  shows card S = card fixed-points + ( $\sum X \in \text{nontrivial-orbits. card } X$ )
proof -
  have card-nontrivial:
    card ( $\bigcup \text{nontrivial-orbits}$ ) = ( $\sum X \in \text{nontrivial-orbits. card } X$ )
  proof (rule card-Union-disjoint)
  show disjoint nontrivial-orbits
    using orbits-disjoint nontrivial-orbits-subset
    by (auto simp: disjoint-def)
  show  $\bigwedge X. X \in \text{nontrivial-orbits} \implies \text{finite } X$ 
  proof -
    fix X assume X  $\in \text{nontrivial-orbits}$ 
    then obtain s where s  $\in S$  X = orbit s
      using nontrivial-orbits-def orbits-coverE by auto
    then show finite X
      using orbit-subset[OF  $\langle s \in S \rangle$ ] fin by (meson finite-subset)
  qed
  qed
  have fin-fixed: finite fixed-points
    using fixed-points-subset fin by (rule finite-subset)
  have fin-nonfixed: finite (S - fixed-points) using fin by simp
  have split: fixed-points  $\cup$  (S - fixed-points) = S
    using fixed-points-subset by auto
  have card S = card fixed-points + card (S - fixed-points)
    using card-Un-disjoint[OF fin-fixed fin-nonfixed] split by simp
  also have ... = card fixed-points + card ( $\bigcup \text{nontrivial-orbits}$ )
    by (simp add: nontrivial-orbits-cover)
  also have ... = card fixed-points + ( $\sum X \in \text{nontrivial-orbits. card } X$ )
    by (simp add: card-nontrivial)
  finally show ?thesis .
qed

```

If every nontrivial orbit has cardinality divisible by k , then the same is true of the set of non-fixed points. This is the divisibility form of the fixed-point congruence.

```

lemma dvd-card-nonfixed-points:
  assumes fin: finite S
  and dvd:  $\bigwedge s. \llbracket s \in S; s \notin \text{fixed-points} \rrbracket \implies k \text{ dvd card (orbit } s)$ 
  shows k dvd card (S - fixed-points)
proof -
  have k dvd card ( $\bigcup \text{nontrivial-orbits}$ )

```

```

proof (intro dvd-partition strip)
  show finite ( $\bigcup$  nontrivial-orbits)
    using fin nontrivial-orbits-cover by simp
  show  $k$  dvd card  $X$  if  $X$ :  $X \in$  nontrivial-orbits for  $X$ 
  proof –
    obtain  $s$  where  $s$ :  $s \in S$   $X =$  orbit  $s$ 
    using  $X$  nontrivial-orbits-def orbits-coverE by auto
    have  $s \notin$  fixed-points
    using  $X$  s fixed-point-iff-orbit-singleton[OF s(1)]
    by (auto simp: nontrivial-orbits-def)
    then show ?thesis using dvd[OF s(1)] s(2) by simp
  qed
qed (use orbits-disjoint nontrivial-orbits-subset in ⟨auto simp: disjoint-def⟩)
then show ?thesis by (simp add: nontrivial-orbits-cover)
qed

```

If k divides the cardinality of every orbit, then k divides card S .

```

lemma dvd-orbits-dvd-S:
  assumes finite  $S \wedge X$ .  $X \in$  orbits  $\implies k$  dvd card  $X$ 
  shows  $k$  dvd card  $S$ 
proof –
  have  $k$  dvd card ( $\bigcup$  orbits)
    using orbits-disjoint unfolding disjoint-def
    by (metis assms dvd-partition orbits-cover)
  then show ?thesis by (simp add: orbits-cover)
qed

```

end

8 Sylow's Theorem

```

context Monoid
begin

```

```

definition power :: 'a  $\Rightarrow$  nat  $\Rightarrow$  'a
  where power  $a$   $n \equiv (((\cdot) \ a)^\wedge n) \ 1$ 

```

```

definition element-order :: 'a  $\Rightarrow$  nat
  where element-order  $a \equiv (LEAST \ n. \ n > 0 \wedge \text{power } a \ n = 1)$ 

```

```

lemma power-0 [simp]: power  $a$  0 = 1
  by (simp add: power-def)

```

```

lemma power-Suc [simp]: power  $a$  (Suc  $n$ ) =  $a \cdot$  power  $a$   $n$ 
  by (simp add: power-def)

```

```

lemma power-1 [simp]:  $a \in M \implies$  power  $a$  1 =  $a$ 
  by simp

```

```

lemma power-closed [intro, simp]:  $a \in M \implies \text{power } a \ n \in M$ 
  by (induction n) auto

lemma power-add:  $a \in M \implies \text{power } a \ (m + n) = \text{power } a \ m \cdot \text{power } a \ n$ 
  by (induction m) (simp-all add: associative)

lemma power-mult:  $a \in M \implies \text{power } a \ (m * n) = \text{power } (\text{power } a \ m) \ n$ 
  by (induction n) (simp-all add: power-add)

lemma power-unit [simp]:  $\text{power } 1 \ n = 1$ 
  by (induction n) simp-all

lemma power-Suc-right:  $a \in M \implies \text{power } a \ (\text{Suc } n) = \text{power } a \ n \cdot a$ 
proof (induction n)
  case 0
  then show ?case by simp
next
  case (Suc n) then show ?case
    using Suc-eq-plus1 local.power-add power-1 by presburger
qed

lemma power-commutes:  $a \in M \implies a \cdot \text{power } a \ n = \text{power } a \ n \cdot a$ 
  by (metis power-Suc power-Suc-right)

lemma power-commutes-power:
   $a \in M \implies \text{power } a \ m \cdot \text{power } a \ n = \text{power } a \ n \cdot \text{power } a \ m$ 
  by (simp add: power-add[symmetric] add.commute)

lemma element-order-unit [simp]:  $\text{element-order } 1 = 1$ 
  unfolding element-order-def by (rule Least-equality) auto

end

context Group begin

lemma power-inverse:
  assumes  $g \in G$ 
  shows  $\text{power } (\text{inverse } g) \ n = \text{inverse } (\text{power } g \ n)$ 
proof (induction n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have  $\text{inverse } g \cdot \text{inverse } (\text{power } g \ n) = \text{inverse } (\text{power } g \ n) \cdot \text{inverse } g$ 
    using power-commutes[of inverse g n] assms Suc by simp
  then have  $\text{power } (\text{inverse } g) \ (\text{Suc } n) = \text{inverse } (\text{power } g \ n) \cdot \text{inverse } g$ 
    using Suc by simp
  also have  $\dots = \text{inverse } (\text{power } g \ (\text{Suc } n))$ 
    using inverse-composition-commute[of g power g n] assms by simp

```

```

    finally show ?case .
qed

lemma finite-power-order-exists:
  assumes finite G g ∈ G
  shows ∃ n > 0. power g n = 1
proof -
  have False if inj-on (power g) {.. $\text{card } G$ }
  proof -
    have card (power g ‘ {.. $\text{card } G$ }) = Suc (card G)
    by (simp add: card-image that)
    then show False
    using card-mono assms by (metis Suc-n-not-le-n imageE power-closed subsetI)
  qed
  then obtain i j where i ≤ card G j ≤ card G i < j and ij-eq: power g i = power
  g j
  by (metis (mono-tags, opaque-lifting) inj-on-def linorder-cases atMost-iff)
  then have j = i + (j - i) by simp
  then have eq: power g j = power g i · power g (j - i)
  using power-add assms(2) by metis
  have power g (j - i) = 1
  by (metis Monoid.invertible-left-cancel Monoid-axioms assms(2) eq ij-eq invertible
  power-closed right-unit unit-closed)
  with ⟨i < j⟩ show ?thesis by (intro exI[of - j - i]) auto
qed

lemma element-order-pos:
  assumes finite G g ∈ G
  shows element-order g > 0
proof -
  obtain n where n > 0 power g n = 1
  using finite-power-order-exists[OF assms] by auto
  then show ?thesis
  unfolding element-order-def using LeastI[of λn. n > 0 ∧ power g n = 1 n] by
  auto
qed

lemma power-element-order:
  assumes finite G g ∈ G
  shows power g (element-order g) = 1
proof -
  obtain n where n > 0 power g n = 1
  using finite-power-order-exists[OF assms] by auto
  then show ?thesis
  unfolding element-order-def using LeastI[of λn. n > 0 ∧ power g n = 1 n]
  by auto
qed

lemma element-order-minimal:
  assumes finite G g ∈ G n > 0 power g n = 1

```

shows $\text{element-order } g \leq n$
unfolding element-order-def **by** (rule *Least-le*) (use *assms in auto*)

lemma *element-order-dvd*:
assumes *finite* G $g \in G$ $\text{power } g \ n = 1$
shows $\text{element-order } g \ \text{dvd } n$
proof (rule *ccontr*)
assume *ndvd*: $\neg \text{element-order } g \ \text{dvd } n$
have *ord-pos*: $\text{element-order } g > 0$
using $\text{element-order-pos}[OF \ \text{assms}(1,2)]$.
have *mod-pos*: $n \bmod \text{element-order } g > 0$
using *ndvd ord-pos* **by** (metis *dvd-eq-mod-eq-0 gr0I*)
have $\text{power } g \ n = \text{power } g \ (\text{element-order } g * (n \text{ div } \text{element-order } g) + n \bmod \text{element-order } g)$
by *simp*
also have $\dots = \text{power } g \ (\text{element-order } g * (n \text{ div } \text{element-order } g)) \cdot \text{power } g \ (n \bmod \text{element-order } g)$
using $\text{assms}(2)$ *power-add* **by** *blast*
also have $\text{power } g \ (\text{element-order } g * (n \text{ div } \text{element-order } g)) = \text{power} \ (\text{power } g \ (\text{element-order } g)) \ (n \text{ div } \text{element-order } g)$
using $\text{assms}(2)$ *power-mult* **by** *blast*
also have $\dots = \text{power } 1 \ (n \text{ div } \text{element-order } g)$
using $\text{power-element-order}[OF \ \text{assms}(1,2)]$ **by** *simp*
also have $\dots = 1$ **by** *simp*
finally have $\text{power } g \ (n \bmod \text{element-order } g) = 1$
using *assms* **by** *simp*
moreover have $n \bmod \text{element-order } g < \text{element-order } g$
using *ord-pos* **by** *simp*
ultimately show *False*
using $\text{element-order-minimal}[OF \ \text{assms}(1,2) \ \text{mod-pos}]$ **by** *linarith*
qed

lemma *power-mod-order*:
assumes *finite* G $g \in G$
shows $\text{power } g \ n = \text{power } g \ (n \bmod \text{element-order } g)$
proof –
have $n = \text{element-order } g * (n \text{ div } \text{element-order } g) + n \bmod \text{element-order } g$
by *simp*
then have $\text{power } g \ n = \text{power } g \ (\text{element-order } g * (n \text{ div } \text{element-order } g)) \cdot \text{power } g \ (n \bmod \text{element-order } g)$
using $\text{assms}(2)$ *power-add* **by** *metis*
also have $\text{power } g \ (\text{element-order } g * (n \text{ div } \text{element-order } g)) = \text{power} \ (\text{power } g \ (\text{element-order } g)) \ (n \text{ div } \text{element-order } g)$
using $\text{assms}(2)$ *power-mult* **by** *blast*
also have $\dots = 1$
using $\text{power-element-order}[OF \ \text{assms}]$ **by** *simp*
finally show *?thesis* **using** $\text{assms}(2)$ **by** *simp*
qed

lemma *power-eq-order-False*:

```

    assumes finite G g ∈ G
      and i < j j < element-order g
      and power g i = power g j
    shows False
  proof -
    from ⟨i < j⟩ have j = i + (j - i) by simp
    then have power g j = power g i · power g (j - i)
      using power-add ⟨g ∈ G⟩ by metis
    with assms(5) have power g i = power g i · power g (j - i) by simp
    then have power g (j - i) = 1
      using ⟨g ∈ G⟩
      by (metis invertible invertible-left-inverse invertible-left-inverse2 power-closed)
    moreover from assms have 0 < j - i j - i < element-order g by auto
    ultimately show False
      using assms element-order-dvd nat-dvd-not-less by blast
  qed

```

```

lemma power-inj-on-order:
  assumes finite G g ∈ G
  shows inj-on (power g) {..element-order g}
  unfolding inj-on-def
  by (metis assms lessThan-iff linorder-neqE-nat power-eq-order-False)

```

end

8.1 Cyclic subgroups

The cyclic subgroup generated by an element *g*: the set of its powers. For *g* of finite order this is ($\cap$) *g* ‘ {..*element-order g*}.

```

definition (in Group) cyclic-subgroup :: 'a ⇒ 'a set
  where cyclic-subgroup g = power g ‘ {..element-order g}

```

```

context Group
begin

```

```

lemma cyclic-subgroup-eq-range:
  assumes fin: finite G and gG: g ∈ G
  shows cyclic-subgroup g = range (power g)
proof (rule set-eqI)
  fix x
  show x ∈ cyclic-subgroup g ⟷ x ∈ range (power g)
proof
  assume x ∈ range (power g)
  then obtain n where x = power g n by auto
  then have x = power g (n mod element-order g)
    using power-mod-order[OF fin gG] by simp
  moreover have n mod element-order g < element-order g
    using element-order-pos[OF fin gG] by simp
  ultimately show x ∈ cyclic-subgroup g unfolding cyclic-subgroup-def by auto

```


qed (auto simp: cyclic-subgroup-def)
qed

lemma cyclic-subgroup-subset:
assumes $g \in G$ **shows** cyclic-subgroup $g \subseteq G$
unfolding cyclic-subgroup-def **using** assms **by** auto

lemma power-mem-cyclic-subgroup:
assumes fin: finite G **and** gG : $g \in G$
shows power g $i \in$ cyclic-subgroup g
using cyclic-subgroup-eq-range[OF fin gG] **by** blast

The cyclic subgroup is indeed a subgroup.

theorem cyclic-subgroup-is-subgroup:
assumes fin: finite G **and** gG : $g \in G$
shows Subgroup (cyclic-subgroup g) G ($\cdot$) **1**

proof –
have ord-pos: element-order $g > 0$ **using** element-order-pos[OF fin gG] .
note $C\text{-alt} =$ cyclic-subgroup-eq-range[OF fin gG]
have unit- C : **1** $\in$ cyclic-subgroup g **unfolding** $C\text{-alt}$
by (metis local.power-0 rangeI)
have comp- C : $\bigwedge a\ b. a \in$ cyclic-subgroup $g \implies b \in$ cyclic-subgroup $g \implies a \cdot b \in$
cyclic-subgroup g

proof –
fix $a\ b$ **assume** $a \in$ cyclic-subgroup $g\ b \in$ cyclic-subgroup g
then obtain $i\ j$ **where** $a =$ power $g\ i\ b =$ power $g\ j$ **unfolding** $C\text{-alt}$ **by** auto
then have $a \cdot b =$ power $g\ (i + j)$ **using** gG power-add **by** auto
then show $a \cdot b \in$ cyclic-subgroup g **unfolding** $C\text{-alt}$ **by** blast

qed
have inv- g : inverse $g =$ power $g\ (\text{element-order } g - 1)$
proof (rule inverse-equality)
show $g \cdot$ power $g\ (\text{element-order } g - 1) = \mathbf{1}$
using power-element-order[OF fin gG] ord-pos
by (metis Suc-diff-1 local.power-Suc)
then show power $g\ (\text{element-order } g - 1) \cdot g = \mathbf{1}$
using power-element-order[OF fin gG] ord-pos gG local.power-commutes **by** auto
show $g \in G$ **by** fact
show power $g\ (\text{element-order } g - 1) \in G$ **using** gG **by** simp

qed
have inv- C : $\bigwedge a. a \in$ cyclic-subgroup $g \implies$ inverse $a \in$ cyclic-subgroup g
proof –
fix a **assume** $a \in$ cyclic-subgroup g
then obtain i **where** $a =$ power $g\ i$ **unfolding** $C\text{-alt}$ **by** auto
have inverse $a =$ power (inverse g) i **using** a power-inverse[OF gG] **by** simp
also have $\dots =$ power (power $g\ (\text{element-order } g - 1)$) i **using** inv- g **by** simp
also have $\dots =$ power $g\ ((\text{element-order } g - 1) * i)$ **using** gG power-mult **by**

simp
finally **have** inverse $a =$ power $g\ ((\text{element-order } g - 1) * i)$.
then show inverse $a \in$ cyclic-subgroup g **unfolding** $C\text{-alt}$ **by** blast
qed

```

show ?thesis
  using subgroupI[of cyclic-subgroup g] cyclic-subgroup-subset[OF gG] unit-C comp-C
  inv-C
  by auto
qed

```

Its cardinality is the order of the generator.

```

theorem card-cyclic-subgroup:
  assumes fin: finite G and gG: g ∈ G
  shows card (cyclic-subgroup g) = element-order g
proof –
  have card (cyclic-subgroup g) = card (power g ‘ {..unfolding cyclic-subgroup-def by simp
  also have ... = card {..using card-image power-inj-on-order[OF fin gG] by blast
  also have ... = element-order g by simp
  finally show ?thesis .
qed

```

end

```

lemma (in Group) element-order-dvd-card:
  assumes fin: finite G and gG: g ∈ G
  shows element-order g dvd (card G)
proof –
  interpret CG: subgroup-of-group cyclic-subgroup g G (·) 1
  by (simp add: Group-axioms cyclic-subgroup-is-subgroup[OF fin gG] subgroup-of-group-def)
  from CG.lagrange[OF fin]
  have card G = element-order g * CG.index
    using card-cyclic-subgroup[OF fin gG] by simp
  then show ?thesis by auto
qed

```

```

lemma (in Group) power-order-eq-1 [simp]: g ∈ G ⇒ finite G ⇒ power g (card G)
= 1
  by (simp add: element-order-dvd-card power-mod-order)

```

```

lemma (in Monoid-homomorphism) commutes-with-power:
  assumes g ∈ M
  shows η (source.power g n) = target.power (η g) n
  by (induction n) (simp-all add: commutes-with-unit commutes-with-composition
  assms)

```

```

lemma (in group-homomorphism) element-order-image-dvd:
  assumes finite G finite G' g ∈ G
  shows target.element-order (η g) dvd source.element-order g
  by (metis assms commutes-with-power commutes-with-unit map-closed
  source.power-element-order target.element-order-dvd)

```

```

lemma (in group-isomorphism) element-order-preserved:
  assumes finite G finite G' g ∈ G
  shows target.element-order (η g) = source.element-order g
proof (rule dvd-antisym)
  show target.element-order (η g) dvd source.element-order g
    using element-order-image-dvd assms by blast
  show source.element-order g dvd target.element-order (η g)
proof –
  have eta-g: η g ∈ G' using assms map-closed by blast
  have target.power (η g) (target.element-order (η g)) = 1'
    using target.power-element-order[OF assms(2) eta-g] .
  then have img-eq: η (source.power g (target.element-order (η g))) = η 1
    using commutes-with-power assms commutes-with-unit by simp
  have inj: inj-on η G using bij-betw-def bijective by blast
  have source.power g (target.element-order (η g)) = 1
    using inj-onD[OF inj img-eq - source.unit-closed] assms by simp
  then show ?thesis
    using source.element-order-dvd[OF assms(1,3)] by blast
qed
qed

```

The same fact stated with the raw *Monoid.element-order* applied to the explicit operations and units, rather than the locale-qualified *source.element-order* / *target.element-order*. This is the form a caller working with concrete groups (e.g. the symmetric group under $(\circ)/id$) can unify against directly — the prefixed versions are definitionally equal but do not match by name across interpretations.

```

lemma (in group-isomorphism) element-order-preserved':
  assumes finite G finite G' g ∈ G
  shows Monoid.element-order (·') (1') (η g) = Monoid.element-order (·) 1 g
  using element-order-preserved[OF assms] .

```

```

lemma (in Group) element-order-power-prime:
  assumes fin: finite G and gG: g ∈ G and prime: prime p and dvd: p dvd element-order g
  shows element-order (power g (element-order g div p)) = p
proof –
  define k where k = element-order g div p
  have ord-eq: element-order g = p * k
    using dvd by (simp add: k-def)
  have k-pos: k > 0
    using element-order-pos[OF fin gG] ord-eq prime by (metis grOI mult-is-0)
  have gk: power g k ∈ G using gG by simp
  have power (power g k) p = power g (element-order g)
    by (simp add: gG local.power-mult mult.commute ord-eq)
  then have dvd-p: element-order (power g k) dvd p
    using element-order-dvd[OF fin gk]
    by (simp add: fin gG power-element-order)
  have element-order (power g k) ≠ 1

```

```

    using power-element-order[of power  $g$   $k$ ] fin  $gG$   $gk$   $k$ -pos
    by (metis dvd-mult-cancel2 element-order-dvd not-prime-1 ord-eq power-1 prime)
  with dvd-p prime show ?thesis
    unfolding  $k$ -def by (metis prime-nat-iff)
qed

end
theory Sylow-Theorems
  imports Group-Theory Group-Operations HOL-Computational-Algebra.Primes
begin

```

9 The First Sylow Theorem

```

context Group
begin

```

interpretation *left*: left-translations-of-group G $(\cdot)$ 1 ..

The set of n -element subsets of G .

```

definition  $\Omega$  :: nat  $\Rightarrow$  'a set set
  where  $\Omega$   $n$  = { $B$ .  $B \subseteq G \wedge \text{card } B = n$ }

```

The left-translation action on Ω n : each $g \in G$ acts by mapping a subset B to $(\cdot) g \cdot B = \{g \cdot b \mid b \in B\}$. We make the map extensional on Ω n so it lies in Ω $n \rightarrow_E \Omega$ n .

```

definition  $\varphi$ -left :: nat  $\Rightarrow$  'a  $\Rightarrow$  'a set  $\Rightarrow$  'a set
  where  $\varphi$ -left  $n$   $g$  = restrict ( $\lambda B$ .  $(\cdot) g \cdot B$ ) ( $\Omega$   $n$ )

```

Membership in Ω n .

```

lemma  $\Omega$ -memI [intro]:  $\llbracket B \subseteq G; \text{card } B = n \rrbracket \Longrightarrow B \in \Omega$   $n$ 
  unfolding  $\Omega$ -def by simp

```

```

lemma  $\Omega$ -memD [dest]:  $B \in \Omega$   $n \Longrightarrow B \subseteq G \wedge \text{card } B = n$ 
  unfolding  $\Omega$ -def by simp

```

The cardinality of Ω n is the binomial coefficient $\text{card } G \text{ choose } n$.

```

lemma card- $\Omega$ :
  assumes finite  $G$ 
  shows card ( $\Omega$   $n$ ) = card  $G$  choose  $n$ 
  using assms by (simp add:  $\Omega$ -def  $n$ -subsets)

```

Left multiplication by $g \in G$ is injective on the carrier.

```

lemma left-mult-inj-on:
  assumes  $g \in G$ 
  shows inj-on  $((\cdot) g)$   $G$ 
  by (simp add: assms inj-on-def)

```

Left multiplication maps subsets of G to subsets of G .

lemma *left-mult-image-subset*:

assumes $g \in G \ B \subseteq G$
shows $(\cdot) \ g \ ' \ B \subseteq G$
using *assms* **by** *auto*

Left multiplication preserves cardinality of subsets of G .

lemma *left-mult-image-card*:

assumes $g \in G \ B \subseteq G$
shows $\text{card } ((\cdot) \ g \ ' \ B) = \text{card } B$
by (*meson assms left-mult-inj-on card-image inj-on-subset*)

The action $\varphi\text{-left } n \ g$ maps $\Omega \ n$ to itself.

lemma *$\varphi\text{-left-closed}$* :

assumes $g \in G \ B \in \Omega \ n$
shows $(\cdot) \ g \ ' \ B \in \Omega \ n$
using *assms left-mult-image-subset left-mult-image-card* **by** (*fastforce simp: $\Omega\text{-def}$*)

The action applied to an element of $\Omega \ n$.

lemma *$\varphi\text{-left-apply [simp]}$* :

assumes $B \in \Omega \ n$
shows $\varphi\text{-left } n \ g \ B = (\cdot) \ g \ ' \ B$
using *assms* **unfolding** $\varphi\text{-left-def}$ **by** *simp*

Outside $\Omega \ n$, the action is undefined.

lemma *$\varphi\text{-left-undefined}$* :

assumes $B \notin \Omega \ n$
shows $\varphi\text{-left } n \ g \ B = \text{undefined}$
using *assms* **unfolding** $\varphi\text{-left-def}$ **by** *simp*

The action is extensional on $\Omega \ n$.

lemma *$\varphi\text{-left-extensional}$* : $\varphi\text{-left } n \ g \in \text{extensional } (\Omega \ n)$

unfolding $\varphi\text{-left-def}$ **by** (*rule restrict-extensional*)

The action maps $\Omega \ n$ into itself.

lemma *$\varphi\text{-left-mapsto}$* :

assumes $g \in G$
shows $\varphi\text{-left } n \ g \in \Omega \ n \rightarrow_E \Omega \ n$
using *assms $\varphi\text{-left-closed}$ $\varphi\text{-left-extensional}$*
by (*auto simp: $\Pi E\text{-def}$ $\Pi\text{-def}$ extensional-def $\varphi\text{-left-def}$ restrict-def $\Omega\text{-def}$*)

Left multiplication by $g \in G$ followed by *local.inverse* g is the identity on subsets of G .

lemma *left-mult-image-inverse*:

assumes $g \in G \ B \subseteq G$
shows $(\cdot) \ (\text{inverse } g) \ ' \ ((\cdot) \ g \ ' \ B) = B$
proof (*intro iffI set-eqI*)
fix x

```

assume  $x \in (\cdot) (\text{inverse } g) ' ((\cdot) g ' B)$ 
then obtain  $b \in B$  where  $x = \text{inverse } g \cdot (g \cdot b)$  by blast
then show  $x \in B$ 
  by (metis assms invertible invertible-left-inverse2 subsetD)
next
fix  $x$ 
assume  $x \in B$  then show  $x \in (\cdot) (\text{inverse } g) ' ((\cdot) g ' B)$ 
  by (metis assms imageI in-mono invertible invertible-left-inverse2)
qed

```

The action $\varphi\text{-left } n$ g is a bijection on Ω n .

```

lemma  $\varphi\text{-left-bij}$ :
  assumes  $g \in G$ 
  shows bij-betw ( $\varphi\text{-left } n$   $g$ ) ( $\Omega$   $n$ ) ( $\Omega$   $n$ )
proof (intro bij-betw-byWitness strip)
  fix  $B$  assume  $B: B \in \Omega$   $n$ 
  then have  $BG: B \subseteq G$  and  $gB: (\cdot) g ' B \in \Omega$   $n$ 
    using assms B  $\varphi\text{-left-closed}$  by auto
  with  $B$  show  $\varphi\text{-left } n$  ( $\text{inverse } g$ ) ( $\varphi\text{-left } n$   $g$   $B$ ) =  $B$ 
    using assms BG  $\varphi\text{-left-apply left-mult-image-inverse$ 
    by presburger
next
  fix  $B$  assume  $B: B \in \Omega$   $n$ 
  then have  $BG: B \subseteq G$  by auto
  have  $gB: (\cdot) (\text{inverse } g) ' B \in \Omega$   $n$ 
    using B  $\varphi\text{-left-closed}$  assms invertible invertible-inverse-closed by meson
  have  $\varphi\text{-left } n$   $g$  ( $\varphi\text{-left } n$  ( $\text{inverse } g$ )  $B$ ) =  $(\cdot) g ' ((\cdot) (\text{inverse } g) ' B)$ 
    using B gB  $\varphi\text{-left-apply}$  by presburger
  also have  $\dots = B$ 
    using BG assms invertible invertible-inverse-closed invertible-inverse-inverse
    left-mult-image-inverse by metis
  finally show  $\varphi\text{-left } n$   $g$  ( $\varphi\text{-left } n$  ( $\text{inverse } g$ )  $B$ ) =  $B$  .
next
  show  $\varphi\text{-left } n$   $g ' \Omega$   $n \subseteq \Omega$   $n$ 
    using assms  $\varphi\text{-left-closed}$  by auto
  show  $\varphi\text{-left } n$  ( $\text{inverse } g$ ) '  $\Omega$   $n \subseteq \Omega$   $n$ 
    by (simp add:  $\varphi\text{-left-closed}$  assms image-iff image-subset-iff)
qed

```

The action satisfies the unit law.

```

lemma  $\varphi\text{-left-unit}$ :  $\varphi\text{-left } n$  1 = identity ( $\Omega$   $n$ )
  by (force simp: image-def left-unit  $\varphi\text{-left-def}$  subsetD)

```

The action satisfies the composition law.

```

lemma  $\varphi\text{-left-compose}$ :
  assumes  $g \in G$   $h \in G$ 
  shows  $\varphi\text{-left } n$  ( $g \cdot h$ ) = compose ( $\Omega$   $n$ ) ( $\varphi\text{-left } n$   $g$ ) ( $\varphi\text{-left } n$   $h$ )
proof (rule ext)
  fix  $B$ 

```

```

show  $\varphi\text{-left } n \ (g \cdot h) \ B = \text{compose } (\Omega \ n) \ (\varphi\text{-left } n \ g) \ (\varphi\text{-left } n \ h) \ B$ 
proof (cases  $B \in \Omega \ n$ )
  case True
    then have  $\text{compose } (\Omega \ n) \ (\varphi\text{-left } n \ g) \ (\varphi\text{-left } n \ h) \ B = \varphi\text{-left } n \ g \ ((\cdot) \ h \ ' B)$ 
      by (simp add: compose-eq)
    also have  $\dots = (\cdot) \ g \ ' ((\cdot) \ h \ ' B)$ 
      using assms True  $\varphi\text{-left-closed}$  by simp
    also have  $\dots = (\cdot) \ (g \cdot h) \ ' B$ 
    proof -
      have  $B \subseteq G$  using True by auto
      then show ?thesis
        using associative [OF assms]
        by (smt (cvc5, best) compose-eq image-cong in-mono surj-compose)
    qed
  also have  $\dots = \varphi\text{-left } n \ (g \cdot h) \ B$  using True by simp
  finally show ?thesis by simp
next
  case False
  then show ?thesis by (simp add:  $\varphi\text{-left-def}$  compose-def)
qed
qed

```

The left-translation action on $\Omega \ n$ is a *Group-Action*.

theorem *subset-action*:

```

  Group-Action  $G \ (\cdot) \ \mathbf{1} \ (\varphi\text{-left } n) \ (\Omega \ n)$ 
proof
  fix  $g$  assume  $g \in G$ 
  then show  $\varphi\text{-left } n \ g \in \text{Monoid.Units } (\Omega \ n \rightarrow_E \Omega \ n) \ (\text{compose } (\Omega \ n)) \ (\text{identity } (\Omega \ n))$ 
    by (simp add:  $\varphi\text{-left-bij } \varphi\text{-left-mapsto transformations.Units-bij-betwD}$ )
qed (auto simp:  $\varphi\text{-left-unit } \varphi\text{-left-compose}$ )

```

end

lemma (in *Group*) *Sylow-abelian*:

```

  assumes grp: Abelian-Group  $G \ (\cdot) \ \mathbf{1}$  finite  $G$ 
  and prime: prime  $p$  and dvd:  $p \ \text{dvd} \ \text{card } G$ 
  shows  $\exists g \in G. \ \text{element-order } g = p$ 
  using dvd grp
proof (induction card  $G$  arbitrary:  $G \ (\cdot) \ \mathbf{1}$  rule: less-induct)
  case less
  then interpret  $G$ : Abelian-Group  $G$  composition unit by simp
  have  $\text{card } G \neq 1$ 
    using less.prems prime by force
  with less obtain  $a$  where  $a: a \in G \ a \neq \text{unit}$ 
    using  $G.\text{unit-closed}$  by (metis empty-iff is-singletonI' is-singleton-altdef)
  let ? $r = G.\text{element-order } a$ 
  show ?case
proof (cases  $p \ \text{dvd} \ ?r$ )
  case True

```

```

    then obtain  $r'$  where  $r'$ :  $?r = p * r'$  and pow-eq:  $G.\text{power } (G.\text{power } a \ r') \ p =$ 
unit
    using  $G.\text{power-element-order } G.\text{power-mult } \langle a \in G \rangle \text{ less.prem}(3)$  by (metis dvdE
mult.commute)
    have fin: finite  $G$  using less by simp
    have aG:  $G.\text{power } a \ r' \in G$  using a by simp
    have dvd-p:  $G.\text{element-order } (G.\text{power } a \ r') \text{ dvd } p$ 
      using  $G.\text{element-order-dvd}[OF \text{ fin } aG]$  pow-eq by blast
    have False if  $G.\text{element-order } (G.\text{power } a \ r') = 1$ 
    proof -
      have  $G.\text{element-order } a \text{ dvd } r'$ 
      using  $G.\text{element-order-dvd } G.\text{power-1 } G.\text{power-element-order } \langle a \in G \rangle \ aG$  fin that
    by metis
      then show False
        using  $G.\text{element-order-pos}[OF \text{ fin } \langle a \in G \rangle] \ r' \text{ prime}$  by simp
    qed
    with dvd-p show ?thesis using aG
      using prime prime-nat-iff by blast
  next
    case not-dvd: False
    have fin: finite  $G$  using less by simp
    define  $C$  where  $C \equiv G.\text{cyclic-subgroup } a$ 
    have C-sub:  $C \subseteq G$  unfolding C-def using  $G.\text{cyclic-subgroup-subset } a$  by simp
    have card-C:  $\text{card } C = ?r$  unfolding C-def using  $G.\text{card-cyclic-subgroup}[OF \text{ fin } \langle a \in G \rangle]$  .
    interpret sub-C: Subgroup  $C \ G$  composition unit
      unfolding C-def using  $G.\text{cyclic-subgroup-is-subgroup}[OF \text{ fin } \langle a \in G \rangle]$  .
    interpret CG: subgroup-of-abelian-group  $C \ G$  composition unit ..
    have card-quotient:  $\text{card } CG.\text{Factor-Group} = CG.\text{index}$ 
      using  $CG.\text{card-Factor-Group}$  less by simp
    then have  $\langle \text{card } C < \text{card } G \rangle$ 
      by (metis C-sub card-C card-mono fin le-neq-implies-less less.prem(1) not-dvd)

    define FactG where  $\text{FactG} \equiv \text{some-elem } \langle CG.\text{Factor-Group}$ 
    define comp where  $\text{comp} \equiv \text{class-representatives.rep-comp } G \ CG.\text{Factor-Group}$ 
    CG.quotient-composition
    interpret iso: group-isomorphism restrict some-elem  $CG.\text{Factor-Group}$ 
       $CG.\text{Factor-Group}$   $CG.\text{quotient-composition } CG.\text{Class unit}$ 
       $\text{FactG comp some-elem } (CG.\text{Class unit})$ 
      using  $CG.\text{group-isomorphism-Factor-Group-class-reps}$ 
      by (force simp: FactG-def local.comp-def)
    have *:  $\text{card } C * \text{card } \text{FactG} = \text{card } G$ 
      using  $CG.\text{card-class-reps } CG.\text{lagrange } \text{FactG-def card-quotient fin}$  by presburger
    then have  $\text{card } \text{FactG} > 1$ 
      by (metis  $\langle \text{card } C < \text{card } G \rangle$  mult.commute nat-mult-1 nat-mult-less-cancel-disj)
    have finite FactG
      using  $CG.\text{class-reps-subset } \text{FactG-def fin finite-subset}$  by blast
    have  $\langle p \text{ dvd } \text{card } \text{FactG} \rangle$ 
      by (metis * card-C less.prem(1) not-dvd prime prime-dvd-multD)
    have abel-FactG: Abelian-Group  $\text{FactG comp (some-elem } (CG.\text{Class unit}))$ 

```



```

    using CG.abelian-class-reps[OF CG.quotient-abelian] by (simp add: FactG-def
local.comp-def)
  have card C > 1
    using G.element-order-pos G.power-1 G.power-element-order a(1,2) card-C fin
    by (metis One-nat-def antisym-conv3 not-less-eq)
  then have card FactG < card G
    using * ⟨1 < card FactG⟩
    by (metis One-nat-def bot-nat-0.not-eq-extremum n-less-m-mult-n not-less-iff-gr-or-eq)
  then obtain g where g: g ∈ FactG Monoid.element-order comp (some-elem
(CG.Class unit)) g = p
    using less(1) ⟨p dvd card FactG⟩ abel-FactG ⟨finite FactG⟩ ⟨1 < card FactG⟩
  by blast
  then obtain X where X: X ∈ CG.Factor-Group some-elem X = g by (auto
simp: FactG-def)
  obtain g' where g': g' ∈ G X = CG.Class g'
    using CG.representant-exists[OF X(1)] by auto
  have finFG: finite CG.Factor-Group
    using CG.Partition-def fin finite-imageI by auto
  then have nat-dvd: CG.quotient.element-order (CG.Class g') dvd G.element-order
g'
    by (metis CG.natural.commutates-with-power CG.quotient.element-order-dvd G.power-element-order
X(1) fin g')
  have iso.image.sub.element-order (restrict some-elem CG.Factor-Group X)
    dvd CG.quotient.element-order X
    using iso.element-order-image-dvd[OF finFG ⟨finite FactG⟩ X(1)] by simp
  then have p dvd CG.quotient.element-order (CG.Class g')
    using X g g' by auto
  then show ?thesis
    using G.element-order-power-prime fin g'(1) gcd-nat.trans nat-dvd prime by
blast
qed
qed

```

9.1 Conjugation, Center, and Centralizer

context *Group* **begin**

The conjugation action: g acts on x by $g x g^{-1}$. We use $g x g^{-1}$ rather than $g^{-1} x g$ so that the map $g \mapsto (x \mapsto g x g^{-1})$ is a group homomorphism (not anti-homomorphism).

definition *conjugation* :: $'a \Rightarrow 'a \Rightarrow 'a$
where *conjugation* $g x \equiv$ (if $g \in G \wedge x \in G$ then $g \cdot x \cdot \text{inverse } g$ else x)

lemma *conjugation-closed*:
assumes $g \in G \ x \in G$
shows *conjugation* $g x \in G$
using *assms* **by** (simp add: *conjugation-def*)

lemma *conjugation-unit*:

assumes $x \in G$
shows $\text{conjugation } 1 \ x = x$
using *assms* **by** (*simp add: conjugation-def*)

lemma *conjugation-compose*:
assumes $g \in G \ h \in G$
shows $\text{conjugation } (g \cdot h) \ x = \text{conjugation } g \ (\text{conjugation } h \ x)$
by (*simp add: assms associative conjugation-def inverse-composition-commute*)

lemma *conjugation-outside* [*simp*]:
 $x \notin G \implies \text{conjugation } g \ x = x$
by (*simp add: conjugation-def*)

lemma *conjugation-bij*:
assumes $g \in G$
shows *bij-betw* (*conjugation* g) $G \ G$
proof (*rule bij-betwI*)
show $\bigwedge x. x \in G \implies \text{conjugation } (\text{inverse } g) \ (\text{conjugation } g \ x) = x$
using *assms* **by** (*simp add: conjugation-def associative invertible-left-inverse2*)
show $\bigwedge y. y \in G \implies \text{conjugation } g \ (\text{conjugation } (\text{inverse } g) \ y) = y$
using *assms* **by** (*simp add: conjugation-def associative invertible-right-inverse2*)
qed (*auto simp: conjugation-closed assms*)

The restricted conjugation map, suitable for the *Group-Action* locale.

definition $\varphi\text{-conj} :: 'a \Rightarrow 'a \Rightarrow 'a$
where $\varphi\text{-conj } g = \text{restrict } (\text{conjugation } g) \ G$

lemma $\varphi\text{-conj-apply}$ [*simp*]:
assumes $g \in G \ x \in G$
shows $\varphi\text{-conj } g \ x = g \cdot x \cdot \text{inverse } g$
using *assms* **by** (*simp add: $\varphi\text{-conj-def}$ conjugation-def*)

lemma $\varphi\text{-conj-outside}$ [*simp*]:
 $x \notin G \implies \varphi\text{-conj } g \ x = \text{undefined}$
by (*simp add: $\varphi\text{-conj-def}$*)

lemma $\varphi\text{-conj-extensional}$: $\varphi\text{-conj } g \in \text{extensional } G$
unfolding $\varphi\text{-conj-def}$ **by** (*rule restrict-extensional*)

lemma $\varphi\text{-conj-mapsto}$:
assumes $g \in G$
shows $\varphi\text{-conj } g \in G \rightarrow_E G$
using *assms* **by** (*auto simp: PiE-def Pi-def $\varphi\text{-conj-def}$ conjugation-closed*)

lemma $\varphi\text{-conj-bij}$:
assumes $g \in G$
shows *bij-betw* ($\varphi\text{-conj } g$) $G \ G$
by (*simp add: $\varphi\text{-conj-def}$ assms conjugation-bij*)

lemma $\varphi\text{-conj-unit}$: $\varphi\text{-conj } 1 = \text{identity } G$

by *auto*

lemma φ -conj-compose:

assumes $g \in G$ $h \in G$

shows $\varphi\text{-conj } (g \cdot h) = \text{compose } G (\varphi\text{-conj } g) (\varphi\text{-conj } h)$

by (*force simp: conjugation-closed φ -conj-def assms compose-def conjugation-compose*)

theorem *conjugation-action*: $\text{Group-Action } G (\cdot) \mathbf{1} \varphi\text{-conj } G$

proof

fix g **assume** $g \in G$

then show $\varphi\text{-conj } g \in \text{Monoid.Units } (G \rightarrow_E G) (\text{compose } G) (\text{identity } G)$

by (*simp add: φ -conj-bij φ -conj-mapsto transformations.Units-bijective*)

qed (*auto simp: φ -conj-unit φ -conj-compose*)

The center of G : elements that commute with everything.

definition *center* :: 'a set

where $\text{center} = \{z \in G. \forall g \in G. z \cdot g = g \cdot z\}$

The centralizer of an element y : elements that commute with y .

definition *centralizer* :: 'a $\Rightarrow$ 'a set

where $\text{centralizer } y = \{g \in G. g \cdot y = y \cdot g\}$

lemma *center-subset*: $\text{center} \subseteq G$

by (*auto simp: center-def*)

lemma *center-mem-iff*:

assumes $z \in G$

shows $z \in \text{center} \iff (\forall g \in G. z \cdot g = g \cdot z)$

using *assms* **by** (*simp add: center-def*)

lemma *centralizer-subset*: $\text{centralizer } y \subseteq G$

by (*auto simp: centralizer-def*)

lemma *centralizer-mem-iff*:

assumes $g \in G$ $y \in G$

shows $g \in \text{centralizer } y \iff g \cdot y = y \cdot g$

using *assms* **by** (*simp add: centralizer-def*)

Interpret the conjugation action to access orbits, stabilizers, and fixed points.

interpretation *conj*: $\text{Group-Action } G (\cdot) \mathbf{1} \varphi\text{-conj } G$

by (*rule conjugation-action*)

lemma *center-eq-fixed-points*: $\text{center} = \text{conj.fixed-points}$

unfolding *center-def conj.fixed-points-def conj.stabilizer-def*

by (*force simp: subset-iff simp flip: commute-iff-inverse*)

The class equation separates the singleton conjugacy classes, which are exactly the center, from the nontrivial conjugacy classes.

theorem *class-equation*:

```

assumes finite G
shows  $\text{card } G = \text{card center} + (\sum X \in \text{conj.nontrivial-orbits. card } X)$ 
using conj.fixed-point-class-equation[OF assms]
by (simp add: center-eq-fixed-points)

```

```

lemma centralizer-eq-stabilizer:
  assumes  $y \in G$ 
  shows  $\text{centralizer } y = \text{conj.stabilizer } y$ 
  using assms
  by (auto simp: centralizer-def commute-iff-inverse conj.stabilizer-def)

```

The center is a subgroup of G .

```

lemma inverse-commutes:
  assumes  $x \in G \ \forall g \in G. x \cdot g = g \cdot x$ 
  shows  $\forall g \in G. \text{inverse } x \cdot g = g \cdot \text{inverse } x$ 
proof
  fix  $g$  assume  $gG: g \in G$ 
  have  $x \cdot (\text{inverse } x \cdot g) = g$ 
    using assms(1) gG by (simp add: associative [symmetric])
  moreover have  $x \cdot (g \cdot \text{inverse } x) = g$ 
    by (simp add: assms associative gG)
  ultimately show  $\text{inverse } x \cdot g = g \cdot \text{inverse } x$ 
    using assms(1) gG
    by (metis composition-closed invertible invertible-inverse-closed
      invertible-left-inverse2)
qed

```

```

lemma center-subgroup: Subgroup center G (·) 1
proof (rule subgroupI)
  fix  $x \ y$  assume  $x \in \text{center} \ y \in \text{center}$ 
  then show  $x \cdot y \in \text{center}$ 
    unfolding center-def mem-Collect-eq
    by (metis associative composition-closed)
  next
  fix  $x$  assume  $x: x \in \text{center}$ 
  then show  $\text{invertible } x$  using center-subset by auto
  show  $\text{inverse } x \in \text{center}$ 
    using  $x$  center-subset inverse-commutes by (auto simp: center-def)
qed (auto simp add: center-def)

```

The center is an abelian group.

```

lemma center-abelian: Abelian-Group center (·) 1
proof –
  interpret  $CG: \text{subgroup-of-group center } G (\cdot) 1$ 
    by (simp add: Group-axioms center-subgroup subgroup-of-group-def)
  show ?thesis
  proof qed (use center-subset center-def in blast)
qed

```

The centralizer of $y \in G$ is a subgroup of G .

```

lemma centralizer-subgroup:
  assumes  $y \in G$ 
  shows Subgroup (centralizer  $y$ )  $G$  ( $\cdot$ ) 1
  using conj.stabilizer-subgroup[OF assms] centralizer-eq-stabilizer[OF assms] by simp

```

If y is not in the center, then its centralizer is a proper subgroup.

```

lemma centralizer-proper:
  assumes  $y \in G$   $y \notin \text{center } G$ 
  shows  $\text{card } (\text{centralizer } y) < \text{card } G$ 
proof –
  have  $\text{centralizer } y \subseteq G$  by (rule centralizer-subset)
  moreover have  $\text{centralizer } y \neq G$ 
    by (metis assms center-mem-iff centralizer-mem-iff)
  ultimately show ?thesis using assms( $\beta$ ) by (simp add: psubsetI psubset-card-mono)
qed

```

end

Preimage of a subgroup under a group homomorphism is a subgroup.

```

lemma (in group-homomorphism) preimage-subgroup:
  assumes sub: Subgroup  $S$   $G'$  ( $\cdot'$ ) 1'
  defines  $H \equiv \{g \in G. \eta \ g \in S\}$ 
  shows Subgroup  $H$   $G$  ( $\cdot$ ) 1
proof (rule source.subgroupI)
  show  $H \subseteq G$  1  $\in H$ 
    using commutes-with-unit sub Submonoid.sub-unit-closed by (fastforce simp: H-def Subgroup-def)
  next
    fix  $x \ y$  assume  $x \in H$   $y \in H$ 
    then have  $xG: x \in G$  and  $yG: y \in G$  and  $xS: \eta \ x \in S$  and  $yS: \eta \ y \in S$ 
      unfolding H-def by auto
    have  $\eta \ (x \cdot y) = \eta \ x \cdot' \eta \ y$  using commutes-with-composition  $xG \ yG$  by auto
    also have  $\dots \in S$  using sub xS yS
      by (meson Group-def Monoid.composition-closed Subgroup-def)
    finally show  $x \cdot y \in H$  unfolding H-def using  $xG \ yG$  by auto
  next
    fix  $x$  assume  $x \in H$ 
    then have  $xG: x \in G$  and  $xS: \eta \ x \in S$  unfolding H-def by auto
    show source.invertible  $x$  using  $xG$  by auto
    have  $\eta \ (\text{source.inverse } x) = \text{target.inverse } (\eta \ x)$ 
      using invertible-commutes-with-inverse  $xG$  by blast
    also have  $\dots \in S$ 
      using sub target.Gen-def target.Gen-subgroup-eq target.generate.inv xS by blast
    finally show source.inverse  $x \in H$ 
      by (auto simp: H-def xG)
qed

```

Cardinality of the preimage: each coset in the subgroup contributes $\text{card } \text{Ker}$ elements.

lemma (in *group-homomorphism*) *preimage-card*:
assumes *sub*: Subgroup $S \subseteq G$ ($\cdot$) $1'$ **and** *fin*: finite G **and** *fin'*: finite G'
and *img*: $S \subseteq \eta^{-1} G$
defines $H \equiv \{g \in G. \eta g \in S\}$
shows $\text{card } H = \text{card } S * \text{card } \text{Ker}$
proof –
– Partition H into fibers.
define *fiber* **where** $\text{fiber } s \equiv \{g \in G. \eta g = s\}$ **for** s
have *H-eq*: $H = (\bigcup s \in S. \text{fiber } s)$
unfolding *H-def fiber-def* **by** *auto*
have *fib-class*: $\exists g \in G. \text{fiber } s = \text{kernel.Class } g$ **if** $s \in S$ **for** s
– Each fiber over $s \in S$ is nonempty.
proof –
obtain g **where** gG : $g \in G$ **and** sg : $\eta g = s$
using $\langle s \in S \rangle \text{img}$ **by** *auto*
have $\text{fiber } s = \text{kernel.Class } g$
proof (*intro set-eqI iffI*)
fix x **assume** $x \in \text{fiber } s$
then have xG : $x \in G$ **and** $\eta x = s$ $\eta x = \eta g$
unfolding *fiber-def* **using** sg **by** *auto*
then have $\eta (\text{inverse } x \cdot g) = 1'$
using *invertible-image-lemma* sg xG $\langle g \in G \rangle$ **by** (*auto simp add: commutes-with-composition*)
then have $\text{inverse } x \cdot g \in \text{Ker}$
using *Ker-memI* gG xG **by** *blast*
then have $(x, g) \in \text{kernel.Congruence}$
by (*metis* gG *kernel.CongruenceI* *kernel.symmetric* *source.invertible* *source.invertible-right-inverse2* xG)
then show $x \in \text{kernel.Class } g$
by (*rule* *kernel.ClassI*)
next
fix x **assume** $x \in \text{kernel.Class } g$
then have $(x, g) \in \text{kernel.Congruence}$
using *kernel.ClassD* gG **by** *blast*
then have xG : $x \in G$ **using** *kernel.left-closed* **by** *auto*
from $\langle (x, g) \in \text{kernel.Congruence} \rangle$ **obtain** k **where** kK : $k \in \text{Ker}$ **and** xgk : $x = g \cdot k$
using *kernel.CongruenceD* **by** *auto*
then have $\eta x = \eta g \cdot \eta k$
using *commutes-with-composition* gG *Ker-closed[OF kK]* **by** *auto*
then show $x \in \text{fiber } s$ **unfolding** *fiber-def* **using** xG
using *Ker-image* gG kK sg **by** *force*
qed
then show $\exists g \in G. \text{fiber } s = \text{kernel.Class } g$ **using** gG **by** *auto*
qed
– Each fiber has exactly $\text{card } \text{Ker}$ elements.
have *fib-card*: $\bigwedge s. s \in S \implies \text{card } (\text{fiber } s) = \text{card } \text{Ker}$
by (*metis* *fib-class* *kernel.Class-cardinality*)
– The fibers are pairwise disjoint.
have *fib-disj*: *disjoint-family-on fiber S*

```

    unfolding disjoint-family-on-def fiber-def by auto
    — Fibers are finite (since  $G$  is finite).
    have fib-fin:  $\bigwedge s. s \in S \implies \text{finite } (\text{fiber } s)$ 
    unfolding fiber-def using fin by auto
    have finite  $S$ 
    using fin' sub by (meson finite-subset Subgroup-def Submonoid.subset)
    then show ?thesis
    using card-UN-disjoint'[OF fib-disj fib-fin] by (simp add: H-eq fib-card)
qed

```

9.2 First Sylow Theorem

```

theorem (in Group) Sylow-I:
  fixes  $p :: \text{nat}$  and  $m :: \text{nat}$ 
  assumes prime  $p$ 
    and finite  $G$ 
    and  $p \wedge^m \text{dvd card } G$ 
  shows  $\exists H. \text{Subgroup } H \ G \ (\cdot) \ 1 \wedge \text{card } H = p \wedge^m$ 
  using assms(3,2) Group-axioms
proof (induction card  $G$  arbitrary:  $G \ (\cdot) \ 1 \ m$  rule: less-induct)
case less
then interpret  $G$ : Group  $G$  composition unit
by meson
show ?case
proof (cases  $m = 0$ )
case True
then have Subgroup  $\{\text{unit}\} \ G$  composition unit  $\wedge \text{card } \{\text{unit}\} = p \wedge^m$ 
by (auto intro: G.subgroupI)
then show ?thesis by blast
next
case False
have fin: finite  $G$  using less.prem by simp
have pa-dvd:  $p \wedge^m \text{dvd card } G$  using less.prem by simp
have p-dvd-G:  $p \text{dvd card } G$ 
using False pa-dvd by auto
show ?thesis
proof (cases  $p \text{dvd card } G.\text{center}$ )
case True
— By Cauchy's theorem for abelian groups, the center contains an element of
order  $p$ .
have center-fin: finite  $G.\text{center}$ 
using fin G.center-subset finite-subset by blast
interpret  $Z$ : Group  $G.\text{center}$  composition unit
using G.center-subgroup Subgroup.axioms(2) by force
obtain  $z$  where  $zZ: z \in G.\text{center}$  and  $z\text{-ord}: Z.\text{element-order } z = p$ 
using Z.Sylow-abelian[OF G.center-abelian center-fin  $\langle \text{prime } p \rangle$  True]
by auto
have zG:  $z \in G$  using zZ G.center-subset by auto
have z-comm:  $\bigwedge g. g \in G \implies \text{composition } z \ g = \text{composition } g \ z$ 
using zZ G.center-def by auto

```

— The element orders in the center and in G coincide.

```

have Z.power z n = G.power z n for n
  by (induction n) auto
then have z-ord-G: G.element-order z = p using z-ord
  unfolding Z.element-order-def G.element-order-def by simp
have p-pos: p > 0 using ⟨prime p⟩ prime-gt-0-nat by blast
— Build the cyclic subgroup  $N = \langle z \rangle$  of order  $p$ .
define N where N = G.power z ‘{.. $p$ }
have N-alt: N = range (G.power z)
  using G.power-mod-order[OF fin zG] p-pos z-ord-G
  by (fastforce simp: N-def)
have N-sub: N ⊆ G unfolding N-def using zG by auto
have card-N: card N = p
  using G.power-inj-on-order[OF fin zG] z-ord-G
  by (simp add: N-def card-image)
have unit-N: unit ∈ N unfolding N-alt
  by (metis G.power-0 rangeI)
have comp-N: composition x y ∈ N if x ∈ N y ∈ N for x y
  using that zG by (force simp: N-alt simp flip: G.power-add)
have inv-N: G.inverse x ∈ N if x ∈ N for x
proof —
  obtain i where xi: x = G.power z i
    using that N-alt ⟨x ∈ N⟩ by blast
  have xG: x ∈ G using that N-sub by auto
  define y where y ≡ G.power z ((p - 1) * i)
  have yN: y ∈ N unfolding y-def N-alt by auto
  have yG: y ∈ G using yN N-sub by auto
  have composition x y = G.power z (i + (p - 1) * i)
    unfolding xi y-def using zG G.power-add by auto
  also have i + (p - 1) * i = p * i using p-pos by (simp add: algebra-simps)
  also have G.power z (p * i) = G.power (G.power z p) i
    using zG G.power-mult by simp
  also have G.power (G.power z p) i = unit
    using G.power-element-order[OF fin zG] z-ord-G by simp
  finally have xy: composition x y = unit .
  then show ?thesis
    by (metis G.center-mem-iff G.inverse-equality Z.power-closed xG xi yG yN
zZ)
qed
have sub-N: Subgroup N G composition unit
  using G.subgroupI N-sub comp-N inv-N unit-N by blast
— Step 3:  $N$  is m normal subgroup of  $G$  (since  $z$  is central).
interpret NG: subgroup-of-group N G composition unit
  by (simp add: G.Group-axioms sub-N subgroup-of-group-def)
have N-normal: normal-subgroup N G composition unit
proof (rule NG.Left-equals-Right-coset-implies-normality)
  fix g assume gG: g ∈ G
  show NG.Left-Coset g N = NG.Right-Coset N g
proof —
  have power-comm: composition g (G.power z i) = composition (G.power z i)

```



```

g for i
  proof (induction i)
    case 0
    then show ?case by (simp add: gG)
  next
    case (Suc i) then show ?case
      by (metis G.center-mem-iff G.power-closed Z.power-closed gG zG zZ)
  qed
  show ?thesis
  proof (intro set-eqI iffI)
    fix x
    assume x ∈ NG.Left-Coset g N
    then obtain n where nN: n ∈ N and xeq: x = composition g n
      by (auto simp: NG.Left-Coset-def)
    from nN obtain i where ni: n = G.power z i
      using N-alt by blast
    then show x ∈ NG.Right-Coset N g
      using nN power-comm xeq by auto
  next
    fix x
    assume x ∈ NG.Right-Coset N g
    then obtain n i where n ∈ N x = composition n g n = G.power z i
      using N-alt by blast
    then show x ∈ NG.Left-Coset g N
      using NG.Left-Coset-memI power-comm by metis
  qed
qed
qed
— Step 4: Form the quotient group  $G/N$  and apply the induction hypothesis.
interpret NG: normal-subgroup N G composition unit
  using N-normal .
have card-FG: card NG.Factor-Group = NG.index
  using NG.card-Factor-Group fin by simp
have card-G-eq: card G = card N * NG.index
  using NG.lagrange fin by simp
then have card-G-eq': card G = p * card NG.Factor-Group
  using card-N card-FG by simp
have FG-lt: card NG.Factor-Group < card G
  using assms(1) card-G-eq' fin prime-nat-iff zG by fastforce
show ?thesis
proof —
  — The quotient group is finite.
  have fin-FG: finite NG.Factor-Group
    using fin NG.partition partition-on-def by (metis finite-UnionD)
  — Divisibility:  $p^{m-1}$  divides card NG.Factor-Group.
  have p ^ m dvd p * card NG.Factor-Group
    using pa-dvd card-G-eq' by simp
  then have pa1-dvd: p ^ (m - 1) dvd card NG.Factor-Group
    by (metis False dvd-mult-cancel p-pos power-eq-if)

```

— Apply the induction hypothesis to the class-representative group (type '*a* set').

```

define FactG where FactG  $\equiv$  some-elem ' NG.Factor-Group
define fcomp where fcomp  $\equiv$  class-representatives.rep-comp G NG.Factor-Group
NG.quotient-composition
interpret iso: group-isomorphism restrict some-elem NG.Factor-Group
NG.Factor-Group NG.quotient-composition NG.Class unit
FactG fcomp some-elem (NG.Class unit)
using NG.group-isomorphism-Factor-Group-class-reps
by (force simp: FactG-def fcomp-def)
interpret Group-FactG: Group FactG fcomp some-elem (NG.Class unit)
using iso.target.Group-axioms .
have fin-FactG: finite FactG
using NG.class-reps-subset FactG-def fin finite-subset by blast
have card-FactG: card FactG = card NG.Factor-Group
using NG.card-class-reps FactG-def by simp
have FG-lt': card FactG < card G
using FG-lt card-FactG by simp
obtain K' where K'-sub: Subgroup K' FactG fcomp (some-elem (NG.Class
unit))
and K'-card: card K' =  $p \wedge (m - 1)$ 
by (metis FG-lt' card-FactG fin-FactG iso.target.Group-axioms less.hyps
pa1-dvd)
— Transfer K' back to NG.Factor-Group via the inverse isomorphism.
define inv- $\eta$  where inv- $\eta$   $\equiv$  restrict (inv-into NG.Factor-Group (restrict
some-elem NG.Factor-Group)) FactG
interpret inv-iso: group-isomorphism inv- $\eta$  FactG fcomp some-elem (NG.Class
unit)
NG.Factor-Group NG.quotient-composition NG.Class unit
using iso.inverse-group-isomorphism by (simp add: inv- $\eta$ -def)
define K where K  $\equiv$  inv- $\eta$  ' K'
have K-sub: Subgroup K NG.Factor-Group NG.quotient-composition (NG.Class
unit)
unfolding K-def using inv-iso.image-subgroup[OF K'-sub] .
have inj-on inv- $\eta$  K'
using inv-iso.injective K'-sub
by (meson inj-on-subset Subgroup-def Submonoid.subset)
then have K-card: card K =  $p \wedge (m - 1)$ 
unfolding K-def using K'-card card-image by metis
— Pull back K to G via the natural homomorphism.
define H where H  $\equiv$  {g  $\in$  G. NG.Class g  $\in$  K}
— H is m subgroup of G.
interpret NG-hom: group-homomorphism NG.Class G composition unit
NG.Factor-Group NG.quotient-composition NG.Class unit
by (simp add: G.Group-axioms NG.natural.Monoid-homomorphism-axioms
NG.quotient.Group-axioms
group-homomorphism-def)
have H-sub: Subgroup H G composition unit
unfolding H-def using K-sub NG-hom.preimage-subgroup by blast
— Compute card H.

```

```

have K-in-image:  $K \subseteq NG.Class \text{ ' } G$ 
using K-sub NG.natural.surjective by (metis Subgroup-def Submonoid.subset)
have card H = card K * card NG-hom.Ker
  unfolding H-def
  using NG-hom.preimage-card[OF K-sub fin fin-FG K-in-image] .
  — The kernel of the natural map is  $N$ , so  $card\ NG-hom.Ker = p$ .
moreover have NG-hom.Ker =  $N$ 
proof
  show NG-hom.Ker  $\subseteq N$ 
  using NG.Normal-def NG-hom.Ker-image subsetI by blast
  show  $N \subseteq NG-hom.Ker$ 
  using G.left-unit G.unit-closed NG.Class-eq NG.CongruenceI NG.sub
  NG-hom.Ker-memI
  by (metis subsetI)
qed
moreover have card K * card N =  $p \wedge m$ 
  by (simp add: False power-eq-if K-card card-N)
ultimately have card H =  $p \wedge m$  by simp
with H-sub show ?thesis by blast
qed
next
case False
— Case 1:  $p$  does not divide  $card\ G.center$ .
interpret conj: Group-Action G composition unit G. $\varphi$ -conj G
  by (rule G.conjugation-action)
— The conjugation action partitions  $G$  into orbits. By orbit-stabilizer, the class
equation gives  $|G| = |Z(G)| + \sum [G : C(y-j)]$ . Since  $p$  divides  $card\ G$  but not  $card\ G.center$ ,
some non-trivial orbit must have size not divisible by  $p$ .
have False if  $\forall y \in G. y \notin G.center \longrightarrow p\ dvd\ card\ (conj.orbit\ y)$ 
proof —
  have  $p\ dvd\ card\ (G - G.center)$ 
  using conj.dvd-card-nonfixed-points[OF fin]
  by (simp add: G.center-eq-fixed-points that)
  with False show False
  by (metis G.center-subset card-Diff-subset card-mono dvd-diffD1 fin finite-subset
  p-dvd-G)
qed
then obtain y where yG:  $y \in G$  and y-not-center:  $y \notin G.center$ 
  and not-dvd-orbit:  $\neg p\ dvd\ card\ (conj.orbit\ y)$ 
  by blast
define X where  $X \equiv conj.orbit\ y$ 
— The class equation gives  $|G| = |Z(G)| + \sum [G : C(y-j)]$ . Since  $p$  divides  $card\ G$ 
but not  $card\ G.center$ , some non-trivial conjugacy class must have size coprime to
 $p$ .
have orbit-stab:  $card\ G = card\ X * card\ (conj.stabilizer\ y)$ 
  using conj.orbit-stabilizer-card[OF yG fin] X-def by simp
have stab-eq:  $conj.stabilizer\ y = G.centralizer\ y$ 
  using G.centralizer-eq-stabilizer[OF yG] by simp
have coprime (card X) ( $p \wedge m$ )
  using X-def assms(1) not-dvd-orbit prime-imp-power-coprime by blast

```

```

then have pa-dvd-cent:  $p \wedge m \text{ dvd card } (G.\text{centralizer } y)$ 
by (metis pa-dvd orbit-stab stab-eq coprime-dvd-mult-right-iff coprime-commute)
have cent-grp: Group (G.centralizer y) composition unit
  using G.centralizer-subgroup yG Subgroup-def by meson
have cent-fin: finite (G.centralizer y)
  using fin G.centralizer-subset by (rule rev-finite-subset)
  — By the induction hypothesis, the centralizer has m subgroup of order  $p^m$ .
have card (G.centralizer y) < card G
  using G.centralizer-proper[OF yG y-not-center fin] .
then obtain H where H-sub: Subgroup H (G.centralizer y) composition unit
  and H-card: card H =  $p \wedge m$ 
  using cent-fin cent-grp less.hyps pa-dvd-cent by blast
then show ?thesis
  using subgroup-transitive[OF H-sub] G.centralizer-subgroup[OF yG] by auto
qed
qed
qed

```

Cauchy's theorem for arbitrary finite groups is now an immediate corollary of the first Sylow theorem: taking $m = 1$ yields a subgroup of order p , and a group of prime order is cyclic, so any of its non-unit elements has order exactly p .)

corollary (in Group) *Cauchy*:

```

fixes p :: nat
assumes prime: prime p and fin: finite G and dvd: p dvd card G
shows  $\exists g \in G. \text{element-order } g = p$ 
proof —
  have  $p \wedge 1 \text{ dvd card } G$  using dvd by simp
  then obtain H where H: Subgroup H G ( $\cdot$ ) 1 and cardH: card H = p
    using Sylow-I[OF prime fin, of 1] by auto
  interpret H: Subgroup H G ( $\cdot$ ) 1 by (rule H)
  have finH: finite H using fin H.subset by (rule rev-finite-subset)
  have card H  $\neq 1$ 
    using cardH prime by force
  then obtain g where gH:  $g \in H$  and gne:  $g \neq 1$ 
    using is-singleton-altdef is-singleton-iff-ex1[of H] by blast
  — The element order comes from ( $\cdot$ ) and 1 alone, so it agrees H and G.
  have dvdp: H.sub.element-order g dvd p
    using H.sub.element-order-dvd-card[OF finH gH] cardH by simp
  have H.sub.element-order g  $\neq 1$ 
    using fin gH gne power-element-order by force
  with gH H.subset show ?thesis
    using dvdp prime by (auto simp: prime-nat-iff)
qed

```

10 The Second Sylow Theorem

If H is a subgroup of a group G and g is in G , then the conjugate $g H g^{-1}$ is again a subgroup of G .

context *Group* **begin**

definition *conjugate-subgroup* :: 'a $\Rightarrow$ 'a set $\Rightarrow$ 'a set
where *conjugate-subgroup* *g* *H* = *conjugation* *g* ' *H*

lemma *conjugate-subgroup-mem-iff*:
assumes *g* $\in$ *G* *H* $\subseteq$ *G* *x* $\in$ *G*
shows *x* $\in$ *conjugate-subgroup* *g* *H* $\longleftrightarrow$ ($\exists$ *h* $\in$ *H*. *x* = *g* $\cdot$ *h* $\cdot$ *inverse* *g*)
unfolding *conjugate-subgroup-def* **using** *assms*
by (*auto simp: conjugation-def*)

lemma *conjugate-subgroup-subset*:
assumes *H* $\subseteq$ *G*
shows *conjugate-subgroup* *g* *H* $\subseteq$ *G*
using *assms* **by** (*auto simp: conjugate-subgroup-def conjugation-def*)

lemma *conjugate-subgroup-card*:
assumes *H* $\subseteq$ *G* *finite* *H*
shows *card* (*conjugate-subgroup* *g* *H*) = *card* *H*
using *card-image*[*of* *conjugation* *g* *H*] *inj-on-subset*[*OF* *bij-betw-imp-inj-on*[*OF* *conjugation-bij*] 'H $\subseteq$ *G*']
by (*metis* *conjugation-def* *conjugate-subgroup-def* *conjugation-unit* *ext* *unit-closed*)

Conjugation is a homomorphism: it distributes over products and inverses.

lemma *conjugation-hom*:
assumes *h1* $\in$ *G* *h2* $\in$ *G*
shows *conjugation* *g* *h1* $\cdot$ *conjugation* *g* *h2* = *conjugation* *g* (*h1* $\cdot$ *h2*)
using *assms* **by** (*simp add: conjugation-def associative invertible-left-inverse2*)

lemma *conjugation-inverse*:
assumes *h* $\in$ *G*
shows *conjugation* *g* (*inverse* *h*) = *inverse* (*conjugation* *g* *h*)
using *conjugation-def* *assms* *inverse-composition-commute* **by** *auto*

lemma *conjugate-subgroup-is-subgroup*:
assumes *sub*: *Subgroup* *H* *G* ($\cdot$) **1** **and** *g*: *g* $\in$ *G*
shows *Subgroup* (*conjugate-subgroup* *g* *H*) *G* ($\cdot$) **1**
proof –
interpret *H*: *Subgroup* *H* *G* ($\cdot$) **1** **using** *sub* .
show ?*thesis*
proof (*rule* *subgroupI*)
show *conjugate-subgroup* *g* *H* $\subseteq$ *G*
by (*simp add: H.subset conjugate-subgroup-subset* *g*)
next
have *conjugation* *g* **1** = **1** **using** *g* **by** (*simp add: conjugation-def*)
then show **1** $\in$ *conjugate-subgroup* *g* *H*
using *H.subset conjugate-subgroup-mem-iff* *g* **by** *force*
next
fix *x* *y* **assume** *x* $\in$ *conjugate-subgroup* *g* *H* *y* $\in$ *conjugate-subgroup* *g* *H*
then obtain *hx* *hy* **where** *hx* $\in$ *H* *x* = *conjugation* *g* *hx*

```

    and  $hy \in H$   $y = \text{conjugation } g \text{ } hy$  and  $x \cdot y = \text{conjugation } g \text{ } (hx \cdot hy)$ 
    using conjugate-subgroup-def conjugation-hom by auto
  then show  $x \cdot y \in \text{conjugate-subgroup } g \text{ } H$ 
    using conjugate-subgroup-def by blast
next
fix  $x$  assume  $x \in \text{conjugate-subgroup } g \text{ } H$ 
then obtain  $h$  where  $h: h \in H$   $x = \text{conjugation } g \text{ } h$ 
  unfolding conjugate-subgroup-def by auto
have  $hG: h \in G$  using  $h(1)$   $H.\text{subset}$  by auto
show invertible  $x$  using  $h(2)$  conjugation-closed[OF  $g \text{ } hG$ ] by auto
show inverse  $x \in \text{conjugate-subgroup } g \text{ } H$ 
  using  $h$  conjugation-inverse[OF  $hG$ ]
  by (metis  $H.\text{sub.invertible}$   $H.\text{submonoid-inverse-closed}$  conjugate-subgroup-def
imageI)
qed
qed

```

```

lemma conjugate-subgroup-self:
  assumes  $\text{sub}: \text{Subgroup } H \text{ } G \text{ } (\cdot)$  1 and  $gH: g \in H$  and  $\text{fin}: \text{finite } G$ 
  shows  $\text{conjugate-subgroup } g \text{ } H = H$ 
proof -
  interpret  $H: \text{Subgroup } H \text{ } G \text{ } (\cdot)$  1 using sub .
  have  $\text{finH}: \text{finite } H$  using  $\text{fin}$   $H.\text{subset}$   $\text{finite-subset}$  by auto
  have  $\text{conjugate-subgroup } g \text{ } H \subseteq H$ 
    using  $gH$  by (auto simp: conjugate-subgroup-def conjugation-def)
  then show ?thesis
    by (simp add:  $H.\text{subset}$   $\text{card-subset-eq}$  conjugate-subgroup-card  $\text{finH}$   $gH$ )
qed

```

```

lemma conjugate-subgroup-unit:
  assumes  $H \subseteq G$ 
  shows  $\text{conjugate-subgroup } 1 \text{ } H = H$ 
  using assms by (force simp: conjugate-subgroup-def conjugation-unit)

```

```

lemma conjugate-subgroup-compose:
  assumes  $g \in G$   $h \in G$   $H \subseteq G$ 
  shows  $\text{conjugate-subgroup } (g \cdot h) \text{ } H = \text{conjugate-subgroup } g \text{ } (\text{conjugate-subgroup } h \text{ } H)$ 
  unfolding conjugate-subgroup-def
  using assms by (auto simp: conjugation-compose intro: imageI)

```

If H normalises K (i.e. every element of H conjugates K to itself), then the set product $HK = H \times K$ is a subgroup of G .

definition $\text{subgroup-product} :: 'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$
 where $\text{subgroup-product } H \text{ } K = (\lambda(h,k). h \cdot k) \text{ } (H \times K)$

```

lemma subgroup-product-mem-iff:
   $x \in \text{subgroup-product } H \text{ } K \iff (\exists h \in H. \exists k \in K. x = h \cdot k)$ 
  unfolding subgroup-product-def by auto

```

```

lemma subgroup-product-subset:
  assumes  $H \subseteq G$   $K \subseteq G$ 
  shows  $\text{subgroup-product } H K \subseteq G$ 
  unfolding subgroup-product-def using assms by auto

lemma
  assumes Subgroup  $K G (\cdot) 1$   $H \subseteq G$ 
  shows subgroup-product-superset-left:  $H \subseteq \text{subgroup-product } H K$ 
    and subgroup-product-superset-right:  $H \subseteq \text{subgroup-product } K H$ 
proof -
  interpret  $K$ : Subgroup  $K G (\cdot) 1$  using assms(1) .
  show  $H \subseteq \text{subgroup-product } H K$   $H \subseteq \text{subgroup-product } K H$ 
    using assms(2) subgroup-product-mem-iff by force+
qed

lemma normalizes-subgroup-product-is-subgroup:
  assumes Hsub: Subgroup  $H G (\cdot) 1$ 
    and Ksub: Subgroup  $K G (\cdot) 1$ 
    and norm:  $\forall h \in H. \text{conjugate-subgroup } h K = K$ 
  shows Subgroup  $(\text{subgroup-product } H K) G (\cdot) 1$ 
proof -
  interpret  $H$ : Subgroup  $H G (\cdot) 1$  using Hsub .
  interpret  $K$ : Subgroup  $K G (\cdot) 1$  using Ksub .
  have swap:  $\exists k' \in K. k \cdot h = h \cdot k'$  if  $k \in K$   $h \in H$  for  $k h$ 
  proof -
    have conjugation (inverse  $h$ )  $k \in K$ 
      by (metis  $H.\text{image-of-inverse-iff}$  conjugate-subgroup-def image-eqI norm that)
    moreover have conjugation (inverse  $h$ )  $k = \text{inverse } h \cdot k \cdot h$ 
      by (simp add: conjugation-def that)
    moreover have  $k \cdot h = h \cdot (\text{inverse } h \cdot k \cdot h)$ 
      by (simp add: that flip: associative)
    ultimately show ?thesis by auto
  qed
  show ?thesis
proof (rule subgroupI)
  show  $\text{subgroup-product } H K \subseteq G$ 
    by (simp add:  $H.\text{subset } K.\text{subset}$  subgroup-product-subset)
  show  $1 \in \text{subgroup-product } H K$ 
    using subgroup-product-mem-iff by force
next
  fix  $x y$ 
  assume  $xHK$ :  $x \in \text{subgroup-product } H K$  and  $yHK$ :  $y \in \text{subgroup-product } H K$ 
  obtain  $h1 k1$  where  $h1H$ :  $h1 \in H$  and  $k1K$ :  $k1 \in K$  and  $xeq$ :  $x = h1 \cdot k1$ 
    using  $xHK$  unfolding subgroup-product-mem-iff by auto
  obtain  $h2 k2$  where  $h2H$ :  $h2 \in H$  and  $k2K$ :  $k2 \in K$  and  $yeq$ :  $y = h2 \cdot k2$ 
    using  $yHK$  unfolding subgroup-product-mem-iff by auto
  obtain  $k'$  where  $k'K$ :  $k' \in K$  and  $k'eq$ :  $k1 \cdot h2 = h2 \cdot k'$ 
    using swap[OF  $k1K$   $h2H$ ] by auto
  have  $x \cdot y = h1 \cdot k1 \cdot (h2 \cdot k2)$  using  $xeq$   $yeq$  by simp

```

```

    also have ... = (h1 · h2) · (k' · k2)
    by (metis H.sub K.sub associative composition-closed h1H h2H k'K k'eq k1K
k2K)
    finally have x · y = (h1 · h2) · (k' · k2) .
    then show x · y ∈ subgroup-product H K
    using h1H h2H k'K k2K subgroup-product-mem-iff by auto
next
fix g assume g ∈ subgroup-product H K
then obtain h k where hH: h ∈ H and kK: k ∈ K and geq: g = h · k
    unfolding subgroup-product-mem-iff by auto
show invertible g
    by (simp add: geq hH kK)
have inverse g = inverse k · inverse h
    by (simp add: geq hH inverse-composition-commute kK)
moreover obtain ikK: inverse k ∈ K and ihH: inverse h ∈ H
    using hH kK by blast
ultimately show inverse g ∈ subgroup-product H K
    using local.swap subgroup-product-mem-iff by metis
qed
qed

```

The normalizer of a subgroup H in G : the set of elements that normalize H under conjugation.

definition *normalizer* :: 'a set $\Rightarrow$ 'a set
where *normalizer* $H \equiv \{g \in G. \text{conjugate-subgroup } g \ H = H\}$

lemma *normalizer-subset*: *normalizer* $H \subseteq G$
unfolding *normalizer-def* **by** auto

lemma *normalizer-mem-iff*:
 $g \in \text{normalizer } H \longleftrightarrow g \in G \wedge \text{conjugate-subgroup } g \ H = H$
unfolding *normalizer-def* **by** auto

lemma *normalizer-subgroup*:
assumes *sub*: Subgroup $H \ G \ (\cdot) \ 1$ **and** *fin*: finite G
shows Subgroup (*normalizer* H) $G \ (\cdot) \ 1$
proof (rule subgroupI)
interpret H : Subgroup $H \ G \ (\cdot) \ 1$ **using** *sub* .
show $1 \in \text{normalizer } H$
by (simp add: H.subset conjugate-subgroup-unit normalizer-mem-iff)
fix $g \ h$ **assume** gN : $g \in \text{normalizer } H$ **and** hN : $h \in \text{normalizer } H$
then show $g \cdot h \in \text{normalizer } H$
using H.subset conjugate-subgroup-compose normalizer-mem-iff **by** force
fix g **assume** gN : $g \in \text{normalizer } H$
then have gG : $g \in G$ **and** gH : conjugate-subgroup $g \ H = H$
unfolding normalizer-mem-iff **by** auto
have conjugate-subgroup (inverse g) $H = \text{conjugate-subgroup } (\text{inverse } g \cdot g) \ H$
using H.subset conjugate-subgroup-compose $gG \ gH$ invertible invertible-inverse-closed
by presburger


```

also have ... = H
  by (simp add: conjugate-subgroup-self fin gG sub)
finally show inverse g ∈ normalizer H
  by (simp add: normalizer-mem-iff gG)
qed (auto simp: normalizer-subset normalizer-mem-iff)

```

```

lemma subgroup-subset-normalizer:
  assumes sub: Subgroup H G (·) 1 and finite G
  shows H ⊆ normalizer H
proof
  fix h assume hH: h ∈ H
  interpret H: Subgroup H G (·) 1 using sub .
  show h ∈ normalizer H
    by (simp add: conjugate-subgroup-self hH normalizer-mem-iff assms)
qed

```

```

lemma normal-in-normalizer:
  assumes sub: Subgroup H G (·) 1 and fin: finite G
  shows normal-subgroup H (normalizer H) (·) 1
proof -
  interpret H: Subgroup H G (·) 1 using sub .
  interpret N: Subgroup normalizer H G (·) 1
    using normalizer-subgroup[OF sub fin] .
  have sub-H-N: Subgroup H (normalizer H) (·) 1
  proof
    qed (auto simp: subgroup-subset-normalizer[OF sub fin])
    show ?thesis
  proof (intro normal-subgroup.intro subgroup-of-group.intro sub-H-N N.sub.Group-axioms)
    show normal-subgroup-axioms H (normalizer H) (·) 1
    proof
      fix g k assume gN: g ∈ normalizer H and kH: k ∈ H
      have igH: conjugate-subgroup (inverse g) H = H
        using normalizer-subgroup[OF sub fin] gN normalizer-mem-iff by blast
      have conjugation (inverse g) k ∈ conjugate-subgroup (inverse g) H
        unfolding conjugate-subgroup-def using kH by auto
      then show N.sub.inverse g · k · g ∈ H
        using conjugation-def gN igH kH H.subset by fastforce
    qed
  qed
qed

```

The order formula for a product whose factors normalize one another in the relevant direction. The second isomorphism theorem supplies the quotient cardinalities, so this bridge remains entirely within the native set-based API.

```

lemma normalizes-subgroup-product-card:
  assumes Hsub: Subgroup H G (·) 1
  and Ksub: Subgroup K G (·) 1
  and norm: ∀ h ∈ H. conjugate-subgroup h K = K
  and fin: finite G

```

```

shows card (subgroup-product H K) * card (H ∩ K) = card H * card K
proof -
  interpret H: Subgroup H G (·) 1 using Hsub .
  interpret K: Subgroup K G (·) 1 using Ksub .
  have product-sub: Subgroup (subgroup-product H K) G (·) 1
    by (rule normalizes-subgroup-product-is-subgroup[OF Hsub Ksub norm])
  then interpret product: Subgroup subgroup-product H K G (·) 1 .
  interpret N: Subgroup normalizer K G (·) 1
    by (rule normalizer-subgroup[OF Ksub fin])
  have product-subset-N: subgroup-product H K ⊆ normalizer K
  proof
    fix x
    assume x: x ∈ subgroup-product H K
    obtain h k where hH: h ∈ H and kK: k ∈ K and xeq: x = h · k
      using x unfolding subgroup-product-mem-iff by auto
    obtain h ∈ normalizer K and k ∈ normalizer K
      using Ksub conjugate-subgroup-self fin hH kK norm normalizer-mem-iff by auto
    then show x ∈ normalizer K
      using xeq N.sub-composition-closed by simp
  qed
  have product-N-sub: Subgroup (subgroup-product H K) (normalizer K) (·) 1
    by (meson N.Subgroup-axioms product-sub product-subset-N subgroup-restrict)
  interpret KN: normal-subgroup K normalizer K (·) 1
    by (simp add: Ksub fin normal-in-normalizer)
  have K-product: K ⊆ subgroup-product H K
    by (rule subgroup-product-superset-right[OF Hsub K.subset])
  have K-product-sub: Subgroup K (subgroup-product H K) (·) 1
    by (meson K-product Ksub product-sub subgroup-restrict)
  have K-normal-product: normal-subgroup K (subgroup-product H K) (·) 1
  proof (intro normal-subgroup.intro subgroup-of-group.intro K-product-sub product.sub.Group-axioms)
    show normal-subgroup-axioms K (subgroup-product H K) (·) 1
  proof
    fix g k
    assume gP: g ∈ subgroup-product H K and kK: k ∈ K
    have inverse-eq: product.sub.inverse g = N.sub.inverse g
      using gP product-N-sub by (metis Subgroup.subgroup-inverse-equality)
    show product.sub.inverse g · k · g ∈ K
      using KN.normal gP inverse-eq kK product-subset-N by auto
  qed
  qed
  then interpret KP: normal-subgroup K subgroup-product H K (·) 1 .
  have H-product-sub: Subgroup H (subgroup-product H K) (·) 1
    by (meson H.subset Hsub Ksub product-sub subgroup-product-superset-left subgroup-restrict)
  have HIntK-product-sub:
    Subgroup (H ∩ K) (subgroup-product H K) (·) 1
    by (rule product.sub.subgroup-intersection[OF H-product-sub K-product-sub])
  have HIntK-H-sub: Subgroup (H ∩ K) H (·) 1
    using HIntK-product-sub H-product-sub subgroup-restrict by fastforce
  interpret HIntK: normal-subgroup H ∩ K H (·) 1

```

```

proof (intro normal-subgroup.intro subgroup-of-group.intro HIntK-H-sub H.sub.Group-axioms)
  show normal-subgroup-axioms ( $H \cap K$ )  $H$  ( $\cdot$ ) 1
    by (simp add: H-product-sub K-normal-product normal-subgroup.axioms(2)
      product.sub.normal-subgroup-intersection)
qed
have H-product-group: subgroup-of-group  $H$  (subgroup-product  $H$   $K$ ) ( $\cdot$ ) 1
  by (rule subgroup-of-groupI[OF H-product-sub product.sub.Group-axioms])
have product-locale:
  normal-subgroup-product  $K$   $H$  (subgroup-product  $H$   $K$ ) ( $\cdot$ ) 1
  by (rule normal-subgroup-product.intro[OF K-normal-product H-product-group])
interpret iso: second-iso-theorem  $K$   $H$  subgroup-product  $H$   $K$  ( $\cdot$ ) 1
  by (rule second-iso-theorem.intro[OF product-locale])
have iso-product: iso.HK = subgroup-product  $H$   $K$ 
  by (simp add: iso.HK-def subgroup-product-def)
have fract-left: iso.Fract-HK-K = KP.Partition
  unfolding iso.Fract-HK-K-def iso-product KP.Partition-def by rule
have fract-right: iso.Fract-H-HIntK = HIntK.Partition
  unfolding iso.Fract-H-HIntK-def HIntK.Partition-def by rule
have finH: finite  $H$  using finite-subset[OF H.subset fin] .
have fin-product: finite (subgroup-product  $H$   $K$ )
  using finite-subset[OF subgroup-product-subset[OF H.subset K.subset] fin] .
have card-left: card iso.Fract-HK-K = KP.index
  using KP.card-Factor-Group[OF fin-product] fract-left by simp
have card-right: card iso.Fract-H-HIntK = HIntK.index
  using HIntK.card-Factor-Group[OF finH] fract-right by simp
have iso-card: card iso.Fract-HK-K = card iso.Fract-H-HIntK
proof -
  obtain f composition unit composition' unit' where f:
    group-isomorphism f iso.Fract-HK-K composition unit
    iso.Fract-H-HIntK composition' unit'
  using iso.second-isomorphism unfolding isomorphic-as-groups-def by auto
interpret f: group-isomorphism f iso.Fract-HK-K composition unit
  iso.Fract-H-HIntK composition' unit' using f .
  show ?thesis using f.bijjective by (rule bij-betw-same-card)
qed
have card-product: card (subgroup-product  $H$   $K$ ) = card  $K$  * card iso.Fract-HK-K
  using KP.lagrange[OF fin-product] card-left iso-product by simp
have card-H: card  $H$  = card ( $H \cap K$ ) * card iso.Fract-H-HIntK
  using HIntK.lagrange[OF finH] card-right fract-right by simp
have card (subgroup-product  $H$   $K$ ) * card ( $H \cap K$ )
  = card  $K$  * (card ( $H \cap K$ ) * card iso.Fract-H-HIntK)
  by (simp add: card-product iso-card)
also have ... = card  $H$  * card  $K$ 
  by (simp add: card-H)
finally show ?thesis .
qed

```

Sylow p -subgroups: subgroups whose order is the largest power of p dividing $\text{card } G$.

definition *Sylow-subgroups* :: nat $\Rightarrow$ 'a set set
where *Sylow-subgroups* p =
 {H. Subgroup H G ($\cdot$) 1 $\wedge$ card H = p $\wedge$ multiplicity p (card G)}

Sylow p -subgroups exist by the first Sylow theorem.

lemma *Sylow-subgroups-nonempty*:
assumes prime p finite G
shows *Sylow-subgroups* p $\neq$ {}
using *Sylow-I*[OF assms] *multiplicity-dvd* **by** (auto simp: *Sylow-subgroups-def*)

Conjugation preserves Sylow p -subgroups.

lemma *conjugate-Sylow-is-Sylow*:
assumes P $\in$ *Sylow-subgroups* p g $\in$ G finite G
shows *conjugate-subgroup* g P $\in$ *Sylow-subgroups* p
proof –
interpret Subgroup P G ($\cdot$) 1
using *Sylow-subgroups-def* assms **by** blast
show ?thesis
using assms *conjugate-subgroup-card*
by (simp add: *Sylow-subgroups-def* *conjugate-subgroup-is-subgroup* *finite-subset subset*)
qed

The conjugation action on subsets of G : each $g \in G$ maps a subset H to *conjugation* g ' H . We restrict to *Sylow-subgroups* p to obtain a group action.

definition φ -Sylow :: nat $\Rightarrow$ 'a $\Rightarrow$ 'a set $\Rightarrow$ 'a set
where φ -Sylow p g = *restrict* (*conjugate-subgroup* g) (*Sylow-subgroups* p)

lemma *Sylow-subgroups-subgroup*:
P $\in$ *Sylow-subgroups* p $\implies$ Subgroup P G ($\cdot$) 1
unfolding *Sylow-subgroups-def* **by** auto

lemma *Sylow-subgroups-subset*:
P $\in$ *Sylow-subgroups* p $\implies$ P $\subseteq$ G
unfolding *Sylow-subgroups-def* **using** Submonoid.subset Subgroup-def **by** fastforce

lemma *Sylow-subgroups-card*:
P $\in$ *Sylow-subgroups* p $\implies$ card P = p $\wedge$ multiplicity p (card G)
unfolding *Sylow-subgroups-def* **by** auto

lemma φ -Sylow-mapsto:
assumes g $\in$ G prime p finite G
shows φ -Sylow p g $\in$ *Sylow-subgroups* p $\rightarrow_E$ *Sylow-subgroups* p
unfolding φ -Sylow-def PiE-def Pi-def
using *conjugate-Sylow-is-Sylow* assms **by** auto

lemma φ -Sylow-bij:
assumes g: g $\in$ G **and** p: prime p **and** fin: finite G
shows *bij-betw* (φ -Sylow p g) (*Sylow-subgroups* p) (*Sylow-subgroups* p)

```

proof (intro bij-betw-byWitness strip)
  show  $\varphi\text{-Sylow } p \text{ (inverse } g) (\varphi\text{-Sylow } p \ g \ P) = P$  if  $P: P \in \text{Sylow-subgroups } p$  for
   $P$ 
  proof –
    have  $P_{\text{sub}}: P \subseteq G$  using Sylow-subgroups-subset[OF P] .
    have  $gP$ : conjugate-subgroup  $g \ P \in \text{Sylow-subgroups } p$ 
      using conjugate-Sylow-is-Sylow[OF P g fin] .
    show  $\varphi\text{-Sylow } p \text{ (inverse } g) (\varphi\text{-Sylow } p \ g \ P) = P$ 
      unfolding  $\varphi\text{-Sylow-def}$  using  $P \ gP$ 
      by (metis conjugate-subgroup-unit  $P_{\text{sub}}$  conjugate-subgroup-compose  $g$ 
        invertible invertible-inverse-closed invertible-left-inverse restrict-apply)
  qed
  show  $\varphi\text{-Sylow } p \ g (\varphi\text{-Sylow } p \text{ (inverse } g) \ Q) = Q$  if  $Q: Q \in \text{Sylow-subgroups } p$  for
   $Q$ 
  proof –
    have  $igQ$ : conjugate-subgroup (inverse  $g$ )  $Q \in \text{Sylow-subgroups } p$ 
      using conjugate-Sylow-is-Sylow[OF Q - fin]  $g$  by auto
    show  $\varphi\text{-Sylow } p \ g (\varphi\text{-Sylow } p \text{ (inverse } g) \ Q) = Q$ 
      unfolding  $\varphi\text{-Sylow-def}$  using  $Q \ igQ$  Sylow-subgroups-subset[OF Q]
      by (metis conjugate-subgroup-unit conjugate-subgroup-compose  $g$ 
        invertible invertible-inverse-closed invertible-right-inverse restrict-apply)
  qed
qed (use  $\varphi\text{-Sylow-mapsto}$ [OF - p fin]  $g$  in  $\langle \text{auto simp: PiE-def Pi-def} \rangle$ )

lemma  $\varphi\text{-Sylow-unit}$ :
  assumes prime  $p$  finite  $G$ 
  shows  $\varphi\text{-Sylow } p \ \mathbf{1} = \text{identity } (\text{Sylow-subgroups } p)$ 
  using Sylow-subgroups-subset  $\varphi\text{-Sylow-def}$  conjugate-subgroup-unit by fastforce

lemma  $\varphi\text{-Sylow-compose}$ :
  assumes  $g: g \in G \ h \in G$  prime  $p$  finite  $G$ 
  shows  $\varphi\text{-Sylow } p \ (g \cdot h) = \text{compose } (\text{Sylow-subgroups } p) (\varphi\text{-Sylow } p \ g) (\varphi\text{-Sylow } p \ h)$ 
  unfolding  $\varphi\text{-Sylow-def}$  compose-def
  by (force simp: Sylow-subgroups-subset conjugate-Sylow-is-Sylow conjugate-subgroup-compose
    assms)

lemma  $\varphi\text{-Sylow-action}$ :
  assumes prime  $p$  finite  $G$ 
  shows Group-Action  $G \ (\cdot) \ \mathbf{1} \ (\varphi\text{-Sylow } p) (\text{Sylow-subgroups } p)$ 
proof
  fix  $g$  assume  $g \in G$ 
  then show  $\varphi\text{-Sylow } p \ g \in \text{Monoid.Units } (\text{Sylow-subgroups } p \rightarrow_E \text{ Sylow-subgroups } p)$ 
   $(\text{compose } (\text{Sylow-subgroups } p)) (\text{identity } (\text{Sylow-subgroups } p))$ 
    using  $\varphi\text{-Sylow-mapsto}$   $\varphi\text{-Sylow-bij}$  assms
    by (intro transformations.Units-bij-betwD) auto
qed (use  $\varphi\text{-Sylow-unit}$   $\varphi\text{-Sylow-compose}$  assms in auto)

lemma prime-power-dvd-not-prime-dvd:

```

```

fixes  $p\ d :: \text{nat}$  and  $k :: \text{nat}$ 
assumes  $\text{prime } p\ d\ \text{dvd } p \wedge k \neg p\ \text{dvd } d$ 
shows  $d = 1$ 
by (metis assms dvd-trans prime-dvd-power prime-factor-nat primes-dvd-imp-eq)

```

The second Sylow theorem: any two Sylow p -subgroups of G are conjugate. That is, the conjugation action of G on *Sylow-subgroups* p is transitive.

theorem *Sylow-II*:

```

assumes  $\text{prime } p\ \text{finite } G$ 
and  $P \in \text{Sylow-subgroups } p\ Q \in \text{Sylow-subgroups } p$ 
shows  $\exists g \in G. \text{conjugate-subgroup } g\ P = Q$ 

```

proof –

```

interpret act: Group-Action  $G\ (\cdot)\ 1\ \varphi\text{-Sylow } p\ \text{Sylow-subgroups } p$ 
using  $\varphi\text{-Sylow-action}[OF\ \text{assms}(1,2)]$  .

```

Let P act on *Sylow-subgroups* p by conjugation. Consider the orbits of this restricted action. The key combinatorial argument shows that Q must lie in the orbit of P .

```

have  $P\text{-self-orbit}: \forall g \in P. \text{conjugate-subgroup } g\ P = P$ 
using Sylow-subgroups-subgroup assms conjugate-subgroup-self by blast

```

Transitivity of the conjugation action via the left-coset counting argument. P acts on the left cosets of Q by left multiplication. Since the number of cosets is coprime to p , there must be a fixed coset, which yields the conjugating element.

```

interpret Psub: Subgroup  $P\ G\ (\cdot)\ 1$ 
using Sylow-subgroups-subgroup[OF assms(3)] .
interpret Qsub: Subgroup  $Q\ G\ (\cdot)\ 1$ 
using Sylow-subgroups-subgroup[OF assms(4)] .
interpret QG: subgroup-of-group  $Q\ G\ (\cdot)\ 1$ 
by (simp add: Qsub.Subgroup-axioms Group-axioms subgroup-of-group-def)
have finP:  $\text{finite } P$  using assms(2) Psub.subset finite-subset by auto
have finQ:  $\text{finite } Q$  using assms(2) Qsub.subset finite-subset by auto
define  $k$  where  $k \equiv \text{multiplicity } p\ (\text{card } G)$ 
have  $\text{card } P' : \text{card } P = p \wedge k$  and  $\text{card } Q' : \text{card } Q = p \wedge k$ 
using Sylow-subgroups-card assms k-def by blast
have  $p\text{-pos} : p > 0$ 
by (simp add: assms(1) prime-gt-0-nat)
have  $\text{card } G\text{-pos} : \text{card } G > 0$ 
using Psub.subset assms(2) card-0-eq by blast
  — The index  $[G:Q]$  is not divisible by  $p$ .
then have  $QG.\text{index} = \text{card } G \text{ div } (p \wedge k)$ 
using QG.lagrange assms(2) cardQ' by auto
then have  $\text{not-}p\text{-dvd-index} : \neg p\ \text{dvd } QG.\text{index}$ 
using multiplicity-decompose[of concl:  $p$ ]
using assms(1) cardG-pos k-def not-prime-unit by auto
  —  $P$  acts on the left cosets  $G/Q$  by left multiplication. A coset  $gQ$  is fixed by  $P$ 
  iff  $P \subseteq gQg^{-1}$ . Since  $[G:Q]$  is not divisible by  $p$  and every orbit of the  $P$ -action has
  size dividing  $p \wedge k$ , there is a fixed coset.
define  $L$  where  $L = (\lambda g. g \cdot | Q) \text{ ` } G$ 

```

```

have finL: finite L
  using L-def assms(2) by blast
  — The left cosets of  $Q$  have cardinality  $[G:Q]$ .
have L-sub-Omega:  $L \subseteq \Omega(p \wedge k)$ 
  by (auto simp: L-def Qsub.subset  $\Omega$ -memI  $\varphi$ -left-closed act.Left-Coset-def cardQ')
interpret P-act: Group-Action P ( $\cdot$ ) 1  $\varphi$ -left ( $p \wedge k$ )  $\Omega(p \wedge k)$ 
proof
  fix g assume gP:  $g \in P$ 
  then show  $\varphi$ -left ( $p \wedge k$ ) g
     $\in \text{Monoid.Units } (\Omega(p \wedge k) \rightarrow_E \Omega(p \wedge k)) \text{ (compose } (\Omega(p \wedge k)))$ 
  (identity ( $\Omega(p \wedge k)$ ))
  by (simp add:  $\varphi$ -left-bij  $\varphi$ -left-mapsto transformations.Units-bijection)
qed (auto simp add:  $\varphi$ -left-unit  $\varphi$ -left-compose)
  —  $Q$  is an element of  $\Omega p^k$ .
have Q-in-Omega:  $Q \in \Omega(p \wedge k)$ 
  using Qsub.subset cardQ' by auto
  — The stabilizer of  $Q$  under the  $P$ -action is  $P \cap Q$ .
have stab-Q:  $P\text{-act.stabilizer } Q = P \cap Q$ 
proof (intro antisym subsetI)
  fix a
  assume a  $\in P\text{-act.stabilizer } Q$ 
  then have aP:  $a \in P$  and eq:  $\varphi$ -left ( $p \wedge k$ ) a  $Q = Q$ 
    unfolding P-act.stabilizer-def by auto
  have ( $\cdot$ ) a ' $Q = Q$  using Q-in-Omega eq by simp
  with aP show a  $\in P \cap Q$ 
    by (metis IntI Psub.sub.right-unit Qsub.sub-unit-closed image-eqI)
next
  fix a
  assume a:  $a \in P \cap Q$ 
  then have ( $\cdot$ ) a ' $Q = Q$ 
    by (simp add: Qsub.sub.left-mult-image-card Qsub.sub.left-mult-image-subset
      card-subset-eq finQ)
  then show a  $\in P\text{-act.stabilizer } Q$ 
    unfolding P-act.stabilizer-def using a by (simp add: Q-in-Omega)
qed
  — By orbit-stabilizer, the orbit size divides  $p \wedge k$ .
have orbit-dvd:  $\text{card } (P\text{-act.orbit } Q) \text{ dvd } p \wedge k$ 
  by (metis P-act.orbit-stabilizer-card Q-in-Omega cardP' dvd-triv-left finP)
  — The orbit of  $Q$  under  $P$  is contained in  $L$ .
have orbit-sub-L:  $P\text{-act.orbit } Q \subseteq L$ 
  using P-act.orbit-mem-iff Q-in-Omega by (auto simp: L-def coset-notation.Left-Coset-def)
  —  $L$  has cardinality  $[G:Q]$ , not divisible by  $p$ .
have union-L:  $\bigcup L = G$ 
proof
  show  $\bigcup L \subseteq G$ 
    unfolding L-def coset-notation.Left-Coset-def using Qsub.subset by auto
  show  $G \subseteq \bigcup L$ 
    using L-def Qsub.sub-unit-closed act.Left-Coset-memI right-unit by (metis UN-iff
      subsetI)

```

```

qed
have disj-L:  $C1 \cap C2 = \{\}$ 
  if C1L:  $C1 \in L$  and C2L:  $C2 \in L$  and neq:  $C1 \neq C2$  for C1 C2
proof -
  obtain g1 g2 where g1G:  $g1 \in G$  and C1eq:  $C1 = g1 \cdot Q$ 
    and g2G:  $g2 \in G$  and C2eq:  $C2 = g2 \cdot Q$ 
    using C1L C2L unfolding L-def by auto
  show  $C1 \cap C2 = \{\}$ 
  proof (rule ccontr)
    assume  $C1 \cap C2 \neq \{\}$ 
    then obtain x where xC1:  $x \in C1$  and xC2:  $x \in C2$  by auto
    obtain q1 q2 where q1Q:  $q1 \in Q$  and xeq1:  $x = g1 \cdot q1$ 
      and q2Q:  $q2 \in Q$  and xeq2:  $x = g2 \cdot q2$ 
      and eq12:  $g1 \cdot q1 = g2 \cdot q2$ 
      using C1eq C2eq xC1 xC2 by fastforce
    have q1G:  $q1 \in G$  and q2G:  $q2 \in G$  using q1Q q2Q Qsub.subset by auto
      — Show  $C1 = C2$  by showing every element of one is in the other.
    have  $C1 = C2$ 
  proof (intro antisym subsetI)
    fix y
    assume  $y \in C1$ 
    then obtain r where rQ:  $r \in Q$  and yeq:  $y = g1 \cdot r$ 
      unfolding C1eq coset-notation.Left-Coset-def by auto
    have rG:  $r \in G$  using rQ Qsub.subset by auto
    have iq1Q: inverse  $q1 \in Q$ 
      using q1Q by (intro Qsub.submonoid-inverse-closed) auto
    have  $q2 \cdot \text{inverse } q1 \cdot r \in Q$ 
      using q2Q iq1Q rQ by (intro Qsub.sub-composition-closed) auto
    moreover have  $y = g2 \cdot (q2 \cdot \text{inverse } q1 \cdot r)$ 
  proof -
    have  $g2 \cdot (q2 \cdot \text{inverse } q1 \cdot r) = g2 \cdot q2 \cdot \text{inverse } q1 \cdot r$ 
      using g2G q2G q1G rG by (simp add: associative)
    also have  $\dots = g1 \cdot r$ 
      by (metis Qsub.sub associative eq12 g1G invertible invertible-right-inverse
iq1Q q1G right-unit)
    finally show ?thesis using yeq by simp
  qed
  ultimately show  $y \in C2$ 
    unfolding C2eq coset-notation.Left-Coset-def by auto
next
fix y
assume  $y \in C2$ 
then obtain r where rQ:  $r \in Q$  and yeq:  $y = g2 \cdot r$ 
  unfolding C2eq coset-notation.Left-Coset-def by auto
have rG:  $r \in G$  using rQ Qsub.subset by auto
have  $q1 \cdot \text{inverse } q2 \cdot r \in Q$ 
  using q1Q rQ q2Q by blast
moreover have  $y = g1 \cdot (q1 \cdot \text{inverse } q2 \cdot r)$ 
proof -
  have  $g1 \cdot (q1 \cdot \text{inverse } q2 \cdot r) = g1 \cdot q1 \cdot \text{inverse } q2 \cdot r$ 

```



```

    using g1G q1G q2G rG by (simp add: associative)
    also have ... = g2 · r
      using eq12 g2G q2G rG by (simp add: associative invertible-right-inverse)
    finally show ?thesis using yeq by simp
  qed
  ultimately show y ∈ C1
    unfolding C1eq coset-notation.Left-Coset-def by auto
  qed
  with neq show False by contradiction
  qed
  qed
  have card-coset:  $\bigwedge C. C \in L \implies \text{card } C = \text{card } Q$ 
    using L-sub-Omega cardQ' by auto
  have card G =  $(\sum_{C \in L. \text{card } C})$ 
    unfolding union-L [symmetric]
  proof (intro card-Union-disjoint)
    show disjoint L using disj-L unfolding disjoint-def by auto
    show  $\bigwedge A. A \in L \implies \text{finite } A$ 
      using union-L ⟨finite G⟩ by (meson Union-upper finite-subset)
  qed
  also have ... =  $(\sum_{C \in L. \text{card } Q})$ 
    by (simp add: card-coset)
  finally have card G = card Q * card L by simp
  with cardG-pos have card-L: card L = QG.index
    using QG.lagrange by fastforce
    — Every orbit of P on L has size a power of p, and [G:Q] is not divisible by p.
  So there is a fixed coset gQ.
  have  $\exists C \in L. \forall a \in P. (\cdot) a \cdot C = C$ 
  proof (rule ccontr)
    assume nofix:  $\neg (\exists C \in L. \forall a \in P. (\cdot) a \cdot C = C)$ 
    — The P-orbit of a coset in L stays inside L.
    have orbit-in-L:  $P\text{-act.orbit } C \subseteq L \text{ if } C \in L \text{ for } C$ 
  proof
    fix D assume D ∈ P-act.orbit C
    then obtain a where aP: a ∈ P and aG: a ∈ G and Deq:  $\varphi\text{-left } (p \wedge k) a C$ 
    = D
      by (auto simp: P-act.orbit-def)
    obtain g where gG: g ∈ G and Ceq:  $C = g \cdot | Q$  using ⟨C ∈ L⟩ unfolding
    L-def by auto
    have D =  $(\cdot) a \cdot C$  using Deq that L-sub-Omega by auto
    also have ... =  $(a \cdot g) \cdot | Q$ 
      unfolding coset-notation.Left-Coset-def using aG gG Qsub.subset by (auto
    simp: Ceq associative)
    finally show D ∈ L unfolding L-def using aG gG by auto
  qed
  — By orbit-stabilizer, each P-orbit size divides card P =  $p \wedge k$ .
  have orbit-dvd-gen: card (P-act.orbit C) dvd  $p \wedge k$  if C ∈ L for C
    unfolding dvd-def
    using L-sub-Omega P-act.orbit-stabilizer-card cardP' finP that by auto
    — A singleton orbit means the coset is fixed by all of P.

```

have *orbit-singleton-fixed*: $\forall a \in P. (\cdot) a \cdot C = C$
if $C \in L$ *P-act.orbit* $C = \{C\}$ **for** C
proof
 fix a **assume** aP : $a \in P$
 have $COmega$: $C \in \Omega (p \wedge k)$ **using** *that(1)* *L-sub-Omega* **by** *auto*
 then show $(\cdot) a \cdot C = C$
 by (*metis P-act.orbit-memI* φ -left-apply aP *emptyE* *insert-iff* *that(2)*)
qed
 — A nontrivial divisor of $p \wedge k$ is divisible by p .
have *div-ppow*: $p \text{ dvd } d$ **if** $d \text{ dvd } p \wedge k$ $d > 1$ **for** $d :: \text{nat}$
 using *assms(1)* *prime-power-dvd-not-prime-dvd* **that** **by** *blast*
 — Assuming no fixed coset, every P-orbit on L has size divisible by p.
have *orbit-p-dvd*: $p \text{ dvd } \text{card } (P\text{-act.orbit } C)$ **if** CL : $C \in L$ **for** C
proof —
 have $C \in P\text{-act.orbit } C$
 using *L-sub-Omega P-act.orbit-self* **that** **by** *blast*
 moreover have $P\text{-act.orbit } C \neq \{C\}$
 by (*metis nofix orbit-singleton-fixed* *that*)
 ultimately have $\text{card } (P\text{-act.orbit } C) \neq 1$ **using** *card-1-singletonE* **by** *force*
 then show *?thesis* **using** *div-ppow*
 using *assms(1)* *orbit-dvd-gen* *prime-power-dvd-not-prime-dvd* **that** **by** *blast*
qed
 — The P-orbits partition L, so p divides $\text{card } L = [G:Q]$.
define *Lorbs* **where** $Lorbs \equiv P\text{-act.orbit } L$
have *cover*: $\bigcup Lorbs = L$
 using *L-sub-Omega P-act.orbit-self* *orbit-in-L* **by** (*auto simp: Lorbs-def*)
have *disj*: *disjoint* *Lorbs*
 using *P-act.orbit-disjoint L-sub-Omega*
 by (*intro disjointI*) (*auto simp: Lorbs-def*)
have $\bigwedge A. A \in Lorbs \implies \text{finite } A$
 using *cover finL* **by** (*metis Union-upper rev-finite-subset*)
then have *cardL-sum*: $\text{card } L = (\sum X \in Lorbs. \text{card } X)$
 using *disj card-Union-disjoint* *cover* **by** *blast*
have $p \text{ dvd } (\sum X \in Lorbs. \text{card } X)$
 using *orbit-p-dvd* **by** (*intro dvd-sum*) (*auto simp: Lorbs-def*)
with *cardL-sum not-p-dvd-index* **show** *False*
 by (*simp add: card-L*)
qed
then obtain g **where** gG : $g \in G$ **and** *fixed*: $\forall a \in P. (\cdot) a \cdot (g \cdot Q) = g \cdot Q$
 using *L-def* **by** *blast*
 — A fixed coset gQ yields $g^{-1}Pg \subseteq Q$.
have *conj-sub*: *conjugate-subgroup* (*inverse* g) $P \subseteq Q$
proof
 fix x **assume** *xconj*: $x \in \text{conjugate-subgroup } (\text{inverse } g) P$
 then obtain a **where** aP : $a \in P$ **and** *xeg*: $x = \text{conjugation } (\text{inverse } g) a$
 unfolding *conjugate-subgroup-def* **by** *auto*
 — Since $a \in P$, left multiplication by a fixes $g \cdot Q$.
then have $a \cdot (g \cdot 1) \in g \cdot Q$
 by (*metis Qsub.sub-unit-closed coset-notation.Left-Coset-def* *fixed imageI*)
then show $x \in Q$

```

    using gG aP associative conjugation-def gG invertible-left-inverse2 xeq by auto
  qed
  — By equal cardinality, we get equality.
  have conjugate-subgroup (inverse g) P = Q
  by (simp add: Psub.subset cardP' cardQ' card-subset-eq conj-sub conjugate-subgroup-card
    finP finQ gG)
  then show ?thesis
  using gG by blast
  qed

```

A Sylow subgroup cannot normalize a distinct Sylow subgroup. The normalized product is a finite subgroup whose order is forced between the common Sylow order and the maximal p -power dividing $\text{card } G$.

```

lemma normalizes-Sylow-subgroups-eq:
  assumes prime: prime  $p$  and fin: finite  $G$ 
    and P:  $P \in \text{Sylow-subgroups } p$  and Q:  $Q \in \text{Sylow-subgroups } p$ 
    and norm:  $\forall g \in P. \text{conjugate-subgroup } g \ Q = Q$ 
  shows  $Q = P$ 
proof —
  interpret Psub: Subgroup  $P \ G \ (\cdot) \ 1$ 
    using Sylow-subgroups-subgroup[OF P] .
  interpret Qsub: Subgroup  $Q \ G \ (\cdot) \ 1$ 
    using Sylow-subgroups-subgroup[OF Q] .
  have finP: finite  $P$  using finite-subset[OF Psub.subset fin] .
  have finQ: finite  $Q$  using finite-subset[OF Qsub.subset fin] .
  define  $m$  where  $m \equiv \text{multiplicity } p \ (\text{card } G)$ 
  have cardP:  $\text{card } P = p^m$  and cardQ:  $\text{card } Q = p^m$ 
    using Sylow-subgroups-card  $P \ Q$  by (simp-all add: m-def)
  have product-sub: Subgroup (subgroup-product  $P \ Q$ )  $G \ (\cdot) \ 1$ 
    by (rule normalizes-subgroup-product-is-subgroup[OF Psub.Subgroup-axioms Qsub.Subgroup-axioms
      norm])
  interpret product: Subgroup subgroup-product  $P \ Q \ G \ (\cdot) \ 1$ 
    using product-sub .
  interpret productG: subgroup-of-group subgroup-product  $P \ Q \ G \ (\cdot) \ 1 \ ..$ 
  have fin-product: finite (subgroup-product  $P \ Q$ )
    using finite-subset[OF subgroup-product-subset[OF Psub.subset Qsub.subset] fin] .
  have card (subgroup-product  $P \ Q$ ) * card  $(P \cap Q) = \text{card } P * \text{card } Q$ 
    by (rule normalizes-subgroup-product-card[OF Psub.Subgroup-axioms Qsub.Subgroup-axioms
      norm fin])
  then have  $\text{card } (\text{subgroup-product } P \ Q) * \text{card } (P \cap Q) = p^{m+m}$ 
    by (simp add: cardP cardQ monoid-mult-class.power-add)
  then have product-dvd-power:  $\text{card } (\text{subgroup-product } P \ Q) \text{ dvd } p^{m+m}$ 
    by (metis dvd-def)
  obtain  $i$  where  $i \leq m + m$  and card-product:  $\text{card } (\text{subgroup-product } P \ Q) =$ 
 $p^i$ 
    using divides-primelow-nat[OF prime] product-dvd-power by blast
  have p-i-dvd-G:  $p^i \text{ dvd } \text{card } G$ 
    using card-product by (simp add: productG.lagrange[OF fin])
  have  $G \neq \{\}$ 

```

```

    using Psub.subset Psub.sub-unit-closed by blast
  then have card G > 0
    using fin by (simp add: card-gt-0-iff)
  then have multiplicity p (p ^ i) ≤ multiplicity p (card G)
    using dvd-imp-multiplicity-le p-i-dvd-G by blast
  then have i-le-m: i ≤ m using prime by (simp add: m-def)
  have P-product: P ⊆ subgroup-product P Q
    by (rule subgroup-product-superset-left[OF Qsub.Subgroup-axioms Psub.subset])
  have p ^ m ≤ p ^ i
    using card-mono[OF fin-product P-product] cardP card-product by simp
  then have m ≤ i
    using prime prime-gt-1-nat by force
  with i-le-m have exp-eq: i = m by auto
  have P-eq-product: P = subgroup-product P Q
    by (simp add: P-product cardP card-product card-subset-eq exp-eq fin-product)
  have Q-product: Q ⊆ subgroup-product P Q
    by (rule subgroup-product-superset-right[OF Psub.Subgroup-axioms Qsub.subset])
  show ?thesis
    using P-eq-product Q-product cardP cardQ finP by (metis card-subset-eq)
qed

```

Conjugation is transitive on the Sylow p -subgroups.

lemma *Sylow-subgroups-orbit*:

```

  assumes prime: prime p and fin: finite G
    and P: P ∈ Sylow-subgroups p
  shows (λg. φ-Sylow p g P) ‘ G = Sylow-subgroups p
proof -
  interpret act: Group-Action G (·) 1 φ-Sylow p Sylow-subgroups p
  using φ-Sylow-action[OF prime fin] .
  show (λg. φ-Sylow p g P) ‘ G = Sylow-subgroups p
proof (rule equalityI)
  show Sylow-subgroups p ⊆ (λg. φ-Sylow p g P) ‘ G
proof
  fix Q assume Q: Q ∈ Sylow-subgroups p
  obtain g where gG: g ∈ G and gPQ: conjugate-subgroup g P = Q
    using Sylow-II[OF prime fin P Q] by blast
  have φ-Sylow p g P = Q
    unfolding φ-Sylow-def using P gPQ by simp
  then show Q ∈ (λg. φ-Sylow p g P) ‘ G
    using gG by blast
qed
qed (use P in blast)
qed

```

The number of Sylow p -subgroups divides the group order and is congruent to one modulo p .

theorem

```

  assumes prime: prime p and fin: finite G
  shows Sylow-subgroups-card-dvd: card (Sylow-subgroups p) dvd card G

```

```

    and Sylow-III:  $p \text{ dvd } \text{card } (\text{Sylow-subgroups } p) - 1$ 
  proof -
    have fin-S: finite (Sylow-subgroups  $p$ )
    using Sylow-subgroups-orbit fin prime by (metis equals0I finite.emptyI finite-imageI)
    obtain P where P:  $P \in \text{Sylow-subgroups } p$ 
    using Sylow-subgroups-nonempty fin prime by blast
    interpret Psub: Subgroup  $P \ G \ (\cdot) \ 1$ 
    using Sylow-subgroups-subgroup[OF P] .
    interpret act: Group-Action  $G \ (\cdot) \ 1 \ \varphi\text{-Sylow } p \ \text{Sylow-subgroups } p$ 
    using  $\varphi\text{-Sylow-action}$ [OF prime fin] .
    have orbit-S:  $\text{act.orbit } P = \text{Sylow-subgroups } p$ 
    using Sylow-subgroups-orbit[OF prime fin P]
    by (simp add: act.orbit-eq)
    have stab-normalizer:  $\text{act.stabilizer } P = \text{normalizer } P$ 
    unfolding act.stabilizer-def normalizer-def  $\varphi\text{-Sylow-def}$ 
    using P by simp
    have orbit-stab:
       $\text{card } G = \text{card } (\text{Sylow-subgroups } p) * \text{card } (\text{normalizer } P)$ 
    using act.orbit-stabilizer-card[OF P fin] orbit-S stab-normalizer by simp
    show  $\text{card } (\text{Sylow-subgroups } p) \text{ dvd } \text{card } G$ 
    using dvd-def orbit-stab by blast
    have P-act-map:
       $\varphi\text{-Sylow } p \ g \in \text{Monoid.Units}$ 
       $(\text{Sylow-subgroups } p \rightarrow_E \text{Sylow-subgroups } p)$ 
       $(\text{compose } (\text{Sylow-subgroups } p)) (\text{identity } (\text{Sylow-subgroups } p))$ 
    if  $g \in P$  for  $g$ 
    using that by blast
    interpret P-act: Group-Action  $P \ (\cdot) \ 1 \ \varphi\text{-Sylow } p \ \text{Sylow-subgroups } p$ 
    proof qed (use act.hom-unit act.hom-compose P-act-map in auto)
    have fixed-P:  $P \in P\text{-act.fixed-points}$ 
    by (simp add: Group.conjugate-subgroup-self Group-axioms P P-act.fixed-point-char
      Psub.Subgroup-axioms  $\varphi\text{-Sylow-def fin}$ )
    have  $Q = P$  if Qfixed:  $Q \in P\text{-act.fixed-points}$  for  $Q$ 
    proof -
      have Qmem:  $Q \in \text{Sylow-subgroups } p$ 
      using subsetD[OF P-act.fixed-points-subset Qfixed] .
      have normQ:  $\forall g \in P. \text{conjugate-subgroup } g \ Q = Q$ 
      using P-act.fixed-point-char Qfixed Qmem  $\varphi\text{-Sylow-def}$  by auto
      show ?thesis
      by (rule normalizes-Sylow-subgroups-eq[OF prime fin P Qmem normQ])
    qed
    then have  $P\text{-act.fixed-points} = \{P\}$ 
    using fixed-P by auto
    moreover have  $p \text{ dvd } \text{card } (\text{Sylow-subgroups } p - P\text{-act.fixed-points})$ 
    proof (rule P-act.dvd-card-nonfixed-points[OF fin-S])
      fix Q
      assume Q:  $Q \in \text{Sylow-subgroups } p$  and Qnot:  $Q \notin P\text{-act.fixed-points}$ 
      have orbit-ne:  $P\text{-act.orbit } Q \neq \{Q\}$ 
      using P-act.fixed-point-iff-orbit-singleton Q Qnot by presburger
      have orbit-card-ne:  $\text{card } (P\text{-act.orbit } Q) \neq 1$ 

```

```

    using P-act.orbit-self[OF Q] orbit-ne by (metis card-1-singletonE emptyE insert-iff)
  have orbit-dvd: card (P-act.orbit Q) dvd p ^ multiplicity p (card G)
    using P-act.orbit-stabilizer-card Q Sylow-subgroups-card[OF P] by force
  show p dvd card (P-act.orbit Q)
    using orbit-card-ne orbit-dvd prime prime-power-dvd-not-prime-dvd by blast
qed
ultimately show p dvd card (Sylow-subgroups p) - 1
  using P by auto
qed

end

end

```

11 Commutators and the Derived Subgroup

```

theory Commutator
  imports Sylow-Theorems
begin

```

The commutator of two elements, the commutator subgroup of two subsets, and the derived (commutator) subgroup of a group. These underlie the derived series and the notion of a solvable group. (Custom notation is deferred; we use plain constant names while developing the theory.)

```

context Group
begin

```

The commutator of two elements.

```

definition commutator-elt :: 'a ⇒ 'a ⇒ 'a
  where commutator-elt a b = a · b · inverse a · inverse b

```

Membership rule for commutators. Declared *simp* only; as an *intro* rule it back-chains on *every* carrier-membership goal ($?x \in G$ via $?x = \text{commutator-elt } ?a ?b$), which interacts badly with large introduction-rule sets and can loop.

```

lemma commutator-elt-closed [simp]:
  [| a ∈ G; b ∈ G |] ⇒ commutator-elt a b ∈ G
  unfolding commutator-elt-def by simp

```

The commutator subgroup of two subsets of the carrier.

```

definition commutator-subgroup :: 'a set ⇒ 'a set ⇒ 'a set
  where commutator-subgroup H K = { commutator-elt h k | h k. h ∈ H ∧ k ∈ K }

```

The derived subgroup of the whole group.

```

definition derived :: 'a set
  where derived = commutator-subgroup G G

```

lemma *commutator-subgroup-is-subgroup*:
Subgroup (commutator-subgroup H K) G (·) 1
unfolding *commutator-subgroup-def* **by** (rule *Gen-is-subgroup*)

lemma *commutator-subgroup-subset*:
commutator-subgroup H K $\subseteq$ G
unfolding *commutator-subgroup-def* **by** (rule *Gen-subset-carrier*)

lemma *derived-is-subgroup*: *Subgroup derived G (·) 1*
unfolding *derived-def* **by** (rule *commutator-subgroup-is-subgroup*)

lemma *derived-subset*: *derived $\subseteq$ G*
unfolding *derived-def* **by** (rule *commutator-subgroup-subset*)

Generators of the commutator subgroup are commutators.

lemma *commutator-gen-mem*:
 $\llbracket h \in H; k \in K; H \subseteq G; K \subseteq G \rrbracket \implies \text{commutator-elt } h \ k \in \text{commutator-subgroup } H \ K$
unfolding *commutator-subgroup-def* **using** *Gen-incl* **by** *force*

Conjugation distributes over a commutator.

lemma *conjugation-commutator*:
assumes $g \in G \ a \in G \ b \in G$
shows $\text{conjugation } g \ (\text{commutator-elt } a \ b) = \text{commutator-elt } (\text{conjugation } g \ a) \ (\text{conjugation } g \ b)$
using *assms* **by** (simp add: *conjugation-hom commutator-elt-def flip: conjugation-inverse*)

The derived subgroup is normal.

lemma *derived-normal*: *normal-subgroup derived G (·) 1*
proof –
interpret *D*: *Subgroup derived G (·) 1* **by** (rule *derived-is-subgroup*)
interpret *DG*: *subgroup-of-group derived G (·) 1*
by (simp add: *D.Subgroup-axioms Group-axioms subgroup-of-group-def*)
show *?thesis*
proof
fix $g \ k$ **assume** $g: g \in G$ **and** $k: k \in \text{derived}$
have $ig: \text{inverse } g \in G$ **using** g **by** *simp*
have $\text{conj-in}: \text{conjugation } (\text{inverse } g) \ x \in \text{derived}$
if $x \in \text{generate } (\{ \text{commutator-elt } a \ b \mid a \ b. \ a \in G \wedge b \in G \} \cap G)$ **for** x
using *that*
proof *induction*
case *unit* **then show** *?case*
using *conjugation-def* **by** *auto*
next
case (*incl x*)
then obtain $a \ b$ **where** $xab: x = \text{commutator-elt } a \ b$ **and** $ab: a \in G \ b \in G$ **by** *auto*
have $\text{conjugation } (\text{inverse } g) \ x =$
 $\text{commutator-elt } (\text{conjugation } (\text{inverse } g) \ a) \ (\text{conjugation } (\text{inverse } g) \ b)$

```

    using ig ab xab by (simp add: conjugation-commutator)
  moreover have conjugation (inverse g) a ∈ G conjugation (inverse g) b ∈ G
    using ig ab by (auto simp: conjugation-closed)
  ultimately show ?case
    by (metis commutator-gen-mem derived-def set-eq-subset)
next
case (comp x y)
have xy: x ∈ G y ∈ G using comp.hyps generate-subset-carrier by auto
with conjugation-hom[OF xy] comp.IH D.sub-composition-closed
show ?case by metis
next
case (inv x)
have xG: x ∈ G using inv.hyps generate-subset-carrier by auto
have conjugation (inverse g) (inverse x) = inverse (conjugation (inverse g) x)
  using conjugation-inverse[OF xG] .
then show ?case using inv.IH D.submonoid-inverse-closed D.sub.invertible by
simp
qed
have inverse g · k · g = conjugation (inverse g) k
  using g k derived-subset
  by (auto simp: conjugation-def associative invertible-inverse-inverse subsetD)
moreover have k ∈ generate ({ commutator-elt a b | a b. a ∈ G ∧ b ∈ G } ∩ G)
  using k unfolding derived-def commutator-subgroup-def Gen-def .
ultimately show inverse g · k · g ∈ derived
  using conj-in by simp
qed
qed

```

The commutator subgroup is monotone in both arguments.

lemma *commutator-subgroup-mono*:

```

assumes A ⊆ A' B ⊆ B'
shows commutator-subgroup A B ⊆ commutator-subgroup A' B'
unfolding commutator-subgroup-def Gen-def
using assms by (intro generate-mono) blast

```

A commutator computed inside a subgroup (with the restricted operation) equals the ambient commutator. A locality property of *commutator-elt*.

lemma *commutator-elt-subgroup-eq*:

```

assumes H: Subgroup H G (·) 1 and ab: a ∈ H b ∈ H
shows Group.commutator-elt H (·) 1 a b = commutator-elt a b

```

proof –

```

interpret S: Subgroup H G (·) 1 by (rule H)
show ?thesis
  by (simp add: assms S.sub.commutator-elt-def commutator-elt-def)
also have ... = commutator-elt a b by (simp add: commutator-elt-def)

```

qed

The commutator subgroup computed inside a subgroup is contained in the ambient one.

lemma *commutator-subgroup-subgroup-subset*:
assumes H : Subgroup H G $(\cdot)$ **1** **and** AB : $A \subseteq H$ $B \subseteq H$
shows $\text{Group.commutator-subgroup } H$ $(\cdot)$ **1** A $B \subseteq \text{commutator-subgroup } A$ B
proof –
interpret S : Subgroup H G $(\cdot)$ **1** **by** (rule H)
have AG : $A \subseteq G$ $B \subseteq G$ **using** AB S .subset **by** auto
have genset: $(\{ \text{Group.commutator-elt } H$ $(\cdot)$ **1** h $k \mid h$ $k. h \in A \wedge k \in B \} \cap H)$
 $\subseteq \{ \text{commutator-elt } h$ $k \mid h$ $k. h \in A \wedge k \in B \}$
using *assms commutator-elt-subgroup-eq* **by** blast
have absorb: $\{ \text{commutator-elt } h$ $k \mid h$ $k. h \in A \wedge k \in B \} \cap G$
 $= \{ \text{commutator-elt } h$ $k \mid h$ $k. h \in A \wedge k \in B \}$
using AG **by** (force simp: commutator-elt-closed)
have $\text{Group.commutator-subgroup } H$ $(\cdot)$ **1** A B
 $= \text{Group.generate } H$ $(\cdot)$ **1** $(\{ \text{Group.commutator-elt } H$ $(\cdot)$ **1** h $k \mid h$ $k. h \in A$
 $\wedge k \in B \} \cap H)$
by (simp add: S .sub.commutator-subgroup-def S .sub.Gen-def)
also have $\dots \subseteq \text{generate } (\{ \text{Group.commutator-elt } H$ $(\cdot)$ **1** h $k \mid h$ $k. h \in A \wedge k \in B$
 $B \} \cap H)$
by (rule generate-subgroup-subset[OF H])
also have $\dots \subseteq \text{generate } \{ \text{commutator-elt } h$ $k \mid h$ $k. h \in A \wedge k \in B \}$
using genset **by** (rule generate-mono)
also have $\dots = \text{commutator-subgroup } A$ B
by (simp add: commutator-subgroup-def Gen-def absorb)
finally show ?thesis .
qed

Generation is *absolute* for a subgroup: for $S \subseteq H$, the subgroup generated by S inside G coincides with the one generated inside H (the reverse inclusion to *generate-subgroup-subset*). The H -generated set is a subgroup of G containing S , so the minimal such (*generate*) is contained in it.

lemma *generate-subgroup-superset*:
assumes H : Subgroup H G $(\cdot)$ **1** **and** S : $S \subseteq H$
shows $\text{generate } S \subseteq \text{Group.generate } H$ $(\cdot)$ **1** S
proof (rule generate-minimal)
interpret S' : Subgroup H G $(\cdot)$ **1** **by** (rule H)
have Subgroup $(\text{Group.generate } H$ $(\cdot)$ **1** $S)$ H $(\cdot)$ **1** **by** (rule S' .sub.generate-is-subgroup)
then show Subgroup $(\text{Group.generate } H$ $(\cdot)$ **1** $S)$ G $(\cdot)$ **1**
using H **by** (rule subgroup-transitive)
have $S \subseteq \text{Group.generate } H$ $(\cdot)$ **1** S **using** S S' .sub.generate-incl **by** auto
then show $S \cap G \subseteq \text{Group.generate } H$ $(\cdot)$ **1** S **by** blast
qed

The commutator subgroup of subsets of H computed in G is contained in the one computed in H (the reverse of *commutator-subgroup-subgroup-subset*; the two are in fact equal).

lemma *commutator-subgroup-subset-subgroup*:
assumes H : Subgroup H G $(\cdot)$ **1** **and** AB : $A \subseteq H$ $B \subseteq H$
shows $\text{commutator-subgroup } A$ $B \subseteq \text{Group.commutator-subgroup } H$ $(\cdot)$ **1** A B
proof –

```

interpret  $S$ : Subgroup  $H$   $G$   $(\cdot)$  1 by (rule  $H$ )
have  $AG$ :  $A \subseteq G$   $B \subseteq G$  using  $AB$   $S.subset$  by auto
  — The commutator generators computed in  $H$  coincide with the ambient ones and
  lie in  $H$ .
have absorb:  $\{ commutator\text{-}elt\ h\ k \mid h\ k. h \in A \wedge k \in B \} \cap G$ 
  =  $\{ commutator\text{-}elt\ h\ k \mid h\ k. h \in A \wedge k \in B \}$ 
  using  $AG$  by (force simp: commutator-elt-closed)
have commutator-subgroup  $A\ B$  = generate  $(\{ commutator\text{-}elt\ h\ k \mid h\ k. h \in A \wedge k \in B \})$ 
  by (simp add: commutator-subgroup-def Gen-def absorb)
also have  $\dots \subseteq generate\ (\{ Group.commutator\text{-}elt\ H\ (\cdot)\ 1\ h\ k \mid h\ k. h \in A \wedge k \in B \} \cap H)$ 
  using assms by (intro generate-mono) (force simp: commutator-elt-def  $S.sub.commutator\text{-}elt\text{-}def$ )
also have  $\dots \subseteq Group.generate\ H\ (\cdot)\ 1\ (\{ Group.commutator\text{-}elt\ H\ (\cdot)\ 1\ h\ k \mid h\ k. h \in A \wedge k \in B \} \cap H)$ 
  by (rule generate-subgroup-superset[OF H]) auto
also have  $\dots = Group.commutator\text{-}subgroup\ H\ (\cdot)\ 1\ A\ B$ 
  by (simp add:  $S.sub.commutator\text{-}subgroup\text{-}def\ S.sub.Gen\text{-}def$ )
finally show ?thesis .
qed

```

11.1 The Derived Series and Solvable Groups

The derived series of a group is obtained by iterating the derived-subgroup construction. A group is solvable when its derived series reaches the trivial subgroup. As a first milestone we show that every abelian group is solvable.

The derived series, as a sequence of subsets of the carrier.

```

primrec derivedSeries ::  $nat \Rightarrow 'a\ set$ 
  where
    derivedSeries  $0 = G$ 
     $| derivedSeries\ (Suc\ n) = commutator\text{-}subgroup\ (derivedSeries\ n)\ (derivedSeries\ n)$ 

```

A group is solvable if its derived series reaches the trivial subgroup.

```

definition solvable ::  $bool$ 
  where solvable  $\equiv (\exists n. derivedSeries\ n = \{1\})$ 

```

Each term of the derived series is a subgroup.

```

lemma derivedSeries-subgroup: Subgroup  $(derivedSeries\ n)\ G\ (\cdot)\ 1$ 
  by (cases  $n$ ) (auto simp: commutator-subgroup-is-subgroup group-self-subgroup)

```

```

lemma derivedSeries-subset:  $derivedSeries\ n \subseteq G$  and derivedSeries-unit-closed:  $1 \in derivedSeries\ n$ 

```

```

proof —
  interpret  $S$ : Subgroup  $derivedSeries\ n\ G\ (\cdot)\ 1$  by (rule derivedSeries-subgroup)
  show  $derivedSeries\ n \subseteq G$  by (rule  $S.subset$ )
  show  $1 \in derivedSeries\ n$  by (rule  $S.sub\text{-}unit\text{-}closed$ )
qed

```

The commutator subgroup of a subgroup H (with itself) is contained in H : the commutator generators lie in H (closed under the operation and inverse), and H , being a subgroup, contains the subgroup they generate.

```

lemma commutator-subgroup-self-subset:
  assumes  $H$ : Subgroup  $H$   $G$   $(\cdot)$  1
  shows commutator-subgroup  $H$   $H \subseteq H$ 
proof –
  interpret  $S$ : Subgroup  $H$   $G$   $(\cdot)$  1 by (rule  $H$ )
  show ?thesis
  using  $S$ .sub.commutator-subgroup-subset assms commutator-subgroup-subset-subgroup
  by blast
qed

```

The derived series is decreasing: $\text{derivedSeries } (\text{Suc } n) \subseteq \text{derivedSeries } n$.

```

lemma derivedSeries-Suc-subset:  $\text{derivedSeries } (\text{Suc } n) \subseteq \text{derivedSeries } n$ 
  using commutator-subgroup-self-subset[OF derivedSeries-subgroup] by simp

```

```

lemma derivedSeries-antimono:  $m \leq n \implies \text{derivedSeries } n \subseteq \text{derivedSeries } m$ 
  by (induction  $n$ ) (use derivedSeries-Suc-subset not-less-eq-eq in fastforce)+

```

In an abelian group every commutator is the unit.

```

lemma commutator-elt-abelian:
  assumes commutative-monoid  $G$   $(\cdot)$  1 and  $a \in G$   $b \in G$ 
  shows commutator-elt  $a$   $b = \mathbf{1}$ 
proof –
  interpret  $\text{comm}$ : commutative-monoid  $G$   $(\cdot)$  1 by (rule assms(1))
  show ?thesis
  using assms(2,3)  $\text{comm.commutative}$  commutator-elt-def commute-iff-inverse by
  auto
qed

```

In an abelian group the derived subgroup is trivial.

```

lemma derived-trivial-if-abelian:
  assumes commutative-monoid  $G$   $(\cdot)$  1
  shows  $\text{derived} = \{\mathbf{1}\}$ 
proof –
  interpret  $D$ : Subgroup  $\text{derived } G$   $(\cdot)$  1 by (rule derived-is-subgroup)
  have  $\text{derived} \subseteq \{\mathbf{1}\}$ 
  unfolding derived-def commutator-subgroup-def
  using commutator-elt-abelian[OF assms]
  by (intro Gen-minimal[OF trivial-subgroup]) auto
  then show ?thesis using  $D$ .sub-unit-closed by auto
qed

```

Milestone: every abelian group is solvable.

```

theorem abelian-imp-solvable:
  assumes commutative-monoid  $G$   $(\cdot)$  1
  shows solvable

```

```

unfolding solvable-def
by (metis assms derivedSeries.simps derived-def derived-trivial-if-abelian)

end

end

```

12 Transfer of Solvability

```

theory Solvable-Transfer
imports Commutator
begin

```

Solvability transfers along the basic group constructions: it passes to homomorphic (in particular quotient) images, and to subgroups. These are the load-bearing facts connecting a solvable group to its subquotients.

```

context group-homomorphism
begin

```

A homomorphism carries commutators to commutators.

```

lemma image-commutator-elt:
  assumes  $a \in G$   $b \in G$ 
  shows  $\eta$  (source.commutator-elt  $a$   $b$ ) = target.commutator-elt ( $\eta$   $a$ ) ( $\eta$   $b$ )
proof –
  have  $\eta$  (source.commutator-elt  $a$   $b$ ) =  $\eta$   $a$   $\cdot'$   $\eta$   $b$   $\cdot'$  inverse' ( $\eta$   $a$ )  $\cdot'$  inverse' ( $\eta$   $b$ )
    using assms by (simp add: source.commutator-elt-def commutes-with-composition

      invertible-commutes-with-inverse)
  also have  $\dots$  = target.commutator-elt ( $\eta$   $a$ ) ( $\eta$   $b$ )
    using assms by (simp add: target.commutator-elt-def)
  finally show ?thesis .
qed

```

The image of a generated subgroup is contained in the subgroup generated by the image.

```

lemma image-generate-sub:  $\eta$  ‘ source.generate  $S \subseteq$  target.generate ( $\eta$  ‘ ( $S \cap G$ ))
proof
  fix  $y$  assume  $y \in \eta$  ‘ source.generate  $S$ 
  then obtain  $x$  where  $x: x \in$  source.generate  $S$  and  $yx: y = \eta$   $x$  by auto
  from  $x$  have  $\eta$   $x \in$  target.generate ( $\eta$  ‘ ( $S \cap G$ ))
  proof induction
    case unit
    then show ?case using commutes-with-unit target.generate.unit by simp
  next
    case (incl  $x$ )
    then have  $\eta$   $x \in \eta$  ‘ ( $S \cap G$ )  $\cap G'$  by auto
    then show ?case by (rule target.generate.incl)
  next

```

```

    case (comp x y)
    have xG:  $x \in G$   $y \in G$  using comp.hyps source.generate-subset-carrier by auto
    have  $\eta (x \cdot y) = \eta x \cdot' \eta y$  using xG by (simp add: commutes-with-composition)
    then show ?case using comp.IH by (simp add: target.generate.comp)
  next
    case (inv x)
    have xG:  $x \in G$  using inv.hyps source.generate-subset-carrier by auto
    have  $\eta (\text{inverse } x) = \text{inverse}' (\eta x)$  using xG by (simp add: invertible-commutes-with-inverse)
    then show ?case using inv.IH by (simp add: target.generate.inv)
  qed
  then show  $y \in \text{target.generate } (\eta^{-1} (S \cap G))$  using yx by simp
qed

```

The reverse: the subgroup generated by the image is contained in the image of the generated subgroup. (No surjectivity needed.)

```

lemma image-generate-super:  $\text{target.generate } (\eta^{-1} (S \cap G)) \subseteq \eta^{-1} \text{source.generate } S$ 
proof
  fix y assume  $y \in \text{target.generate } (\eta^{-1} (S \cap G))$ 
  then show  $y \in \eta^{-1} \text{source.generate } S$ 
  proof induction
    case unit
    then show ?case using commutes-with-unit source.generate.unit by force
  next
    case (incl x)
    then show ?case using source.generate.incl by force
  next
    case (comp x y)
    then obtain a b where  $\S: a \in \text{source.generate } S$   $x = \eta a$   $b \in \text{source.generate } S$   $y = \eta b$   $a \in G$   $b \in G$ 
    using source.generate-subset-carrier by auto
    then have  $a \cdot b \in \text{source.generate } S$  using source.generate.comp by blast
    with  $\S$  show ?case by (metis commutes-with-composition image-eqI)
  next
    case (inv x)
    then obtain a where  $a: a \in \text{source.generate } S$   $x = \eta a$   $a \in G$ 
    using source.generate-subset-carrier by auto
    then show ?case
    by (metis source.generate.inv image-eqI invertible-commutes-with-inverse source.invertible)
  qed
qed

```

Hence the image of a commutator subgroup is the commutator subgroup of the images.

```

lemma image-commutator-subgroup-sub:
  assumes  $H \subseteq G$   $K \subseteq G$ 
  shows  $\eta^{-1} \text{source.commutator-subgroup } H K \subseteq \text{target.commutator-subgroup } (\eta^{-1} H)$ 
  ( $\eta^{-1} K$ )
proof -
  have genset-sub:  $\{ \text{target.commutator-elt } h' k' \mid h' k'. h' \in \eta^{-1} H \wedge k' \in \eta^{-1} K \} \subseteq$ 

```

G'
using *assms* **by** *force*
have $\eta \text{ ' source.commutator-subgroup } H K$
 $\subseteq \text{target.generate } (\eta \text{ ' } (\{ \text{source.commutator-elt } h k \mid h k. h \in H \wedge k \in K \}$
 $\cap G) \cap G))$
using *image-generate-sub* *source.Gen-def* *source.commutator-subgroup-def* **by** *pres-*
burger
also have $\eta \text{ ' } (\{ \text{source.commutator-elt } h k \mid h k. h \in H \wedge k \in K \} \cap G) \cap G)$
 $\subseteq \{ \text{target.commutator-elt } h' k' \mid h' k'. h' \in \eta \text{ ' } H \wedge k' \in \eta \text{ ' } K \}$
using *assms* **by** (*force simp: image-def image-commutator-elt*)
then have $\text{target.generate } (\eta \text{ ' } (\{ \text{source.commutator-elt } h k \mid h k. h \in H \wedge k \in K$
 $\} \cap G) \cap G))$
 $\subseteq \text{target.generate } \{ \text{target.commutator-elt } h' k' \mid h' k'. h' \in \eta \text{ ' } H \wedge k' \in$
 $\eta \text{ ' } K \}$
by (*rule target.generate-mono*)
also have $\dots = \text{target.commutator-subgroup } (\eta \text{ ' } H) (\eta \text{ ' } K)$
using *genset-sub* **by** (*simp add: target.commutator-subgroup-def target.Gen-def*
Int-absorb2)
finally show *?thesis* .
qed

lemma *image-commutator-subgroup-super*:
assumes $H \subseteq G \ K \subseteq G$
shows $\text{target.commutator-subgroup } (\eta \text{ ' } H) (\eta \text{ ' } K) \subseteq \eta \text{ ' source.commutator-subgroup}$
 $H K$
proof –
have *seteq*: $\{ \text{target.commutator-elt } h' k' \mid h' k'. h' \in \eta \text{ ' } H \wedge k' \in \eta \text{ ' } K \} \cap G'$
 $\subseteq \eta \text{ ' } (\{ \text{source.commutator-elt } h k \mid h k. h \in H \wedge k \in K \} \cap G)$
using *assms* **by** (*force simp: image-def image-commutator-elt*)
have $\text{target.commutator-subgroup } (\eta \text{ ' } H) (\eta \text{ ' } K)$
 $\subseteq \text{target.generate } (\eta \text{ ' } (\{ \text{source.commutator-elt } h k \mid h k. h \in H \wedge k \in K \} \cap$
 $G))$
by (*simp add: seteq target.Gen-def target.commutator-subgroup-def target.generate-mono*)
also have $\dots \subseteq \eta \text{ ' source.commutator-subgroup } H K$
using *image-generate-super*
by (*fastforce simp add: source.commutator-subgroup-def source.Gen-def*)
finally show *?thesis* .
qed

lemma *image-commutator-subgroup-eq*:
assumes $H \subseteq G \ K \subseteq G$
shows $\eta \text{ ' source.commutator-subgroup } H K = \text{target.commutator-subgroup } (\eta \text{ ' } H)$
 $(\eta \text{ ' } K)$
by (*meson assms image-commutator-subgroup-sub image-commutator-subgroup-super*
equalityI)

For a surjective homomorphism, the derived series of the target is the image of the derived series of the source.

lemma *image-derivedSeries*:

```

assumes  $\eta \text{ ' } G = G'$ 
shows  $\eta \text{ ' source.derivedSeries } n = \text{target.derivedSeries } n$ 
by (induction n) (auto simp: assms image-commutator-subgroup-eq source.derivedSeries-subset)

```

A surjective homomorphic image of a solvable group is solvable.

```

lemma solvable-image:
  assumes  $\eta \text{ ' } G = G'$  and source.solvable
  shows target.solvable
  using assms unfolding source.solvable-def target.solvable-def
  by (metis commutes-with-unit image-derivedSeries image-empty image-insert)

```

Unconditionally (no surjectivity), the image of the source derived series is contained in the target derived series.

```

lemma image-derivedSeries-subset:  $\eta \text{ ' source.derivedSeries } n \subseteq \text{target.derivedSeries } n$ 
proof (induction n)
  case 0
  show ?case using map-closed by auto
next
  case (Suc n) then show ?case
    by (simp add: image-commutator-subgroup-eq source.derivedSeries-subset
      target.commutator-subgroup-mono)
qed

```

A group admitting an injective homomorphism into a solvable group is solvable. If the target is solvable, its derived series reaches $\{1'\}$ at some step n ; the image of the source derived series is contained there (*image-derivedSeries-subset*), so η maps *source.derivedSeries* n to $1'$, and injectivity (with $\eta \mathbf{1} = 1'$) forces it to be $\{1\}$.

```

lemma solvable-of-injective:
  assumes inj: inj-on  $\eta$  G and solv: target.solvable
  shows source.solvable
proof –
  from solv obtain n where n: target.derivedSeries n = {1'}
  unfolding target.solvable-def by blast
  — Every element of source.derivedSeries  $n$  maps to  $1'$ .
  have mapsto:  $\eta x = 1'$  if  $x: x \in \text{source.derivedSeries } n$  for  $x$ 
    by (metis image-derivedSeries-subset image-subset-iff n singletonD x)
  have source.derivedSeries n  $\subseteq \{1\}$ 
    using inj source.unit-closed source.derivedSeries-subset
    using injective-iff-kernel-unit mapsto by blast
  with source.derivedSeries-unit-closed show ?thesis unfolding source.solvable-def
by blast
qed

```

end

A surjective homomorphic image of a solvable group is solvable (top level).

```

theorem (in group-epimorphism) solvable-image-epi:
  assumes source.solvable
  shows target.solvable
  using solvable-image surjective assms by simp

```

12.1 Subgroups

We work in the ambient *Group* and refer to a subgroup's own group operations through the locale-qualified constants *Group.commutator-elt* *H* $(\cdot)$ **1**, *Group.generate* *H* $(\cdot)$ **1** and *Group.derivedSeries* *H* $(\cdot)$ **1**. This bridge avoids the awkward parameter naming of the bare *Subgroup* locale and is reusable for any subgroup.

```

context Group
begin

```

The subgroup's derived series is contained in the ambient derived series.

```

lemma derivedSeries-subgroup-subset:
  assumes H: Subgroup H G  $(\cdot)$  1
  shows Group.derivedSeries H  $(\cdot)$  1 n  $\subseteq$  derivedSeries n
proof –
  interpret S: Subgroup H G  $(\cdot)$  1 by (rule H)
  show ?thesis
  proof (induction n)
    case 0
    show ?case using S.subset by simp
  next
    case (Suc n)
    have Group.derivedSeries H  $(\cdot)$  1 (Suc n)
       $\subseteq$  commutator-subgroup (S.sub.derivedSeries n) (S.sub.derivedSeries n)
      using commutator-subgroup-subgroup-subset[OF H] S.sub.derivedSeries-subset
    by simp
    also have  $\dots \subseteq$  derivedSeries (Suc n)
      using commutator-subgroup-mono[OF Suc.IH Suc.IH] by simp
    finally show ?case .
  qed
qed

```

A subgroup of a solvable group is solvable.

```

theorem solvable-subgroup:
  assumes H: Subgroup H G  $(\cdot)$  1 and solv: solvable
  shows Group.solvable H  $(\cdot)$  1
proof –
  interpret S: Subgroup H G  $(\cdot)$  1 by (rule H)
  show ?thesis
    using solv H S.sub.derivedSeries-unit-closed derivedSeries-subgroup-subset
    unfolding S.sub.solvable-def solvable-def by blast
qed

```


Shift lemma. Once the derived series has entered a subgroup H (at step k), the later terms are dominated by H 's own derived series: $\text{derivedSeries } (k + i)$ is contained in $H^{(i)}$. (Induction on i , using commutator monotonicity and that a commutator of subsets of H is computed inside H .)

lemma *derivedSeries-shift-subgroup:*

assumes H : Subgroup H G $(\cdot)$ **1** **and** k : $\text{derivedSeries } k \subseteq H$

shows $\text{derivedSeries } (k + i) \subseteq \text{Group.derivedSeries } H$ $(\cdot)$ **1** i

proof –

interpret S : Subgroup H G $(\cdot)$ **1** **by** (rule H)

show ?thesis

proof (induction i)

case 0

show ?case **using** k **by** simp

next

case (Suc i) **then**

have $\text{derivedSeries } (k + \text{Suc } i) \subseteq \text{commutator-subgroup } (S.\text{sub.derivedSeries } i)$
 $(S.\text{sub.derivedSeries } i)$

using commutator-subgroup-mono[OF Suc.IH Suc.IH] **by** simp

also have $\dots \subseteq S.\text{sub.derivedSeries } (\text{Suc } i)$

using commutator-subgroup-subset-subgroup[OF H] $S.\text{sub.derivedSeries-subset}$ **by**
 simp

finally show ?case .

qed

qed

end

12.2 Solvability of an extension

An extension of a solvable group by a solvable group is solvable. For a surjective homomorphism η : if the kernel and the image G' are both solvable, so is G . The derived series of G maps onto that of G' (*image-derivedSeries*), which reaches $\{1\}$; so $\text{source.derivedSeries } k \subseteq \text{Ker}$, and the kernel's own solvability then drives the remaining terms to $\{1\}$ via the shift lemma.

theorem (in group-epimorphism) *solvable-extension:*

assumes solvKer : Group.solvable $\text{Ker } (\cdot)$ **1**

and solvImg : target.solvable

shows source.solvable

proof –

— The derived series of G maps onto that of the image, which vanishes at some step.

from solvImg **obtain** k **where** k : $\text{target.derivedSeries } k = \{1\}$

unfolding target.solvable-def **by** blast

— Everything in $\text{source.derivedSeries } k$ maps to $1'$, hence lies in the kernel.

have subKer : $\text{source.derivedSeries } k \subseteq \text{Ker}$

using image-derivedSeries[OF surjective] k $\text{source.derivedSeries-subset}$ **by** blast

— The kernel is a subgroup; its solvability drives the shifted series to $\{1\}$.

from solvKer **obtain** m **where** m : $\text{Group.derivedSeries } \text{Ker } (\cdot)$ **1** $m = \{1\}$

using kernel.sub.solvable-def **by** auto

```

have source.derivedSeries (k + m) ⊆ Group.derivedSeries Ker (·) 1 m
  by (simp add: kernel.Subgroup-axioms source.derivedSeries-shift-subgroup subKer)
then show ?thesis unfolding source.solvable-def
  using m source.derivedSeries-unit-closed by auto
qed

end

```

13 The external direct product of two groups

```

theory Group-Product
  imports Solvable-Transfer
begin

```

The external direct product of two groups: carrier the Cartesian product of the carriers, with componentwise operation and unit. This is the group-theoretic prerequisite for combining two solvable Galois groups (the compositum of two splitting fields embeds into the product of their Galois groups); its headline consequence is that a *product of solvable groups is solvable*.

```

locale group-product =
  G: Group G (·) 1 + G': Group G' (·) 1'
  for G and composition (infixl ⟨·⟩ 70) and unit (⟨1⟩)
  and G' and composition' (infixl ⟨·'⟩ 70) and unit' (⟨1'⟩)
begin

```

```

definition pcarrier :: ('a × 'b) set where pcarrier = G × G'

```

```

definition pcomp :: 'a × 'b ⇒ 'a × 'b ⇒ 'a × 'b (infixl ⟨⊗×⟩ 70)
  where x ⊗× y = (fst x · fst y, snd x ·' snd y)

```

```

definition punit :: 'a × 'b (⟨1×⟩) where 1× = (1, 1')

```

```

lemma pcomp-closed [intro, simp]:
  x ∈ pcarrier ⇒ y ∈ pcarrier ⇒ x ⊗× y ∈ pcarrier
  by (auto simp: pcarrier-def pcomp-def)

```

```

lemma punit-closed [intro, simp]: 1× ∈ pcarrier
  by (auto simp: pcarrier-def punit-def)

```

The direct product is a group.

```

theorem Group-product: Group pcarrier (⊗×) 1×
proof (rule GroupI)
  fix x y z assume x ∈ pcarrier y ∈ pcarrier z ∈ pcarrier
  then show (x ⊗× y) ⊗× z = x ⊗× (y ⊗× z)
    by (auto simp: pcarrier-def pcomp-def G.associative G'.associative)
next
  fix u assume u: u ∈ pcarrier
  then show 1× ⊗× u = u ⊗× 1× = u

```

```

  by (auto simp: pcarrier-def pcomp-def punit-def)
have fu: fst u ∈ G and su: snd u ∈ G'
  using u by (auto simp: pcarrier-def)
obtain a where a: a ∈ G fst u · a = 1 a · fst u = 1
  using G.invertible[OF fu] by (auto simp: G.invertible-def)
obtain b where b: b ∈ G' snd u · b = 1' b · snd u = 1'
  using G'.invertible[OF su] by (auto simp: G'.invertible-def)
show ∃ v ∈ pcarrier. u ⊗× v = 1× ∧ v ⊗× u = 1×
  using a b pcomp-def punit-def pcarrier-def by auto
qed auto

```

```

sublocale prod: Group pcarrier (⊗×) 1×
  by (rule Group-product)

```

13.1 Projections are homomorphisms; the derived series is componentwise-bounded

The (extensional) first and second projections are group homomorphisms onto G and G' . We use *restrict fst pcarrier* so the maps are extensional, as the homomorphism locale requires; on *pcarrier* they agree with *fst* / *snd*.

```

definition p1 :: 'a × 'b ⇒ 'a
  where p1 = restrict fst pcarrier

```

```

definition p2 :: 'a × 'b ⇒ 'b
  where p2 = restrict snd pcarrier

```

```

lemma p1-eq: x ∈ pcarrier ⇒ p1 x = fst x
  by (simp add: p1-def)

```

```

lemma p2-eq: x ∈ pcarrier ⇒ p2 x = snd x
  by (simp add: p2-def)

```

```

lemma p1-hom: group-homomorphism p1 pcarrier (⊗×) 1× G (·) 1
proof

```

```

  show p1 ∈ pcarrier →E G
  by (auto simp: p1-def pcarrier-def)
  show ∧ x y. x ∈ pcarrier ⇒ y ∈ pcarrier ⇒ p1 (x ⊗× y) = p1 x · p1 y
  using p1-eq pcomp-closed pcomp-def by auto
  show p1 1× = 1
  using p1-eq punit-closed punit-def by auto
qed

```

```

lemma p2-hom: group-homomorphism p2 pcarrier (⊗×) 1× G' (·) 1'
proof

```

```

  show p2 ∈ pcarrier →E G'
  by (auto simp: p2-def pcarrier-def)
  show ∧ x y. x ∈ pcarrier ⇒ y ∈ pcarrier ⇒ p2 (x ⊗× y) = p2 x ·' p2 y
  using p2-eq pcomp-closed pcomp-def by auto
  show p2 1× = 1'

```

using p2-eq punit-closed punit-def by auto
qed

An element of the product's derived series projects into each factor's derived series.

lemma *derivedSeries-p1*: $p1 \text{ ' } prod.derivedSeries \ n \subseteq G.derivedSeries \ n$
proof (*induction n*)
 case 0
 show ?case
 by (*meson group-homomorphism.image-derivedSeries-subset p1-hom*)
next
 case (*Suc n*)
 interpret *F*: *group-homomorphism p1 pcarrier* ($\otimes_{\times}$) $\mathbf{1}_{\times} G (\cdot) \mathbf{1}$ **by** (*rule p1-hom*)
 have $p1 \text{ ' } prod.derivedSeries \ (Suc \ n)$
 $\subseteq G.commutator-subgroup \ (p1 \text{ ' } prod.derivedSeries \ n) \ (p1 \text{ ' } prod.derivedSeries \ n)$
 using *F.image-commutator-subgroup-sub*[*OF prod.derivedSeries-subset prod.derivedSeries-subset*]
 by *simp*
 also have $\dots \subseteq G.commutator-subgroup \ (G.derivedSeries \ n) \ (G.derivedSeries \ n)$
 using *G.commutator-subgroup-mono*[*OF Suc.IH Suc.IH*] .
 also have $\dots = G.derivedSeries \ (Suc \ n)$ **by** *simp*
 finally show ?case .
qed

lemma *derivedSeries-p2*: $p2 \text{ ' } prod.derivedSeries \ n \subseteq G'.derivedSeries \ n$
proof (*induction n*)
 case 0 **show** ?case
 by (*meson group-homomorphism.image-derivedSeries-subset p2-hom*)
next
 case (*Suc n*)
 interpret *S*: *group-homomorphism p2 pcarrier* ($\otimes_{\times}$) $\mathbf{1}_{\times} G' (\cdot) \mathbf{1}'$ **by** (*rule p2-hom*)
 have $p2 \text{ ' } prod.derivedSeries \ (Suc \ n)$
 $\subseteq G'.commutator-subgroup \ (p2 \text{ ' } prod.derivedSeries \ n) \ (p2 \text{ ' } prod.derivedSeries \ n)$
 using *S.image-commutator-subgroup-sub*[*OF prod.derivedSeries-subset prod.derivedSeries-subset*]
 by *simp*
 also have $\dots \subseteq G'.commutator-subgroup \ (G'.derivedSeries \ n) \ (G'.derivedSeries \ n)$
 using *G'.commutator-subgroup-mono*[*OF Suc.IH Suc.IH*] .
 also have $\dots = G'.derivedSeries \ (Suc \ n)$ **by** *simp*
 finally show ?case .
qed

13.2 Product of solvable groups is solvable

The direct product of two solvable groups is solvable. If both factors' derived series reach the trivial subgroup (at steps m, k), then at step $\max m \ k$ the product's derived series projects to $\{\mathbf{1}\}$ in each factor (the derived series is monotone decreasing), hence consists of the single element $\mathbf{1}_{\times}$.

theorem *solvable-product*:

```

    assumes solvG:  $G$ .solvable and solvG':  $G'$ .solvable
    shows prod.solvable
  proof -
    obtain  $m$  where  $m$ :  $G$ .derivedSeries  $m = \{1\}$  using solvG unfolding  $G$ .solvable-def
    by blast
    obtain  $k$  where  $k$ :  $G'$ .derivedSeries  $k = \{1\}$  using solvG' unfolding  $G'$ .solvable-def
    by blast
    define  $N$  where  $N = \max m k$ 
    have dGN:  $G$ .derivedSeries  $N = \{1\}$ 
      using  $m$   $G$ .derivedSeries-antimono[of  $m$   $N$ ]  $G$ .derivedSeries-unit-closed by (auto simp:  $N$ -def)
    have dG'N:  $G'$ .derivedSeries  $N = \{1\}$ 
      using  $k$   $G'$ .derivedSeries-antimono[of  $k$   $N$ ]  $G'$ .derivedSeries-unit-closed by (auto simp:  $N$ -def)
    — At step  $N$  the product's derived series projects to a point in each factor.
    have prod.derivedSeries  $N \subseteq \{1_\times\}$ 
    proof
      fix  $x$  assume  $x$ :  $x \in \text{prod.derivedSeries } N$ 
      have  $x$ :  $x \in \text{pcarrier}$  using  $x$  prod.derivedSeries-subset by blast
      then have  $p1\ x = 1$   $p2\ x = 1'$  using derivedSeries-p1 dGN derivedSeries-p2 dG'N
        using  $x$  by blast+
      then show  $x \in \{1_\times\}$ 
        using  $p1\text{-eq}$   $p2\text{-eq}$   $\text{punit-def}$   $x$  by auto
    qed
    moreover have  $1_\times \in \text{prod.derivedSeries } N$  by (rule prod.derivedSeries-unit-closed)
    ultimately have prod.derivedSeries  $N = \{1_\times\}$  by auto
    then show ?thesis unfolding prod.solvable-def by blast
  qed

end

end

```

14 The direct product of an indexed family of groups

```

theory Group-Family-Product
  imports Group-Theory
begin

```

The external direct product of a family of groups $G\ i\ (i \in I)$: the carrier is the *extensional* dependent function space $\Pi_{i \in I} G$ — functions defined on I with $x\ i \in G\ i$ for each index — with pointwise composition and unit. This generalises the binary external direct product of theory *Group-Product* to an arbitrary index set, mirroring *product-group* of *HOL-Algebra.Product-Groups* but in the locale idiom.

14.1 The product locale

A family of groups indexed by I : for each $i \in I$ a carrier $G\ i$ with composition $(\circ)\ i$ and unit $unit\ i$ forming a *comp-group*.

```

locale group-family =
  fixes  $I :: 'i\ set$ 
    and  $G :: 'i \Rightarrow 'a\ set$ 
    and  $comp :: 'i \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ 
    and  $unit :: 'i \Rightarrow 'a$ 
  assumes comp-group:  $\bigwedge i. i \in I \implies Group\ (G\ i)\ (comp\ i)\ (unit\ i)$ 
begin

```

14.2 Carrier, composition and unit

The product carrier: extensional functions x on I with $x\ i \in G\ i$.

```

definition Pcarrier ::  $('i \Rightarrow 'a)\ set\ (\langle \prod_G \rangle)$ 
  where  $\prod_G = (\Pi_{i \in I}. G\ i)$ 

```

Pointwise composition (extensional on I).

```

definition Pcomp ::  $('i \Rightarrow 'a) \Rightarrow ('i \Rightarrow 'a) \Rightarrow ('i \Rightarrow 'a)\ (\text{infixl } \langle \otimes_\Pi \rangle\ 70)$ 
  where  $x \otimes_\Pi y = (\lambda i \in I. comp\ i\ (x\ i)\ (y\ i))$ 

```

The pointwise unit.

```

definition Punit ::  $'i \Rightarrow 'a\ (\langle 1_\Pi \rangle)$ 
  where  $1_\Pi = (\lambda i \in I. unit\ i)$ 

```

```

lemma Pcarrier-memI:
   $\llbracket \bigwedge i. i \in I \implies x\ i \in G\ i; x \in extensional\ I \rrbracket \implies x \in \prod_G$ 
  by (auto simp: Pcarrier-def PiE-iff)

```

```

lemma Pcarrier-mem-component:  $\llbracket x \in \prod_G; i \in I \rrbracket \implies x\ i \in G\ i$ 
  by (auto simp: Pcarrier-def PiE-iff)

```

```

lemma Pcarrier-extensional:  $x \in \prod_G \implies x \in extensional\ I$ 
  by (auto simp: Pcarrier-def PiE-iff)

```

```

lemma Pcomp-closed [intro, simp]:
  assumes  $x \in \prod_G$  and  $y \in \prod_G$  shows  $x \otimes_\Pi y \in \prod_G$ 
proof (rule Pcarrier-memI)
  fix  $i$  assume  $i: i \in I$ 
  interpret  $Gi: Group\ G\ i\ comp\ i\ unit\ i$  using  $i$  by (rule comp-group)
  show  $(x \otimes_\Pi y)\ i \in G\ i$ 
    using  $assms\ i$  by (auto simp: Pcomp-def Pcarrier-mem-component)
qed (auto simp: Pcomp-def)

```

```

lemma Punit-closed [iff]:  $1_\Pi \in \prod_G$ 
  using Group-def Monoid.unit-closed Pcarrier-def Punit-def comp-group by fastforce

```

lemma *Pcomp-apply*: $i \in I \implies (x \otimes_{\Pi} y) i = \text{comp } i (x i) (y i)$
by (*simp add: Pcomp-def*)

lemma *Punit-apply*: $i \in I \implies \mathbf{1}_{\Pi} i = \text{unit } i$
by (*simp add: Punit-def*)

14.3 The product is a group

Two members of the product agree iff they agree componentwise on I (both being extensional).

lemma *Pcarrier-eqI*:
assumes $x \in \prod_G$ **and** $y \in \prod_G$ **and** $\bigwedge i. i \in I \implies x i = y i$
shows $x = y$
using *assms* **by** (*metis Pcarrier-extensional extensionalityI*)

theorem *Group-Pcarrier*: $\text{Group } \prod_G (\otimes_{\Pi}) \mathbf{1}_{\Pi}$

proof (*rule GroupI*)

show $\bigwedge x y. x \in \prod_G \implies y \in \prod_G \implies x \otimes_{\Pi} y \in \prod_G$ **by** (*rule Pcomp-closed*)

show $\mathbf{1}_{\Pi} \in \prod_G$ **by** (*rule Punit-closed*)

next

fix $x y z$ **assume** $xyz: x \in \prod_G y \in \prod_G z \in \prod_G$

show $(x \otimes_{\Pi} y) \otimes_{\Pi} z = x \otimes_{\Pi} (y \otimes_{\Pi} z)$

proof (*rule Pcarrier-eqI*)

fix i **assume** $i: i \in I$

interpret $Gi: \text{Group } G i \text{ comp } i \text{ unit } i$ **using** i **by** (*rule comp-group*)

show $((x \otimes_{\Pi} y) \otimes_{\Pi} z) i = (x \otimes_{\Pi} (y \otimes_{\Pi} z)) i$

using $xyz i$ **by** (*simp add: Pcomp-apply Pcarrier-mem-component Gi.associative*)

qed (*use xyz in auto*)

next

fix x **assume** $x: x \in \prod_G$

show $\mathbf{1}_{\Pi} \otimes_{\Pi} x = x$

proof (*rule Pcarrier-eqI*)

fix i **assume** $i: i \in I$

interpret $Gi: \text{Group } G i \text{ comp } i \text{ unit } i$ **using** i **by** (*rule comp-group*)

show $(\mathbf{1}_{\Pi} \otimes_{\Pi} x) i = x i$

using $x i$ **by** (*simp add: Pcomp-apply Punit-apply Pcarrier-mem-component*)

qed (*use x in auto*)

next

fix x **assume** $x: x \in \prod_G$

show $x \otimes_{\Pi} \mathbf{1}_{\Pi} = x$

proof (*rule Pcarrier-eqI*)

fix i **assume** $i: i \in I$

interpret $Gi: \text{Group } G i \text{ comp } i \text{ unit } i$ **using** i **by** (*rule comp-group*)

show $(x \otimes_{\Pi} \mathbf{1}_{\Pi}) i = x i$

using $x i$ **by** (*simp add: Pcomp-apply Punit-apply Pcarrier-mem-component*)

qed (*use x in auto*)

next

fix x **assume** $x: x \in \prod_G$

— The inverse is the pointwise inverse.

— Choose a two-sided inverse in each factor.

have $\exists v. \forall i \in I. v \ i \in G \ i \wedge \text{comp } i \ (x \ i) \ (v \ i) = \text{unit } i \wedge \text{comp } i \ (v \ i) \ (x \ i) = \text{unit } i$

proof (*intro bchoice ballI*)

fix i **assume** $i: i \in I$

interpret Gi : *Group* $G \ i \ \text{comp } i \ \text{unit } i$ **using** i **by** (*rule comp-group*)

have xi : $x \ i \in G \ i$ **using** $x \ i$ **by** (*rule Pcarrier-mem-component*)

then have $Gi.\text{invertible} \ (x \ i)$ **by** *simp*

then show $\exists v. v \in G \ i \wedge \text{comp } i \ (x \ i) \ v = \text{unit } i \wedge \text{comp } i \ v \ (x \ i) = \text{unit } i$

using xi **by** (*auto simp: Gi.invertible-def*)

qed

then obtain v **where** $v: \bigwedge i. i \in I \implies v \ i \in G \ i$

$\wedge \text{comp } i \ (x \ i) \ (v \ i) = \text{unit } i \wedge \text{comp } i \ (v \ i) \ (x \ i) = \text{unit } i$ **by** *blast*

show $\exists y \in \prod_G. x \otimes_\Pi y = \mathbf{1}_\Pi \wedge y \otimes_\Pi x = \mathbf{1}_\Pi$

proof

define y **where** $y = (\lambda i \in I. v \ i)$

have yi : $y \ i = v \ i$ **if** $i \in I$ **for** i **using** *that* **by** (*simp add: y-def*)

show yG : $y \in \prod_G$

by (*rule Pcarrier-memI*) (*use v in <auto simp: y-def>*)

have $x \otimes_\Pi y = \mathbf{1}_\Pi \wedge y \otimes_\Pi x = \mathbf{1}_\Pi$

proof (*intro conjI Pcarrier-eqI Punit-closed Pcomp-closed x yG*)

fix i **assume** $i: i \in I$

show $(x \otimes_\Pi y) \ i = \mathbf{1}_\Pi \ i \ (y \otimes_\Pi x) \ i = \mathbf{1}_\Pi \ i$

using $v[OF \ i] \ i$ **by** (*simp-all add: Pcomp-apply Punit-apply yi*)

qed

then show $x \otimes_\Pi y = \mathbf{1}_\Pi \wedge y \otimes_\Pi x = \mathbf{1}_\Pi$.

qed

qed

sublocale *product*: *Group* $Pcarrier \ (\otimes_\Pi) \ Punit$

by (*rule Group-Pcarrier*)

14.4 Projections are homomorphisms

The projection onto the i th factor is a *comp-group* homomorphism from the product onto $G \ i$.

definition $Pproj :: 'i \Rightarrow ('i \Rightarrow 'a) \Rightarrow 'a$

where $Pproj \ i = \text{restrict} \ (\lambda x. x \ i) \ \prod_G$

lemma *Pproj-apply*: $x \in \prod_G \implies Pproj \ i \ x = x \ i$

by (*simp add: Pproj-def*)

lemma *Pproj-hom*:

assumes $i: i \in I$

shows *group-homomorphism* $(Pproj \ i) \ \prod_G \ (\otimes_\Pi) \ \mathbf{1}_\Pi \ (G \ i) \ (\text{comp } i) \ (\text{unit } i)$

proof —

interpret Gi : *Group* $G \ i \ \text{comp } i \ \text{unit } i$ **using** i **by** (*rule comp-group*)

have *graph*: $Pproj \ i \in \prod_G \rightarrow_E \ G \ i$

by (*simp add: Pcarrier-def PiE-mem Pproj-def i*)

show *?thesis*


```

proof
  qed (use i in ⟨auto simp: graph Pproj-apply Punit-apply Pcomp-apply⟩)
qed

end

```

14.5 The product of a family of abelian groups is abelian

```

locale abelian-group-family = group-family +
  assumes abelian:  $\bigwedge i. i \in I \implies \text{Abelian-Group } (G\ i) \ (comp\ i) \ (unit\ i)$ 
begin

theorem abelian-group-Pcarrier:  $\text{Abelian-Group } \prod_G (\otimes_\Pi) \mathbf{1}_\Pi$ 
proof (rule Abelian-Group.intro)
  show  $\text{Group } \prod_G (\otimes_\Pi) \mathbf{1}_\Pi$  by (rule Group-Pcarrier)
  — The product operation is commutative, so the (already-established) monoid is
  commutative.
  have comm:  $x \otimes_\Pi y = y \otimes_\Pi x$  if xy:  $x \in \prod_G y \in \prod_G$  for x y
  proof (rule Pcarrier-eqI)
    fix i assume i:  $i \in I$ 
    interpret Gi:  $\text{Abelian-Group } G\ i\ comp\ i\ unit\ i$  using i by (rule abelian)
    show  $(x \otimes_\Pi y)\ i = (y \otimes_\Pi x)\ i$ 
    using xy i by (simp add: Pcomp-apply Pcarrier-mem-component Gi.commutative)
  qed (use xy in auto)
  show commutative-monoid  $\prod_G (\otimes_\Pi) \mathbf{1}_\Pi$ 
    by (simp add: comm commutative-monoid-axioms.intro commutative-monoid-def
product.Monoid-axioms)
  qed

end

end

```

```

theory Zassenhaus-Lemma
imports Group-Theory
begin

```

15 The Zassenhaus Lemma

Suppose that $H1$ is normal in H , and $K1$ is normal in K . The two wings of the butterfly are obtained by multiplying each normal subgroup by $H \cap K$; their denominators use the smaller intersections $H \cap K1$ and $K \cap H1$.

A locale records this symmetric configuration once. In particular, no ambient record updates or HOL-Algebra structures occur in the statement.

```

locale zassenhaus =
  G:  $\text{Group } G\ (\cdot)\ \mathbf{1} +$ 
  H:  $\text{subgroup-of-group } H\ G\ (\cdot)\ \mathbf{1} +$ 

```

```

K: subgroup-of-group K G (·) 1 +
H1: normal-subgroup H1 H (·) 1 +
K1: normal-subgroup K1 K (·) 1
for G H K H1 K1 and composition (infixl <·> 70) and unit (<1>)
begin

interpretation HK-sub: Subgroup H ∩ K G (·) 1
  by (simp add: G.subgroup-intersection H.Subgroup-axioms K.Subgroup-axioms)

interpretation HK1-subG: Subgroup H ∩ K1 G (·) 1
  using G.subgroup-intersection H.Subgroup-axioms K.Subgroup-axioms K1.Subgroup-axioms

  by (meson subgroup-transitive)

interpretation HK1-sub: Subgroup H ∩ K1 H (·) 1
  using H.Subgroup-axioms HK1-subG.Subgroup-axioms subgroup-restrict by fastforce

interpretation KH1-sub: Subgroup K ∩ H1 K (·) 1
proof -
  have Subgroup (K ∩ H1) G (·) 1
    using G.subgroup-intersection H.Subgroup-axioms H1.Subgroup-axioms K.Subgroup-axioms
    by (meson subgroup-transitive)
  then show Subgroup (K ∩ H1) K (·) 1
    using K.Subgroup-axioms subgroup-restrict by force
qed

interpretation HK1-normal: normal-subgroup H ∩ K1 H ∩ K (·) 1
proof -
  have I-sub: Subgroup (H ∩ K) K (·) 1
    using HK-sub.Subgroup-axioms K.Subgroup-axioms subgroup-restrict by fastforce
  have Subgroup ((H ∩ K) ∩ K1) K (·) 1
    by (rule K.sub.subgroup-intersection[OF I-sub K1.Subgroup-axioms])
  then have C-K: Subgroup (H ∩ K1) K (·) 1
    using K1.subset by (smt (cvc5, best) dual-order.refl inf.bounded-iff subset-antisym)
  have C-sub: Subgroup (H ∩ K1) (H ∩ K) (·) 1
    using C-K I-sub subgroup-restrict by fastforce
  interpret C: Subgroup H ∩ K1 H ∩ K (·) 1 by fact
  interpret I: Subgroup H ∩ K K (·) 1 by fact
  have I-H-sub: Subgroup (H ∩ K) H (·) 1
    using H.Subgroup-axioms HK-sub.Subgroup-axioms by (meson inf-sup-ord(1) subgroup-restrict)
  interpret I-H: Subgroup H ∩ K H (·) 1 by fact
  show normal-subgroup (H ∩ K1) (H ∩ K) (·) 1
  proof
    fix g c
    assume g: g ∈ H ∩ K and c: c ∈ H ∩ K1
    obtain inv-K: Monoid.inverse (H ∩ K) (·) 1 g = K.sub.inverse g
      and inv-H: Monoid.inverse (H ∩ K) (·) 1 g = H.sub.inverse g
    using I-H-sub I-sub g by (metis Subgroup.subgroup-inverse-equality)
    have g-H: g ∈ H and c-H: c ∈ H and g-K: g ∈ K and c-K1: c ∈ K1

```

```

    using g c by blast+
  have in-H:  $H.sub.inverse\ g \cdot c \cdot g \in H$ 
    using Monoid.invertible-inverse-closed c-H g-H by blast
  have in-K1:  $K.sub.inverse\ g \cdot c \cdot g \in K1$ 
    by (rule K1.normal[OF g-K c-K1])
  show Monoid.inverse  $(H \cap K) (\cdot) 1\ g \cdot c \cdot g \in H \cap K1$ 
    using in-H in-K1 inv-H inv-K by (metis IntI)
qed
qed

interpretation KH1-normal: normal-subgroup  $K \cap H1\ H \cap K (\cdot) 1$ 
proof -
  have I-sub: Subgroup  $(H \cap K)\ H (\cdot) 1$ 
    using H.Subgroup-axioms HK-sub.Subgroup-axioms subgroup-restrict by force
  have Subgroup  $((H \cap K) \cap H1)\ H (\cdot) 1$ 
    by (simp add: H.sub.subgroup-intersection H1.Subgroup-axioms I-sub)
  then have C-H: Subgroup  $(K \cap H1)\ H (\cdot) 1$ 
    using H1.subset by (smt (cvc5, best) dual-order.refl inf.bounded-iff subset-antisym)
  have C-sub: Subgroup  $(K \cap H1)\ (H \cap K) (\cdot) 1$ 
    using C-H I-sub subgroup-restrict by fastforce
  interpret C: Subgroup  $K \cap H1\ H \cap K (\cdot) 1$  by fact
  interpret I: Subgroup  $H \cap K\ H (\cdot) 1$  by fact
  have I-K-sub: Subgroup  $(H \cap K)\ K (\cdot) 1$ 
    using HK-sub.Subgroup-axioms K.Subgroup-axioms subgroup-restrict by fastforce
  interpret I-K: Subgroup  $H \cap K\ K (\cdot) 1$  by fact
  show normal-subgroup  $(K \cap H1)\ (H \cap K) (\cdot) 1$ 
  proof
    fix g c
    assume g:  $g \in H \cap K$  and c:  $c \in K \cap H1$ 
    have inv-H: Monoid.inverse  $(H \cap K) (\cdot) 1\ g = H.sub.inverse\ g$ 
      using I.subgroup-inverse-equality g by presburger
    have inv-K: Monoid.inverse  $(H \cap K) (\cdot) 1\ g = K.sub.inverse\ g$ 
      using I-K.subgroup-inverse-equality g by presburger
    have g-K:  $g \in K$  and c-K:  $c \in K$  and g-H:  $g \in H$  and c-H1:  $c \in H1$  and in-K:
       $K.sub.inverse\ g \cdot c \cdot g \in K$ 
      using g c by blast+
    have in-H1:  $H.sub.inverse\ g \cdot c \cdot g \in H1$ 
      by (rule H1.normal[OF g-H c-H1])
    show Monoid.inverse  $(H \cap K) (\cdot) 1\ g \cdot c \cdot g \in K \cap H1$ 
      using in-H1 in-K inv-H inv-K by (metis IntI)
  qed
qed

interpretation middle-H: Subgroup  $H \cap K\ H (\cdot) 1$ 
  using H.Subgroup-axioms HK-sub.Subgroup-axioms subgroup-restrict by force

interpretation middle-K: Subgroup  $H \cap K\ K (\cdot) 1$ 
  using HK-sub.Subgroup-axioms K.Subgroup-axioms subgroup-restrict by force

interpretation middle-H-group: subgroup-of-group  $H \cap K\ H (\cdot) 1$ 

```

by (*simp add: H.sub.Group-axioms middle-H.Subgroup-axioms subgroup-of-group-def*)

interpretation *middle-K-group: subgroup-of-group $H \cap K K (\cdot) \mathbf{1}$*
by (*simp add: K.sub.Group-axioms middle-K.Subgroup-axioms subgroup-of-group-def*)

interpretation *HK1-group: subgroup-of-group $H \cap K1 H (\cdot) \mathbf{1}$*
by (*simp add: H.sub.Group-axioms HK1-sub.Subgroup-axioms subgroup-of-groupI*)

interpretation *KH1-group: subgroup-of-group $K \cap H1 K (\cdot) \mathbf{1}$*
by (*simp add: K.sub.Group-axioms KH1-sub.Subgroup-axioms subgroup-of-group-def*)

The four subgroup products making up the two wings.

definition *left-top where*
 $left-top \equiv (case-prod (\cdot)) ' ((H \cap K) \times H1)$

definition *left-bottom where*
 $left-bottom \equiv (case-prod (\cdot)) ' ((H \cap K1) \times H1)$

definition *right-top where*
 $right-top \equiv (case-prod (\cdot)) ' ((H \cap K) \times K1)$

definition *right-bottom where*
 $right-bottom \equiv (case-prod (\cdot)) ' ((K \cap H1) \times K1)$

interpretation *left-top-product: normal-subgroup-product $H1 H \cap K H (\cdot) \mathbf{1} ..$*

interpretation *left-bottom-product: normal-subgroup-product $H1 H \cap K1 H (\cdot) \mathbf{1} ..$*

interpretation *right-top-product: normal-subgroup-product $K1 H \cap K K (\cdot) \mathbf{1} ..$*

interpretation *right-bottom-product: normal-subgroup-product $K1 K \cap H1 K (\cdot) \mathbf{1} ..$*

lemma *left-top-eq [simp]: left-top = left-top-product.HK*
by (*simp add: left-top-def left-top-product.HK-def*)

lemma *left-bottom-eq [simp]: left-bottom = left-bottom-product.HK*
by (*simp add: left-bottom-def left-bottom-product.HK-def*)

lemma *right-top-eq [simp]: right-top = right-top-product.HK*
by (*simp add: right-top-def right-top-product.HK-def*)

lemma *right-bottom-eq [simp]: right-bottom = right-bottom-product.HK*
by (*simp add: right-bottom-def right-bottom-product.HK-def*)

lemma *left-top-subgroup: Subgroup left-top $H (\cdot) \mathbf{1}$*
by (*simp add: left-top-product.HK-subgroup*)

lemma *left-bottom-subgroup: Subgroup left-bottom $H (\cdot) \mathbf{1}$*
by (*simp add: left-bottom-product.HK-subgroup*)

lemma *right-top-subgroup*: Subgroup *right-top* K $(\cdot)$ **1**
by (*simp add: right-top-product.HK-subgroup*)

lemma *right-bottom-subgroup*: Subgroup *right-bottom* K $(\cdot)$ **1**
by (*simp add: right-bottom-product.HK-subgroup*)

lemma *left-bottom-subset-top*: *left-bottom* $\subseteq$ *left-top*
unfolding *left-bottom-def left-top-def*
by *blast*

lemma *right-bottom-subset-top*: *right-bottom* $\subseteq$ *right-top*
unfolding *right-bottom-def right-top-def*
by *blast*

lemma *left-top-memE*:
assumes $x \in \text{left-top}$
obtains $b\ n$ **where** $b \in H \cap K\ n \in H1\ x = b \cdot n$
using *assms left-top-eq left-top-product.HK-memE* **by** *metis*

lemma *left-bottom-memI*:
 $\llbracket c \in H \cap K1; n \in H1 \rrbracket \implies c \cdot n \in \text{left-bottom}$
unfolding *left-bottom-eq* **by** (*rule left-bottom-product.HK-memI*)

lemma *left-bottom-memE*:
assumes $x \in \text{left-bottom}$
obtains $c\ n$ **where** $c \in H \cap K1\ n \in H1\ x = c \cdot n$
using *assms left-bottom-eq left-bottom-product.HK-memE* **by** *blast*

The lower subgroup in the left wing is normal in the upper one.

lemma *left-bottom-normal*: *normal-subgroup left-bottom left-top* $(\cdot)$ **1**
proof –
have *bottom-top*: Subgroup *left-bottom left-top* $(\cdot)$ **1**
by (*rule subgroup-restrict[OF left-bottom-subgroup left-top-subgroup left-bottom-subset-top]*)
interpret *bottom*: Subgroup *left-bottom left-top* $(\cdot)$ **1** **by** *fact*
interpret *top-H*: Subgroup *left-top H* $(\cdot)$ **1** **by** (*rule left-top-subgroup*)
show *?thesis*
proof
fix $x\ y$
assume $x: x \in \text{left-top}$ **and** $y: y \in \text{left-bottom}$
obtain $b\ n$ **where** *b-middle*: $b \in H \cap K$ **and** $n: n \in H1$ **and** *x-eq*: $x = b \cdot n$
using x **by** (*rule left-top-memE*)
obtain $c\ m$ **where** $c: c \in H \cap K1$ **and** $m: m \in H1$ **and** *y-eq*: $y = c \cdot m$
using y **by** (*rule left-bottom-memE*)
have *bH*: $b \in H$ **and** *cH*: $c \in H$ **and** *nH*: $n \in H$ **and** *mH*: $m \in H$
using *b-middle c n m H1.sub* **by** *blast+*
have *inv-middle*: $H.\text{sub.inverse } b = \text{Monoid.inverse } (H \cap K) (\cdot)$ **1** b
using *b-middle middle-H.subgroup-inverse-equality* **by** *blast*
have *conjugate-c*: $H.\text{sub.inverse } b \cdot c \cdot b \in H \cap K1$
using *HK1-normal.normal b-middle c inv-middle* **by** *presburger*
have *conjugate-y*: $H.\text{sub.inverse } b \cdot y \cdot b \in \text{left-bottom}$

```

using  $H1.normal[OF\ bH\ m]$   $H.sub.conjugate-composition\ H.sub.invertible\ bH\ cH$ 
conjugate-c
  left-bottom-memI mH y-eq by presburger
have inv-n:  $H.sub.inverse\ n \in H1$ 
  by (rule  $H1.submonoid-inverse-closed[OF\ H1.sub.invertible[OF\ n]\ n]$ )
have inv-n-bottom:  $H.sub.inverse\ n \in left-bottom$ 
  using inv-n left-bottom-eq left-bottom-product.K-subset-HK by blast
have n-bottom:  $n \in left-bottom$ 
  using left-bottom-product.K-subset-HK n by auto
have inv-x:  $H.sub.inverse\ x = H.sub.inverse\ n \cdot H.sub.inverse\ b$ 
  unfolding x-eq using  $H.sub.inverse-composition-commute\ bH\ nH$  by blast
obtain inv-b-H:  $H.sub.inverse\ b \in H$  and inv-n-H:  $H.sub.inverse\ n \in H$ 
  using  $H.sub.invertible-inverse-closed\ H1.sub\ bH\ inv-n$  by blast
have y-top:  $y \in left-top$  using bottom.subset y by blast
have yH:  $y \in H$  using top-H.subset y-top by blast
have conjugate-expansion:  $H.sub.inverse\ x \cdot y \cdot x = H.sub.inverse\ n \cdot (H.sub.inverse\ b \cdot y \cdot b) \cdot n$ 
  using  $H.sub.associative\ H.sub.composition-closed\ bH\ inv-b-H\ inv-n-H\ inv-x\ nH$ 
x-eq yH
  by metis
then have  $H.sub.inverse\ x \cdot y \cdot x \in left-bottom$ 
  using bottom.sub-composition-closed conjugate-y inv-n-bottom n-bottom by presburger
then show  $Monoid.inverse\ left-top\ (\cdot)\ 1\ x \cdot y \cdot x \in left-bottom$ 
  using top-H.subgroup-inverse-equality x by metis
qed
qed

```

The following two identities are the algebraic centre of the butterfly. An element of the middle intersection that is written as a product in one wing forces its normal-subgroup factor into the opposite subgroup.

lemma *middle-inter-left-bottom:*

$(H \cap K) \cap left-bottom =$
 $(case-prod\ (\cdot))\ '((H \cap K1) \times (K \cap H1))$

proof (intro equalityI subsetI)

fix x

assume x: $x \in (H \cap K) \cap left-bottom$

then obtain p **where** p-in: $p \in (H \cap K1) \times H1$ **and** p-eq: $x = case-prod\ (\cdot)\ p$

unfolding left-bottom-def **by** blast

obtain c h **where** p: $p = (c,h)$ **by** (cases p)

have c: $c \in H \cap K1$ **and** h: $h \in H1$

using p-in **unfolding** p **by** blast+

have x-eq: $x = c \cdot h$ **using** p-eq **by** (simp only: p prod.case)

have cK: $c \in K$ **using** c K1.sub **by** blast

have xK: $x \in K$ **using** x **by** blast

have cG: $c \in G$ **and** hG: $h \in G$

using c h H.sub H1.sub **by** blast+

then have G.inverse c · x = h

using G.invertible-left-inverse2 x-eq **by** blast

moreover have $inv\text{-}c\text{-}x\text{-}K$: $G.inverse\ c \cdot x \in K$
using $cK\ xK$ **by** $blast$
ultimately have hK : $h \in K$ **by** $meson$
show $x \in (case\text{-}prod\ (\cdot))\ ' ((H \cap K1) \times (K \cap H1))$
using $c\ h\ hK\ x\text{-}eq$ **by** $blast$
next
fix x
assume $x \in (case\text{-}prod\ (\cdot))\ ' ((H \cap K1) \times (K \cap H1))$
then obtain $p\ c\ h$ **where** $p\text{-}in$: $p \in (H \cap K1) \times (K \cap H1)$
and $p\text{-}eq$: $x = case\text{-}prod\ (\cdot)\ p$ **and** p : $p = (c, h)$ **by** $blast$
have c : $c \in H \cap K1$ **and** h : $h \in K \cap H1$
using $p\text{-}in$ **unfolding** p **by** $blast+$
have $x\text{-}eq$: $x = c \cdot h$ **using** $p\text{-}eq$ **by** $(simp\ only: p\ prod.case)$
have $x \in H \cap K$
unfolding $x\text{-}eq$ **using** $c\ h$ **by** $blast$
moreover have $x \in left\text{-}bottom$
unfolding $left\text{-}bottom\text{-}def$ **using** $c\ h\ x\text{-}eq$ **by** $blast$
ultimately show $x \in (H \cap K) \cap left\text{-}bottom$ **by** $blast$
qed

interpretation $symmetric$: $zassenhaus\ G\ K\ H\ K1\ H1\ (\cdot)\ 1\ ..$

lemma $middle\text{-}inter\text{-}right\text{-}bottom$:

$(H \cap K) \cap right\text{-}bottom = (case\text{-}prod\ (\cdot))\ ' ((K \cap H1) \times (H \cap K1))$
using $right\text{-}bottom\text{-}def\ symmetric.left\text{-}bottom\text{-}def\ symmetric.middle\text{-}inter\text{-}left\text{-}bottom$
by $(metis\ Int\text{-}commute)$

The normality argument for the right wing is exactly the symmetric instance.

lemma $right\text{-}bottom\text{-}normal$: $normal\text{-}subgroup\ right\text{-}bottom\ right\text{-}top\ (\cdot)\ 1$

using $right\text{-}bottom\text{-}eq\ right\text{-}top\text{-}eq\ symmetric.left\text{-}bottom\text{-}eq\ symmetric.left\text{-}bottom\text{-}normal$
 $symmetric.left\text{-}top\text{-}eq$ **by** $(metis\ inf.\text{commute})$

15.1 The quotient isomorphism

interpretation $left\text{-}quotient$: $normal\text{-}subgroup\ left\text{-}bottom\ left\text{-}top\ (\cdot)\ 1$

by $(rule\ left\text{-}bottom\text{-}normal)$

interpretation $right\text{-}quotient$: $normal\text{-}subgroup\ right\text{-}bottom\ right\text{-}top\ (\cdot)\ 1$

by $(rule\ right\text{-}bottom\text{-}normal)$

lemma $middle\text{-}subset\text{-}left\text{-}top$: $H \cap K \subseteq left\text{-}top$

using $left\text{-}top\text{-}product.H\text{-}subset\text{-}HK$ **unfolding** $left\text{-}top\text{-}eq$.

lemma $middle\text{-}subset\text{-}right\text{-}top$: $H \cap K \subseteq right\text{-}top$

using $right\text{-}top\text{-}product.H\text{-}subset\text{-}HK$ **unfolding** $right\text{-}top\text{-}eq$.

lemma $middle\text{-}left\text{-}top\text{-}subgroup$: $Subgroup\ (H \cap K)\ left\text{-}top\ (\cdot)\ 1$

by $(rule\ subgroup\text{-}restrict[OF\ middle.H.Subgroup\text{-}axioms\ left\text{-}top\text{-}subgroup\ middle\text{-}subset\text{-}left\text{-}top])$

lemma $middle\text{-}right\text{-}top\text{-}subgroup$: $Subgroup\ (H \cap K)\ right\text{-}top\ (\cdot)\ 1$

by (rule subgroup-restrict[OF middle-K.Subgroup-axioms right-top-subgroup middle-subset-right-top])

interpretation middle-left-top: subgroup-of-group $H \cap K$ left-top $(\cdot)$ **1**
by (rule subgroup-of-groupI[OF middle-left-top-subgroup left-quotient.Group-axioms])

interpretation middle-right-top: subgroup-of-group $H \cap K$ right-top $(\cdot)$ **1**
by (rule subgroup-of-groupI[OF middle-right-top-subgroup right-quotient.Group-axioms])

interpretation left-second:
second-iso-theorem left-bottom $H \cap K$ left-top $(\cdot)$ **1** ..

interpretation right-second:
second-iso-theorem right-bottom $H \cap K$ right-top $(\cdot)$ **1** ..

lemma left-second-product: left-second.HK = left-top

proof (intro equalityI subsetI)

fix x

assume $x \in \text{left-second.HK}$

then obtain $h\ k$ **where** $h: h \in H \cap K$ **and** $k: k \in \text{left-bottom}$

and $x\text{-eq}: x = h \cdot k$ **by** (rule left-second.HK-memE)

show $x \in \text{left-top}$

unfolding $x\text{-eq}$ **using** $h\ k$ **by** blast

next

fix x

assume $x: x \in \text{left-top}$

then obtain $h\ n$ **where** $h: h \in H \cap K$ **and** $n: n \in H1$ **and** $x\text{-eq}: x = h \cdot n$

by (rule left-top-memE)

have unit-inter: $1 \in H \cap K1$

using $H.\text{sub-unit-closed}\ K1.\text{sub-unit-closed}$ **by** blast

then have $n \in \text{left-bottom}$

using $H1.\text{sub.left-unit}\ \text{left-bottom-memI}\ n$ **by** metis

then show $x \in \text{left-second.HK}$

using $h\ \text{left-second.HK-memI}\ x\text{-eq}$ **by** blast

qed

lemma right-second-product: right-second.HK = right-top

proof (intro equalityI subsetI)

fix x

assume $x \in \text{right-second.HK}$

then obtain $h\ k$ **where** $h: h \in H \cap K$ **and** $k: k \in \text{right-bottom}$

and $x\text{-eq}: x = h \cdot k$ **by** (rule right-second.HK-memE)

show $x \in \text{right-top}$

unfolding $x\text{-eq}$ **using** $h\ k$ **by** blast

next

fix x

assume $x: x \in \text{right-top}$

then obtain $h\ n$ **where** $h': h \in K \cap H$ **and** $n: n \in K1$ **and** $x\text{-eq}: x = h \cdot n$

using $\text{right-top-eq}\ \text{right-top-product.HK-memE}$ **by** (metis Int-commute)

have unit-inter: $1 \in K \cap H1$

using $K.sub\text{-}unit\text{-}closed$ $H1.sub\text{-}unit\text{-}closed$ by blast
 then have $n \in right\text{-}bottom$
 using $K1.sub.left\text{-}unit$ n $right\text{-}bottom\text{-}eq$ $right\text{-}bottom\text{-}product.HK\text{-}memI$ by metis
 then show $x \in right\text{-}second.HK$ unfolding $x\text{-}eq$
 using h' $right\text{-}second.HK\text{-}memI$ by blast
 qed

lemma *left-second-factor-group*:
 $left\text{-}second.Fract\text{-}HK\text{-}K = left\text{-}quotient.Factor\text{-}Group$
 using $left\text{-}quotient.Partition\text{-}def$ $left\text{-}second.Fract\text{-}HK\text{-}K\text{-}def$ $left\text{-}second\text{-}product$ by
 argo

lemma *right-second-factor-group*:
 $right\text{-}second.Fract\text{-}HK\text{-}K = right\text{-}quotient.Factor\text{-}Group$
 using $right\text{-}quotient.Partition\text{-}def$ $right\text{-}second.Fract\text{-}HK\text{-}K\text{-}def$ $right\text{-}second\text{-}product$
 by argo

lemma *common-denominator*:
 $(H \cap K) \cap left\text{-}bottom = (H \cap K) \cap right\text{-}bottom$
 unfolding $middle\text{-}inter\text{-}left\text{-}bottom$ $middle\text{-}inter\text{-}right\text{-}bottom$
 using $HK1\text{-}normal.subset$ $KH1\text{-}normal.set\text{-}prod\text{-}commute$ by metis

interpretation *left-common*:
 $normal\text{-}subgroup (H \cap K) \cap left\text{-}bottom$ $H \cap K (\cdot) 1$
 using $left\text{-}bottom\text{-}normal$ $left\text{-}quotient.normal\text{-}subgroup\text{-}intersection$ $middle\text{-}left\text{-}top\text{-}subgroup$
 by blast

interpretation *right-common*:
 $normal\text{-}subgroup (H \cap K) \cap right\text{-}bottom$ $H \cap K (\cdot) 1$
 using $common\text{-}denominator$ $left\text{-}common.normal\text{-}subgroup\text{-}axioms$ by force

lemma *left-wing-isomorphism*:
 $(left\text{-}quotient.Factor\text{-}Group, left\text{-}quotient.quotient\text{-}composition,$
 $left\text{-}quotient.Class\ 1) \cong_G$
 $(left\text{-}second.Fract\text{-}H\text{-}HIntK, left\text{-}common.quotient\text{-}composition,$
 $left\text{-}common.Class\ 1)$
 using $left\text{-}second.second\text{-}isomorphism$
 unfolding $left\text{-}second\text{-}factor\text{-}group$.

lemma *right-wing-isomorphism*:
 $(right\text{-}quotient.Factor\text{-}Group, right\text{-}quotient.quotient\text{-}composition,$
 $right\text{-}quotient.Class\ 1) \cong_G$
 $(right\text{-}second.Fract\text{-}H\text{-}HIntK, right\text{-}common.quotient\text{-}composition,$
 $right\text{-}common.Class\ 1)$
 using $right\text{-}second.second\text{-}isomorphism$
 unfolding $right\text{-}second\text{-}factor\text{-}group$.

lemma *common-factor-group*:
 $(left\text{-}second.Fract\text{-}H\text{-}HIntK, left\text{-}common.quotient\text{-}composition,$

```

    left-common.Class 1) =
    (right-second.Fract-H-HIntK, right-common.quotient-composition,
     right-common.Class 1)
using common-denominator left-second.Fract-H-HIntK-def right-second.Fract-H-HIntK-def
by presburger

```

The Zassenhaus lemma: the quotients represented by the two wings of the butterfly are isomorphic. The quotient operations and units are displayed explicitly, as is customary in New-Algebra's set-based API.

```

theorem butterfly-lemma:
  (left-quotient.Factor-Group, left-quotient.quotient-composition,
   left-quotient.Class 1)  $\cong_G$ 
  (right-quotient.Factor-Group, right-quotient.quotient-composition,
   right-quotient.Class 1)
using common-factor-group left-wing-isomorphism right-wing-isomorphism
by (metis isomorphic-as-groups-symmetric isomorphic-as-groups-transitive)

end

end

```

16 The Schreier refinement theorem

```

theory Schreier-Refinement
imports Normal-Chain Zassenhaus-Lemma
begin

```

16.1 Refining pairs of finite normal series

The Zassenhaus lemma compares one refinement cell determined by a step from each of two normal series. We name the two families of subgroup products independently of a particular cell. Fixing one index produces a row of intermediate subgroups refining the corresponding step of the other series.

```

locale normal-series-pair =
  A: normal-series G (·) 1 A m +
  B: normal-series G (·) 1 B n
  for G and composition (infixl ⟨·⟩ 70) and unit (⟨1⟩)
    and A :: nat  $\Rightarrow$  'a set and m
    and B :: nat  $\Rightarrow$  'a set and n
begin

definition left-refinement-term :: nat  $\Rightarrow$  nat  $\Rightarrow$  'a set
  where left-refinement-term i j  $\equiv$  (case-prod (·)) ' ((A (Suc i)  $\cap$  B j)  $\times$  A i)

definition right-refinement-term :: nat  $\Rightarrow$  nat  $\Rightarrow$  'a set
  where right-refinement-term j i  $\equiv$  (case-prod (·)) ' ((B (Suc j)  $\cap$  A i)  $\times$  B j)

end

```

```

locale series-refinement-cell =
  normal-series-pair  $G (\cdot) \mathbf{1} A m B n$ 
  for  $G$  and composition (infixl  $\langle \cdot \rangle$  70) and unit ( $\langle \mathbf{1} \rangle$ )
    and  $A :: \text{nat} \Rightarrow 'a \text{ set}$  and  $m$ 
    and  $B :: \text{nat} \Rightarrow 'a \text{ set}$  and  $n$ 
    and  $i j +$ 
  assumes i-lt:  $i < m$  and j-lt:  $j < n$ 
begin

lemma ambient-group:
  Group  $G (\cdot) \mathbf{1}$ 
  by (rule  $A.G.Group\text{-}axioms$ )

lemma A-step-top-subgroup:
  subgroup-of-group ( $A (Suc\ i)$ )  $G (\cdot) \mathbf{1}$ 
  by (simp add:  $A.term\text{-}subgroup\ Suc\ leI\ ambient\text{-}group\ i\text{-}lt\ subgroup\text{-}of\text{-}group\text{-}def$ )

lemma B-step-top-subgroup:
  subgroup-of-group ( $B (Suc\ j)$ )  $G (\cdot) \mathbf{1}$ 
  by (simp add:  $B.term\text{-}subgroup\ Suc\ leI\ ambient\text{-}group\ j\text{-}lt\ subgroup\text{-}of\text{-}group\text{-}def$ )

interpretation cell: zassenhaus
   $G A (Suc\ i) B (Suc\ j) A\ i\ B\ j (\cdot) \mathbf{1}$ 
  by (simp add:  $zassenhaus.intro\ A.normal\text{-}step\ A\text{-}step\text{-}top\text{-}subgroup\ B.normal\text{-}step\ B\text{-}step\text{-}top\text{-}subgroup\ ambient\text{-}group\ i\text{-}lt\ j\text{-}lt$ )

lemma left-refinement-bottom-eq:
  left-refinement-term  $i\ j = cell.left\text{-}bottom$ 
  using  $cell.left\text{-}bottom\text{-}def\ left\text{-}refinement\text{-}term\text{-}def$  by auto

lemma left-refinement-top-eq:
  left-refinement-term  $i (Suc\ j) = cell.left\text{-}top$ 
  using  $cell.left\text{-}top\text{-}def\ left\text{-}refinement\text{-}term\text{-}def$  by auto

lemma right-refinement-bottom-eq:
  right-refinement-term  $j\ i = cell.right\text{-}bottom$ 
  using  $cell.right\text{-}bottom\text{-}def\ right\text{-}refinement\text{-}term\text{-}def$  by auto

lemma right-refinement-top-eq:
  right-refinement-term  $j (Suc\ i) = cell.right\text{-}top$ 
  using  $cell.right\text{-}top\text{-}def\ right\text{-}refinement\text{-}term\text{-}def$  by auto

lemma left-refinement-normal:
  normal-subgroup (left-refinement-term  $i\ j$ ) (left-refinement-term  $i (Suc\ j)$ )  $(\cdot) \mathbf{1}$ 
  using  $cell.left\text{-}bottom\text{-}normal\ left\text{-}refinement\text{-}bottom\text{-}eq\ left\text{-}refinement\text{-}top\text{-}eq$  by argo

lemma right-refinement-normal:
  normal-subgroup (right-refinement-term  $j\ i$ ) (right-refinement-term  $j (Suc\ i)$ )  $(\cdot) \mathbf{1}$ 

```

using *cell.right-bottom-normal right-refinement-bottom-eq right-refinement-top-eq* **by**
argo

Opposite sides of every refinement cell have the same native quotient-group isomorphism class. This is the factor-level form of Zassenhaus needed by a later Schreier refinement theorem; the statement contains no HOL-Algebra records and no representatives of quotient isomorphisms.

theorem *refinement-cell-factor-class-eq*:

normal-factor-class (*left-refinement-term* *i j*) (*left-refinement-term* *i* (*Suc j*)) (*·*) **1** =
normal-factor-class (*right-refinement-term* *j i*) (*right-refinement-term* *j* (*Suc i*)) (*·*)
1
proof –
have *normal-factor* (*left-refinement-term* *i j*) (*left-refinement-term* *i* (*Suc j*)) (*·*) **1**
 $\cong_G$
normal-factor (*right-refinement-term* *j i*) (*right-refinement-term* *j* (*Suc i*)) (*·*) **1**
unfolding *normal-factor-def left-refinement-bottom-eq left-refinement-top-eq*
right-refinement-bottom-eq right-refinement-top-eq
by (*rule cell.butterfly-lemma*)
then show *?thesis*
by (*simp add: left-refinement-normal normal-factor-class-eq-iff right-refinement-normal*)
qed
end

The factors of a refinement are read row by row. The following elementary lemmas turn a rectangular list of entries into finite sums of singleton multisets; this makes transposition of the refinement matrix explicit in the final proof.

lemma *mset-map-upt*:

mset (*List.map* *f* [*0..<n*]) = ($\sum i < n. \{\#f\ i\ \#\}$)
by (*induction n*) (*simp-all add: sum.lessThan-Suc*)

lemma *mset-concat-map-upt*:

mset (*concat* (*List.map* ($\lambda i. \text{List.map } (f\ i)\ [0..<n]$) [*0..<m*])) = ($\sum i < m. \sum j < n. \{\#f\ i\ j\ \#\}$)
proof (*induction m*)
case *0*
show *?case* **by** *simp*
next
case (*Suc m*)
have *row*: *mset* (*List.map* (*f m*) [*0..<n*]) = ($\sum j < n. \{\#f\ m\ j\ \#\}$)
by (*rule mset-map-upt*)
have *mset* (*concat* (*List.map* ($\lambda i. \text{List.map } (f\ i)\ [0..<n]$) [*0..<Suc m*])) =
mset (*concat* (*List.map* ($\lambda i. \text{List.map } (f\ i)\ [0..<n]$) [*0..<m*])) + *mset* (*List.map*
(*f m*) [*0..<n*])
by *simp*
also have ... = ($\sum i < \text{Suc } m. \sum j < n. \{\#f\ i\ j\ \#\}$)
using *Suc row* **by** *auto*
finally show *?case* .
qed

lemma *length-concat-map-upt*:

*length (concat (List.map (λi . List.map ($f i$) [0.. n]) [0.. m])) = $m * n$*
by (*induction m*) (*simp-all add: algebra-simps*)

At the endpoints of a refinement row, one factor in a subgroup product is either trivial or the whole carrier. For intermediate terms, intersecting the carrier with another ambient subgroup and multiplying by a normal subgroup again gives an ambient subgroup. These generic facts keep the row proofs below independent of quotient representations.

lemma *subgroup-product-with-trivial-left*:

assumes *K*: Subgroup *K* *G* composition unit
shows (*case-prod composition*) ‘($\{unit\} \times K$) = *K*

proof –

interpret *K*: Subgroup *K* *G* composition unit **by** *fact*

show *?thesis*

by *force*

qed

lemma *subgroup-product-with-carrier-left*:

assumes *K*: Subgroup *K* *H* composition unit
shows (*case-prod composition*) ‘($H \times K$) = *H*

proof –

obtain *K-submonoid*: Submonoid *K* *H* composition unit **and** *H-monoid*: Monoid *H* composition unit

using *assms* **by** (*meson Subgroup-def Submonoid.axioms(1)*)

have *unit-K*: *unit* $\in K$

by (*rule Submonoid.sub-unit-closed[OF K-submonoid]*)

show *?thesis*

proof (*intro equalityI subsetI*)

show $\bigwedge x. x \in (\lambda(x, y). \text{composition } x \ y) \text{ ‘ } (H \times K) \implies x \in H$

using *H-monoid K-submonoid Monoid.composition-closed Submonoid.sub* **by**

fastforce

show $\bigwedge x. x \in H \implies x \in (\lambda(x, y). \text{composition } x \ y) \text{ ‘ } (H \times K)$

using *H-monoid unit-K* **by** (*metis Monoid.right-unit SigmaI pair-imageI*)

qed

qed

lemma *intersection-normal-product-subgroup*:

assumes *G*: Group *G* composition unit

and *N*: normal-subgroup *K* *H* composition unit

and *H*: Subgroup *H* *G* composition unit

and *L*: Subgroup *L* *G* composition unit

shows Subgroup ((*case-prod composition*) ‘($(H \cap L) \times K$))
G composition unit

proof –

interpret *N*: normal-subgroup *K* *H* composition unit **by** *fact*

have *intersection-group*:

subgroup-of-group ($H \cap L$) *H* composition unit

```

    using G H L N.Group-axioms
    by (meson Group.subgroup-intersection inf-le1 subgroup-of-group-def subgroup-restrict)
  show ?thesis
    using H N intersection-group
    by (metis normal-subgroup-product.HK-def normal-subgroup-product.HK-subgroup
        normal-subgroup-product.intro subgroup-transitive)
qed

```

```

context normal-series-pair
begin

```

For every pair of valid step indices, the assumptions of *series-refinement-cell* follow directly from the two series. We construct the locale predicate without interpreting it, so clients can reuse the proved cell API without re-elaborating the Zassenhaus locale.

```

lemma series-refinement-cellI:
  assumes i:  $i < m$  and j:  $j < n$ 
  shows series-refinement-cell G ( $\cdot$ ) 1 A m B n i j
  by (simp add: i j normal-series-pair-axioms series-refinement-cell-axioms.intro
      series-refinement-cell-def)

```

16.2 Refinement rows

The left row at index i runs from $A\ i$ to $A\ (Suc\ i)$; the right row at index j runs from $B\ j$ to $B\ (Suc\ j)$. Thus concatenating rows in increasing outer index refines the original series. The normal-chain theorems verify every intermediate subgroup and adjacent normality condition.

```

lemma left-refinement-start:
  assumes i:  $i < m$ 
  shows left-refinement-term i 0 = A i
proof -
  interpret ASi: Subgroup A (Suc i) G ( $\cdot$ ) 1
    by (rule A.term-subgroup) (use i in simp)
  have intersection:  $A\ (Suc\ i) \cap B\ 0 = \{1\}$ 
    using B.bottom ASi.sub-unit-closed by auto
  show ?thesis
    unfolding left-refinement-term-def intersection
    using subgroup-product-with-trivial-left[OF A.term-subgroup] i by simp
qed

```

```

lemma left-refinement-end:
  assumes i:  $i < m$ 
  shows left-refinement-term i n = A (Suc i)
proof -
  interpret ASi: Subgroup A (Suc i) G ( $\cdot$ ) 1
    by (rule A.term-subgroup) (use i in simp)
  have intersection:  $A\ (Suc\ i) \cap B\ n = A\ (Suc\ i)$ 
    using B.top ASi.subset by auto

```

show *?thesis*
unfolding *left-refinement-term-def intersection*
by (rule *subgroup-product-with-carrier-left*[*OF A.normal-step-subgroup*[*OF i*]])
qed

lemma *right-refinement-start*:
assumes *j: j < n*
shows *right-refinement-term j 0 = B j*
proof –
interpret *BSj: Subgroup B (Suc j) G (·) 1*
by (rule *B.term-subgroup*) (use *j* **in** *simp*)
have *intersection: B (Suc j) ∩ A 0 = {1}*
using *A.bottom BSj.sub-unit-closed* **by** *auto*
show *?thesis*
unfolding *right-refinement-term-def intersection*
using *subgroup-product-with-trivial-left*[*OF B.term-subgroup*] *j* **by** *simp*
qed

lemma *right-refinement-end*:
assumes *j: j < n*
shows *right-refinement-term j m = B (Suc j)*
proof –
interpret *BSj: Subgroup B (Suc j) G (·) 1*
by (rule *B.term-subgroup*) (use *j* **in** *simp*)
have *intersection: B (Suc j) ∩ A m = B (Suc j)*
using *A.top BSj.subset* **by** *auto*
show *?thesis*
unfolding *right-refinement-term-def intersection*
by (rule *subgroup-product-with-carrier-left*[*OF B.normal-step-subgroup*[*OF j*]])
qed

lemma *left-refinement-step*:
assumes *i: i < m and j: j < n*
shows *normal-subgroup (left-refinement-term i j) (left-refinement-term i (Suc j)) (·) 1*
using *i j series-refinement-cellI* **by** (meson *series-refinement-cell.left-refinement-normal*)

lemma *left-refinement-term-subgroup*:
assumes *i: i < m and j: j ≤ n*
shows *Subgroup (left-refinement-term i j) G (·) 1*
unfolding *left-refinement-term-def*
by (*simp add: A.G.Group-axioms A.normal-step A.term-subgroup B.term-subgroup*
Suc-leI i
intersection-normal-product-subgroup j)

lemma *right-refinement-step*:
assumes *i: i < m and j: j < n*
shows *normal-subgroup (right-refinement-term j i) (right-refinement-term j (Suc i)) (·) 1*
using *i j series-refinement-cellI* **by** (meson *series-refinement-cell.right-refinement-normal*)

```

lemma right-refinement-term-subgroup:
  assumes  $i: i \leq m$  and  $j: j < n$ 
  shows Subgroup (right-refinement-term  $j$   $i$ )  $G$   $(\cdot)$  1
  unfolding right-refinement-term-def
  by (simp add:  $A.G.Group\text{-}axioms$   $A.term\text{-}subgroup$   $B.normal\text{-}step$   $B.term\text{-}subgroup$ 
Suc-leI  $i$ 
intersection-normal-product-subgroup  $j$ )

lemma left-refinement-row:
  assumes  $i: i < m$ 
  shows normal-chain  $G$   $(\cdot)$  1 (left-refinement-term  $i$ )  $n$ 
proof (intro normal-chain.intro  $A.G.Group\text{-}axioms$ )
  show normal-chain-axioms  $G$   $(\cdot)$  1 (left-refinement-term  $i$ )  $n$ 
  by (simp add:  $i$  left-refinement-step left-refinement-term-subgroup normal-chain-axioms-def)
qed

lemma right-refinement-row:
  assumes  $j: j < n$ 
  shows normal-chain  $G$   $(\cdot)$  1 (right-refinement-term  $j$ )  $m$ 
proof (intro normal-chain.intro  $A.G.Group\text{-}axioms$ )
  show normal-chain-axioms  $G$   $(\cdot)$  1 (right-refinement-term  $j$ )  $m$ 
  by (simp add:  $j$  normal-chain-axioms-def right-refinement-step right-refinement-term-subgroup)
qed

```

16.3 Factor-class refinements

The two lists below flatten the left and right refinement rows in row-major order. Each entry is the native isomorphism class of the corresponding normal-chain factor; repeated or trivial factors are retained.

```

definition left-refinement-factor-class ::  $nat \Rightarrow nat \Rightarrow 'a$  set monoid-iso-class
where
  left-refinement-factor-class  $i$   $j \equiv$ 
    normal-factor-class (left-refinement-term  $i$   $j$ ) (left-refinement-term  $i$  (Suc  $j$ ))  $(\cdot)$ 
1

```

```

definition right-refinement-factor-class ::  $nat \Rightarrow nat \Rightarrow 'a$  set monoid-iso-class
where
  right-refinement-factor-class  $j$   $i \equiv$ 
    normal-factor-class (right-refinement-term  $j$   $i$ ) (right-refinement-term  $j$  (Suc  $i$ ))
     $(\cdot)$  1

```

```

definition left-refinement-factor-classes ::  $'a$  set monoid-iso-class list
where
  left-refinement-factor-classes  $\equiv$ 
    concat (List.map  $(\lambda i. List.map$  (left-refinement-factor-class  $i$ )  $[0..<n]$ )  $[0..<m]$ )

```

```

definition right-refinement-factor-classes ::  $'a$  set monoid-iso-class list
where

```



```

right-refinement-factor-classes ≡
  concat (List.map
    (λj. List.map (right-refinement-factor-class j) [0..<m]) [0..<n])

```

lemma *length-left-refinement-factor-classes*:
 $\text{length left-refinement-factor-classes} = m * n$
unfolding *left-refinement-factor-classes-def* **by** (rule *length-concat-map-upt*)

lemma *length-right-refinement-factor-classes*:
 $\text{length right-refinement-factor-classes} = n * m$
unfolding *right-refinement-factor-classes-def* **by** (rule *length-concat-map-upt*)

lemma *refinement-factor-class-eq*:
assumes $i: i < m$ **and** $j: j < n$
shows $\text{left-refinement-factor-class } i \ j = \text{right-refinement-factor-class } j \ i$
proof –
have $C: \text{series-refinement-cell } G \ (\cdot) \ \mathbf{1} \ A \ m \ B \ n \ i \ j$
by (rule *series-refinement-cell[OF i j]*)
show ?thesis
unfolding *left-refinement-factor-class-def right-refinement-factor-class-def*
by (rule *series-refinement-cell.refinement-cell-factor-class-eq[OF C]*)
qed

The two row-major factor lists need not have the same order. Their multisets agree: transposing the refinement matrix sends the factor at position (i, j) to the isomorphic factor at (j, i) . This is the native, factor-class form of the Schreier refinement theorem.

theorem *schreier-refinement*:
 $\text{mset left-refinement-factor-classes} = \text{mset right-refinement-factor-classes}$
proof –
have *transpose*:
 $(\sum i < m. \sum j < n. \{\#\text{left-refinement-factor-class } i \ j\# \}) =$
 $(\sum j < n. \sum i < m. \{\#\text{right-refinement-factor-class } j \ i\# \})$
using *refinement-factor-class-eq sum.swap* **by** *auto*
show ?thesis
unfolding *left-refinement-factor-classes-def right-refinement-factor-classes-def mset-concat-map-upt*
using *transpose* **by** *blast*
qed
end

16.4 Reduced Schreier refinements

A finite normal chain may repeat a term. Its corresponding quotient is the one-element group and contributes no genuine factor. The operation below removes precisely the canonical trivial isomorphism class while retaining multiplicity and forgetting order.

definition *nontrivial-factor-multiset* ::

'a monoid-iso-class list $\Rightarrow$ *'a monoid-iso-class multiset*
where
nontrivial-factor-multiset factors $\equiv$ *filter-mset* ($\lambda C. C \neq$ *trivial-monoid-iso-class*)
(mset factors)

lemma *nontrivial-factor-multiset-cong*:
assumes *mset factors = mset factors'*
shows *nontrivial-factor-multiset factors = nontrivial-factor-multiset factors'*
using *assms* **unfolding** *nontrivial-factor-multiset-def* **by** *simp*

lemma *nontrivial-factor-multiset-append* [*simp*]:
nontrivial-factor-multiset (factors @ factors') =
nontrivial-factor-multiset factors + nontrivial-factor-multiset factors'
unfolding *nontrivial-factor-multiset-def* **by** *simp*

lemma *nontrivial-factor-multiset-Nil* [*simp*]:
nontrivial-factor-multiset [] = {#}
unfolding *nontrivial-factor-multiset-def* **by** *simp*

lemma *nontrivial-factor-multiset-singleton* [*simp*]:
nontrivial-factor-multiset [C] =
(if C = trivial-monoid-iso-class then {#} else {#C#})
unfolding *nontrivial-factor-multiset-def* **by** *simp*

lemma *nontrivial-factor-multiset-concat-map-upt*:
*nontrivial-factor-multiset (concat (List.map f [0..*m*]))* =
($\sum i < m. \text{nontrivial-factor-multiset } (f\ i)$)
by (*induction m*) (*simp-all add: sum.lessThan-Suc*)

context *normal-series-pair*
begin

The trivial-class test has a concrete chain interpretation: it detects exactly an adjacent repetition. These equivalences are the bridge used by the later Jordan–Hölder argument when a refinement row lies inside a simple factor.

lemma *left-refinement-factor-class-eq-trivial-iff*:
assumes *i: i < m and j: j < n*
shows *left-refinement-factor-class i j = trivial-monoid-iso-class*
 $\longleftrightarrow$ *left-refinement-term i j = left-refinement-term i (Suc j)*
by (*simp add: i j left-refinement-factor-class-def left-refinement-step normal-factor-class-eq-trivial-iff*)

lemma *right-refinement-factor-class-eq-trivial-iff*:
assumes *i: i < m and j: j < n*
shows *right-refinement-factor-class j i = trivial-monoid-iso-class*
 $\longleftrightarrow$ *right-refinement-term j i = right-refinement-term j (Suc i)*
by (*simp add: i j normal-factor-class-eq-trivial-iff right-refinement-factor-class-def right-refinement-step*)

definition *left-reduced-refinement-factor-multiset* :: *'a set monoid-iso-class multiset*

where *left-reduced-refinement-factor-multiset* $\equiv$ *nontrivial-factor-multiset* *left-refinement-factor-classes*

definition *right-reduced-refinement-factor-multiset* $::$ 'a set monoid-iso-class multiset
where *right-reduced-refinement-factor-multiset* $\equiv$ *nontrivial-factor-multiset* *right-refinement-factor-classes*

Removing repetition factors from equivalent Schreier refinements preserves equality. Unlike *mset left-refinement-factor-classes* = *mset right-refinement-factor-classes*, this form is ready to compare directly with the factor multiset of a composition series.

theorem *reduced-schreier-refinement*:

left-reduced-refinement-factor-multiset = *right-reduced-refinement-factor-multiset*

by (*simp add: left-reduced-refinement-factor-multiset-def nontrivial-factor-multiset-def right-reduced-refinement-factor-multiset-def schreier-refinement*)

end

end

17 Composition series, etc.

theory *Simple-Factor-Chain*

imports *Schreier-Refinement Normal-Series*

begin

17.1 Simple groups

A simple group is a non-trivial group whose only normal subgroups are the trivial subgroup and the whole group.

locale *Simple-Group* = *Group* *G* ($\cdot$) **1**

for *G* **and** *composition* (**infixl** $\langle \cdot \rangle$ 70) **and** *unit* ($\langle \mathbf{1} \rangle$) +

assumes *nontrivial*: $G \neq \{\mathbf{1}\}$

and *normal-subgroup-eq-trivial-or-top*:

$\bigwedge K. \text{normal-subgroup } K \ G \ (\cdot) \ \mathbf{1} \implies K = \{\mathbf{1}\} \vee K = G$

begin

Simple groups have no proper non-trivial normal subgroups.

lemma *proper-normal-subgroup-is-trivial*:

assumes *K*: *normal-subgroup* *K* *G* ($\cdot$) **1** **and** *proper*: $K \neq G$

shows $K = \{\mathbf{1}\}$

using *normal-subgroup-eq-trivial-or-top*[*OF K*] *proper* **by** *blast*

lemma *nontrivial-normal-subgroup-is-top*:

assumes *K*: *normal-subgroup* *K* *G* ($\cdot$) **1** **and** *K-nontrivial*: $K \neq \{\mathbf{1}\}$

shows $K = G$

using *normal-subgroup-eq-trivial-or-top*[*OF K*] *K-nontrivial* **by** *blast*

end

A simple factor group is the quotient associated with a normal subgroup whose factor group is simple. Naming this local construction keeps the quotient operations attached to the existing *normal-subgroup* interpretation.

```

locale simple-factor-group =
  N: normal-subgroup K H (·) 1 +
  Q: Simple-Group N.Factor-Group N.quotient-composition N.Class 1
  for K and H and composition (infixl <·> 70) and unit (<1>)
begin

lemma factor-group-is-simple:
  Simple-Group N.Factor-Group N.quotient-composition (N.Class 1)
  by (rule Q.Simple-Group-axioms)

lemma factor-group-nontrivial:
  N.Factor-Group ≠ {N.Class 1}
  by (rule Q.nontrivial)

lemma proper:
  K ≠ H
  using N.Factor-Group-eq-singleton-iff Q.nontrivial by blast

end

A composition series is a finite normal series whose successive factor groups are
simple. The indexed representation is inherited from normal-series.

locale composition-series = normal-series +
  assumes factor-simple:  $\bigwedge i. i < n \implies \text{simple-factor-group } (H\ i) (H\ (\text{Suc } i)) (\cdot) 1$ 

begin

lemma factor-simple-step:
  assumes i:  $i < n$ 
  shows simple-factor-group (H i) (H (Suc i)) (·) 1
  by (rule factor-simple[OF i])

lemma series-factor-simple:
  assumes i:  $i < n$ 
  shows Simple-Group (fst (series-factor i))
    (fst (snd (series-factor i))) (snd (snd (series-factor i)))
  using factor-simple-step i series-factor-def simple-factor-group-def by fastforce

lemma series-factors-nth-simple:
  assumes i:  $i < n$ 
  shows Simple-Group (fst (series-factors ! i))
    (fst (snd (series-factors ! i))) (snd (snd (series-factors ! i)))
  using series-factors-nth[OF i] series-factor-simple[OF i] by simp

lemma composition-step-strict:
  assumes i:  $i < n$ 

```

```

shows  $H\ i \neq H\ (Suc\ i)$ 
using factor-simple-step i by (metis simple-factor-group.proper)

```

Every prefix of a composition series is again a composition series.

```

lemma prefix-composition-series:
  assumes  $k: k \leq n$ 
  shows composition-series ( $H\ k$ )  $(\cdot)\ 1\ H\ k$ 
proof (intro composition-series.intro composition-series-axioms.intro prefix-normal-series[OF
 $k]$ )
qed (use factor-simple-step k in auto)

```

A composition series has positive length exactly when its group is nontrivial.

```

lemma zero-length-iff-trivial:  $n = 0 \longleftrightarrow G = \{1\}$ 
proof
  show  $n = 0 \implies G = \{1\}$ 
  by (rule zero-length-implies-trivial)
next
  assume G-trivial:  $G = \{1\}$ 
  show  $n = 0$ 
  proof (rule ccontr)
    assume n-nonzero:  $n \neq 0$ 
    interpret H1: Subgroup  $H\ (Suc\ 0)\ G\ (\cdot)\ 1$ 
    using term-subgroup n-nonzero by auto
    have  $H\ (Suc\ 0) \subseteq \{1\}$ 
    using H1.subset G-trivial by simp
    then show False
    using bottom composition-step-strict n-nonzero by fastforce
  qed
qed
end

```

17.2 Normal chains inside a simple factor

The subgroup correspondence detects the bottom and top subgroups of a quotient without choosing representatives. These two characterisations are the order-theoretic part of the correspondence needed below.

```

context normal-subgroup-in-subgroup
begin

```

```

lemma Fract-H-K-eq-bottom-iff:
   $Fract-H-K = \{K.Class\ 1\} \longleftrightarrow H = K$ 
proof
  assume quotient:  $Fract-H-K = \{K.Class\ 1\}$ 
  show  $H = K$ 
  proof (rule equalityI)
    show  $H \subseteq K$ 
    proof

```

```

    fix h
    assume h: h ∈ H
    have K.Class h ∈ Fract-H-K by (rule Class-H-closed[OF h])
    then have class-eq: K.Class h = K.Class 1
      using quotient by simp
    have h ∈ K.Class h by (rule K.Class-self[OF H.sub[OF h]])
    then show h ∈ K
      using class-eq K.Class-unit-normal-subgroup by simp
  qed
  show K ⊆ H by (rule K-contained)
qed
next
assume HK: H = K
show Fract-H-K = {K.Class 1}
proof (rule equalityI)
  show Fract-H-K ⊆ {K.Class 1}
  proof
    fix A
    assume A ∈ Fract-H-K
    then obtain h where h: h ∈ H and A: A = K.Class h
      by (rule Fract-H-K-memE)
    have hK: h ∈ K using h HK by simp
    have class-eq: K.Class h = K.Class 1
    proof (rule K.Class-eq)
      show (h, 1) ∈ K.Congruence
      proof (rule K.CongruenceI)
        show h = 1 · h by (rule K.left-unit[OF K.sub[OF hK], symmetric])
        show h ∈ G by (rule K.sub[OF hK])
        show 1 ∈ G by (rule K.unit-closed)
        show h ∈ K by fact
      qed
    qed
    show A ∈ {K.Class 1} using A class-eq by simp
  qed
  show {K.Class 1} ⊆ Fract-H-K
  by (simp add: Fract-H-K-unit-closed)
qed
qed

lemma Fract-H-K-eq-top-iff:
  Fract-H-K = K.Factor-Group ⟷ H = G
proof
  assume quotient: Fract-H-K = K.Factor-Group
  show H = G
  proof (rule equalityI)
    show H ⊆ G by (rule H.subset)
    show G ⊆ H
    proof
      fix g
      assume g: g ∈ G

```

```

    have  $K.Class\ g \in K.Factor-Group$ 
      by (rule  $K.Class-in-Partition[OF\ g]$ )
    then have  $K.Class\ g \in Fract-H-K$  using quotient by simp
    then have  $K.Class\ g \subseteq H$  by (rule  $Fract-H-K-into-subset-H$ )
    moreover have  $g \in K.Class\ g$  by (rule  $K.Class-self[OF\ g]$ )
    ultimately show  $g \in H$  by blast
  qed
qed
next
  assume  $HG: H = G$ 
  show  $Fract-H-K = K.Factor-Group$ 
  proof (rule equalityI)
    show  $Fract-H-K \subseteq K.Factor-Group$ 
      by (rule  $Fract-H-K-subset$ )
    show  $K.Factor-Group \subseteq Fract-H-K$ 
    proof
      fix  $A$ 
      assume  $A \in K.Factor-Group$ 
      then obtain  $g$  where  $g: g \in G$  and  $A: A = K.Class\ g$ 
        using  $K.representant-exists$  by blast
      have  $g \in H$  using  $g\ HG$  by simp
      then show  $A \in Fract-H-K$  unfolding  $A$  by (rule  $Class-H-closed$ )
    qed
  qed
qed
end

```

An intermediate subgroup that is normal in the numerator of a simple factor is necessarily one of the endpoints. The proof passes to the quotient using the existing correspondence theorem and then reflects the two possible quotient subgroups back to the original carrier sets.

```

lemma simple-factor-intermediate-normal:
  assumes factor: simple-factor-group  $K\ H$  composition unit
    and contained:  $K \subseteq L$ 
    and normal: normal-subgroup  $L\ H$  composition unit
  shows  $L = K \vee L = H$ 
proof -
  interpret factor: simple-factor-group  $K\ H$  composition unit by fact
  interpret normal: normal-subgroup  $L\ H$  composition unit by fact
  have Subgroup: subgroup-of-group  $L\ H$  composition unit
    by (rule subgroup-of-groupI[OF
      normal.Subgroup-axioms factor.N.Group-axioms])
  interpret corr: normal-subgroup-in-subgroup  $K\ L\ H$  composition unit
  proof (rule normal-subgroup-in-subgroup.intro)
    show normal-subgroup  $K\ H$  composition unit
      by (rule factor.N.normal-subgroup-axioms)
    show subgroup-of-group  $L\ H$  composition unit by (rule Subgroup)
    show normal-subgroup-in-subgroup-axioms  $K\ L$ 

```

```

    by (rule normal-subgroup-in-subgroup-axioms.intro[OF contained])
  qed
  have quotient-normal:
    normal-subgroup corr.Fract-H-K factor.N.Factor-Group
    factor.N.quotient-composition (factor.N.Class unit)
  by (rule iffD1[OF corr.normal-iff normal])
  have corr.Fract-H-K = {factor.N.Class unit} ∨
    corr.Fract-H-K = factor.N.Factor-Group
  by (rule factor.Q.normal-subgroup-eq-trivial-or-top[OF quotient-normal])
  then show ?thesis
  using corr.Fract-H-K-eq-bottom-iff corr.Fract-H-K-eq-top-iff by blast
qed

```

definition *normal-chain-factor-classes* ::

```

(nat ⇒ 'a set) ⇒ nat ⇒
('a ⇒ 'a ⇒ 'a) ⇒ 'a ⇒
'a set monoid-iso-class list

```

where

```

normal-chain-factor-classes C n composition unit =
  List.map
    (λi. normal-factor-class (C i) (C (Suc i)) composition unit)
    [0.. $n$ ]

```

definition *reduced-normal-chain-factor-multiset* ::

```

(nat ⇒ 'a set) ⇒ nat ⇒
('a ⇒ 'a ⇒ 'a) ⇒ 'a ⇒
'a set monoid-iso-class multiset

```

where

```

reduced-normal-chain-factor-multiset C n composition unit =
  nontrivial-factor-multiset
    (normal-chain-factor-classes C n composition unit)

```

lemma *length-normal-chain-factor-classes*:

```

length (normal-chain-factor-classes C n composition unit) = n
unfolding normal-chain-factor-classes-def by simp

```

A finite normal chain between the endpoints of a simple factor has exactly one nontrivial factor. Repetitions may occur on either side of that strict step, but filtering the canonical trivial class leaves the original factor class with multiplicity one.

theorem *simple-factor-chain-reduction*:

```

assumes chain: normal-chain G composition unit C n
  and factor: simple-factor-group K H composition unit
  and bottom: C 0 = K
  and top: C n = H
shows reduced-normal-chain-factor-multiset C n composition unit =
  {#normal-factor-class K H composition unit#}
using chain factor bottom top
proof (induction n arbitrary: C)

```



```

case 0
then show ?case by (metis simple-factor-group.proper)
next
case (Suc n)
interpret C: normal-chain G composition unit C Suc n by fact
interpret factor: simple-factor-group K H composition unit by fact

have last-normal: normal-subgroup (C n) H composition unit
  using C.normal-step[of n] Suc.prem(4) by simp
have K-subset-last:  $K \subseteq C\ n$ 
  using C.chain-mono[of 0 n] Suc.prem(3) by simp
have last-cases:  $C\ n = K \vee C\ n = H$ 
  by (rule simple-factor-intermediate-normal[
    OF Suc.prem(2) K-subset-last last-normal])

have factors-suc:
  normal-chain-factor-classes C (Suc n) composition unit =
    normal-chain-factor-classes C n composition unit @
    [normal-factor-class (C n) (C (Suc n)) composition unit]
unfolding normal-chain-factor-classes-def by simp

show ?case
proof (rule disjE[OF last-cases])
  assume 1:  $C\ n = K$ 
  have earlier-terms:  $C\ i = K$  if  $i \leq n$  for  $i$ 
  proof (rule equalityI)
    show  $C\ i \subseteq K$ 
    using C.chain-mono[of i n] that 1 by simp
    show  $K \subseteq C\ i$ 
    using C.chain-mono[of 0 i] that Suc.prem(3) by simp
  qed
  have earlier-trivial:
    normal-factor-class (C i) (C (Suc i)) composition unit =
      trivial-monoid-iso-class if  $i < n$  for  $i$ 
  proof -
    have step: normal-subgroup (C i) (C (Suc i)) composition unit
      by (rule C.normal-step) (use that in simp)
    show ?thesis
    using earlier-terms[of i] earlier-terms[of Suc i] that
      normal-factor-class-eq-trivial-iff[OF step]
    by simp
  qed
  have prefix-classes:
    normal-chain-factor-classes C n composition unit =
      List.map ( $\lambda$ -. trivial-monoid-iso-class) [0.. $n$ ]
    unfolding normal-chain-factor-classes-def
    by (rule map-cong) (auto intro: earlier-trivial)
  have reduced-prefix:
    nontrivial-factor-multiset
      (normal-chain-factor-classes C n composition unit) = {#}

```

```

unfolding prefix-classes nontrivial-factor-multiset-def
by (induction n) simp-all
have last-class:
  normal-factor-class (C n) (C (Suc n)) composition unit =
    normal-factor-class K H composition unit
using 1 Suc.prem(4) by simp
show ?thesis
unfolding reduced-normal-chain-factor-multiset-def factors-suc
using reduced-prefix last-class factor.proper
  normal-factor-class-eq-trivial-iff[OF factor.N.normal-subgroup-axioms]
by simp
next
assume 2: C n = H
have prefix: normal-chain G composition unit C n
by (rule C.prefix-normal-chain) simp
have reduced-prefix:
  reduced-normal-chain-factor-multiset C n composition unit =
    {#normal-factor-class K H composition unit#}
by (rule Suc.IH[OF prefix Suc.prem(2) Suc.prem(3) 2])
have last-trivial:
  normal-factor-class (C n) (C (Suc n)) composition unit =
    trivial-monoid-iso-class
using 2 Suc.prem(4)
  normal-factor-class-eq-trivial-iff[OF C.normal-step[of n]] by simp
show ?thesis
using reduced-prefix last-trivial
unfolding reduced-normal-chain-factor-multiset-def factors-suc
by simp
qed
qed

end
theory Jordan-Hoelder-Uniqueness
imports Simple-Factor-Chain
begin

```

18 The Jordan–Hölder theorem

Two composition series of the same set-based group form a normal-series pair, while simplicity of their factors supplies the additional rigidity needed to collapse each Schreier refinement row. Keeping these assumptions in one locale makes the final comparison symmetric and explicit.

```

locale composition-series-pair =
  normal-series-pair G (·) 1 A m B n +
  A-comp: composition-series G (·) 1 A m +
  B-comp: composition-series G (·) 1 B n
for G and composition (infixl <·> 70) and unit (<1>)
and A :: nat ⇒ 'a set and m

```

and $B :: \text{nat} \Rightarrow 'a \text{ set}$ and n
begin

Each refinement row spans one simple composition factor. The generic simple-factor chain theorem therefore leaves exactly that factor after deleting repetitions.

lemma *left-refinement-row-reduction*:

assumes $i: i < m$

shows $\text{nontrivial-factor-multiset } (\text{List.map } (\text{left-refinement-factor-class } i) [0..<n])$

=

$\{\#A.\text{series-factor-class } i\# \}$

proof –

have $\text{reduced-normal-chain-factor-multiset } (\text{left-refinement-term } i) \ n \ (\cdot) \ 1 =$

$\{\# \text{normal-factor-class } (A \ i) \ (A \ (\text{Suc } i)) \ (\cdot) \ 1\# \}$

using $A.\text{comp.factor-simple-step } i \ \text{left-refinement-end} \ \text{left-refinement-row} \ \text{left-refinement-start}$

by $(\text{metis } \text{simple-factor-chain-reduction})$

then show $?thesis$

unfolding $\text{reduced-normal-chain-factor-multiset-def}$

$\text{normal-chain-factor-classes-def} \ \text{left-refinement-factor-class-def}$

$A.\text{series-factor-class-eq-normal-factor-class}$

by simp

qed

lemma *right-refinement-row-reduction*:

assumes $j: j < n$

shows $\text{nontrivial-factor-multiset } (\text{List.map } (\text{right-refinement-factor-class } j) [0..<m])$

=

$\{\#B.\text{series-factor-class } j\# \}$

proof –

have $\text{reduced-normal-chain-factor-multiset } (\text{right-refinement-term } j) \ m \ (\cdot) \ 1 =$

$\{\# \text{normal-factor-class } (B \ j) \ (B \ (\text{Suc } j)) \ (\cdot) \ 1\# \}$

using $B.\text{comp.factor-simple-step } j \ \text{right-refinement-end} \ \text{right-refinement-row} \ \text{right-refinement-start}$

by $(\text{metis } \text{simple-factor-chain-reduction})$

then show $?thesis$

unfolding $\text{reduced-normal-chain-factor-multiset-def}$

$\text{normal-chain-factor-classes-def} \ \text{right-refinement-factor-class-def}$

$B.\text{series-factor-class-eq-normal-factor-class}$

by simp

qed

lemma *left-reduced-refinement-eq-series-factor-multiset*:

$\text{left-reduced-refinement-factor-multiset} = A.\text{series-factor-multiset}$

proof –

have $\text{left-reduced-refinement-factor-multiset} =$

$(\sum_{i < m. \text{nontrivial-factor-multiset } (\text{List.map } (\text{left-refinement-factor-class } i) [0..<n])})$

by $(\text{simp add: } \text{left-reduced-refinement-factor-multiset-def} \ \text{left-refinement-factor-classes-def} \ \text{nontrivial-factor-multiset-concat-map-upt})$

also have $\dots = (\sum_{i < m. \{\#A.\text{series-factor-class } i\# \})$

```

    using left-refinement-row-reduction by auto
    also have ... = A.series-factor-multiset
    using A.series-factor-classes-def A.series-factor-multiset-def by (metis mset-map-upt)
    finally show ?thesis .
qed

lemma right-reduced-refinement-eq-series-factor-multiset:
  right-reduced-refinement-factor-multiset = B.series-factor-multiset
proof -
  have right-reduced-refinement-factor-multiset =
    ( $\sum j < n. \text{nontrivial-factor-multiset } (\text{List.map } (\text{right-refinement-factor-class } j) [0..<m])$ )
  by (simp add: nontrivial-factor-multiset-concat-map-upt right-reduced-refinement-factor-multiset-def
    right-refinement-factor-classes-def)
  also have ... = ( $\sum j < n. \{\#B.\text{series-factor-class } j\}$ )
  using right-refinement-row-reduction by auto
  also have ... = B.series-factor-multiset
  using B.series-factor-classes-def B.series-factor-multiset-def by (metis mset-map-upt)
  finally show ?thesis .
qed

```

The factor-class multiset of a composition series is independent of the series. Equality is obtained by identifying each original series with its reduced Schreier refinement and applying the reduced refinement theorem.

```

theorem jordan-hoelder:
  A.series-factor-multiset = B.series-factor-multiset
  using left-reduced-refinement-eq-series-factor-multiset reduced-schreier-refinement
  right-reduced-refinement-eq-series-factor-multiset by presburger

```

```

corollary jordan-hoelder-length: m = n
  using A.size-series-factor-multiset B.size-series-factor-multiset jordan-hoelder by
  argo

```

end

end

theory Maximal-Normal-Subgroup

imports Simple-Factor-Chain

begin

19 Maximal normal subgroups

A maximal normal subgroup is proper and admits no proper normal enlargement. Stating maximality with non-strict inclusion makes the locale convenient to consume in finite-choice and composition-series arguments.

```

locale maximal-normal-subgroup =
  N: normal-subgroup K G (·) 1
  for K and G and composition (infixl <·> 70) and unit (<1>) +

```

```

assumes proper:  $K \neq G$ 
and maximal:
   $\bigwedge L. \text{normal-subgroup } L \ G \ (\cdot) \ 1 \implies K \subseteq L \implies$ 
     $L = K \vee L = G$ 
begin

```

```

lemma strict:
   $K \subset G$ 
  using N.subset proper by blast

```

```

end

```

19.1 The simple-quotient characterisation

The quotient of a group by a proper normal subgroup is simple exactly when the subgroup is maximal normal. The forward correspondence uses the quotient endpoint characterisations developed for simple factors; the reverse direction is precisely normal-subgroup rigidity inside a simple factor.

```

theorem maximal-normal-subgroup-iff-simple-factor-group:
  maximal-normal-subgroup  $K \ G \ \text{composition unit} \longleftrightarrow$ 
    simple-factor-group  $K \ G \ \text{composition unit}$ 

```

```

proof

```

```

  assume maximal: maximal-normal-subgroup  $K \ G \ \text{composition unit}$ 
  interpret maximal: maximal-normal-subgroup  $K \ G \ \text{composition unit}$  by fact

```

```

  have quotient-nontrivial:
    maximal.N.Factor-Group  $\neq \{ \text{maximal.N.Class unit} \}$ 

```

```

  proof

```

```

    assume quotient-trivial:
      maximal.N.Factor-Group  $= \{ \text{maximal.N.Class unit} \}$ 

```

```

    have subgroup-top:

```

```

      subgroup-of-group  $G \ G \ \text{composition unit}$ 
      by (rule subgroup-of-groupI[OF maximal.N.group-self-subgroup
        maximal.N.Group-axioms])

```

```

    interpret corr: normal-subgroup-in-subgroup  $K \ G \ G \ \text{composition unit}$ 

```

```

    proof (rule normal-subgroup-in-subgroup.intro)

```

```

      show normal-subgroup  $K \ G \ \text{composition unit}$ 
      by (rule maximal.N.normal-subgroup-axioms)

```

```

      show subgroup-of-group  $G \ G \ \text{composition unit}$  by fact

```

```

      show normal-subgroup-in-subgroup-axioms  $K \ G$ 

```

```

      by (rule normal-subgroup-in-subgroup-axioms.intro[OF maximal.N.subset])

```

```

    qed

```

```

    have corr.Fract-H-K  $= \{ \text{maximal.N.Class unit} \}$ 

```

```

      using corr.Fract-H-K-eq-top-iff quotient-trivial by simp

```

```

    then have  $G = K$ 

```

```

      using corr.Fract-H-K-eq-bottom-iff by blast

```

```

    then show False using maximal.proper by blast

```

```

  qed

```

```

have quotient-normal-cases:
  Q = {maximal.N.Class unit} ∨ Q = maximal.N.Factor-Group
  if Q-normal:
    normal-subgroup Q maximal.N.Factor-Group
    maximal.N.quotient-composition (maximal.N.Class unit)
  for Q
proof –
interpret Q: normal-subgroup Q maximal.N.Factor-Group
  maximal.N.quotient-composition maximal.N.Class unit
  by (rule Q-normal)
define L where
  L = {g ∈ G. maximal.N.Class g ∈ Q}

have L-subgroup: Subgroup L G composition unit
proof (rule maximal.N.subgroupI)
  show L ⊆ G unfolding L-def by blast
next
  show unit ∈ L
    unfolding L-def using Q.sub-unit-closed maximal.N.unit-closed by simp
next
  fix g h
  assume g: g ∈ L and h: h ∈ L
  have classes:
    maximal.N.quotient-composition (maximal.N.Class g)
    (maximal.N.Class h) ∈ Q
  using g h unfolding L-def by (intro Q.sub-composition-closed) auto
  show composition g h ∈ L
    using g h classes unfolding L-def
    by (simp add: maximal.N.Class-commutes-with-composition)
next
  fix g
  assume g ∈ L
  then show maximal.N.invertible g
    unfolding L-def by auto
next
  fix g
  assume g: g ∈ L
  have g-G: g ∈ G and class-g: maximal.N.Class g ∈ Q
  using g unfolding L-def by auto
  have maximal.N.quotient.inverse (maximal.N.Class g) ∈ Q
  by (rule Q.submonoid-inverse-closed[OF
    Q.sub.invertible[OF class-g] class-g])
  then show maximal.N.inverse g ∈ L
    using g-G unfolding L-def
    by (simp add: maximal.N.Class-commutes-with-inverse)
qed
have L-group: subgroup-of-group L G composition unit
  by (rule subgroup-of-groupI[OF L-subgroup maximal.N.Group-axioms])
interpret L: subgroup-of-group L G composition unit by (rule L-group)
have L-normal: normal-subgroup L G composition unit

```

```

proof
  fix g l
  assume g: g ∈ G and l: l ∈ L
  have class-g: maximal.N.Class g ∈ maximal.N.Factor-Group
    by (rule maximal.N.Class-in-Partition[OF g])
  have inverse-g: maximal.N.inverse g ∈ G
    by (rule maximal.N.invertible-inverse-closed[OF
      maximal.N.invertible[OF g] g])
  have l-G: l ∈ G
    using l unfolding L-def by simp
  have class-l: maximal.N.Class l ∈ Q
    using l unfolding L-def by simp
  have quotient-conjugate:
    maximal.N.quotient-composition
      (maximal.N.quotient-composition
        (maximal.N.quotient.inverse (maximal.N.Class g))
        (maximal.N.Class l))
      (maximal.N.Class g) ∈ Q
    by (rule Q.normal[OF class-g class-l])
  show composition (composition (maximal.N.inverse g) l) g ∈ L
    using g inverse-g l-G quotient-conjugate
    unfolding L-def
    by (simp add: maximal.N.Class-commutes-with-composition
      maximal.N.Class-commutes-with-inverse)
qed
have K-subset-L: K ⊆ L
proof
  fix k
  assume k: k ∈ K
  have maximal.N.Class k = maximal.N.Class unit
  proof (rule maximal.N.Class-eq)
    show (k, unit) ∈ maximal.N.Congruence
    proof (rule maximal.N.CongruenceI)
      show k = composition unit k
        by (rule maximal.N.left-unit[OF maximal.N.sub[OF k], symmetric])
      show k ∈ G by (rule maximal.N.sub[OF k])
      show unit ∈ G by (rule maximal.N.unit-closed)
      show k ∈ K by fact
    qed
  qed
  then show k ∈ L
    unfolding L-def using k maximal.N.sub Q.sub-unit-closed by simp
qed

interpret corr: normal-subgroup-in-subgroup K L G composition unit
proof (rule normal-subgroup-in-subgroup.intro)
  show normal-subgroup K G composition unit
    by (rule maximal.N.normal-subgroup-axioms)
  show subgroup-of-group L G composition unit by fact
  show normal-subgroup-in-subgroup-axioms K L

```

```

    by (rule normal-subgroup-in-subgroup-axioms.intro[OF K-subset-L])
qed
have corr-eq-Q: corr.Fract-H-K = Q
proof (rule equalityI)
  show corr.Fract-H-K  $\subseteq$  Q
  proof
    fix A
    assume A  $\in$  corr.Fract-H-K
    then obtain l where l: l  $\in$  L and A-eq: A = maximal.N.Class l
      by (rule corr.Fract-H-K-memE)
    show A  $\in$  Q
    using l A-eq unfolding L-def by simp
  qed
next
  show Q  $\subseteq$  corr.Fract-H-K
  proof
    fix A
    assume A: A  $\in$  Q
    then have A  $\in$  maximal.N.Factor-Group by (rule Q.sub)
    then obtain g where g: g  $\in$  G and A-eq: A = maximal.N.Class g
      using maximal.N.representant-exists by blast
    have g  $\in$  L using A A-eq g unfolding L-def by simp
    then show A  $\in$  corr.Fract-H-K
    unfolding corr.Fract-H-K-def A-eq by blast
  qed
qed

have L = K  $\vee$  L = G
  by (rule maximal.maximal[OF L-normal K-subset-L])
then show ?thesis
  using corr-eq-Q corr.Fract-H-K-eq-bottom-iff
    corr.Fract-H-K-eq-top-iff
  by blast
qed

show simple-factor-group K G composition unit
proof (intro simple-factor-group.intro)
  show normal-subgroup K G composition unit
    by (rule maximal.N.normal-subgroup-axioms)
next
  show Simple-Group maximal.N.Factor-Group
    maximal.N.quotient-composition (maximal.N.Class unit)
proof (unfold-locales)
  show maximal.N.Factor-Group  $\neq$  {maximal.N.Class unit}
    by (rule quotient-nontrivial)
next
  show  $\bigwedge Q$ . normal-subgroup Q maximal.N.Factor-Group
    maximal.N.quotient-composition (maximal.N.Class unit)  $\implies$ 
    Q = {maximal.N.Class unit}  $\vee$  Q = maximal.N.Factor-Group
    by (rule quotient-normal-cases)

```



```

    qed
  qed
next
  assume factor: simple-factor-group K G composition unit
  interpret factor: simple-factor-group K G composition unit by fact
  show maximal-normal-subgroup K G composition unit
  proof (unfold-locales)
    show  $K \neq G$  by (rule factor.proper)
  next
    fix L
    assume normal: normal-subgroup L G composition unit
    and contained:  $K \subseteq L$ 
    show  $L = K \vee L = G$ 
    by (rule simple-factor-intermediate-normal[OF factor contained normal])
  qed
qed

```

19.2 Existence in finite groups

In a finite group, every proper normal subgroup lies below a maximal normal subgroup. Finiteness is used only to select a maximal member of the finite set of proper normal enlargements; the selected subgroup is therefore also suitable for induction on the carrier cardinality.

```

theorem finite-maximal-normal-subgroup-above:
  assumes G: Group G composition unit
  and finite-G: finite G
  and K-normal: normal-subgroup K G composition unit
  and K-proper:  $K \neq G$ 
  obtains M where
    maximal-normal-subgroup M G composition unit  $K \subseteq M$ 
proof -
  let ?candidates =
    {L. normal-subgroup L G composition unit  $\wedge K \subseteq L \wedge L \neq G$ }
  interpret K: normal-subgroup K G composition unit by (rule K-normal)
  have candidates-finite: finite ?candidates
  proof (rule finite-subset[OF - finite-Pow-iff[THEN iffD2, OF finite-G]])
    show ?candidates  $\subseteq$  Pow G
  proof
    fix L
    assume L  $\in$  ?candidates
    then have normal-subgroup L G composition unit by simp
    then interpret L: normal-subgroup L G composition unit .
    show L  $\in$  Pow G using L.subset by simp
  qed
qed
have K-candidate: K  $\in$  ?candidates
using K-normal K-proper by simp
have candidates-nonempty: ?candidates  $\neq \{\}$ 
using K-candidate by blast

```

```

obtain  $M$  where  $M$ -candidate:  $M \in ?candidates$ 
and  $M$ -maximal:
   $\forall L \in ?candidates. M \subseteq L \longrightarrow M = L$ 
using finite-has-maximal[OF candidates-finite candidates-nonempty]
by auto
have  $M$ -normal: normal-subgroup  $M$   $G$  composition unit
and  $K$ -subset- $M$ :  $K \subseteq M$  and  $M$ -proper:  $M \neq G$ 
using  $M$ -candidate by auto
have  $M$ -is-maximal: maximal-normal-subgroup  $M$   $G$  composition unit
proof (rule maximal-normal-subgroup.intro[OF M-normal])
  show maximal-normal-subgroup-axioms  $M$   $G$  composition unit
  proof
    show  $M \neq G$  by fact
  next
    fix  $L$ 
    assume  $L$ -normal: normal-subgroup  $L$   $G$  composition unit
      and  $M$ -subset- $L$ :  $M \subseteq L$ 
    show  $L = M \vee L = G$ 
    proof (cases  $L = G$ )
      case True
        then show ?thesis by simp
      next
        case False
          have  $L \in ?candidates$ 
            using  $L$ -normal  $K$ -subset- $M$   $M$ -subset- $L$  False by auto
          then have  $M = L$  using  $M$ -maximal  $M$ -subset- $L$  by blast
          then show ?thesis by simp
        qed
      qed
    qed
  show ?thesis by (rule that[OF M-is-maximal K-subset-M])
qed

corollary finite-nontrivial-group-has-maximal-normal-subgroup:
assumes  $G$ : Group  $G$  composition unit
and finite-G: finite  $G$  and nontrivial:  $G \neq \{unit\}$ 
obtains  $M$  where maximal-normal-subgroup  $M$   $G$  composition unit
proof –
interpret  $G$ : Group  $G$  composition unit by (rule  $G$ )
have trivial-normal: normal-subgroup  $\{unit\}$   $G$  composition unit
  by (rule  $G$ .trivial-normal-subgroup)
have trivial-proper:  $\{unit\} \neq G$ 
using nontrivial by blast
obtain  $M$  where  $M$ -maximal: maximal-normal-subgroup  $M$   $G$  composition unit
and  $\{unit\} \subseteq M$ 
using finite-maximal-normal-subgroup-above[OF
   $G$  finite-G trivial-normal trivial-proper] .
show ?thesis by (rule that[OF M-maximal])
qed

```

```

end
theory Finite-Composition-Series
  imports Maximal-Normal-Subgroup
begin

```

20 Composition series of finite groups

A composition series for a normal subgroup can be extended by one simple factor. The old terms retain their indices and the new ambient group is placed at the successor of the old length. Keeping this construction separate makes the finite-cardinality induction below reusable.

```

definition extend-composition-series ::
  (nat  $\Rightarrow$  'a set)  $\Rightarrow$  nat  $\Rightarrow$  'a set  $\Rightarrow$  nat  $\Rightarrow$  'a set
where
  extend-composition-series H n G i = (if i  $\leq$  n then H i else G)

```

```

lemma extend-composition-series-old [simp]:
  i  $\leq$  n  $\implies$  extend-composition-series H n G i = H i
by (simp add: extend-composition-series-def)

```

```

lemma extend-composition-series-top [simp]:
  extend-composition-series H n G (Suc n) = G
by (simp add: extend-composition-series-def)

```

```

theorem composition-series-extend-simple-factor:
  assumes series: composition-series K composition unit H n
  and factor: simple-factor-group K G composition unit
  shows composition-series G composition unit
  (extend-composition-series H n G) (Suc n)
proof -
  interpret series: composition-series K composition unit H n
  by (rule series)
  interpret factor: simple-factor-group K G composition unit
  by (rule factor)
  show ?thesis
proof (intro composition-series.intro)
  show normal-series G composition unit
  (extend-composition-series H n G) (Suc n)
proof (intro normal-series.intro)
  show Group G composition unit by (rule factor.N.Group-axioms)
next
  show normal-series-axioms G composition unit
  (extend-composition-series H n G) (Suc n)
proof (intro normal-series-axioms.intro)
  show extend-composition-series H n G 0 = {unit}
  using series.bottom by simp
next
  show extend-composition-series H n G (Suc n) = G by simp

```

```

next
  fix i
  assume i:  $i \leq \text{Suc } n$ 
  show Subgroup (extend-composition-series H n G i) G composition unit
  proof (cases  $i \leq n$ )
    case True
      have old: Subgroup (H i) K composition unit
      by (rule series.term-subgroup[OF True])
      show ?thesis
      using subgroup-transitive[OF old factor.N.Subgroup-axioms] True
      by simp
    next
      case False
      then have  $i = \text{Suc } n$  using i by arith
      then show ?thesis
      by (simp add: factor.N.group-self-subgroup)
  qed
next
  fix i
  assume i:  $i < \text{Suc } n$ 
  show normal-subgroup (extend-composition-series H n G i)
    (extend-composition-series H n G (Suc i)) composition unit
  proof (cases  $i < n$ )
    case True
      then show ?thesis
      using series.normal-step[OF True] by simp
    next
      case False
      then have  $i = n$  using i by arith
      then show ?thesis
      using series.top factor.N.normal-subgroup-axioms by simp
  qed
qed
qed
next
  show composition-series-axioms composition unit
    (extend-composition-series H n G) (Suc n)
  proof (intro composition-series-axioms.intro)
    fix i
    assume i:  $i < \text{Suc } n$ 
    show simple-factor-group (extend-composition-series H n G i)
      (extend-composition-series H n G (Suc i)) composition unit
    proof (cases  $i < n$ )
      case True
        then show ?thesis
        using series.factor-simple[OF True] by simp
      next
        case False
        then have  $i = n$  using i by arith
        then show ?thesis using series.top factor by simp
    qed
  qed

```

qed
 qed
 qed
 qed

20.1 Existence

Every finite group has a composition series. For a nontrivial group, choose a maximal normal subgroup. Its quotient is simple, while its proper carrier has smaller cardinality, so the induction hypothesis supplies the prefix.

```

theorem finite-group-has-composition-series-ex:
  assumes G: Group G composition unit and finite-G: finite G
  shows  $\exists H\ n.$  composition-series G composition unit H n
  using G finite-G
proof (induction card G arbitrary: G composition unit rule: less-induct)
  case less
  interpret G: Group G composition unit by (rule less.prem(1))
  show ?case
  proof (cases G = {unit})
    case True
    have composition-series G composition unit ( $\lambda.$  G) 0
    proof (intro composition-series.intro normal-series.intro)
      show Group G composition unit by (rule G.Group-axioms)
    next
      show normal-series-axioms G composition unit ( $\lambda.$  G) 0
      using True G.group-self-subgroup
      by unfold-locales auto
    next
      show composition-series-axioms composition unit ( $\lambda.$  G) 0
      proof (rule composition-series-axioms.intro)
        fix i :: nat
        assume i < 0
        then show simple-factor-group (( $\lambda.$  G) i)
          (( $\lambda.$  G) (Suc i)) composition unit
          by simp
        qed
      qed
      then show ?thesis by blast
  next
  case False
  obtain M where maximal: maximal-normal-subgroup M G composition unit
    using finite-nontrivial-group-has-maximal-normal-subgroup[OF
      G.Group-axioms less.prem(2) False] .
  interpret maximal: maximal-normal-subgroup M G composition unit
    by (rule maximal)
  have finite-M: finite M
    by (rule finite-subset[OF maximal.N.subset less.prem(2)])
  have card-M-less: card M < card G
    by (rule psubset-card-mono[OF less.prem(2) maximal.strict])

```

```

have M-group: Group M composition unit
  by (rule subgroup-imp-Group[OF maximal.N.Subgroup-axioms])
obtain H n where prefix: composition-series M composition unit H n
  using less.hyps[OF card-M-less M-group finite-M] by blast
have factor: simple-factor-group M G composition unit
  by (rule iffD1[OF maximal-normal-subgroup-iff-simple-factor-group maximal])
have composition-series G composition unit
  (extend-composition-series H n G) (Suc n)
  by (rule composition-series-extend-simple-factor[OF prefix factor])
then show ?thesis by blast
qed
qed

corollary finite-group-has-composition-series:
  assumes G: Group G composition unit and finite-G: finite G
  obtains H n where composition-series G composition unit H n
  using finite-group-has-composition-series-ex[OF G finite-G] that by blast

end
theory Composition-Factors-Solvable
  imports Finite-Composition-Series Jordan-Hoelder-Uniqueness Solvable-Transfer
begin

```

21 Solvability and composition factors

21.1 Auxiliary group-theoretic bridges

The kernel of the canonical quotient map is the normal subgroup itself. Although this is implicit in the quotient construction, an explicit lemma is useful whenever a theorem is phrased using the generic homomorphism kernel, as is the solvable-extension theorem.

```

lemma (in normal-subgroup) quotient-map-group-epimorphism:
  group-epimorphism Class G (·) unit
  Factor-Group quotient-composition (Class unit)
proof (rule group-epimorphism.intro)
  show group-homomorphism Class G (·) unit
    Factor-Group quotient-composition (Class unit)
proof (rule group-homomorphism.intro)
  show Monoid-homomorphism Class G (·) unit
    Factor-Group quotient-composition (Class unit)
    by (rule natural.Monoid-homomorphism-axioms)
  show Group G (·) unit by (rule Group-axioms)
  show Group Factor-Group quotient-composition (Class unit)
    by (rule quotient.Group-axioms)
qed
show Monoid-epimorphism Class G (·) unit
  Factor-Group quotient-composition (Class unit)
  by (rule Monoid-epimorphism.intro[OF

```

```

      natural.Monoid-homomorphism-axioms natural.surjective-map-axioms])
qed

lemma (in normal-subgroup) quotient-map-kernel:
  group-homomorphism.Ker Class G (Class unit) = K
proof -
interpret quotient-map: group-epimorphism Class G (·) unit
  Factor-Group quotient-composition Class unit
  by (rule quotient-map-group-epimorphism)
show ?thesis
proof (rule equalityI)
  show quotient-map.Ker  $\subseteq$  K
proof
  fix x
  assume x: x  $\in$  quotient-map.Ker
  have x-G: x  $\in$  G by (rule quotient-map.Ker-closed[OF x])
  have class-eq: Class x = Class unit
    by (rule quotient-map.Ker-image[OF x])
  have x  $\in$  Class x by (rule Class-self[OF x-G])
  then have x  $\in$  Class unit using class-eq by simp
  then show x  $\in$  K using Class-unit-normal-subgroup by simp
qed
next
show K  $\subseteq$  quotient-map.Ker
proof
  fix k
  assume k: k  $\in$  K
  have class-eq: Class k = Class unit
proof (rule Class-eq)
  show (k, unit)  $\in$  Congruence
proof (rule CongruenceI)
  show k = unit  $\cdot$  k by (rule left-unit[OF sub[OF k], symmetric])
  show k  $\in$  G by (rule sub[OF k])
  show unit  $\in$  G by (rule unit-closed)
  show k  $\in$  K by fact
qed
qed
show k  $\in$  quotient-map.Ker
  by (rule quotient-map.Ker-memI[OF class-eq sub[OF k]])
qed
qed
qed

```

A simple group is solvable exactly when it is abelian. The nontrivial direction uses simplicity on the derived subgroup: if that subgroup were the whole carrier, every term of the derived series would remain the whole carrier, contradicting solvability.

theorem *simple-group-solvable-iff-abelian:*
assumes *simple: Simple-Group G composition unit*

```

shows Group.solvable G composition unit  $\longleftrightarrow$ 
  Abelian-Group G composition unit
proof -
interpret simple: Simple-Group G composition unit by (rule simple)
show ?thesis
proof
  assume solvable: simple.solvable
  have derived-cases: simple.derived = {unit}  $\vee$  simple.derived = G
    by (rule simple.normal-subgroup-eq-trivial-or-top[OF simple.derived-normal])
  have derived-not-top: simple.derived  $\neq$  G
  proof
    assume derived-top: simple.derived = G
    have series-top: simple.derivedSeries i = G for i
    proof (induction i)
      case 0
      show ?case by simp
    next
      case (Suc i)
      have simple.commutator-subgroup G G = G
        using derived-top unfolding simple.derived-def .
      then show ?case using Suc.IH by simp
    qed
  obtain i where simple.derivedSeries i = {unit}
    using solvable unfolding simple.solvable-def by blast
  then show False using series-top[of i] simple.nontrivial by simp
qed
have derived-bottom: simple.derived = {unit}
  using derived-cases derived-not-top by blast
have commutative: commutative-monoid G composition unit
proof (rule commutative-monoid.intro[OF simple.Monoid-axioms])
  show commutative-monoid-axioms G composition
  proof
    fix a b
    assume a:  $a \in G$  and b:  $b \in G$ 
    have commutator-in: simple.commutator-elt a b  $\in$  simple.derived
      unfolding simple.derived-def
      by (rule simple.commutator-gen-mem[OF a b subset-refl subset-refl])
    have commutator-unit: simple.commutator-elt a b = unit
      using commutator-in derived-bottom by simp
    have conjugate-G:
      composition (composition a b) (simple.inverse a)  $\in$  G
      using a b by simp
    have conjugate-inverse:
      composition (composition (composition a b) (simple.inverse a))
        (simple.inverse b) = unit
      using commutator-unit unfolding simple.commutator-elt-def .
    have conjugate-eq:
      composition (composition a b) (simple.inverse a) = b
    proof (rule simple.inverse-unique[where u=simple.inverse b])
      show composition (simple.inverse b) b = unit using b by simp
    qed
  qed

```



```

    show composition (composition (composition a b) (simple.inverse a))
      (simple.inverse b) = unit by (rule conjugate-inverse)
    show simple.inverse b ∈ G using b by simp
    show composition (composition a b) (simple.inverse a) ∈ G
      by (rule conjugate-G)
    show b ∈ G by fact
  qed
  show composition a b = composition b a
    using conjugate-eq simple commute-iff-inverse[OF a b] by simp
  qed
  show Abelian-Group G composition unit
    by (rule Abelian-Group.intro[OF simple.Group-axioms commutative])
next
  assume abelian: Abelian-Group G composition unit
  interpret abelian: Abelian-Group G composition unit by (rule abelian)
  show simple.solvable
    by (rule simple.abelian-imp-solvable[OF
      abelian.commutative-monoid-axioms])
  qed
qed

```

21.2 The factor criterion

```

context composition-series
begin

```

```

definition abelian-factors :: bool
where
  abelian-factors  $\longleftrightarrow$  ( $\forall i < n.$ 
    Abelian-Group (fst (series-factor i))
      (fst (snd (series-factor i))) (snd (snd (series-factor i))))

```

```

lemma abelian-factorsI:
  assumes  $\bigwedge i. i < n \implies$  Abelian-Group (fst (series-factor i))
    (fst (snd (series-factor i))) (snd (snd (series-factor i)))
  shows abelian-factors
  using assms unfolding abelian-factors-def by blast

```

```

lemma abelian-factorsD:
  assumes abelian: abelian-factors and i:  $i < n$ 
  shows Abelian-Group (fst (series-factor i))
    (fst (snd (series-factor i))) (snd (snd (series-factor i)))
  using abelian i unfolding abelian-factors-def by blast

```

Solvability is equivalent to abelianness of every composition factor. The forward implication passes solvability to each subgroup and then to its quotient. The reverse implication reconstructs solvability one extension at a time along the series.

```

theorem solvable-iff-abelian-factors:
   $G.\text{solvable} \longleftrightarrow \text{abelian-factors}$ 
proof
  assume solvable:  $G.\text{solvable}$ 
  show abelian-factors
    unfolding abelian-factors-def
  proof (intro allI impI)
    fix i
    assume i:  $i < n$ 
    interpret step: simple-factor-group  $H\ i\ H\ (Suc\ i)\ (\cdot)\ unit$ 
      by (rule factor-simple[OF i])
    have numerator-subgroup: Subgroup  $(H\ (Suc\ i))\ G\ (\cdot)\ unit$ 
      by (rule term-subgroup) (use i in simp)
    have numerator-solvable: Group.solvable  $(H\ (Suc\ i))\ (\cdot)\ unit$ 
      by (rule G.solvable-subgroup[OF numerator-subgroup solvable])
    interpret quotient-map: group-epimorphism step.N.Class  $H\ (Suc\ i)\$ 
       $(\cdot)\ unit\ step.N.Factor-Group\ step.N.quotient-composition\$ 
       $step.N.Class\ unit$ 
      by (rule step.N.quotient-map-group-epimorphism)
    have quotient-solvable: step.N.quotient.solvable
      by (rule quotient-map.solvable-image-epi[OF numerator-solvable])
    have quotient-abelian:
      Abelian-Group  $step.N.Factor-Group\ step.N.quotient-composition\$ 
       $(step.N.Class\ unit)$ 
      by (rule iffD1[OF simple-group-solvable-iff-abelian[
        OF step.factor-group-is-simple] quotient-solvable])
    show Abelian-Group  $(fst\ (series-factor\ i))\$ 
       $(fst\ (snd\ (series-factor\ i)))\ (snd\ (snd\ (series-factor\ i)))$ 
      using quotient-abelian unfolding series-factor-def by simp
    qed
  next
    assume factors: abelian-factors
    have term-solvable:
       $i \leq n \implies \text{Group.solvable}\ (H\ i)\ (\cdot)\ unit$  for i
    proof (induction i)
      case 0
        interpret trivial: Group  $H\ 0\ (\cdot)\ unit$ 
          by (rule subgroup-imp-Group[OF term-subgroup[OF zero-le]])
        have H0:  $H\ 0 = \{unit\}$  by (rule bottom)
        have derived0: trivial.derivedSeries  $0 = \{unit\}$ 
          by (rule trans[OF trivial.derivedSeries.simps(1) H0])
        show ?case
          unfolding trivial.solvable-def using derived0 by blast
      next
        case (Suc i)
        have i:  $i < n$  using Suc.premis by simp
        interpret step: simple-factor-group  $H\ i\ H\ (Suc\ i)\ (\cdot)\ unit$ 
          by (rule factor-simple[OF i])
        interpret quotient-map: group-epimorphism step.N.Class  $H\ (Suc\ i)\$ 
           $(\cdot)\ unit\ step.N.Factor-Group\ step.N.quotient-composition$ 

```

```

    step.N.Class unit
  by (rule step.N.quotient-map-group-epimorphism)
have kernel-eq: quotient-map.Ker = H i
  by (rule step.N.quotient-map-kernel)
have kernel-solvable:
  Group.solvable quotient-map.Ker (·) unit
  using Suc.IH[OF less-imp-le[OF i]] kernel-eq by simp
have factor-abelian:
  Abelian-Group step.N.Factor-Group step.N.quotient-composition
  (step.N.Class unit)
  using abelian-factorsD[OF factors i]
  unfolding series-factor-def by simp
interpret factor-abelian:
  Abelian-Group step.N.Factor-Group step.N.quotient-composition
  step.N.Class unit
  by (rule factor-abelian)
have factor-solvable: step.N.quotient.solvable
  by (rule step.N.quotient.abelian-imp-solvable[OF
    factor-abelian.commutative-monoid-axioms])
show ?case
  by (rule quotient-map.solvable-extension[OF
    kernel-solvable factor-solvable])
qed
show G.solvable
  using term-solvable[OF order-refl] top by simp
qed
end

```

21.3 Independence and finite-group formulations

```

context composition-series-pair
begin

```

The Jordan–Hölder theorem identifies the complete factor multisets. For abelian-ness, the solvability criterion gives the corresponding invariance immediately, without choosing isomorphisms between matched factors.

```

corollary abelian-factors-iff:
  A-comp.abelian-factors  $\longleftrightarrow$  B-comp.abelian-factors
  using A-comp.solvable-iff-abelian-factors
  B-comp.solvable-iff-abelian-factors by blast

```

end

For finite groups the existential and universal formulations are both available because a composition series always exists. The universal form makes independence of the chosen series explicit.

```

theorem finite-group-solvable-iff-has-composition-series-with-abelian-factors:
  assumes G: Group G composition unit and finite-G: finite G

```

```

shows Group.solvable G composition unit  $\longleftrightarrow$ 
  ( $\exists H n.$  composition-series G composition unit H n  $\wedge$ 
    composition-series.abelian-factors composition unit H n)
proof
  assume solvable: Group.solvable G composition unit
  obtain H n where series: composition-series G composition unit H n
    using finite-group-has-composition-series[OF G finite-G] .
  interpret series: composition-series G composition unit H n by (rule series)
  have series.abelian-factors
    using solvable series.solvable-iff-abelian-factors by blast
  then show  $\exists H n.$  composition-series G composition unit H n  $\wedge$ 
    composition-series.abelian-factors composition unit H n
    using series by blast
next
  assume  $\exists H n.$  composition-series G composition unit H n  $\wedge$ 
    composition-series.abelian-factors composition unit H n
  then obtain H n where series: composition-series G composition unit H n
    and factors: composition-series.abelian-factors composition unit H n
    by blast
  interpret series: composition-series G composition unit H n by (rule series)
  show Group.solvable G composition unit
    using factors series.solvable-iff-abelian-factors by blast
qed

theorem finite-group-solvable-iff-all-composition-series-have-abelian-factors:
  assumes G: Group G composition unit and finite-G: finite G
  shows Group.solvable G composition unit  $\longleftrightarrow$ 
    ( $\forall H n.$  composition-series G composition unit H n  $\longrightarrow$ 
      composition-series.abelian-factors composition unit H n)
proof
  assume solvable: Group.solvable G composition unit
  show  $\forall H n.$  composition-series G composition unit H n  $\longrightarrow$ 
    composition-series.abelian-factors composition unit H n
  proof (intro allI impI)
    fix H n
    assume series: composition-series G composition unit H n
    interpret series: composition-series G composition unit H n by (rule series)
    show series.abelian-factors
      using solvable series.solvable-iff-abelian-factors by blast
  qed
next
  assume all-series:
     $\forall H n.$  composition-series G composition unit H n  $\longrightarrow$ 
      composition-series.abelian-factors composition unit H n
  obtain H n where series: composition-series G composition unit H n
    using finite-group-has-composition-series[OF G finite-G] .
  interpret series: composition-series G composition unit H n by (rule series)
  have series.abelian-factors using all-series series by blast
  then show Group.solvable G composition unit
    using series.solvable-iff-abelian-factors by blast

```

qed

Read the other way, the criterion says an abelian group has only abelian composition factors: an abelian group is solvable, and solvability is equivalent to all factors being abelian.

lemma *abelian-group-composition-factors-abelian:*

assumes *ab: Abelian-Group G comp e*

and *series: composition-series G comp e H m*

shows *composition-series.abelian-factors comp e H m*

proof —

interpret *ab: Abelian-Group G comp e* **by** (*rule ab*)

interpret *S: composition-series G comp e H m* **by** (*rule series*)

— *Group.abelian-imp-solvable* wants the *commutative monoid* component, which an abelian group supplies as one of its axioms.

have *Group.solvable G comp e*

by (*rule ab.abelian-imp-solvable*) (*rule ab.commutative-monoid-axioms*)

then show *?thesis* **using** *S.solvable-iff-abelian-factors* **by** *blast*

qed

end

22 The Symmetric and Alternating Groups

theory *Symmetric-Group*

imports *Group-Theory HOL-Combinatorics.Permutations HOL-Combinatorics.Cycles*

begin

We realise the concrete symmetric group S_n as the group of permutations of $\{0..<n\}$ under function composition, reusing the permutation and sign machinery of *HOL-Combinatorics.Permutations*.

Design note: the development's own $\lambda S. \text{Monoid.Units } (S \rightarrow_E S)$ (*compose S*) (*identity S*) consists of *extensional* maps (undefined outside the carrier), which does not interoperate directly with the library predicate *p permutes S* (identity outside the carrier). Building S_n natively on $\{p. p \text{ permutes } S\}$ avoids that impedance mismatch and lets us inherit *sign* and its homomorphism property for free.

22.1 The symmetric group as a Group

definition $S_n :: \text{nat} \Rightarrow (\text{nat} \Rightarrow \text{nat}) \text{ set}$

where $S_n n = \{p. p \text{ permutes } \{0..<n\}\}$

lemma *S_n-iff*: $p \in S_n n \longleftrightarrow p \text{ permutes } \{0..<n\}$

by (*simp add: S_n-def*)

lemma *S_n-permutation*: $p \in S_n n \implies \text{permutation } p$

by (*auto simp: S_n-def permutation-permutes*)

$Sn\ n$ is a group under composition.

theorem *Group-Sn*: $Group\ (Sn\ n)\ (\circ)\ id$
proof (*rule GroupI*)
fix u **assume** $u \in Sn\ n$
then have $perm: u\ permutes\ \{0..<n\}$ **by** (*simp add: Sn-def*)
have $inv\ u \in Sn\ n$ $u \circ inv\ u = id$ $inv\ u \circ u = id$
using $perm$ **by** (*simp-all add: Sn-def permutes-inv permutes-inv-o*)
then show $\exists v \in Sn\ n. u \circ v = id \wedge v \circ u = id$ **by** *blast*
qed (*auto simp: Sn-def permutes-compose permutes-id o-assoc*)

22.2 Relation to the development's abstract symmetric group

The development's own symmetric group $Monoid.Units\ (S \rightarrow_E S)\ (compose\ S)\ (identity\ S)$ consists of the *extensional* bijections of S under *compose* $S/identity\ S$. Our $Sn\ n$ uses library (*permutes*) maps (identity outside). The two are reconciled by *restrict*: restricting a permutation to S yields an element of $Monoid.Units\ (S \rightarrow_E S)\ (compose\ S)\ (identity\ S)$, and this correspondence is bijective. This makes the concrete $Sn\ n$ available to the development's *Group-Action* machinery.

lemma *restrict-permutes-in-Sym*:
assumes $p\ permutes\ S$
shows $restrict\ p\ S \in transformations.Sym\ S$
using *permutes-imp-bij* [*OF assms*]
by (*simp add: bij-betw-imp-funcset transformations.Units-bij-betwD*)

Conversely, an element of $Monoid.Units\ (S \rightarrow_E S)\ (compose\ S)\ (identity\ S)$ permutes S once it is extended by the identity outside S .

lemma *Sym-extend-permutes*:
assumes $\alpha \in transformations.Sym\ S$
shows $(\lambda x. \text{if } x \in S \text{ then } \alpha\ x \text{ else } x)\ permutes\ S$
proof (*rule bij-imp-permutes*)
have $bij-betw\ \alpha\ S\ S$ **using** *assms* **by** (*simp add: transformations.Units-bijective*)
then show $bij-betw\ (\lambda x. \text{if } x \in S \text{ then } \alpha\ x \text{ else } x)\ S\ S$
by (*smt (verit, best) bij-betw-cong*)
qed *simp*

22.3 The sign homomorphism and the alternating group

The two-element group of signs.

lemma *Group-signs*: $Group\ \{1, -1::int\}\ (*)\ 1$
proof (*rule GroupI*)
fix $u :: int$ **assume** $u \in \{1, -1\}$
then show $\exists v \in \{1, -1::int\}. u * v = 1 \wedge v * u = 1$ **by** *force*
qed *auto*

lemma *sign-range*: $p \in Sn\ n \implies sign\ p \in \{1, -1::int\}$

by (auto simp: sign-def)

sign is a group homomorphism $S_n \rightarrow \{1, -1\}$. The multiplicativity is inherited directly from *sign-compose*.

lemma *sign-hom*: group-homomorphism (restrict sign (Sn n)) (Sn n) (○) id {1,-1::int} (*) 1

proof –

interpret *S*: Group Sn n (○) id **by** (rule Group-Sn)

interpret *T*: Group {1,-1::int} (*) 1 **by** (rule Group-signs)

show ?thesis

proof qed (use sign-range in ⟨auto simp: Sn-permutation sign-compose⟩)

qed

The alternating group is the kernel of the sign homomorphism.

definition *An* :: nat $\Rightarrow$ (nat $\Rightarrow$ nat) set

where *An* n = group-homomorphism.Ker (restrict sign (Sn n)) (Sn n) (1::int)

context

fixes n::nat

begin

interpretation *H*: group-homomorphism restrict sign (Sn n) Sn n (○) id {1,-1::int} (*) 1

by (rule sign-hom)

lemma *An-normal*: normal-subgroup (An n) (Sn n) (○) id

using An-def H.kernel.normal-subgroup-axioms **by** presburger

lemma *An-iff*: $p \in An\ n \longleftrightarrow p \in Sn\ n \wedge sign\ p = 1$

by (metis An-def H.Ker-image H.Ker-memI H.kernel.sub restrict-apply)

end

lemma *An-subset-Sn*: $An\ n \subseteq Sn\ n$

by (auto simp: An-iff)

22.4 Three-cycles

A three-cycle on distinct points of $\{0..<n\}$ is a permutation, and it is even, hence lies in the alternating group.

lemma *three-cycle-permutes*:

assumes distinct [a,b,c] $a < n\ b < n\ c < n$

shows cycle-of-list [a,b,c] permutes $\{0..<(n::nat)\}$

using assms cycle-permutes [of [a,b,c]] permutes-subset assms **by** fastforce

lemma *three-cycle-in-Sn*:

assumes distinct [a,b,c] $a < n\ b < n\ c < n$

shows cycle-of-list [a,b,c] $\in Sn\ n$

using *assms three-cycle-permutes* by (simp add: *Sn-def*)

lemma *three-cycle-sign*:
assumes *distinct* [a,b,c]
shows *sign* (*cycle-of-list* [a,b,c]) = 1
proof –
have *sign* (*cycle-of-list* [a,b,c])
= *sign* (*Transposition.transpose* a b) * *sign* (*Transposition.transpose* b c)
by (simp add: *permutation-swap-id sign-compose*)
also have ... = 1 **using** *assms* **by** (simp add: *sign-swap-id*)
finally show ?thesis .
qed

lemma *three-cycle-in-An*:
assumes *distinct* [a,b,c] a < n b < n c < n
shows *cycle-of-list* [a,b,c] ∈ An n
by (meson An-iff *assms three-cycle-in-Sn three-cycle-sign*)
end

23 Insolubility of the Symmetric Group

theory *Sym-Not-Solvable*
imports *Symmetric-Group Solvable-Transfer*
begin

For $n \geq 5$ the symmetric group S_n is not solvable. The argument is the classical one: every three-cycle is a commutator of two three-cycles (using two “spare” points, which exist when $n \geq 5$), so the three-cycles survive in every term of the derived series, which therefore never reaches the trivial subgroup.

23.1 Three-cycles as commutators

The inverse of a three-cycle is the reversed three-cycle.

lemma *rev-inv-3cycle*:
assumes *distinct* [a,b,(c::nat)]
shows *inv* (*cycle-of-list* [a,b,c]) = *cycle-of-list* [c,b,a]
using *assms* **by** (auto simp add: *o-inv-distrib inv-unique-comp transpose-def*)

Inside S_n the group inverse coincides with the functional inverse.

lemma *Sn-inverse*:
assumes $p \in S_n$
shows *Monoid.inverse* (S_n) ($\circ$) id p = *inv* p
proof –
interpret *S*: *Group* S_n ($\circ$) id **by** (rule *Group-Sn*)
show ?thesis
by (metis *S.invertible S.invertible-left-inverse S.invertible-right-inverse assms inv-unique-comp*)

qed

The defining pointwise identity: a three-cycle factors as a product of four three-cycles (which is, up to inverses, a commutator). Verified by a six-way case split on the argument.

lemma *three-cycle-commutator-core*:
assumes *dist*: *distinct* $[a, b, c, d, (e::nat)]$
shows *cycle-of-list* $[a, b, c]$
 $= \text{cycle-of-list } [a, b, d] \circ \text{cycle-of-list } [c, e, a] \circ \text{cycle-of-list } [d, b, a] \circ \text{cycle-of-list } [a, e, c]$
proof (*rule ext*)
fix *x*
have *ne*: $a \neq b \ a \neq c \ a \neq d \ a \neq e \ b \neq c \ b \neq d \ b \neq e \ c \neq d \ c \neq e \ d \neq e$
using *dist* **by** *auto*
then show *cycle-of-list* $[a, b, c] \ x$
 $= (\text{cycle-of-list } [a, b, d] \circ \text{cycle-of-list } [c, e, a] \circ \text{cycle-of-list } [d, b, a] \circ \text{cycle-of-list } [a, e, c]) \ x$
by (*cases* $x \in \{a, b, c, d, e\}$) (*auto simp: ne ne[symmetric]*) +
qed

Hence a three-cycle is a commutator of two three-cycles in S_n .

theorem *three-cycle-commutator*:
assumes *dist*: *distinct* $[a, b, c, d, (e::nat)]$ **and** *lt*: $a < n \ b < n \ c < n \ d < n \ e < n$
shows *cycle-of-list* $[a, b, c]$
 $= \text{Group.commutator-elt } (S_n \ n) \ (\circ) \ \text{id} \ (\text{cycle-of-list } [a, b, d]) \ (\text{cycle-of-list } [c, e, a])$
proof –
interpret *S*: *Group* $S_n \ n \ (\circ) \ \text{id}$ **by** (*rule Group-Sn*)
have *ip*: *inv* (*cycle-of-list* $[a, b, d]$) = *cycle-of-list* $[d, b, a]$
using *dist* **by** (*intro rev-inv-3cycle*) *auto*
have *iq*: *inv* (*cycle-of-list* $[c, e, a]$) = *cycle-of-list* $[a, e, c]$
using *dist* **by** (*intro rev-inv-3cycle*) *auto*
have *p-in*: *cycle-of-list* $[a, b, d] \in S_n \ n$ **using** *dist lt* **by** (*intro three-cycle-in-Sn*) *auto*
have *q-in*: *cycle-of-list* $[c, e, a] \in S_n \ n$ **using** *dist lt* **by** (*intro three-cycle-in-Sn*) *auto*
have *Group.commutator-elt* $(S_n \ n) \ (\circ) \ \text{id} \ (\text{cycle-of-list } [a, b, d]) \ (\text{cycle-of-list } [c, e, a])$
 $= \text{cycle-of-list } [a, b, d] \circ \text{cycle-of-list } [c, e, a] \circ \text{cycle-of-list } [d, b, a] \circ \text{cycle-of-list } [a, e, c]$
using *p-in q-in ip iq* **by** (*simp add: S.commutator-elt-def Sn-inverse*)
also have $\dots = \text{cycle-of-list } [a, b, c]$
using *three-cycle-commutator-core[OF dist]* **by** *simp*
finally show *?thesis* ..
qed

23.2 Persistence of three-cycles in the derived series

A set of cardinality at least two contains two distinct elements. (A direct consequence of the library lemma *card-le-Suc-iff*.)

lemma *card-ge2-two-elems*:

assumes $2 \leq \text{card } R$
shows $\exists d e. d \in R \wedge e \in R \wedge d \neq e$
using *card-le-Suc-iff*[of 1 R] *assms not-less-eq-eq* **by** *fastforce*

For $n \geq 5$ there are two further points distinct from any three.

lemma *two-spare-points*:

assumes $n5: 5 \leq n$ **and** *dist*: $\text{distinct } [a, b, (c::\text{nat})]$ **and** *lt*: $a < n \ b < n \ c < n$
shows $\exists d e. \text{distinct } [a, b, c, d, e] \wedge d < n \wedge e < n$

proof –

let $?R = \{0..<n\} - \{a, b, c\}$
have $\{a, b, c\} \subseteq \{0..<n\}$ **using** *lt* **by** *auto*
then have $\text{card } ?R = \text{card } \{0..<n\} - \text{card } \{a, b, c\}$ **by** (*simp add: card-Diff-subset*)
moreover have $\text{card } \{a, b, c\} = 3$ **using** *dist* **by** *auto*
ultimately have $2 \leq \text{card } ?R$ **using** $n5$ **by** *simp*
then obtain $d e$ **where** $d \in ?R \ e \in ?R \ d \neq e$ **using** *card-ge2-two-elems* **by** *blast*
then show *?thesis* **using** *dist* **by** *auto*

qed

If a subgroup of S_n contains all three-cycles, so does its commutator subgroup.

lemma *three-cycle-in-commutator-subgroup*:

assumes $n5: 5 \leq n$
and H : *Subgroup* $H \ (S_n \ n) \ (\circ) \ \text{id}$
and $H\text{cyc}$: $\bigwedge x \ y \ z. \text{distinct } [x, y, z] \implies x < n \implies y < n \implies z < n \implies \text{cycle-of-list } [x, y, z] \in H$
and *dist*: $\text{distinct } [a, b, c]$ **and** *lt*: $a < n \ b < n \ c < n$
shows $\text{cycle-of-list } [a, b, c] \in \text{Group.commutator-subgroup } (S_n \ n) \ (\circ) \ \text{id} \ H \ H$

proof –

interpret HS : *Subgroup* $H \ S_n \ n \ (\circ) \ \text{id}$
using H **by** *blast*
obtain $d e$ **where** de : $\text{distinct } [a, b, c, d, e] \ d < n \ e < n$
using *two-spare-points*[OF $n5$ *dist* *lt*] **by** *blast*
have hd : $\text{cycle-of-list } [a, b, d] \in H$ **using** de *dist* *lt* **by** (*intro Hcyc*) *auto*
have he : $\text{cycle-of-list } [c, e, a] \in H$ **using** de *dist* *lt* **by** (*intro Hcyc*) *auto*
have $\text{cycle-of-list } [a, b, c]$
 $= \text{Group.commutator-elt } (S_n \ n) \ (\circ) \ \text{id} \ (\text{cycle-of-list } [a, b, d]) \ (\text{cycle-of-list } [c, e, a])$
using de *dist* *lt* **by** (*intro three-cycle-commutator*) *auto*
then show *?thesis*
by (*metis Group.commutator-gen-mem Group-Sn HS.subset hd he*)

qed

Every three-cycle survives in every term of the derived series of S_n .

lemma *three-cycles-in-derivedSeries*:

assumes $n5: 5 \leq n$ **and** *dist*: $\text{distinct } [a, b, c]$ **and** *lt*: $a < n \ b < n \ c < n$
shows $\text{cycle-of-list } [a, b, c] \in \text{Group.derivedSeries } (S_n \ n) \ (\circ) \ \text{id} \ m$

proof –

interpret S : *Group* $S_n \ n \ (\circ) \ \text{id}$ **by** (*rule Group-Sn*)
have $\forall x \ y \ z. \text{distinct } [x, y, z] \implies x < n \implies y < n \implies z < n$
 $\implies \text{cycle-of-list } [x, y, z] \in S.\text{derivedSeries } m$
proof (*induction m*)

```

    case 0
    show ?case
    using S.derivedSeries.simps(1) three-cycle-in-Sn by blast
next
case (Suc m)
then show ?case
using S.derivedSeries.simps(2) S.derivedSeries-subgroup n5 three-cycle-in-commutator-subgroup
by blast
qed
then show ?thesis using dist lt by blast
qed

```

23.3 The main theorem

```

lemma three-cycle-012-neq-id: cycle-of-list [0,1,2] ≠ (id :: nat ⇒ nat)
proof
  assume cycle-of-list [0,1,2] = (id :: nat ⇒ nat)
  then have cycle-of-list [(0::nat),1,2] 0 = 0 by simp
  moreover have cycle-of-list [(0::nat),1,2] 0 = 1 by (simp add: transpose-def)
  ultimately show False by simp
qed

```

```

theorem Sn-not-solvable:
  assumes n5: 5 ≤ n
  shows ¬ Group.solvable (Sn n) (○) id
proof
  interpret S: Group Sn n (○) id by (rule Group-Sn)
  assume S.solvable
  then obtain m where m: S.derivedSeries m = {id} unfolding S.solvable-def by
blast
  have cycle-of-list [0,1,2] ∈ S.derivedSeries m
  using n5 by (intro three-cycles-in-derivedSeries) auto
  with m have cycle-of-list [0,1,2] = (id :: nat ⇒ nat) by simp
  then show False using three-cycle-012-neq-id by simp
qed

```

23.4 The alternating group

The same argument runs inside the alternating group: every three-cycle is even, so it lies in An , and the commutator-of-three-cycles identity stays inside An . The key reuse is *Group.commutator-elt-subgroup-eq*: the commutator computed in the subgroup An agrees with the one in Sn .

```

lemma An-is-subgroup-of-Sn: Subgroup (An n) (Sn n) (○) id
  using An-normal by (simp add: normal-subgroup-def subgroup-of-group-def)

```

```

lemma Group-An: Group (An n) (○) id
  by (rule subgroup-imp-Group[OF An-is-subgroup-of-Sn])

```

```

lemma three-cycle-in-An-commutator-subgroup:

```

assumes $n5: 5 \leq n$
and $H: \text{Subgroup } H \text{ (An } n) \text{ (}\circ\text{) id}$
and $Hcyc: \bigwedge x \ y \ z. \text{ distinct } [x,y,z] \implies x < n \implies y < n \implies z < n \implies \text{cycle-of-list } [x,y,z] \in H$
and $dist: \text{distinct } [a,b,c]$ **and** $lt: a < n \ b < n \ c < n$
shows $\text{cycle-of-list } [a,b,c] \in \text{Group.commutator-subgroup (An } n) \text{ (}\circ\text{) id } H \ H$
proof –
interpret $S: \text{Group } Sn \ n \text{ (}\circ\text{) id}$ **by** (rule Group-Sn)
interpret $HA: \text{Subgroup } H \text{ An } n \text{ (}\circ\text{) id}$ **by** (rule H)
obtain $d \ e$ **where** $de: \text{distinct } [a,b,c,d,e] \ d < n \ e < n$
using $\text{two-spare-points[OF } n5 \ dist \ lt]$ **by** blast
have $hd: \text{cycle-of-list } [a,b,d] \in H$ **using** $de \ dist \ lt$ **by** (intro Hcyc) auto
have $he: \text{cycle-of-list } [c,e,a] \in H$ **using** $de \ dist \ lt$ **by** (intro Hcyc) auto
have $hd-A: \text{cycle-of-list } [a,b,d] \in \text{An } n$ **using** $de \ dist \ lt$ **by** (intro three-cycle-in-An)
auto
– the commutator in $\text{An } n$ agrees with the one in $Sn \ n$
have $\text{cycle-of-list } [a,b,c]$
= $\text{Group.commutator-elt (An } n) \text{ (}\circ\text{) id (cycle-of-list } [a,b,d]) \text{ (cycle-of-list } [c,e,a])}$
using $\text{An-is-subgroup-of-Sn } HA.sub \ S.commutator-elt-subgroup-eq \ de \ hd-A \ he \ lt$
 $\text{three-cycle-commutator}$ **by** presburger
with $HA.subset$ **show** ?thesis
by (metis Group.commutator-gen-mem Group-An hd he)
qed

lemma $\text{three-cycles-in-An-derivedSeries}$:
assumes $n5: 5 \leq n$ **and** $dist: \text{distinct } [a,b,c]$ **and** $lt: a < n \ b < n \ c < n$
shows $\text{cycle-of-list } [a,b,c] \in \text{Group.derivedSeries (An } n) \text{ (}\circ\text{) id } m$
proof –
interpret $A: \text{Group } An \ n \text{ (}\circ\text{) id}$ **by** (rule Group-An)
have ?thesis
using $dist \ lt$
proof (induction m arbitrary: $a \ b \ c$)
case 0
then show ?case
using $A.derivedSeries.simps(1) \ \text{three-cycle-in-An}$ **by** blast
next
case (Suc m)
with $A.derivedSeries.simps(2) \ A.derivedSeries-subgroup \ n5$
 $\text{three-cycle-in-An-commutator-subgroup}$ **show** ?case **by** blast
qed
then show ?thesis **using** $dist \ lt$ **by** blast
qed

theorem An-not-solvable :
assumes $n5: 5 \leq n$
shows $\neg \text{Group.solvable (An } n) \text{ (}\circ\text{) id}$
proof
interpret $A: \text{Group } An \ n \text{ (}\circ\text{) id}$ **by** (rule Group-An)
assume $A.solvable$
then obtain m **where** $m: A.derivedSeries \ m = \{id\}$ **unfolding** $A.solvable-def$ **by**

```

blast
  have cycle-of-list  $[0,1,2] \in A.derivedSeries\ m$ 
    using  $n5$  by (intro three-cycles-in- $An.derivedSeries$ ) auto
  with  $m$  show False using three-cycle-012-neq-id by simp
qed

end

```

24 Generating the Symmetric Group

```

theory Symmetric-Generation
  imports Symmetric-Group Sylow-Theorems
begin

```

S_n n is finite: it is a set of permutations of a finite set. Stated here at the top level — it was previously inside the anonymous context below that fixes a p -cycle g , so the exported fact carried that context's four irrelevant assumptions.

```

lemma finite- $S_n$ : finite ( $S_n\ n$ )
  unfolding  $S_n$ -def using finite-permutations[of  $\{0..<n\}$ ] by simp

```

The symmetric group S_n n is generated by a single transposition together with a full n -cycle, and (for prime p) by *any* transposition together with *any* p -cycle. These generation results are the combinatorial ingredient that, combined with *S_n -not-solvable*, shows that the Galois group of a quintic with exactly two non-real roots is the whole of S_n 5.

24.1 A general lemma on generated subgroups

Powers of an element of a generated subgroup stay inside it.

```

lemma (in Group) generate-funpow:
  assumes  $g \in generate\ S$ 
  shows power  $g\ k \in generate\ S$ 
  by (induction  $k$ ) (auto simp add: assms generate.intros)

```

24.2 Conjugation is an automorphism of the generated subgroup

Conjugation by a group element g fixes the unit and commutes with generation: the image of *generate* S under *conjugation* g is the subgroup generated by the image of S . This lets a generation result be transported along conjugacy, which is what upgrades the standard-cycle generation theorem to an *arbitrary* cycle.

```

lemma (in Group) conjugation-one:  $g \in G \implies conjugation\ g\ 1 = 1$ 
  by (simp add: conjugation-def invertible-right-inverse)

```

```

lemma (in Group) conjugation-inv-cancel:
  assumes  $g: g \in G$  and  $x: x \in G$ 

```

shows $\text{conjugation } (\text{inverse } g) (\text{conjugation } g \ x) = x$
using $g \ x$ **by** (*simp add: conjugation-def associative invertible-left-inverse2 invertible-right-inverse2*)

lemma (in Group) *conjugation-generate-sub*:

assumes $g: g \in G$

shows $\text{conjugation } g \text{ ' generate } S \subseteq \text{generate } (\text{conjugation } g \text{ ' } (S \cap G))$

proof

fix y **assume** $y \in \text{conjugation } g \text{ ' generate } S$

then obtain x **where** $x: x \in \text{generate } S$ **and** $yx: y = \text{conjugation } g \ x$ **by** *auto*

from x **have** $\text{conjugation } g \ x \in \text{generate } (\text{conjugation } g \text{ ' } (S \cap G))$

proof *induction*

case *unit*

then show *?case* **using** *conjugation-one[OF g] generate.unit* **by** *simp*

next

case (*incl x*) **then show** *?case*

by (*simp add: generate.incl conjugation-closed g*)

next

case (*comp x y*)

then show *?case*

using *conjugation-hom generate.comp generate-subset-carrier* **by** (*smt (cvc5, best) in-mono*)

next

case (*inv x*)

then show *?case*

by (*metis IntD2 conjugation-inverse g generate.inv generate-subset-carrier inf.orderE*)

qed

then show $y \in \text{generate } (\text{conjugation } g \text{ ' } (S \cap G))$ **using** yx **by** *simp*

qed

lemma (in Group) *conjugation-generate-eq*:

assumes $g: g \in G$

shows $\text{conjugation } g \text{ ' generate } S = \text{generate } (\text{conjugation } g \text{ ' } (S \cap G))$

proof

show $\text{conjugation } g \text{ ' generate } S \subseteq \text{generate } (\text{conjugation } g \text{ ' } (S \cap G))$

by (*rule conjugation-generate-sub[OF g]*)

next

have $g_{\text{inv}}: \text{inverse } g \in G$ **using** g **by** *blast*

let $?T = \text{conjugation } g \text{ ' } (S \cap G)$

have $\text{sub2}: \text{conjugation } (\text{inverse } g) \text{ ' generate } ?T \subseteq \text{generate } (\text{conjugation } (\text{inverse } g) \text{ ' } (?T \cap G))$

by (*rule conjugation-generate-sub[OF g_{inv}]*)

have $\text{conjugation } (\text{inverse } g) \text{ ' } (?T \cap G) \subseteq S \cap G$

by (*auto simp: conjugation-inv-cancel g*)

then have $\text{step}: \text{generate } (\text{conjugation } (\text{inverse } g) \text{ ' } (?T \cap G)) \subseteq \text{generate } (S \cap G)$

by (*rule generate-mono*)

show $\text{generate } ?T \subseteq \text{conjugation } g \text{ ' generate } S$

proof

fix y **assume** $y: y \in \text{generate } ?T$

have $yG: y \in G$ **using** y *generate-subset-carrier* **by** *blast*

```

    have conjugation g (conjugation (inverse g) y) ∈ conjugation g ‘ generate (S ∩
G)
    using sub2 y step by auto
    moreover have conjugation g (conjugation (inverse g) y) = y
    using ginv yG conjugation-inv-cancel[OF ginv yG] g by simp
    ultimately show y ∈ conjugation g ‘ generate S
    using generate-mono[of S ∩ G S] by auto
qed
qed

```

24.3 Transpositions generate the symmetric group

The set of transpositions of points of $\{0..<n\}$.

definition *transpositions* :: $\text{nat} \Rightarrow (\text{nat} \Rightarrow \text{nat}) \text{ set}$
where *transpositions* $n = \{ \text{Transposition.transpose } a \ b \mid a \ b. \ a \in \{0..<n\} \wedge b \in \{0..<n\} \}$

lemma *transpositions-subset-Sn*: $\text{transpositions } n \subseteq S_n \ n$
by (*auto simp: transpositions-def Sn-def permutes-swap-id*)

24.4 The standard n -cycle

The cyclic shift sending i to $\text{Suc } i \bmod n$ on $\{0..<n\}$, fixing everything outside. We work with this explicit form rather than *cycle-of-list* because its powers have the transparent closed form $(i + k) \bmod n$.

definition *ncycle* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$
where *ncycle* $n \ i = (\text{if } i < n \text{ then } \text{Suc } i \bmod n \text{ else } i)$

lemma *ncycle-lt*: $i < n \implies \text{ncycle } n \ i < n$ **by** (*auto simp: ncycle-def*)

lemma *ncycle-inv-right*: $i < n \implies \text{ncycle } n \ ((i + (n-1)) \bmod n) = i$
by (*simp add: mod-if ncycle-def*)

lemma *ncycle-inv-left*: $i < n \implies (\text{ncycle } n \ i + (n-1)) \bmod n = i$
using *mod-if ncycle-def* **by** *force*

lemma *ncycle-permutes*: $\text{ncycle } n \text{ permutes } \{0..<n\}$
proof (*rule bij-imp-permutes*)
show *bij-betw* $(\text{ncycle } n) \ \{0..<n\} \ \{0..<n\}$
proof (*rule bij-betw-byWitness*)
show $\forall i \in \{0..<n\}. \ \text{ncycle } n \ (\text{if } i < n \text{ then } (i + (n-1)) \bmod n \text{ else } i) = i$
using *ncycle-inv-right* **by** *auto*
qed (*use ncycle-lt ncycle-inv-left in auto*)
qed (*auto simp: ncycle-def*)

lemma *ncycle-in-Sn*: $\text{ncycle } n \in S_n \ n$ **by** (*simp add: Sn-def ncycle-permutes*)

lemma *ncycle-bij*: *bij* $(\text{ncycle } n)$

by (*rule* *permutes-bij*[*OF ncycle-permutes*])

The closed form for the iterates of the n -cycle.

lemma *ncycle-funpow*: $i < n \implies (\text{ncycle } n \text{ } \frown^k) i = (i + k) \bmod n$
by (*induction* k ; *simp* *add: mod-Suc-eq ncycle-def*)

context

fixes $n::\text{nat}$

begin

interpretation S : *Group* $S_n n$ ($\circ$) *id*

by (*rule* *Group-Sn*)

Every permutation of $\{0..<n\}$ is a product of transpositions, so the transpositions generate $S_n n$. This follows directly from the tailored induction rule *permutes-induct*.

lemma *Sn-generated-by-transpositions*:

Group.generate ($S_n n$) ($\circ$) *id* (*transpositions* n) = $S_n n$

proof

show $S.\text{generate} (\text{transpositions } n) \subseteq S_n n$

using *S.generate-subset-carrier* **by** *simp*

next

show $S_n n \subseteq S.\text{generate} (\text{transpositions } n)$

proof

fix p **assume** $p \in S_n n$

then have p *permutes* $\{0..<n\}$ *finite* $\{0..<n\}$ **by** (*simp-all* *add: Sn-def*)

then show $p \in S.\text{generate} (\text{transpositions } n)$

proof (*induction* p *rule: permutes-induct*)

case *id*

show *?case*

using *S.generate.unit* **by** *blast*

next

case (*swap* a b p)

then obtain *Transposition.transpose* a $b \in \text{transpositions } n$

Transposition.transpose a $b \in S_n n$

using *transpositions-def* *transpositions-subset-Sn* **by** *fastforce*

then show *?case*

by (*metis* *Int-iff* *S.generate.comp* *S.generate.incl* *swap.IH*)

qed

qed

qed

24.5 Conjugating a transposition

Conjugating a transposition by a bijection transposes the images of the two points.

lemma *transpose-conjug*:


```

assumes bij q
shows  $q \circ \text{Transposition.transpose } a \ b \circ \text{inv } q = \text{Transposition.transpose } (q \ a) \ (q \ b)$ 
proof –
  have  $\text{Transposition.transpose } (q \ a) \ (q \ b) \circ q = q \circ \text{Transposition.transpose } a \ b$ 
    using assms by (simp add: bij-def transpose-comp-eq)
  then show ?thesis
    by (metis assms bij-betw-inv-into bij-is-inj o-inv-o-cancel)
qed

```

24.6 The symmetric group is generated by a transposition and a cycle

The group power in S_n (n (composition)) is just function iteration.

lemma *Sn-power-funpow*: $\text{Monoid.power } (\circ) \ \text{id} \ (g :: \text{nat} \Rightarrow \text{nat}) \ k = g \ \frown \ k$
by (*metis* *S.power-def* *comp-funpow* *comp-id*)

In S_n the group inverse is the functional inverse.

lemma *Sn-inverse-eq*:
assumes $p \in S_n$
shows $\text{Monoid.inverse } (S_n) \ (\circ) \ \text{id} \ p = \text{inv } p$
using *S.invertible-left-inverse2* *Sn-iff* *assms* *permutes-inv* *permutes-inv-o(1)* **by** *fast-force*

lemma *t01-in-Sn*: $1 < n \implies \text{Transposition.transpose } 0 \ 1 \in S_n$
by (*simp* *add*: *Sn-def* *permutes-swap-id*)

Conjugating $\text{Transposition.transpose } 0 \ 1$ by the k -th power of the n -cycle yields the adjacent transposition $\text{Transposition.transpose } (k \bmod n) \ (\text{Suc } k \bmod n)$.

lemma *adjacent-transpose-conjug*:
assumes $k < n \ 1 < n$
shows $(\text{ncycle } n \ \frown \ k) \circ \text{Transposition.transpose } 0 \ 1 \circ \text{inv } (\text{ncycle } n \ \frown \ k)$
 $= \text{Transposition.transpose } (k \bmod n) \ (\text{Suc } k \bmod n)$
proof –
have $(\text{ncycle } n \ \frown \ k) \circ \text{Transposition.transpose } 0 \ 1 \circ \text{inv } (\text{ncycle } n \ \frown \ k)$
 $= \text{Transposition.transpose } ((\text{ncycle } n \ \frown \ k) \ 0) \ ((\text{ncycle } n \ \frown \ k) \ 1)$
using *ncycle-bij* *bij-fn* *transpose-conjug* **by** *metis*
then show *?thesis*
by (*metis* *One-nat-def* *Suc-lessD* *add-0* *assms(2)* *ncycle-funpow* *plus-1-eq-Suc*)
qed

Hence each adjacent transposition lies in the subgroup generated by $\text{Transposition.transpose } 0 \ 1$ and the n -cycle.

lemma *adjacent-in-generate*:
assumes $k < n \ 1 < n$
defines $S \equiv \{ \text{Transposition.transpose } 0 \ 1, \text{ncycle } n \}$
shows $\text{Transposition.transpose } (k \bmod n) \ (\text{Suc } k \bmod n) \in \text{Group.generate } (S_n \ n)$
 $(\circ) \ \text{id} \ S$
proof –

```

have t01: Transposition.transpose 0 1 ∈ S.generate S
  using Group.generate-incl S.Group-axioms S-def assms(2) t01-in-Sn by fastforce
have nc: ncycle n ∈ S.generate S
  using ncycle-in-Sn by (intro S.generate.incl) (simp add: S-def)
have powk: S.power (ncycle n) k ∈ S.generate S using nc by (rule S.generate-funpow)
have powkSn: S.power (ncycle n) k ∈ Sn n using powk S.generate-subset-carrier
by blast
have conj: S.power (ncycle n) k ∘ Transposition.transpose 0 1 ∘ S.inverse (S.power
(ncycle n) k) ∈ S.generate S
  using powk t01 by (meson S.generate.simps)
have S.inverse (S.power (ncycle n) k) = inv (ncycle n  $\smile$  k)
  using Sn-power-funpow powkSn Sn-inverse-eq by simp
with conj Sn-power-funpow adjacent-transpose-conjug[OF assms(1,2)]
show ?thesis by simp
qed

```

Building from adjacent transpositions, every transposition $\text{transpose } i \ j$ with $i < j < n$ lies in the subgroup.

lemma *transpose-extend*:

```

assumes a ≠ b b ≠ c a ≠ c
shows Transposition.transpose a c = Transposition.transpose a b ∘ Transposition.transpose
b c ∘ Transposition.transpose a b
  using assms by (simp add: transpose-comp-triple)

```

lemma *all-transpose-in-generate-aux*:

```

assumes 1 < n
defines S ≡ { Transposition.transpose 0 1, ncycle n }
shows ∀ i. i + Suc d < n ⟶ Transposition.transpose i (i + Suc d) ∈ Group.generate
(Sn n) (∘) id S
proof (induction d)
case 0
show ?case
  unfolding S-def
  using adjacent-in-generate[OF - assms(1)]
  by (metis One-nat-def Suc-eq-plus1 Suc-lessD mod-less)
next
case (Suc d)
show ?case
proof (intro allI impI)
fix i assume i: i + Suc (Suc d) < n
let ?b = i + Suc d and ?c = i + Suc (Suc d)
have §: i + Suc d < n using i by simp
then have ib: Transposition.transpose i ?b ∈ S.generate S
  using Suc.IH unfolding S-def by blast
have ?b mod n = ?b Suc ?b mod n = Suc ?b using i by auto
then have §: Transposition.transpose ?b (Suc ?b) ∈ S.generate S
  using adjacent-in-generate[OF - assms(1)] unfolding S-def by (metis §)
have Transposition.transpose i ?c
  = Transposition.transpose i ?b ∘ Transposition.transpose ?b ?c ∘ Transposi-

```

```

tion.transpose i ?b
  by (rule transpose-extend) auto
  with ib § show Transposition.transpose i ?c ∈ S.generate S
    by (auto intro: S.generate.comp simp: o-assoc)
qed
qed

lemma transpose-in-generate:
  assumes a < n b < n 1 < n
  shows Transposition.transpose a b ∈ Group.generate (Sn n) (○) id {Transposition.transpose
0 1, ncycle n}
proof (cases a=b)
  case True
  then show ?thesis
    by (simp add: S.generate.unit)
next
  case False
  then show ?thesis
    using assms add-Suc-right less-natE all-transpose-in-generate-aux
    by (metis (no-types, lifting) linorder-not-le order-less-le transpose-commute)
qed

```

The canonical generation theorem: a transposition of two adjacent points together with the standard n -cycle generate the whole symmetric group.

```

theorem Sn-generated-canonical:
  assumes 1 < n
  shows Group.generate (Sn n) (○) id {Transposition.transpose 0 1, ncycle n} = Sn
n
proof
  show S.generate {Transposition.transpose 0 1, ncycle n} ⊆ Sn n
    using S.generate-subset-carrier by simp
next
  have sub: S.generate (transpositions n) ⊆ S.generate {Transposition.transpose 0 1,
ncycle n}
    using transpose-in-generate[OF - - assms]
    by (intro S.generate-minimal S.generate-is-subgroup) (auto simp: transposi-
tions-def)
  then show Sn n ⊆ S.generate {Transposition.transpose 0 1, ncycle n}
    using Sn-generated-by-transpositions by simp
qed

```

24.7 A transposition and a p -cycle generate S_p for prime p

For prime p , the standard p -cycle together with *any* transposition generates S_n . The primality is essential: it guarantees that the conjugates of the transposition by powers of the cycle have supports forming a single orbit covering all points, from which every transposition can be recovered.

Conjugating *Transposition.transpose a b* by the k -th power of the n -cycle shifts both points by k (modulo n); the result stays in the subgroup generated by the cycle and the transposition.

```

lemma ncycle-conj-transpose-in-generate:
  assumes  $a < n \ b < n \ 1 < n$ 
  defines  $S \equiv \{\text{ncycle } n, \text{Transposition.transpose } a \ b\}$ 
  shows  $\text{Transposition.transpose } ((a + k) \bmod n) ((b + k) \bmod n) \in \text{Group.generate } (S \ n \ n) (\circ) \text{ id } S$ 
proof –
  have  $\text{tabSn}: \text{Transposition.transpose } a \ b \in S \ n \ n$ 
    using assms by (auto simp: Sn-def permutes-swap-id)
  have  $\text{tab}: \text{Transposition.transpose } a \ b \in S.\text{generate } S$ 
    using  $\text{tabSn}$  by (intro S.generate.incl (simp add: S-def))
  have  $\text{nc}: \text{ncycle } n \in S.\text{generate } S$ 
    using ncycle-in-Sn by (intro S.generate.incl (simp add: S-def))
  have  $\text{invk}: S.\text{inverse } (S.\text{power } (\text{ncycle } n) \ k) \in S.\text{generate } S$ 
    using  $\text{nc}$  by (intro S.generate.inv S.generate-funpow)
  have  $\text{conj}: S.\text{power } (\text{ncycle } n) \ k \circ \text{Transposition.transpose } a \ b \circ S.\text{inverse } (S.\text{power } (\text{ncycle } n) \ k) \in S.\text{generate } S$ 
    by (simp add: S.generate.comp S.generate-funpow invk nc tab)
  have  $\text{ineq}: S.\text{inverse } (S.\text{power } (\text{ncycle } n) \ k) = \text{inv } (\text{ncycle } n \ \frown \ k)$ 
    using S.power-closed Sn-inverse-eq Sn-power-funpow ncycle-in-Sn by force
  have  $\text{ncycle } n \ \frown \ k \circ \text{Transposition.transpose } a \ b \circ \text{inv } (\text{ncycle } n \ \frown \ k)$ 
     $= \text{Transposition.transpose } ((\text{ncycle } n \ \frown \ k) \ a) ((\text{ncycle } n \ \frown \ k) \ b)$ 
    using ncycle-bij by (intro transpose-conjug bij-fn)
  with ncycle-funpow [of - n k] assms
  have  $\text{ncycle } n \ \frown \ k \circ \text{Transposition.transpose } a \ b \circ \text{inv } (\text{ncycle } n \ \frown \ k)$ 
     $= \text{Transposition.transpose } ((a + k) \bmod n) ((b + k) \bmod n)$  by presburger
  with conj Sn-power-funpow ineq show ?thesis by simp
qed

```

If a subgroup contains two transpositions sharing a point, it contains their “sum”.

```

lemma transpose-chain:
  assumes  $H: \text{Subgroup } H \ (S \ n \ n) (\circ) \text{ id}$ 
  and  $\text{Transposition.transpose } x \ y \in H$ 
  and  $\text{Transposition.transpose } y \ z \in H$ 
  and  $x \neq z \ y \neq z$ 
  shows  $\text{Transposition.transpose } x \ z \in H$ 
proof –
  interpret  $H: \text{Subgroup } H \ S \ n \ n (\circ) \text{ id}$  by (rule H)
  show ?thesis
    using assms by (metis H.sub-composition-closed transpose-comp-triple)
qed

```

end

```

context
  fixes  $p::\text{nat}$ 
  assumes  $p: \text{prime } p$ 

```

begin

interpretation G : *Group* $Sn\ p\ (\circ)\ id$
by (*rule* *Group-Sn*)

lemma $p\text{-gt}1$: $1 < p$
using p **by** (*simp* *add*: *prime-nat-iff*)

Number-theoretic facts about the stride d (the gap of the transposition).

lemma *coprime-prime-stride*:
assumes $0 < d\ d < p$
shows *coprime* $p\ d$
by (*simp* *add*: *assms* *nat-dvd-not-less* p *prime-imp-coprime*)

lemma *kd-mod-zero-iff*:
assumes $0 < d\ d < p$
shows $(k * d) \bmod p = 0 \longleftrightarrow p\ \text{dvd}\ k$
by (*simp* *add*: *coprime-dvd-mult-left-iff* *coprime-prime-stride* *assms* *mod-eq-0-iff-dvd*)

lemma *stride-endpoints-distinct*:
assumes $0 < d\ d < p$
shows $(k * d) \bmod p \neq (Suc\ k * d) \bmod p$
by (*metis* *assms* *add-implies-diff* *le-add2* *mod-eq-dvd-iff-nat* *mult-Suc* *nat-dvd-not-less*)

Each “stride edge” $transpose\ (k * d \bmod p)\ (Suc\ k * d \bmod p)$ lies in the subgroup generated by the cycle and $Transposition.transpose\ 0\ d$.

lemma *stride-edge*:
assumes $d1$: $0 < d$ **and** $d2$: $d < p$
defines $S \equiv \{ncycle\ p,\ Transposition.transpose\ 0\ d\}$
shows $Transposition.transpose\ ((k * d) \bmod p)\ ((Suc\ k * d) \bmod p) \in Group.generate\ (Sn\ p)\ (\circ)\ id\ S$
using $S\text{-def}$ *Symmetric-Generation.ncycle-conj-transpose-in-generate* $d2$
by (*metis* *add-0* *mult-Suc* p *prime-gt-0-nat* *prime-nat-iff*)

Chaining the stride edges, every $transpose\ 0\ (k * d \bmod p)$ (for k not a multiple of p) lies in the subgroup.

lemma *stride-walk*:
fixes $d :: nat$
assumes $d1$: $0 < d$ **and** $d2$: $d < p$
defines $S \equiv \{ncycle\ p,\ Transposition.transpose\ 0\ d\}$
shows $\neg\ p\ \text{dvd}\ k \implies Transposition.transpose\ 0\ ((k * d) \bmod p) \in Group.generate\ (Sn\ p)\ (\circ)\ id\ S$
proof (*induction* k)
case 0
then show *?case* **by** *simp*
next
case $(Suc\ k)$

```

have edge: Transposition.transpose ((k * d) mod p) ((Suc k * d) mod p) ∈ G.generate
S
  using stride-edge[OF d1 d2] unfolding S-def by blast
have suc-ne0: (Suc k * d) mod p ≠ 0
  using Suc.prems d1 d2 kd-mod-zero-iff by blast
show ?case
proof (cases p dvd k)
  case True
  then show ?thesis by (metis edge dvd-imp-mod-0 dvd-mult2)
next
  case False
  have ih: Transposition.transpose 0 ((k * d) mod p) ∈ G.generate S
  using Suc.IH[OF False] unfolding S-def by blast
  have kne0: (k * d) mod p ≠ 0
  using False d1 d2 kd-mod-zero-iff by blast
  have ne: (k * d) mod p ≠ (Suc k * d) mod p
  using d1 d2 stride-endpoints-distinct by blast
  show ?thesis
  using transpose-chain[OF G.generate-is-subgroup ih edge] kne0 suc-ne0 ne
  unfolding S-def by arg0
qed
qed

```

```

lemma pymod: (p * y + 1) mod p = (1::nat) mod p
by (metis mod-mult-self3 mult.commute)

```

Since d is coprime to the prime p , some multiple of d is congruent to 1, so *Transposition.transpose* 0 1 is reached.

```

lemma stride-reaches-one:
  fixes d :: nat
  assumes d1: 0 < d and d2: d < p
  shows ∃ k. (k * d) mod p = 1 ∧ ¬ p dvd k
proof –
  have cop: coprime d p using coprime-prime-stride[OF d1 d2] by (simp add: coprime-commute)
  obtain x y where xy: d * x = p * y + 1 using coprime-bezout-strong[OF cop]
  p-gt1 by auto
  have one: (x * d) mod p = 1
  by (metis mod-less mult.commute p-gt1 pymod xy)
  with one show ?thesis by auto
qed

```

```

lemma t01-in-stride-generate:
  assumes 0 < d < p
  shows Transposition.transpose 0 1 ∈ Group.generate (Sn p) (◦) id {ncycle p, Transposition.transpose 0 d}
  by (metis assms stride-reaches-one stride-walk)

```

The cycle and a stride- d transposition generate $Sn\ p$.

theorem *Sn-generated-cycle-transpose-0d*:
fixes $d :: \text{nat}$
assumes $d1: 0 < d$ **and** $d2: d < p$
shows $\text{Group.generate } (Sn\ p) (\circ) \text{id } \{\text{ncycle } p, \text{Transposition.transpose } 0\ d\} = Sn\ p$
proof
have $nc: \text{ncycle } p \in G.\text{generate } \{\text{ncycle } p, \text{Transposition.transpose } 0\ d\}$
using *ncycle-in-Sn* **by** (intro $G.\text{generate.incl}$) *simp*
have $sub: G.\text{generate } \{\text{Transposition.transpose } 0\ 1, \text{ncycle } p\} \subseteq G.\text{generate } \{\text{ncycle } p, \text{Transposition.transpose } 0\ d\}$
using *t01-in-stride-generate*[OF $d1\ d2$] *nc*
by (intro $G.\text{generate-minimal } G.\text{generate-is-subgroup}$) *auto*
then show $Sn\ p \subseteq G.\text{generate } \{\text{ncycle } p, \text{Transposition.transpose } 0\ d\}$
using *Sn-generated-canonical*[OF $p\text{-gt}1$] **by** *simp*
qed (use $G.\text{generate-subset-carrier}$ **in** *simp*)

lemma *sum-pa-bounds*:
fixes $a\ b :: \text{nat}$
assumes $a < p$ $b < p$ $a \neq b$
shows $0 < (b + (p - a)) \bmod p$
proof (*rule ccontr*)
have $*$: $0 < b + (p - a)$ $b + (p - a) < 2 * p$ $b + (p - a) \neq p$ **using** *assms*
by *auto*
assume $\neg 0 < (b + (p - a)) \bmod p$
then obtain k **where** $b + (p - a) = p * k$ **by** (*auto simp: dvd-def*)
with $*$ **show** *False* **by** *auto*
qed

The main result of this section: for prime p , the standard p -cycle and *any* transposition generate $Sn\ p$.

theorem *Sn-generated-ncycle-transpose*:
fixes $a\ b :: \text{nat}$
assumes $a: a < p$ **and** $b: b < p$ **and** $ab: a \neq b$
shows $\text{Group.generate } (Sn\ p) (\circ) \text{id } \{\text{ncycle } p, \text{Transposition.transpose } a\ b\} = Sn\ p$
proof
define d **where** $d \equiv (b + (p - a)) \bmod p$
obtain $d: 0 < d < p$ **unfolding** *d-def* **using** *p-gt1*
using $a\ ab\ b$ *sum-pa-bounds* **by** (*metis mod-less-divisor not-gr0 not-less0*)
— Conjugating $\text{Transposition.transpose } a\ b$ by $(\text{ncycle } p)^p - a$ maps a to 0 , exhibiting $\text{Transposition.transpose } 0\ d$ inside the subgroup.
have $t0d: \text{Transposition.transpose } 0\ d \in G.\text{generate } \{\text{ncycle } p, \text{Transposition.transpose } a\ b\}$
using *ncycle-conj-transpose-in-generate*[OF $a\ b\ p\text{-gt}1$, of $p - a$] *a d-def* **by** *auto*
have $nc: \text{ncycle } p \in G.\text{generate } \{\text{ncycle } p, \text{Transposition.transpose } a\ b\}$
using *ncycle-in-Sn* **by** (intro $G.\text{generate.incl}$) *simp*
have $sub: G.\text{generate } \{\text{ncycle } p, \text{Transposition.transpose } 0\ d\} \subseteq G.\text{generate } \{\text{ncycle } p, \text{Transposition.transpose } a\ b\}$
using *t0d nc* **by** (intro $G.\text{generate-minimal } G.\text{generate-is-subgroup}$) *auto*
then show $Sn\ p \subseteq G.\text{generate } \{\text{ncycle } p, \text{Transposition.transpose } a\ b\}$
using *Sn-generated-cycle-transpose-0d*[OF d] **by** *simp*

qed (use *G.generate-subset-carrier* in simp)

24.8 An arbitrary cycle and transposition generate S_p

In S_n n the group conjugation specialises to the familiar $g \circ x \circ \text{inv } g$.

lemma *Sn-conjugation-eq*:

assumes $g: g \in S_n \ n$ **and** $x: x \in S_n \ n$
shows $\text{Group.conjugation } (S_n \ n) \ (\circ) \ \text{id } g \ x = g \circ x \circ \text{inv } g$
by (simp add: *Group.conjugation-def* *Group-Sn* *Sn-inverse-eq* $g \ x$)

Transporting *Sn-generated-ncycle-transpose* along conjugacy: for prime p , any conjugate $q \circ \text{ncycle } p \circ \text{inv } q$ of the standard p -cycle, together with the conjugate $q \circ \text{Transposition.transpose } a \ b \circ \text{inv } q$ of a transposition, generates $S_n \ p$.

lemma *Sn-generated-conj-cycle-transpose*:

fixes $q :: \text{nat} \Rightarrow \text{nat}$ **and** $a \ b :: \text{nat}$
assumes $q: q \in S_n \ p$ **and** $a: a < p$ **and** $b: b < p$ **and** $ab: a \neq b$
shows $\text{Group.generate } (S_n \ p) \ (\circ) \ \text{id}$
 $\{q \circ \text{ncycle } p \circ \text{inv } q, q \circ \text{Transposition.transpose } a \ b \circ \text{inv } q\} = S_n \ p$

proof –

interpret $G: \text{Group } S_n \ p \ (\circ) \ \text{id}$ **by** (rule *Group-Sn*)
have $qp: q \text{ permutes } \{0..<p\}$ **using** q **by** (simp add: *Sn-def*)
define S **where** $S = \{\text{ncycle } p, \text{Transposition.transpose } a \ b\}$
have $\text{genS}: G.\text{generate } S = S_n \ p$
unfolding *S-def* **by** (rule *Sn-generated-ncycle-transpose*[*OF* $a \ b \ ab$])
have $S_{\text{subSn}}: S \subseteq S_n \ p$
unfolding *S-def* **using** *ncycle-in-Sn* $a \ b$ **by** (auto simp: *Sn-def* *permutes-swap-id*)
have $G.\text{generate } \{q \circ \text{ncycle } p \circ \text{inv } q, q \circ \text{Transposition.transpose } a \ b \circ \text{inv } q\}$
 $= G.\text{conjugation } q \ ' G.\text{generate } S$
using $G.\text{conjugation-generate-eq}$ [*OF* q] $G.\text{conjugation-def}$ *S-def* *Sn-inverse-eq* S_{subSn}
 q **by** force
also have $\dots = S_n \ p$ **using** $G.\text{conjugation-bij}$ [*OF* q] **by** (simp add: $\text{genS bij-betw-imp-surj-on}$)
finally show *?thesis* .

qed

24.9 An order- p element of S_p is a p -cycle

The combinatorial counterpart of the previous result: every element of $S_n \ p$ of order p (for p prime) is conjugate to the standard cycle, hence is itself a p -cycle. Together with *Sn-generated-conj-cycle-transpose* this lets Cauchy's theorem (which only yields *some* order- p element) feed the generation argument for the quintic.

Iterating g is periodic with period dividing n once $g^n = \text{id}$.

lemma *funpow-mod-period*:

assumes $(g \ \frown^n \ n) = \text{id}$
shows $(g \ \frown^{(k \bmod n)}) = (g \ \frown^k)$
by (metis (no-types, lifting) ext assms funpow-mod-eq id-apply)

If g has order p (prime) and moves x , then no smaller positive power fixes x :
the only powers fixing x are the multiples of p .

lemma *cycle-period-prime*:
assumes p : prime p **and** gp : $(g \text{ } \sim p) = id$ **and** gx : $g\ x \neq x$
and d : $(g \text{ } \sim d)\ x = x$
shows $d \bmod p = 0$
proof (rule ccontr)
assume $d \bmod p \neq 0$
have $dmod$: $(g \text{ } \sim (d \bmod p))\ x = x$
using *funpow-mod-period[OF gp, of d]* **by** *simp*
obtain k **where** k : $(k * (d \bmod p)) \bmod p = 1$
using $\langle d \bmod p \neq 0 \rangle$ *p stride-reaches-one*
by (*meson mod-less-divisor not-gr-zero prime-gt-0-nat*)
then have $(g \text{ } \sim ((k * (d \bmod p)) \bmod p))\ x = x$
using *funpow-mod-period[OF gp]* **by** (*metis dmod funpow-mod-eq mod-mult-self2-is-0 mod-self*)
with gx **show** *False* **using** k **by** *simp*
qed

The conjugator: the map sending i to $g^i\ x$ on $\{0..<p\}$ and fixing the rest.

definition $qmap :: (nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat \Rightarrow nat$
where $qmap\ g\ x\ i = (if\ i < p\ then\ (g \text{ } \sim i)\ x\ else\ i)$

context
fixes $g :: nat \Rightarrow nat$ **and** $x :: nat$
assumes gSn : $g \in Sn\ p$ **and** gp : $(g \text{ } \sim p) = id$
and gx : $g\ x \neq x$ **and** xp : $x < p$
begin

lemma *g-funpow-lt*: $i < p \implies (g \text{ } \sim i)\ x < p$
using gSn *permutes-in-funpow-image xp* **by** (*fastforce simp: Sn-def permutes-funpow*)

lemma *qmap-inj-aux*:
assumes ij : $i < p\ j < p\ i \leq j$ **and** eq : $(g \text{ } \sim i)\ x = (g \text{ } \sim j)\ x$
shows $i = j$
proof –
have $bij-i$: $bij\ (g \text{ } \sim i)$ **using** $Sn-def\ gSn\ permutes-bij$ **by** *fastforce*
have $(g \text{ } \sim i)\ x = (g \text{ } \sim i)\ ((g \text{ } \sim (j - i))\ x)$
by (*metis Sn-iff bij-betw-def eq funpow-diff gSn ij(3) permutes-bij*)
then have $(j - i) \bmod p = 0$
using *cycle-period-prime[OF p gp gx]* **by** (*simp add: bij-i bij-is-inj inj-eq*)
then show $i = j$ **using** ij **by** *auto*
qed

lemma *qmap-inj-on*: $inj-on\ (qmap\ g\ x)\ \{0..<p\}$
proof (rule *inj-onI*)
fix $i\ j$ **assume** $i \in \{0..<p\}\ j \in \{0..<p\}$ $qmap\ g\ x\ i = qmap\ g\ x\ j$
then have $i < p\ j < p\ (g \text{ } \sim i)\ x = (g \text{ } \sim j)\ x$
by (*auto simp: qmap-def*)

```

then show  $i = j$ 
  by (metis nat-le-linear qmap-inj-aux)
qed

lemma qmap-image:  $qmap\ g\ x\ ' \{0..<p\} = \{0..<p\}$ 
proof (rule endo-inj-surj)
  show finite  $\{0..<p\}$  by simp
  show  $qmap\ g\ x\ ' \{0..<p\} \subseteq \{0..<p\}$  using g-funpow-lt by (auto simp: qmap-def)
  show inj-on  $(qmap\ g\ x)\ \{0..<p\}$  by (rule qmap-inj-on)
qed

lemma qmap-permutes:  $qmap\ g\ x$  permutes  $\{0..<p\}$ 
  using qmap-inj-on qmap-image
  by (intro bij-imp-permutes) (auto simp: bij-betw-def qmap-def)

lemma qmap-in-Sn:  $qmap\ g\ x \in Sn\ p$ 
  using qmap-permutes by (simp add: Sn-def)

lemma g-qmap-eq:
  assumes  $i < p$ 
  shows  $g\ (qmap\ g\ x\ i) = qmap\ g\ x\ (ncycle\ p\ i)$ 
  using funpow-mod-period[OF gp, of Suc i] assms by (simp add: qmap-def ncycle-def)

lemma g-comp-qmap:  $g \circ qmap\ g\ x = qmap\ g\ x \circ ncycle\ p$ 
  using g-qmap-eq gSn by (auto simp: Sn-def permutes-def ncycle-def qmap-def)

lemma g-conj-ncycle:  $g = qmap\ g\ x \circ ncycle\ p \circ inv\ (qmap\ g\ x)$ 
  by (metis comp-assoc comp-id g-comp-qmap permutes-inv-o(1) qmap-permutes)

end

theorem order-p-elt-conj-ncycle:
  fixes  $g :: nat \Rightarrow nat$ 
  assumes gSn:  $g \in Sn\ p$  and gp:  $(g \smallfrown p) = id$  and gne:  $g \neq id$ 
  shows  $\exists q \in Sn\ p. g = q \circ ncycle\ p \circ inv\ q$ 
proof -
  obtain  $x$  where  $x: g\ x \neq x$  using gne by (auto simp: fun-eq-iff)
  with gSn have xp:  $x < p$ 
    by (auto simp: Sn-def permutes-def)
  show ?thesis
    using gSn g-conj-ncycle gp qmap-in-Sn x xp by blast
qed

```

The same conclusion phrased through the group's *Monoid.element-order*: an element of $Sn\ p$ of order p is a conjugate of the standard cycle.

```

theorem order-p-elt-conj-ncycle-order:
  fixes  $g :: nat \Rightarrow nat$ 
  assumes gSn:  $g \in Sn\ p$ 
  and ord: Monoid.element-order  $(\circ)\ id\ g = p$ 

```

```

shows  $\exists q \in Sn$   $p. g = q \circ ncycle\ p \circ inv\ q$ 
proof –
  have  $gp: (g \smallfrown p) = id$ 
    using G.power-element-order Sn-power-funpow finite-Sn gSn ord by force
  have  $gne: g \neq id$ 
    using ord p by force
  show ?thesis by (rule order-p-elt-conj-ncycle[OF gSn gp gne])
qed

end

end

```

25 Composition factors of the symmetric and alternating groups

```

theory Sn-Composition-Factors
imports Composition-Factors-Solvable Sym-Not-Solvable Symmetric-Generation
begin

```

These are *consistency* checks, not new mathematics. The composition-series material of *Composition-Factors-Solvable* was contributed as a large block and merged on structural evidence (it builds, contains no *sorry*, and cites the existing solvability lemmas). The checks below test its *content* against results proved independently elsewhere in this development, on the principle that a vacuous or too-weak definition would let a false statement through.

The sharpest available cross-check is *Sn-not-solvable*: this development already proves, by an argument that knows nothing of composition series, that S_n is not solvable for $n \geq 5$. The new criterion says solvability is equivalent to all composition factors being abelian. The two must agree, and forcing them to agree pins down real content: if *composition-series.abelian-factors* were vacuously true — say because *series-factor* returned something degenerate — then S_5 would come out solvable and contradict a theorem already in the library.

25.1 A finite group has a composition series, and S_5 in particular

First that the existence theorem applies at all to a concrete group of interest.

```

lemma Sn-has-composition-series:
   $\exists H\ m. \text{composition-series } (Sn\ n) (\circ)\ id\ H\ m$ 
proof (rule finite-group-has-composition-series-ex)
  show Group  $(Sn\ n) (\circ)\ id$  by (rule Group-Sn)
  show finite  $(Sn\ n)$  by (rule finite-Sn)
qed

```

25.2 The criterion is consistent with S_n being unsolvable

The main cross-check. Every composition series of S_n ($n \geq 5$) has a factor that is *not* abelian. This is not proved by exhibiting one — it is forced, by combining the new criterion with the independently proved unsolvability of S_n . It therefore fails if the new criterion is too weak in either direction.

theorem *Sn-composition-factors-not-all-abelian:*

assumes $n5: 5 \leq n$ **and** *series: composition-series* ($S_n\ n$) ($\circ$) *id* $H\ m$

shows $\neg$ *composition-series.abelian-factors* ($\circ$) *id* $H\ m$

proof

interpret S : *composition-series* $S_n\ n$ ($\circ$) *id* $H\ m$ **by** (*rule series*)

assume S .*abelian-factors*

— The criterion would make S_n solvable...

then have *Group.solvable* ($S_n\ n$) ($\circ$) *id* **using** S .*solvable-iff-abelian-factors* **by** *blast*

— ...contradicting *Sn-not-solvable*, proved independently of all this.

with *Sn-not-solvable*[*OF* $n5$] **show** *False* **by** *simp*

qed

The same for the alternating groups, via *An-not-solvable*.

theorem *An-composition-factors-not-all-abelian:*

assumes $n5: 5 \leq n$ **and** *series: composition-series* ($A_n\ n$) ($\circ$) *id* $H\ m$

shows $\neg$ *composition-series.abelian-factors* ($\circ$) *id* $H\ m$

proof

interpret A : *composition-series* $A_n\ n$ ($\circ$) *id* $H\ m$ **by** (*rule series*)

assume A .*abelian-factors*

then have *Group.solvable* ($A_n\ n$) ($\circ$) *id* **using** A .*solvable-iff-abelian-factors* **by** *blast*

with *An-not-solvable*[*OF* $n5$] **show** *False* **by** *simp*

qed

And the existence and criterion theorems combine: some composition series of S_5 exists, and every one of them has a non-abelian factor. If *composition-series* were unsatisfiable these two would not sit together — the first would be vacuous.

corollary *S5-has-series-with-a-nonabelian-factor:*

$\exists H\ m.$ *composition-series* ($S_n\ 5$) ($\circ$) *id* $H\ m$

$\wedge \neg$ *composition-series.abelian-factors* ($\circ$) *id* $H\ m$

proof —

obtain $H\ m$ **where** *series: composition-series* ($S_n\ 5$) ($\circ$) *id* $H\ m$

using *Sn-has-composition-series*[*of* 5] **by** *blast*

then have $\neg$ *composition-series.abelian-factors* ($\circ$) *id* $H\ m$

by (*rule Sn-composition-factors-not-all-abelian*[*OF* *order-refl*])

with *series* **show** *?thesis* **by** *blast*

qed

end

26 Subfields and Field Homomorphisms (Type-Class Setting)

```
theory Subfield
  imports HOL-Computational-Algebra.Polynomial
begin
```

We develop field-extension theory generically over an ambient field $'a$, taking a *subfield* to be a subset closed under the field operations..)

26.1 The subfield generated by a set

```
inductive-set generate-field :: 'a :: field set  $\Rightarrow$  'a set for K :: 'a set where
  generate-field-base [intro]:    $x \in K \Rightarrow x \in \text{generate-field } K$ 
| generate-field-0 [intro]:       $0 \in \text{generate-field } K$ 
| generate-field-1 [intro]:       $1 \in \text{generate-field } K$ 
| generate-field-minus [intro]:   $x \in \text{generate-field } K \Rightarrow -x \in \text{generate-field } K$ 
| generate-field-add [intro]:     $x \in \text{generate-field } K \Rightarrow y \in \text{generate-field } K \Rightarrow x + y \in \text{generate-field } K$ 
| generate-field-mult [intro]:    $x \in \text{generate-field } K \Rightarrow y \in \text{generate-field } K \Rightarrow x * y \in \text{generate-field } K$ 
| generate-field-inverse [intro]:  $x \in \text{generate-field } K \Rightarrow \text{inverse } x \in \text{generate-field } K$ 
```

```
lemma subset-generate-field:  $A \subseteq \text{generate-field } A$ 
  by (auto intro: generate-field-base)
```

26.2 The subfield locale

```
locale Subfield =
  fixes K :: 'a :: field set
  assumes zero-closed [intro, simp]:  $0 \in K$ 
  assumes one-closed [intro, simp]:  $1 \in K$ 
  assumes add-closed [intro]:  $x \in K \Rightarrow y \in K \Rightarrow x + y \in K$ 
  assumes uminus-closed [intro]:  $x \in K \Rightarrow -x \in K$ 
  assumes mult-closed [intro]:  $x \in K \Rightarrow y \in K \Rightarrow x * y \in K$ 
  assumes inverse-closed [intro]:  $x \in K \Rightarrow \text{inverse } x \in K$ 
begin
```

```
lemma diff-closed [intro]:  $x \in K \Rightarrow y \in K \Rightarrow x - y \in K$ 
  using add-closed[of  $x - y$ ] by auto
```

```
lemma divide-closed [intro]:  $x \in K \Rightarrow y \in K \Rightarrow x / y \in K$ 
  using mult-closed[of  $x$ , OF - inverse-closed[of  $y$ ]] by (simp add: field-simps)
```

```
lemma power-closed [intro]:  $x \in K \Rightarrow x ^ n \in K$ 
  by (induction n) auto
```

```
lemma of-nat-closed [intro]:  $\text{of-nat } n \in K$ 
  by (induction n) auto
```

```

lemma of-int-closed [intro]: of-int  $n \in K$ 
  by (induction n) auto

lemma sum-closed [intro]:  $(\bigwedge x. x \in A \implies f\ x \in K) \implies (\sum x \in A. f\ x) \in K$ 
  by (induction A rule: infinite-finite-induct) auto

lemma prod-closed [intro]:  $(\bigwedge x. x \in A \implies f\ x \in K) \implies (\prod x \in A. f\ x) \in K$ 
  by (induction A rule: infinite-finite-induct) auto

lemma cardK-gt1:
  assumes finite K shows card K > 1
proof –
  have pair:  $\{0, 1\} \subseteq K$  by auto
  with card-mono[OF assms pair] show ?thesis by simp
qed

end

lemma subfield-generate-field [intro]:
  fixes A :: 'a :: field set
  shows Subfield (generate-field A)
  by standard (auto intro: generate-field.intros)

lemma generate-field-least:
  assumes Subfield K  $A \subseteq K$ 
  shows generate-field A  $\subseteq K$ 
proof
  fix x assume x:  $x \in \text{generate-field } A$ 
  interpret Subfield K by fact
  from x show  $x \in K$ 
  by induction (use assms(2) in auto)
qed

lemma generate-field-mono:  $A \subseteq B \implies \text{generate-field } A \subseteq \text{generate-field } B$ 
  by (meson order.trans generate-field-least subfield-generate-field subset-generate-field)

lemma generate-field-mono':  $A \subseteq B \implies x \in \text{generate-field } A \implies x \in \text{generate-field } B$ 
  using generate-field-mono by blast

lemma generate-field-collapse [simp]:  $\text{generate-field } (\text{generate-field } X) = \text{generate-field } X$ 
  using generate-field-least subfield-generate-field subset-generate-field by blast

lemma generate-field-Un-collapse1 [simp]:
   $\text{generate-field } (\text{generate-field } A \cup B) = \text{generate-field } (A \cup B)$ 
proof
  show  $\text{generate-field } (A \cup B) \subseteq \text{generate-field } (\text{generate-field } A \cup B)$ 
  using subset-generate-field by (intro generate-field-mono) auto

```

```

thus generate-field (generate-field  $A \cup B$ )  $\subseteq$  generate-field ( $A \cup B$ )
by (metis generate-field-least le-sup-iff subfield-generate-field subset-generate-field)
qed

```

```

lemma generate-field-Un-collapse2 [simp]:
  generate-field ( $A \cup$  generate-field  $B$ ) = generate-field ( $A \cup B$ )
using generate-field-Un-collapse1 [of  $B$   $A$ ]
by (simp add: Un-commute del: generate-field-Un-collapse1)

```

26.3 Field homomorphisms on a subfield

```

locale field-hom-on = Subfield  $K$  for  $K :: 'a :: \text{field set} +$ 
  fixes  $f :: 'a \Rightarrow 'b :: \text{field}$ 
  assumes hom-0 [simp]:  $f\ 0 = 0$ 
  assumes hom-1 [simp]:  $f\ 1 = 1$ 
  assumes hom-add:  $x \in K \Longrightarrow y \in K \Longrightarrow f\ (x + y) = f\ x + f\ y$ 
  assumes hom-mult:  $x \in K \Longrightarrow y \in K \Longrightarrow f\ (x * y) = f\ x * f\ y$ 
begin

```

```

lemma hom-uminus:  $x \in K \Longrightarrow f\ (-x) = -f\ x$ 
using hom-add [of  $x - x$ ] by (simp add: uminus-closed add-eq-0-iff)

```

```

lemma hom-diff:  $x \in K \Longrightarrow y \in K \Longrightarrow f\ (x - y) = f\ x - f\ y$ 
using hom-add [of  $x - y$ ] hom-uminus [of  $y$ ] uminus-closed [of  $y$ ] by simp

```

```

lemma hom-inverse:  $x \in K - \{0\} \Longrightarrow f\ (\text{inverse } x) = \text{inverse } (f\ x)$ 
using hom-mult [of  $x$  inverse  $x$ ]
by (auto simp: uminus-closed add-eq-0-iff divide-simps divide-closed mult-ac)

```

```

lemma hom-inverse-all:  $x \in K \Longrightarrow f\ (\text{inverse } x) = \text{inverse } (f\ x)$ 
by (cases  $x = 0$ ) (auto simp: hom-inverse)

```

```

lemma hom-divide:  $x \in K \Longrightarrow y \in K - \{0\} \Longrightarrow f\ (x / y) = f\ x / f\ y$ 
using hom-mult [of  $x$  inverse  $y$ ] hom-inverse [of  $y$ ] inverse-closed [of  $y$ ]
by (simp add: field-simps)

```

```

lemma kernel-triv:
  assumes  $u \in K - \{0\} \ f\ u = 0$ 
  shows False
by (metis Diff-iff divide-self-if hom-1 hom-divide insert-iff one-neq-zero assms)

```

A field homomorphism is injective on its domain.

```

lemma inj-on: inj-on  $f\ K$ 
proof
  fix  $x\ y$  assume  $x \in K\ y \in K\ f\ x = f\ y$ 
  thus  $x = y$  using kernel-triv [of  $x - y$ ] by (auto simp: hom-diff diff-closed)
qed

```

```

lemma generate-field-subK:  $S \subseteq K \Longrightarrow \text{generate-field } S \subseteq K$ 
using Subfield-axioms by (rule generate-field-least)

```

The image under f of the subfield generated by a subset $S \subseteq K$ is contained in the subfield generated by the image $f \text{ ' } S$: the homomorphism laws carry each generation step over (here f need only be a homomorphism on K , and $\text{generate-field } S \subseteq K$ keeps every intermediate element in range).

```

lemma image-generate-field-subset:
  assumes  $S: S \subseteq K$ 
  shows  $f \text{ ' } \text{generate-field } S \subseteq \text{generate-field } (f \text{ ' } S)$ 
proof
  fix  $y$  assume  $y \in f \text{ ' } \text{generate-field } S$ 
  then obtain  $x$  where  $x: x \in \text{generate-field } S$  and  $yx: y = f x$  by blast
  from  $x$  have  $f x \in \text{generate-field } (f \text{ ' } S)$ 
  proof (induction rule: generate-field.induct)
    case (generate-field-minus  $x$ )
      then have  $x \in K$  using generate-field-subK  $S$  by auto
      then show  $?case$  using generate-field-minus.IH by (simp add: hom-uminus generate-field.intros)
    next
      case (generate-field-add  $x y$ )
      then have  $x \in K$   $y \in K$  using generate-field-subK  $S$  by auto
      then show  $?case$  using generate-field-add.IH by (simp add: hom-add generate-field.intros)
    next
      case (generate-field-mult  $x y$ )
      then have  $x \in K$   $y \in K$  using generate-field-subK  $S$  by auto
      then show  $?case$  using generate-field-mult.IH by (simp add: hom-mult generate-field.intros)
    next
      case (generate-field-inverse  $x$ )
      then have  $x \in K$  using generate-field-subK  $S$  by auto
      then show  $?case$  using generate-field-inverse.IH by (simp add: hom-inverse-all generate-field.intros)
  qed (auto simp: generate-field.intros)
  then show  $y \in \text{generate-field } (f \text{ ' } S)$  using  $yx$  by simp
qed

```

The image of a generated subfield is itself a subfield (of the target type).

```

lemma image-generate-field-subfield:
  assumes  $S: S \subseteq K$ 
  shows Subfield  $(f \text{ ' } \text{generate-field } S)$ 
proof
  show  $0 \in f \text{ ' } \text{generate-field } S$  using hom-0 by (metis generate-field-0 image-eqI)
  show  $1 \in f \text{ ' } \text{generate-field } S$  using hom-1 by (metis generate-field-1 image-eqI)
next
  fix  $x y$  assume  $x \in f \text{ ' } \text{generate-field } S$   $y \in f \text{ ' } \text{generate-field } S$ 
  then obtain  $a b$  where  $a: a \in \text{generate-field } S$   $x = f a$ 
    and  $b: b \in \text{generate-field } S$   $y = f b$  by blast
  have  $aK: a \in K$  and  $bK: b \in K$  using  $a b$  generate-field-subK  $S$  by auto
  show  $x + y \in f \text{ ' } \text{generate-field } S$ 
    by (metis a aK b bK generate-field.generate-field-add hom-add image-eqI)

```



```

show  $x * y \in f \text{ ' generate-field } S$ 
  by (metis  $a \ aK \ b \ bK$  generate-field.generate-field-mult hom-mult image-iff)
next
  fix  $x$  assume  $x \in f \text{ ' generate-field } S$ 
  then obtain  $a$  where  $a: a \in \text{generate-field } S \ x = f \ a$  by blast
  have  $aK: a \in K$  using  $a$  generate-field-subK S by auto
  show  $- x \in f \text{ ' generate-field } S$ 
    by (metis  $a \ aK$  generate-field.generate-field-minus hom-uminus image-eqI)
  show inverse  $x \in f \text{ ' generate-field } S$ 
    by (metis  $a \ aK$  hom-inverse-all image-iff Subfield.inverse-closed subfield-generate-field)
qed

```

Hence the image of a generated subfield is exactly the subfield generated by the image. (Both sides live in the target type, so this is an unconditional naturality statement.)

```

lemma image-generate-field-eq:
  assumes  $S: S \subseteq K$ 
  shows  $f \text{ ' generate-field } S = \text{generate-field } (f \text{ ' } S)$ 
  by (simp add: assms generate-field-least image-generate-field-subfield
    image-generate-field-subset image-mono order-antisym subset-generate-field)

```

end

end

27 Polynomials over a Subfield; Minimal Polynomials in a Tower

```

theory Field-Extension-Tower
  imports Subfield
begin

```

For the iso-extension step of Galois theory we must reason about the minimal polynomial of an element over an *intermediate* subfield K (not just the prime field). We model a polynomial “over K ” as an *'a poly* all of whose coefficients lie in the subfield K . The key fact is that polynomial division by a *monic* polynomial over K keeps quotient and remainder over K ; everything then takes place inside the single principal ideal domain *'a poly*.

27.1 Polynomials with coefficients in a subfield

```

definition poly-over :: 'a :: field set  $\Rightarrow$  'a poly set where
  poly-over  $K \equiv \{p. \text{set } (\text{coeffs } p) \subseteq K\}$ 

```

```

lemma poly-over-iff-aux:
  assumes  $0 \in K$ 
  shows  $p \in \text{poly-over } K \longleftrightarrow (\forall i. \text{coeff } p \ i \in K)$ 

```

```

using assms forall-coeffs-conv poly-over-def by fastforce

context Subfield
begin

lemma poly-over-iff:  $p \in \text{poly-over } K \longleftrightarrow (\forall i. \text{coeff } p \ i \in K)$ 
  using poly-over-iff-aux[OF zero-closed] .

lemma poly-over-0 [intro, simp]:  $0 \in \text{poly-over } K$ 
  by (simp add: poly-over-iff)

lemma poly-over-coeff:  $p \in \text{poly-over } K \implies \text{coeff } p \ i \in K$ 
  by (simp add: poly-over-iff)

lemma poly-overI:  $(\bigwedge i. \text{coeff } p \ i \in K) \implies p \in \text{poly-over } K$ 
  by (simp add: poly-over-iff)

lemma poly-over-add:  $p \in \text{poly-over } K \implies q \in \text{poly-over } K \implies p + q \in \text{poly-over } K$ 
  by (auto simp: poly-over-iff add-closed)

lemma poly-over-uminus:  $p \in \text{poly-over } K \implies -p \in \text{poly-over } K$ 
  by (auto simp: poly-over-iff uminus-closed)

lemma poly-over-diff:  $p \in \text{poly-over } K \implies q \in \text{poly-over } K \implies p - q \in \text{poly-over } K$ 
  by (auto simp: poly-over-iff diff-closed)

lemma poly-over-smult:  $c \in K \implies p \in \text{poly-over } K \implies \text{smult } c \ p \in \text{poly-over } K$ 
  by (auto simp: poly-over-iff mult-closed)

lemma poly-over-monom:  $c \in K \implies \text{monom } c \ n \in \text{poly-over } K$ 
  by (auto simp: poly-over-iff)

lemma poly-over-pCons:  $c \in K \implies p \in \text{poly-over } K \implies \text{pCons } c \ p \in \text{poly-over } K$ 
  by (auto simp: poly-over-iff coeff-pCons split: nat.split)

lemma poly-over-mult:
  assumes  $p \in \text{poly-over } K \ q \in \text{poly-over } K$  shows  $p * q \in \text{poly-over } K$ 
  using assms
  by (simp add: poly-overI add-closed coeff-mult-semiring-closed mult-closed poly-over-coeff)

lemma poly-over-1 [intro, simp]:  $1 \in \text{poly-over } K$ 
  by (auto simp: poly-over-iff coeff-1)

lemma lead-coeff-closed:  $p \in \text{poly-over } K \implies \text{lead-coeff } p \in K$ 
  by (simp add: poly-over-coeff)

end

```

Enlarging the subfield enlarges the set of polynomials over it: this lets a tower $K \subseteq L$ of intermediate fields reuse the same ambient 'a poly.

lemma *poly-over-mono*: $K \subseteq L \implies \text{poly-over } K \subseteq \text{poly-over } L$
by (*auto simp: poly-over-def*)

27.2 Division by a monic polynomial over a subfield

Long division of a polynomial over K by a *monic* polynomial over K yields a quotient and remainder over K . We first show existence by induction on the degree, then identify the witnesses with the ambient (*div*) and (*mod*) by uniqueness of polynomial division.

lemma *monic-degree-0-eq-1*:
fixes $g :: 'a :: \text{field poly}$
assumes $\text{degree } g = 0$ **lead-coeff** $g = 1$ **shows** $g = 1$
by (*metis assms degree-1 le-0-eq lead-coeff-1 poly-eqI2*)

lemma *degree-drop-aux*:
fixes $q :: 'a :: \text{field poly}$
assumes $z: \text{coeff } q \ n = 0$ **and** $le: \text{degree } q \leq n$ **and** $npos: 0 < n$
shows $\text{degree } q < n$
using *eq-zero-or-degree-less le npoz z* **by** *fastforce*

context *Subfield*
begin

lemma *poly-over-divmod-exists*:
fixes $d :: 'a \text{ poly}$
assumes $d: d \in \text{poly-over } K$ **and** *monic*: $\text{lead-coeff } d = 1$ **and** $ddeg: \text{degree } d > 0$
shows $p \in \text{poly-over } K \implies \exists q \ r. q \in \text{poly-over } K \wedge r \in \text{poly-over } K \wedge p = q * d + r \wedge \text{degree } r < \text{degree } d$
proof (*induction degree p arbitrary: p rule: less-induct*)
case *less*
show *?case*
proof (*cases degree p < degree d*)
case *False*
have $dnz: d \neq 0$ **using** *monic ddeg* **by** *auto*
define c **where** $c \equiv \text{lead-coeff } p$
define t **where** $t \equiv \text{monom } c \ (\text{degree } p - \text{degree } d) * d$
define p' **where** $p' \equiv p - t$
have $cK: c \in K$ **using** *less.prem*s **by** (*simp add: c-def lead-coeff-closed*)
have $tK: t \in \text{poly-over } K$ **using** $cK \ d$ **by** (*auto simp: t-def intro: poly-over-monom poly-over-mult*)
have $p'K: p' \in \text{poly-over } K$ **using** *less.prem*s tK **by** (*simp add: p'-def poly-over-diff*)
— The leading terms cancel, so the degree strictly drops (when $p \neq 0$).
have $pnz: p \neq 0$ **using** *False ddeg* **by** *auto*
have $cnz: c \neq 0$ **using** pnz **by** (*simp add: c-def*)
have $deg-t: \text{degree } t = \text{degree } p$
using dnz *monic cnz False* **by** (*simp add: t-def degree-mult-eq degree-monom-eq*)

```

have lead-t: lead-coeff t = c
  using monic False cnz by (simp add: t-def lead-coeff-mult degree-monom-eq)
have degree p' < degree p
proof -
  have z: coeff p' (degree p) = 0
    using lead-t deg-t by (simp add: p'-def c-def)
  have le: degree p' ≤ degree p
    using deg-t by (simp add: p'-def degree-diff-le)
  then show ?thesis
    using degree-drop-aux[OF z le] False ddeg by linarith
qed
then obtain q r where qr: q ∈ poly-over K r ∈ poly-over K
  p' = q * d + r degree r < degree d
  using less.hyps p'K by blast
have p = (q + monom c (degree p - degree d)) * d + r
  using qr(3) by (simp add: p'-def t-def algebra-simps)
moreover have q + monom c (degree p - degree d) ∈ poly-over K
  using qr(1) cK by (auto intro: poly-over-add poly-over-monom)
ultimately show ?thesis using qr by blast
qed (use less.prem in auto)
qed

```

By uniqueness of division, $p \text{ div } d$ and $p \text{ mod } d$ lie over K .

```

lemma poly-over-div-mod:
  fixes d :: 'a :: field poly
  assumes d: d ∈ poly-over K and monic: lead-coeff d = 1 and ddeg: degree d > 0
  and p: p ∈ poly-over K
  shows p div d ∈ poly-over K ∧ p mod d ∈ poly-over K
proof -
  obtain q r where qr: q ∈ poly-over K r ∈ poly-over K
    p = q * d + r degree r < degree d
  using poly-over-divmod-exists[OF d monic ddeg p] by blast
  have dnz: d ≠ 0 using monic ddeg by auto
  show ?thesis using qr by (simp add: dnz div-poly-less mod-poly-less)
qed

end

```

27.3 Minimal polynomial over a subfield

An element a is *algebraic over K* if some nonzero polynomial over K vanishes at it. A *minimal polynomial* is a monic polynomial over K of least positive degree vanishing at a .

definition *algebraic-over* :: 'a :: field set ⇒ 'a ⇒ bool **where**
algebraic-over K $a \equiv (\exists p \in \text{poly-over } K. p \neq 0 \wedge \text{poly } p \ a = 0)$

definition *is-minpoly* :: 'a :: field set ⇒ 'a ⇒ 'a poly ⇒ bool **where**
is-minpoly K a $m \equiv$
 $m \in \text{poly-over } K \wedge \text{lead-coeff } m = 1 \wedge \text{poly } m \ a = 0 \wedge$

$$(\forall r \in \text{poly-over } K. r \neq 0 \wedge \text{poly } r \ a = 0 \longrightarrow \text{degree } m \leq \text{degree } r)$$

A polynomial is *irreducible over* K if it has positive degree and admits no factorisation over K into two factors of positive degree. (Over the ambient field it may well factor — e.g. split completely — but not within *poly-over* K .)

definition *irreducible-over* :: 'a :: field set $\Rightarrow$ 'a poly $\Rightarrow$ bool **where**

irreducible-over K $p \equiv$
 $p \neq 0 \wedge \text{degree } p > 0 \wedge$
 $(\forall b \ c. b \in \text{poly-over } K \longrightarrow c \in \text{poly-over } K \longrightarrow p = b * c \longrightarrow \text{degree } b = 0 \vee$
 $\text{degree } c = 0)$

Algebraicity is preserved when passing to a larger subfield.

lemma *algebraic-over-mono*:

assumes *algebraic-over* K a **and** $K \subseteq L$
shows *algebraic-over* L a
using *assms* **by** (*meson algebraic-over-def poly-over-mono subsetD*)

context *Subfield*

begin

A minimal polynomial exists for every algebraic element: take a nonzero annihilator of least degree and normalise it to be monic (the normalising scalar lies in K).

lemma *minpoly-exists*:

assumes *algebraic-over* K a
shows $\exists m. \text{is-minpoly } K \ a \ m$

proof —

define Q **where** $Q = (\lambda n. \exists p. p \in \text{poly-over } K \wedge p \neq 0 \wedge \text{poly } p \ a = 0 \wedge \text{degree } p = n)$

then have Q (*LEAST* $n. Q \ n$)

by (*metis LeastI algebraic-over-def assms*)

then obtain p **where** $p: p \in \text{poly-over } K \ p \neq 0 \ \text{poly } p \ a = 0 \ \text{degree } p = (\text{LEAST } n. Q \ n)$

unfolding Q -def **by** *blast*

have *minp*: $\text{degree } p \leq \text{degree } r$ **if** $r \in \text{poly-over } K \ r \neq 0 \ \text{poly } r \ a = 0$ **for** r

by (*metis Least-le Q-def p(4) that*)

define m **where** $m = \text{smult } (\text{inverse } (\text{lead-coeff } p)) \ p$

have *lcp*: $\text{lead-coeff } p \in K$ **using** $p(1)$ **by** (*rule lead-coeff-closed*)

have *lcp-nz*: $\text{lead-coeff } p \neq 0$ **using** $p(2)$ **by** *simp*

have mK : $m \in \text{poly-over } K$

using $p(1)$ *lcp* **by** (*auto simp: m-def intro: poly-over-smult inverse-closed*)

have *m-monic*: $\text{lead-coeff } m = 1$

using *lcp-nz* **by** (*simp add: m-def*)

have *m-root*: $\text{poly } m \ a = 0$ **using** $p(3)$ **by** (*simp add: m-def*)

have $\text{degree } m = \text{degree } p$ **using** *lcp-nz* **by** (*simp add: m-def*)

then have *is-minpoly* $K \ a \ m$

by (*metis is-minpoly-def mK m-monic m-root minp*)

then show *?thesis* **by** *blast*

qed

The kernel characterization: any polynomial over K vanishing at a is a multiple of the minimal polynomial. (Divide by m : the remainder is a lower-degree annihilator over K , hence zero by minimality.)

lemma *minpoly-dvd*:

assumes m : *is-minpoly* K a m **and** r : $r \in \text{poly-over } K$ **and** ra : $\text{poly } r \ a = 0$
shows $m \text{ dvd } r$

proof –

have mK : $m \in \text{poly-over } K$ **and** *monic*: $\text{lead-coeff } m = 1$ **and** $m0$: $\text{poly } m \ a = 0$
and *minimal*: $\bigwedge s. s \in \text{poly-over } K \implies s \neq 0 \implies \text{poly } s \ a = 0 \implies \text{degree } m \leq \text{degree } s$

using m **unfolding** *is-minpoly-def* **by** *auto*

have $mdeg$: $\text{degree } m > 0$

using $m0$ *monic poly-zero* **by** *fastforce*

have rem : $r \bmod m \in \text{poly-over } K$ **using** *poly-over-div-mod*[*OF mK monic mdeg r*]

by *simp*

have mnz : $m \neq 0$ **using** $mdeg$ **by** *auto*

have $rem\text{-root}$: $\text{poly } (r \bmod m) \ a = 0$

by (*simp add: m0 poly-mod ra*)

have $r \bmod m = 0$

by (*meson degree-mod-less' leD minimal mnz rem rem-root*)

then show *?thesis* **by** (*simp add: mod-eq-0-iff-dvd*)

qed

The minimal polynomial has positive degree and is irreducible over K : a proper factorisation over K would give a nonzero factor of smaller degree annihilating a , contradicting minimality.

lemma *minpoly-degree-pos*:

assumes m : *is-minpoly* K a m **shows** $\text{degree } m > 0$

using *is-minpoly-def m poly-zero* **by** *fastforce*

lemma *minpoly-no-proper-factor*:

assumes m : *is-minpoly* K a m

and b : $b \in \text{poly-over } K$ **and** c : $c \in \text{poly-over } K$ **and** eq : $m = b * c$

shows $\text{degree } b = 0 \vee \text{degree } c = 0$

proof (*rule ccontr*)

assume $\neg (\text{degree } b = 0 \vee \text{degree } c = 0)$

then have $bdeg$: $\text{degree } b > 0$ **and** $cdeg$: $\text{degree } c > 0$ **by** *auto*

have bnz : $b \neq 0$ **and** cnz : $c \neq 0$ **using** $bdeg$ $cdeg$ **by** *auto*

have $\text{poly } b \ a = 0 \vee \text{poly } c \ a = 0$

by (*metis divisors-zero eq is-minpoly-def m poly-mult*)

moreover

have $\text{degree } b < \text{degree } m$ $\text{degree } c < \text{degree } m$

using eq bnz cnz $cdeg$ $bdeg$ **by** (*simp-all add: degree-mult-eq*)

ultimately show *False*

by (*meson b bnz c cnz divides-degree leD m minpoly-dvd*)

qed

B

'ezout over K : if m is the (irreducible) minimal polynomial and m does not

divide a polynomial p over K , then 1 is a K -coefficient combination of m and p . Proved via a least-degree element of the combination set, which (by irreducibility of m) must be a unit.

lemma *minpoly-bezout*:

assumes m : *is-minpoly* K a m **and** p : $p \in \text{poly-over } K$ **and** $ndvd$: $\neg m \text{ dvd } p$
shows $\exists u \in \text{poly-over } K. \exists v \in \text{poly-over } K. u * m + v * p = 1$

proof –

have mK : $m \in \text{poly-over } K$ **using** m **by** (*simp add: is-minpoly-def*)

have mnz : $m \neq 0$ **using** *minpoly-degree-pos[OF m]* **by** *auto*

define C **where** $C = (\lambda g. \exists u \in \text{poly-over } K. \exists v \in \text{poly-over } K. g = u * m + v * p)$

have $C\text{-poly}K$: $g \in \text{poly-over } K$ **if** C g **for** g

using *that unfolding C-def by (auto intro: poly-over-add poly-over-mult mK p)*

have $C\text{-}m$: C m **unfolding** $C\text{-def}$ **by** (*rule beXI[of - 1]*) (*auto intro: beXI[of - 0]*)

have $C\text{-}p$: C p **unfolding** $C\text{-def}$ **by** (*rule beXI[of - 0]*) (*auto intro: beXI[of - 1]*)

have $C\text{-sub}$: C $(h - q * g)$ **if** C h C g $q \in \text{poly-over } K$ **for** h g q

proof –

from $\langle C$ $h \rangle$ **obtain** $u1$ $v1$ **where** h : $u1 \in \text{poly-over } K$ $v1 \in \text{poly-over } K$ $h = u1 * m + v1 * p$

unfolding $C\text{-def}$ **by** *blast*

from $\langle C$ $g \rangle$ **obtain** $u2$ $v2$ **where** g : $u2 \in \text{poly-over } K$ $v2 \in \text{poly-over } K$ $g = u2 * m + v2 * p$

unfolding $C\text{-def}$ **by** *blast*

have $h - q * g = (u1 - q * u2) * m + (v1 - q * v2) * p$

using $h(3)$ $g(3)$ **by** (*simp add: algebra-simps*)

moreover have $u1 - q * u2 \in \text{poly-over } K$ $v1 - q * v2 \in \text{poly-over } K$

using h g *that(3)* **by** (*auto intro: poly-over-diff poly-over-mult*)

ultimately show *?thesis* **unfolding** $C\text{-def}$ **by** *blast*

qed

define Q **where** $Q = (\lambda n. \exists g. C$ $g \wedge g \neq 0 \wedge \text{degree } g = n)$

then have Q (*LEAST* n . Q n)

by (*metis C-m LeastI mnz*)

then obtain $g0$ **where** $g0$: C $g0$ $g0 \neq 0$ $\text{degree } g0 = (\text{LEAST } n. Q$ $n)$

unfolding $Q\text{-def}$ **by** *blast*

have *leastdeg*: C $h \implies h \neq 0 \implies \text{degree } g0 \leq \text{degree } h$ **for** h

by (*metis Least-le Q-def g0(3)*)

define g **where** $g = \text{smult } (\text{inverse } (\text{lead-coeff } g0))$ $g0$

have $lg0K$: $\text{lead-coeff } g0 \in K$ **using** $C\text{-poly}K$ [*OF* $g0(1)$] **by** (*rule lead-coeff-closed*)

have $lg0nz$: $\text{lead-coeff } g0 \neq 0$ **using** $g0(2)$ **by** *simp*

from $g0(1)$ **obtain** u v **where** uv : $u \in \text{poly-over } K$ $v \in \text{poly-over } K$ $g0 = u * m + v * p$

unfolding $C\text{-def}$ **by** *blast*

have $g = \text{smult } (\text{inverse } (\text{lead-coeff } g0))$ $u * m + \text{smult } (\text{inverse } (\text{lead-coeff } g0))$ v

$* p$

using $uv(3)$ **by** (*simp add: g-def smult-add-right algebra-simps*)

with uv $lg0K$ **have** gC : C g **unfolding** $C\text{-def}$

using *poly-over-smult* **by** *blast*

have gK : $g \in \text{poly-over } K$ **using** gC **by** (*rule C-polyK*)

have gnz : $g \neq 0$ **using** $g0(2)$ $lg0nz$ **by** (*simp add: g-def*)

have $g\text{-monic}$: $\text{lead-coeff } g = 1$ **using** $lg0nz$ **by** (*simp add: g-def*)

have $g\text{-deg}$: $\text{degree } g = \text{degree } g0$ **using** $lg0nz$ **by** (*simp add: g-def*)

```

have g-dvd: g dvd h if C h for h
proof (cases degree g = 0)
  case True
  then show ?thesis
    using gnz is-unit-iff-degree by blast
next
case False
then have divK: h div g ∈ poly-over K and rem: h mod g ∈ poly-over K
  using poly-over-div-mod[OF gK g-monic - C-polyK[OF that]] by simp-all
have C (h mod g) using C-sub[OF that gC divK]
  by (simp add: minus-div-mult-eq-mod)
then show ?thesis
  using degree-mod-less-degree g-deg gnz leastdeg by fastforce
qed
have gdvdm: g dvd m using C-m by (rule g-dvd)
have gdvdp: g dvd p using C-p by (rule g-dvd)
have degree g = 0
proof (rule ccontr)
  assume gnz0: degree g ≠ 0
  from gdvdm obtain q where mq: m = g * q and qK: q ∈ poly-over K
    using gK g-monic gnz0 mK poly-over-div-mod by force
  have degree q = 0 using minpoly-no-proper-factor[OF m gK qK mq] gnz0 by
simp
  then obtain c where qc: q = [:c:] by (rule degree-eq-zeroE)
  have cnz: c ≠ 0 using mq mnz qc by auto
  have g = m * [:inverse c:]
    by (simp add: cnz mq qc)
  with ndvd show False by (metis dvdI dvd-trans gdvdp)
qed
then show ?thesis
  using C-def gC g-monic monic-degree-0-eq-1 by blast
qed

```

If p is irreducible over K and vanishes at a , then p is an associate (over K) of the minimal polynomial of a : the minimal polynomial divides p , and irreducibility forces the cofactor to be a nonzero constant.

lemma *irreducible-over-minpoly-assoc*:

```

assumes irr: irreducible-over K p and m: is-minpoly K a m
  and pa: poly p a = 0 and pK: p ∈ poly-over K
shows ∃ k. k ≠ 0 ∧ p = smult k m
proof -
  have mK: m ∈ poly-over K and monic: lead-coeff m = 1
    using m by (auto simp: is-minpoly-def)
  have mdeg: degree m > 0 using minpoly-degree-pos[OF m] .
  have m dvd p using minpoly-dvd[OF m pK pa] .
  then obtain c where pc: p = m * c and cK: c ∈ poly-over K
    using poly-over-div-mod[OF mK monic mdeg pK] by force
  have degree m = 0 ∨ degree c = 0
    using irr mK cK by (auto simp: irreducible-over-def pc mult.commute)

```



```

with irr pc mdeg show ?thesis
  by (auto simp: irreducible-over-def elim!: degree-eq-zeroE)
qed

```

Consequently every root of an irreducible p is a root of the minimal polynomial of any other root: this is what makes the minimal-polynomial hypothesis of the transitivity theorem automatic for irreducible polynomials.

```

lemma irreducible-over-root-is-minpoly-root:
  assumes irreducible-over K p p ∈ poly-over K is-minpoly K a m poly p a = 0 poly
p b = 0
  shows poly m b = 0
  using irreducible-over-minpoly-assoc assms by fastforce

```

```

end

```

```

end

```

28 Simple Algebraic Extensions are Subfields

```

theory Simple-Extension
  imports Field-Extension-Tower
begin

```

The simple extension $K(a)$ of a subfield K by an element a algebraic over K is realised as the set of polynomial expressions $\text{poly } p \ a$ for $p \in \text{poly-over } K$. It is a subfield: closure under the ring operations is immediate (evaluation is a ring homomorphism), and closure under inverse is the payoff of Bézout's identity over K .

```

definition eval-img :: 'a :: field set  $\Rightarrow$  'a  $\Rightarrow$  'a set where
  eval-img K a = ( $\lambda p$ . poly p a) ' poly-over K

```

```

lemma eval-imgI: p ∈ poly-over K  $\implies$  poly p a ∈ eval-img K a
  by (auto simp: eval-img-def)

```

```

lemma eval-imgE:
  assumes y ∈ eval-img K a obtains p where p ∈ poly-over K y = poly p a
  using assms by (auto simp: eval-img-def)

```

```

context Subfield
begin

```

```

lemma eval-img-self: a ∈ eval-img K a
proof –
  have poly [:0, 1:] a = a by simp
  then show ?thesis
    by (metis eval-imgI one-closed poly-over-0 poly-over-pCons zero-closed)
qed

```

lemma *eval-img-base*:

assumes $c \in K$

shows $c \in \text{eval-img } K \ a$

proof –

have $[:c:] \in \text{poly-over } K$

by (*simp add: assms poly-over-pCons*)

then show *?thesis*

using *eval-imgI* **by** *fastforce*

qed

lemma *eval-img-0*: $0 \in \text{eval-img } K \ a$

using *eval-img-base zero-closed* **by** *blast*

lemma *eval-img-1*: $1 \in \text{eval-img } K \ a$

using *eval-img-base one-closed* **by** *blast*

lemma *eval-img-add*: $x \in \text{eval-img } K \ a \implies y \in \text{eval-img } K \ a \implies x + y \in \text{eval-img } K \ a$

by (*metis (no-types, lifting) eval-imgE eval-imgI poly-add poly-over-add*)

lemma *eval-img-uminus*: $x \in \text{eval-img } K \ a \implies -x \in \text{eval-img } K \ a$

by (*metis eval-imgE eval-imgI poly-minus poly-over-uminus*)

lemma *eval-img-mult*: $x \in \text{eval-img } K \ a \implies y \in \text{eval-img } K \ a \implies x * y \in \text{eval-img } K \ a$

by (*metis (no-types, lifting) eval-imgE eval-imgI poly-mult poly-over-mult*)

Closure under inverse, the payoff of Bézout.

lemma *eval-img-inverse*:

assumes *alg: algebraic-over* $K \ a$ **and** $y: y \in \text{eval-img } K \ a$ **and** *ynz*: $y \neq 0$

shows *inverse* $y \in \text{eval-img } K \ a$

proof –

obtain p **where** $p: p \in \text{poly-over } K$ **and** $yp: y = \text{poly } p \ a$ **using** y **by** (*rule eval-imgE*)

obtain m **where** $m: \text{is-minpoly } K \ a \ m$ **using** *minpoly-exists[OF alg]* **by** *blast*

have $m0: \text{poly } m \ a = 0$ **using** m **by** (*simp add: is-minpoly-def*)

have $\neg m \ \text{dvd } p$

using $m0 \ ynz \ yp$ **by** *auto*

then obtain $u \ v$ **where** $uv: u \in \text{poly-over } K \ v \in \text{poly-over } K \ u * m + v * p = 1$

using *minpoly-bezout[OF m p]* **by** *blast*

have $\text{poly } (u * m + v * p) \ a = \text{poly } 1 \ a$ **using** $uv(3)$ **by** *simp*

then have *inverse* $y = \text{poly } v \ a$

by (*simp add: inverse-unique m0 mult.commute yp*)

then show *?thesis* **using** $uv(2)$ **by** (*metis eval-imgI*)

qed

Hence, for a algebraic over K , the simple extension is a subfield.

theorem *subfield-eval-img*:

assumes *algebraic-over* $K \ a$ **shows** *Subfield* (*eval-img* $K \ a$)

```

proof
  show  $\bigwedge x. x \in \text{eval-img } K \ a \implies \text{inverse } x \in \text{eval-img } K \ a$ 
    using assms eval-img-inverse by force
qed (auto simp: eval-img-0 eval-img-1 eval-img-add eval-img-uminus eval-img-mult
eval-img-inverse)

```

The simple extension $\text{eval-img } K \ a$ coincides with the generically generated subfield $\text{generate-field } (K \cup \{a\})$ of theory *Subfield*. This connects the polynomial-evaluation construction to the inductive generate-field , so the tower of intermediate fields can be expressed either way. The forward inclusion is unconditional; the reverse uses that $\text{eval-img } K \ a$ is itself a subfield (so generate-field of a generating subset of it stays inside).

```

lemma eval-img-subset-generate-field:
  assumes  $p \in \text{poly-over } K$ 
  shows  $\text{poly } p \ a \in \text{generate-field } (K \cup \{a\})$ 
proof –
  interpret  $G$ : Subfield generate-field  $(K \cup \{a\})$  by (rule subfield-generate-field)
  have  $aG$ :  $a \in \text{generate-field } (K \cup \{a\})$  by (auto intro: generate-field-base)
  have  $cG$ :  $\text{coeff } p \ i \in \text{generate-field } (K \cup \{a\})$  for  $i$ 
    using assms by (auto simp: poly-over-coeff intro: generate-field-base)
  then show ?thesis
    unfolding poly-altdef
    using  $aG \ cG$  by (intro G.sum-closed G.mult-closed G.power-closed)
qed

```

```

lemma eval-img-eq-generate-field:
  assumes  $\text{alg: algebraic-over } K \ a$ 
  shows  $\text{eval-img } K \ a = \text{generate-field } (K \cup \{a\})$ 
proof
  show  $\text{eval-img } K \ a \subseteq \text{generate-field } (K \cup \{a\})$ 
    by (metis eval-imgE eval-img-subset-generate-field subsetI)
next
  have  $K \cup \{a\} \subseteq \text{eval-img } K \ a$ 
    using eval-img-base eval-img-self by auto
  then show  $\text{generate-field } (K \cup \{a\}) \subseteq \text{eval-img } K \ a$ 
    by (rule generate-field-least[OF subfield-eval-img[OF alg]])
qed

```

end

end

29 Extending a Field Homomorphism across a Simple Algebraic Extension

```

theory Iso-Extension
  imports Simple-Extension
begin

```

The keystone of the iso-extension tower. Let f be a homomorphism defined on a subfield K (a *field-hom-on*), let a be algebraic over K with minimal polynomial m , and let b be a root of the f -image $\text{map-poly } f \ m$ of that minimal polynomial. Then f extends to a homomorphism g on the simple extension $\text{eval-img } K \ a (= K(a))$ that sends a to b and restricts to f on K .

The extension is forced: every element of $K \ a$ is $\text{poly } p \ a$ for some $p \in \text{poly-over } K$, and g must send it to $\text{poly } (\text{map-poly } f \ p) \ b$. Well-definedness is the only subtlety: if $\text{poly } p \ a = \text{poly } q \ a$ then m divides $p - q$ (the kernel characterization *minpoly-dvd*), so $\text{map-poly } f \ m$ divides $\text{map-poly } f \ (p - q)$, which therefore vanishes at its root b .

context *field-hom-on*
begin

29.1 The mapped polynomial $\text{map-poly } f$ on $\text{poly-over } K$

Since f is only a homomorphism *on* K , the library's *map-poly* homomorphism lemmas (which assume a global ring homomorphism) do not apply. We re-establish additivity and multiplicativity of $\text{map-poly } f$ restricted to polynomials over K , working coefficientwise.

lemma *hom-sum*:

assumes $\bigwedge i. i \in A \implies x \ i \in K$
shows $f \ (\text{sum } x \ A) = (\sum i \in A. f \ (x \ i))$
using *assms*
by (*induction A rule: infinite-finite-induct*) (*auto simp: hom-add sum-closed*)

The coefficient lemma uses only the global *hom-0*.

lemma *coeff-map-poly-f*: $\text{coeff } (\text{map-poly } f \ p) \ n = f \ (\text{coeff } p \ n)$
by (*rule coeff-map-poly*) (*rule hom-0*)

lemma *map-poly-add-on*:

assumes $p \in \text{poly-over } K \ q \in \text{poly-over } K$
shows $\text{map-poly } f \ (p + q) = \text{map-poly } f \ p + \text{map-poly } f \ q$
by (*intro poly-eqI*) (*simp add: assms coeff-map-poly-f hom-add poly-over-coeff*)

lemma *map-poly-mult-on*:

assumes $p \in \text{poly-over } K \ q \in \text{poly-over } K$
shows $\text{map-poly } f \ (p * q) = \text{map-poly } f \ p * \text{map-poly } f \ q$

proof (*rule poly-eqI*)

fix n
have $\text{coeff } (\text{map-poly } f \ (p * q)) \ n = (\sum k \leq n. f \ (\text{coeff } p \ k * \text{coeff } q \ (n - k)))$
by (*simp add: assms coeff-map-poly-f coeff-mult hom-sum mult-closed poly-over-coeff*)
also have $\dots = (\sum k \leq n. f \ (\text{coeff } p \ k) * f \ (\text{coeff } q \ (n - k)))$
using *assms hom-mult poly-over-iff* **by force**
also have $\dots = \text{coeff } (\text{map-poly } f \ p * \text{map-poly } f \ q) \ n$
by (*simp add: coeff-map-poly-f coeff-mult*)
finally show $\text{coeff } (\text{map-poly } f \ (p * q)) \ n = \text{coeff } (\text{map-poly } f \ p * \text{map-poly } f \ q) \ n$.
qed

lemma *map-poly-diff-on*:
assumes $p \in \text{poly-over } K$ $q \in \text{poly-over } K$
shows $\text{map-poly } f (p - q) = \text{map-poly } f p - \text{map-poly } f q$
using *assms field-hom-on.map-poly-add-on field-hom-on-axioms poly-over-diff* **by**
fastforce

lemma *map-poly-const*: $\text{map-poly } f [:c:] = [:f c:]$
by (*intro poly-eqI*) (*simp add: coeff-map-poly-f coeff-pCons split: nat.split*)

29.2 Evaluation commutes with the homomorphism

f respects powers of elements of K .

lemma *hom-power*: $x \in K \implies f (x \wedge n) = (f x) \wedge n$
by (*induction n*) (*auto simp: hom-mult power-closed*)

For a polynomial over K and an argument in K , evaluating the f -image at $f x$ equals applying f to the value at x : evaluation is natural in f . (The degree of $\text{map-poly } f p$ may drop if f kills the leading coefficient, hence the *sum.mono-neutral-left* step.)

lemma *poly-map-poly-hom*:
assumes $p: p \in \text{poly-over } K$ **and** $x: x \in K$
shows $\text{poly} (\text{map-poly } f p) (f x) = f (\text{poly } p x)$
proof –
have $ck: \bigwedge i. \text{coeff } p i \in K$ **using** p **by** (*simp add: poly-over-coeff*)
have $cmf: \bigwedge i. \text{coeff} (\text{map-poly } f p) i = f (\text{coeff } p i)$ **by** (*rule coeff-map-poly-f*)
have $rng: (\sum i \leq \text{degree} (\text{map-poly } f p). \text{coeff} (\text{map-poly } f p) i * (f x) \wedge i)$
 $= (\sum i \leq \text{degree } p. \text{coeff} (\text{map-poly } f p) i * (f x) \wedge i)$
proof (*rule sum.mono-neutral-left*)
show $\{.. \text{degree} (\text{map-poly } f p)\} \subseteq \{.. \text{degree } p\}$
using *map-poly-degree-leq* **by** *auto*
qed (*auto simp: coeff-eq-0*)
have $\text{poly} (\text{map-poly } f p) (f x) = (\sum i \leq \text{degree } p. f (\text{coeff } p i) * (f x) \wedge i)$
by (*metis (no-types, lifting) ext coeff-map-poly-f poly-altdef rng*)
also have $\dots = (\sum i \leq \text{degree } p. f (\text{coeff } p i * x \wedge i))$
using ck x **by** (*simp add: hom-mult power-closed hom-power*)
also have $\dots = f (\text{poly } p x)$
by (*simp add: ck hom-sum mult-closed poly-altdef power-closed x*)
finally show *?thesis* .
qed

Consequently f sends a root in K of a polynomial over K to a root of the mapped polynomial.

lemma *maps-root*:
assumes $p: p \in \text{poly-over } K$ **and** $x: x \in K$ **and** *root: poly p x = 0*
shows $\text{poly} (\text{map-poly } f p) (f x) = 0$
using *poly-map-poly-hom[OF p x] root* **by** *simp*

29.3 Well-definedness of the extension

If two polynomials over K agree at a , their f -images agree at any root b of $\text{map-poly } f \ m$. This is what makes the extension a function.

lemma *eval-unique*:

assumes m : *is-minpoly* $K \ a \ m$ **and** b : *poly* ($\text{map-poly } f \ m$) $b = 0$
and p : $p \in \text{poly-over } K$ **and** q : $q \in \text{poly-over } K$ **and** eq : *poly* $p \ a = \text{poly } q \ a$
shows *poly* ($\text{map-poly } f \ p$) $b = \text{poly } (\text{map-poly } f \ q) \ b$

proof –

have mK : $m \in \text{poly-over } K$ **and** *monic*: *lead-coeff* $m = 1$
using m **by** (*auto simp: is-minpoly-def*)
have $mdeg$: *degree* $m > 0$ **using** *minpoly-degree-pos*[*OF* m] .
have pq : $p - q \in \text{poly-over } K$ **using** $p \ q$ **by** (*rule poly-over-diff*)
have $root$: *poly* ($p - q$) $a = 0$ **using** eq **by** *simp*
have $m \ \text{dvd}$ ($p - q$) **using** *minpoly-dvd*[*OF* $m \ pq \ root$] .
then obtain s **where** s : $p - q = m * s$ **by** (*elim dvdE*)
have $s = (p - q) \ \text{div } m$ **using** $s \ mdeg \ \text{monic}$ **by** *auto*
then have $\text{map-poly } f \ (p - q) = \text{map-poly } f \ m * \text{map-poly } f \ s$
using s **using** *map-poly-mult-on*[*OF* mK]
by (*metis mK mdeg monic poly-over-div-mod pq*)
then have *poly* ($\text{map-poly } f \ (p - q)$) $b = 0$ **using** b **by** *simp*
moreover have $\text{map-poly } f \ (p - q) = \text{map-poly } f \ p - \text{map-poly } f \ q$
by (*rule map-poly-diff-on*[*OF* $p \ q$])
ultimately show *?thesis* **by** *simp*

qed

29.4 The extension theorem

theorem *iso-extension*:

assumes alg : *algebraic-over* $K \ a$
and m : *is-minpoly* $K \ a \ m$
and b : *poly* ($\text{map-poly } f \ m$) $b = 0$
shows $\exists g$. *field-hom-on* (*eval-img* $K \ a$) $g \wedge g \ a = b \wedge (\forall x \in K. g \ x = f \ x)$

proof –

define g **where** $g = (\lambda x. \text{poly } (\text{map-poly } f \ (\text{SOME } r. r \in \text{poly-over } K \wedge x = \text{poly } r \ a)) \ b)$

have $g\text{-eval}$: $g \ (\text{poly } p \ a) = \text{poly } (\text{map-poly } f \ p) \ b$ **if** p : $p \in \text{poly-over } K$ **for** p

proof –

let $?r = \text{SOME } r. r \in \text{poly-over } K \wedge \text{poly } p \ a = \text{poly } r \ a$
have ex : $\exists r. r \in \text{poly-over } K \wedge \text{poly } p \ a = \text{poly } r \ a$ **using** p **by** *blast*
have r : $?r \in \text{poly-over } K \wedge \text{poly } p \ a = \text{poly } ?r \ a$ **using** ex **by** (*rule someI-ex*)
then show *?thesis* **using** *eval-unique*[*OF* $m \ b \ p$] r
using $g\text{-def}$ **by** *force*

qed

have sf : *Subfield* (*eval-img* $K \ a$) **using** *subfield-eval-img*[*OF* alg] .

interpret E : *Subfield* *eval-img* $K \ a$ **by** (*rule sf*)

have ga : $g \ a = b$

proof –

have $a = \text{poly } [:0,1:] \ a$ **by** *simp*

```

    moreover have  $[:0,1:] \in \text{poly-over } K$  by (intro poly-over-pCons) auto
    ultimately have  $g\ a = \text{poly } (\text{map-poly } f\ [:0,1:])\ b$  using g-eval by metis
    then show ?thesis by (simp add: map-poly-pCons map-poly-const)
  qed
  have  $gK: g\ x = f\ x$  if  $x: x \in K$  for  $x$ 
  proof -
    have  $g\ x = \text{poly } (\text{map-poly } f\ [:x:])\ b$  using g-eval
      using poly-over-pCons  $x$  by force
    then show ?thesis
      by (simp add: map-poly-const)
  qed
  have gadd:  $g\ (x + y) = g\ x + g\ y$  if  $x: x \in \text{eval-img } K\ a$  and  $y: y \in \text{eval-img } K\ a$ 
  for  $x\ y$ 
    using that g-eval map-poly-add-on poly-over-add by (force elim!: eval-imgE)
  have gmult:  $g\ (x * y) = g\ x * g\ y$  if  $x: x \in \text{eval-img } K\ a$  and  $y: y \in \text{eval-img } K\ a$ 
  for  $x\ y$ 
    using that g-eval map-poly-mult-on poly-over-mult by (force elim!: eval-imgE)
  have field-hom-on (eval-img  $K\ a$ )  $g$ 
    by unfold-locale (use gK gadd gmult in auto)
  then show ?thesis using ga gK by blast
  qed

end

end

```

```

theory Ring-Theory
  imports Group-Theory
begin

```

```

no-notation plus (infixl  $\langle + \rangle$  65)
no-notation minus (infixl  $\langle - \rangle$  65)
unbundle no uminus-syntax
no-notation quotient (infixl  $\langle ' / ' \rangle$  90)

```

30 Rings

30.1 Definition and Elementary Properties

Def 2.1

p 86, ll 20–28

```

locale Ring = additive: Abelian-Group  $R\ (+)\ \mathbf{0}$  + multiplicative: Monoid  $R\ (\cdot)\ \mathbf{1}$ 
  for  $R$  and addition (infixl  $\langle + \rangle$  65) and multiplication (infixl  $\langle \cdot \rangle$  70) and zero
  ( $\langle \mathbf{0} \rangle$ ) and unit ( $\langle \mathbf{1} \rangle$ ) +
  assumes distributive:  $\llbracket a \in R; b \in R; c \in R \rrbracket \implies a \cdot (b + c) = a \cdot b + a \cdot c$ 
     $\llbracket a \in R; b \in R; c \in R \rrbracket \implies (b + c) \cdot a = b \cdot a + c \cdot a$ 
begin

```

p 86, ll 20–28

notation *additive.inverse* ($\langle - \rightarrow \rangle$ [66] 65)
abbreviation *subtraction* (**infixl** $\langle - \rangle$ 65) **where** $a - b \equiv a + (-\ b)$

end

Compact version based on the monoid type

locale *Ring'* =
 fixes *additive multiplicative*
 assumes *eq*: *mcarrier additive* = *mcarrier multiplicative*
 assumes *Ring*: *Ring* (*mcarrier additive*) (*mmult additive*) (*mmult multiplicative*)
 (*munit additive*) (*munit multiplicative*)

p 87, ll 10–12

locale *Subring* =
 additive: *Subgroup* *S R* (+) **0** + *multiplicative*: *Submonoid* *S R* ($\cdot$) **1**
 for *S* **and** *R* **and** *addition* (**infixl** $\langle + \rangle$ 65) **and** *multiplication* (**infixl** $\langle \cdot \rangle$ 70) **and**
 zero ($\langle 0 \rangle$) **and** *unit* ($\langle 1 \rangle$)

context *Ring* **begin**

p 88, ll 26–28

lemma *right-zero* [*simp*]:
 assumes [*simp*]: $a \in R$ **shows** $a \cdot 0 = 0$
proof –
 have $a \cdot 0 = a \cdot (0 + 0)$ **by** *simp*
 also have $\dots = a \cdot 0 + a \cdot 0$ **by** (*simp add: distributive del: additive.left-unit additive.right-unit*)
 finally have $a \cdot 0 - a \cdot 0 = a \cdot 0 + a \cdot 0 - a \cdot 0$ **by** *simp*
 then show *?thesis* **by** (*simp add: additive.associative del: additive.invertible-left-cancel*)
qed

p 88, l 29

lemma *left-zero* [*simp*]:
 assumes [*simp*]: $a \in R$ **shows** $0 \cdot a = 0$
proof –
 have $0 \cdot a = (0 + 0) \cdot a$ **by** *simp*
 also have $\dots = 0 \cdot a + 0 \cdot a$ **by** (*simp add: distributive del: additive.left-unit additive.right-unit*)
 finally have $0 \cdot a - 0 \cdot a = 0 \cdot a + 0 \cdot a - 0 \cdot a$ **by** *simp*
 then show *?thesis* **by** (*simp add: additive.associative del: additive.invertible-left-cancel*)
qed

p 88, ll 29–30; p 89, ll 1–2

lemma *left-minus*:
 assumes [*simp*]: $a \in R$ $b \in R$ **shows** $(- \ a) \cdot b = - \ a \cdot b$
proof –
 have $0 = 0 \cdot b$ **by** *simp*

also have $\dots = (a - a) \cdot b$ by *simp*
 also have $\dots = a \cdot b + (-a) \cdot b$ by (*simp add: distributive del: additive.invertible-right-inverse*)
 finally have $-a \cdot b + 0 = -a \cdot b + a \cdot b + (-a) \cdot b$ by (*simp add: additive.associative del: additive.invertible-left-inverse*)
 then show ?thesis by *simp*
 qed

p 89, l 3

lemma *right-minus*:

assumes [*simp*]: $a \in R \ b \in R$ shows $a \cdot (-b) = -a \cdot b$

proof -

have $0 = a \cdot 0$ by *simp*

also have $\dots = a \cdot (b - b)$ by *simp*

also have $\dots = a \cdot b + a \cdot (-b)$ by (*simp add: distributive del: additive.invertible-right-inverse*)

finally have $-a \cdot b + 0 = -a \cdot b + a \cdot b + a \cdot (-b)$ by (*simp add: additive.associative del: additive.invertible-left-inverse*)

then show ?thesis by *simp*

qed

end

30.2 Commutative rings, integral domains, and fields

A commutative ring is a ring whose multiplication is commutative. We reuse the *commutative-monoid* structure on the multiplicative part, mirroring how *Abelian-Group* reuses *commutative-monoid*.

locale *commutative-ring* = *Ring* +

multiplicative: commutative-monoid $R \ (\cdot) \ 1$

begin

Associative-commutative normalisation bundles for the two operations, for on-demand use as *simp add: mult-ac / simp add: add-ac*. The additive group is abelian, so it too supplies the *ac* bundle.

lemmas *mult-ac* = *multiplicative.ac*

lemmas *add-ac* = *additive.ac*

end

A nontrivial ring is one in which the unit and zero differ. Both integral domains and fields require this, so we factor it into a common ancestor: that way the assumption is inherited (and named *nontrivial*) by a single route, avoiding a duplicate when a field is shown to be an integral domain or interpreted.

locale *nontrivial-ring* = *Ring* +

assumes *nontrivial*: $1 \neq 0$

An integral domain is a nontrivial commutative ring without zero divisors.

locale *integral-domain* = *commutative-ring* + *nontrivial-ring* +

assumes *no-zero-divisors*: $\llbracket a \in R; b \in R; a \cdot b = 0 \rrbracket \implies a = 0 \vee b = 0$

A field is a nontrivial commutative ring in which every nonzero element is invertible. Invertibility is expressed through the multiplicative monoid's *Monoid.invertible* predicate, so the multiplicative inverse and unit machinery is available via the *multiplicative* prefix.

locale *Field* = *commutative-ring* + *nontrivial-ring* +
assumes *field-inverse*: $\llbracket a \in R; a \neq 0 \rrbracket \implies \text{multiplicative.invertible } a$
begin

Every field is an integral domain.

sublocale *integral-domain*

proof *unfold-locales*

fix *a b* **assume** *a*: $a \in R$ **and** *b*: $b \in R$ **and** *ab*: $a \cdot b = 0$

show $a = 0 \vee b = 0$

proof (*cases* $a = 0$)

case *False*

then have *inv*: *multiplicative.invertible* *a* **using** *a field-inverse* **by** *blast*

then have *ia*: *multiplicative.inverse* $a \in R$ **using** *a* **by** *simp*

have $b = \text{multiplicative.inverse } a \cdot (a \cdot b)$

using *a b inv* **by** (*simp add: multiplicative.invertible-left-inverse2*)

also have $\dots = \text{multiplicative.inverse } a \cdot 0$ **using** *ab* **by** *simp*

also have $\dots = 0$ **using** *ia* **by** (*rule right-zero*)

finally have $b = 0$.

then show *?thesis* ..

qed *simp*

qed

end

30.3 Ideals, Quotient Rings

p 101, ll 2–5

locale *ring-congruence* = *Ring* +

additive: *group-congruence* $R (+) 0 E +$

multiplicative: *Monoid-congruence* $R (\cdot) 1 E$

for *E*

begin

p 101, ll 2–5

notation *additive.quotient-composition* (**infixl** $\langle [+]\rangle$ 65)

notation *additive.quotient.inverse* ($\langle [-] \rightarrow$ [66] 65)

notation *multiplicative.quotient-composition* (**infixl** $\langle [\cdot]\rangle$ 70)

p 101, ll 5–11

sublocale *quotient*: *Ring* $R / E ([+]) ([\cdot])$ *additive.Class* **0** *additive.Class* **1**

by *unfold-locales*

(*auto simp: additive.Class-commutes-with-composition additive.associative additive.commutative*
multiplicative.Class-commutes-with-composition distributive elim!: additive.quotient-ClassE)

end

p 101, ll 12–13

locale *subgroup-of-additive-group-of-ring* =
additive: Subgroup I R (+) 0 + Ring R (+) (·) 0 1
for *I* **and** *R* **and** *addition* (**infixl** $\langle + \rangle$ 65) **and** *multiplication* (**infixl** $\langle \cdot \rangle$ 70) **and**
zero ($\langle 0 \rangle$) **and** *unit* ($\langle 1 \rangle$)
begin

p 101, ll 13–14

definition *Ring-Congruence* = $\{(a, b). a \in R \wedge b \in R \wedge a - b \in I\}$

p 101, ll 13–14

lemma *Ring-CongruenceI*: $\llbracket a - b \in I; a \in R; b \in R \rrbracket \implies (a, b) \in \text{Ring-Congruence}$
using *Ring-Congruence-def* **by** *blast*

p 101, ll 13–14

lemma *Ring-CongruenceD*: $(a, b) \in \text{Ring-Congruence} \implies a - b \in I$
using *Ring-Congruence-def* **by** *blast*

Jacobson’s definition of Ring congruence deviates from that of group congruence;
this complicates the proof.

p 101, ll 12–14

sublocale *additive: subgroup-of-abelian-group I R (+) 0*
rewrites *additive-congruence: additive.Congruence = Ring-Congruence*
proof –
show *subgroup-of-abelian-group I R (+) 0*
using *additive.commutative additive.invertible-right-inverse2* **by** *unfold-locales auto*
then interpret *additive: subgroup-of-abelian-group I R (+) 0* .
{
 fix *a b*
 assume [*simp*]: $a \in R \ b \in R$
 have $a - b \in I \longleftrightarrow - (a - b) \in I$ **by** *auto*
 also have $\dots \longleftrightarrow - a + b \in I$ **by** (*simp add: additive.commutative additive.inverse-composition-commute*)
 finally have $a - b \in I \longleftrightarrow - a + b \in I$.
}
then show *additive.Congruence = Ring-Congruence*
unfolding *additive.Congruence-def Ring-Congruence-def* **by** *auto*
qed

p 101, l 14

notation *additive.Left-Coset* (**infixl** $\langle + | \rangle$ 65)

end

Def 2.2

p 101, ll 21–22

locale *Ideal* = *subgroup-of-additive-group-of-ring* +
 assumes *Ideal*: $\llbracket a \in R; b \in I \rrbracket \implies a \cdot b \in I \llbracket a \in R; b \in I \rrbracket \implies b \cdot a \in I$

context *subgroup-of-additive-group-of-ring* **begin**

p 101, ll 14–17

theorem *multiplicative-congruence-implies-ideal*:

assumes *Monoid-congruence* *R* $(\cdot)$ **1** *Ring-Congruence*

shows *Ideal* *I* *R* $(+)$ $(\cdot)$ **0** **1**

proof –

interpret *multiplicative*: *Monoid-congruence* *R* $(\cdot)$ **1** *Ring-Congruence* **by** *fact*

show *?thesis*

proof

fix *a b*

assume [*simp*]: $a \in R \ b \in I$

have *congs*: $(a, a) \in \text{Ring-Congruence} \ (b, 0) \in \text{Ring-Congruence}$

by (*auto simp: additive.ClassD additive.Class-unit-normal-subgroup*)

from *congs* **have** $(a \cdot b, 0) \in \text{Ring-Congruence}$ **using** *multiplicative.cong* **by**

fastforce

then show $a \cdot b \in I$ **using** *additive.Class-unit-normal-subgroup* **by** *blast*

from *congs* **have** $(b \cdot a, 0) \in \text{Ring-Congruence}$ **using** *multiplicative.cong* **by**

fastforce

then show $b \cdot a \in I$ **using** *additive.Class-unit-normal-subgroup* **by** *blast*

qed

qed

end

context *Ideal* **begin**

p 101, ll 17–20

theorem *multiplicative-congruence* [*intro*]:

assumes *a*: $(a, a') \in \text{Ring-Congruence}$ **and** *b*: $(b, b') \in \text{Ring-Congruence}$

shows $(a \cdot b, a' \cdot b') \in \text{Ring-Congruence}$

proof –

note *Ring-CongruenceI* [*intro*] *Ring-CongruenceD* [*dest*]

from *a b* **have** [*simp*]: $a \in R \ a' \in R \ b \in R \ b' \in R$ **by** *auto*

from *a* **have** [*simp*]: $a - a' \in I$ **..**

have $a \cdot b - a' \cdot b = (a - a') \cdot b$ **by** (*simp add: distributive left-minus*)

also have $\dots \in I$ **by** (*simp add: Ideal*)

finally have *ab*: $a \cdot b - a' \cdot b \in I$. — ll 18–19

from *b* **have** [*simp*]: $b - b' \in I$ **..**

have $a' \cdot b - a' \cdot b' = a' \cdot (b - b')$ **by** (*simp add: distributive right-minus*)

also have $\dots \in I$ **by** (*simp add: Ideal*)
finally have $a'b': a' \cdot b - a' \cdot b' \in I$. — l 19
have $a \cdot b - a' \cdot b' = (a \cdot b - a' \cdot b) + (a' \cdot b - a' \cdot b')$
by (*simp add: additive.associative*) (*simp add: additive.associative [symmetric]*)
also have $\dots \in I$ **using** *ab a'b'* **by** *simp*
finally show $(a \cdot b, a' \cdot b') \in \text{Ring-Congruence}$ **by** *auto* — ll 19–20
qed

p 101, ll 23–24

sublocale *ring-congruence* **where** $E = \text{Ring-Congruence}$
by *unfold-locale (rule multiplicative-congruence)*

end

p 101, ll 24–26

locale *quotient-ring = Ideal* **begin**

p 101, ll 24–26

sublocale *quotient: Ring R / (subgroup-of-additive-group-of-ring.Ring-Congruence I*
 $R (+) \mathbf{0} ([+]) ([\cdot]) \text{additive.Class } \mathbf{0} \text{additive.Class } \mathbf{1} ..$

p 101, l 26

lemmas *Left-Coset = additive.Left-CosetE*

Equation 17 (1)

p 101, l 28

lemmas *quotient-addition = additive.factor-composition*

Equation 17 (2)

p 101, l 29

theorem *quotient-multiplication [simp]:*
 $\llbracket a \in R; b \in R \rrbracket \implies (a +| I) [\cdot] (b +| I) = a \cdot b +| I$
using *multiplicative.Class-commutes-with-composition additive.Class-is-Left-Coset*
by *auto*

p 101, l 30

lemmas *quotient-zero = additive.factor-unit*
lemmas *quotient-negative = additive.factor-inverse*

end

30.4 Homomorphisms of Rings. Basic Theorems

Def 2.3

p 106, ll 7–9

locale *ring-homomorphism* =
 map η *R* *R'* + source: *Ring* *R* (+) (·) **0** **1** + target: *Ring* *R'* (+') (·') **0'** **1'** +
 additive: group-homomorphism η *R* (+) **0** *R'* (+') **0'** +
 multiplicative: Monoid-homomorphism η *R* (·) **1** *R'* (·') **1'**
for η
 and *R* and addition (infixl <+> 65) and multiplication (infixl <·> 70) and zero
 (<0>) and unit (<1>)
 and *R'* and addition' (infixl <+'> 65) and multiplication' (infixl <·'> 70) and
 zero' (<0'>) and unit' (<1'>)

p 106, l 17

locale *ring-epimorphism* = *ring-homomorphism* + *surjective-map* η *R* *R'*

p 106, ll 14–18

sublocale *quotient-ring* $\subseteq$ *natural: ring-epimorphism*
where η = *additive.Class* and *R'* = *R* / (*subgroup-of-additive-group-of-ring.Ring-Congruence*
I *R* (+) **0**) and addition' = ([+]) and multiplication' = ([·])
 and zero' = *additive.Class* **0** and unit' = *additive.Class* **1**
 ..

context *ring-homomorphism* **begin**

Jacobson reasons via $a - b \in \text{additive.Ker}$ being a congruence; we prefer the
 direct proof, since it is very simple.

p 106, ll 19–21

sublocale *kernel: Ideal* **where** *I* = *additive.Ker*
by *unfold-locale* (*auto simp: additive.Ker-image multiplicative.commutates-with-composition*)

end

p 106, l 22

locale *ring-monomorphism* = *ring-homomorphism* + *injective-map* η *R* *R'*

context *ring-homomorphism* **begin**

p 106, ll 21–23

theorem *ring-monomorphism-iff-kernel-unit*:
ring-monomorphism η *R* (+) (·) **0** *R'* (+') (·') **0'** **1'** $\longleftrightarrow$ *additive.Ker* = {**0**} (**is**
?monom $\longleftrightarrow$ *?ker*)
proof
 assume *?monom* **then interpret** *ring-monomorphism* . **show** *?ker* **by** (*simp add:*
additive.injective-iff-kernel-unit [symmetric])
next
 assume *?ker* **then show** *?monom* **by** *unfold-locale* (*simp add: additive.injective-iff-kernel-unit*)
qed

end

p 106, ll 23–25

sublocale *ring-homomorphism* $\subseteq$ *image*: Subring $\eta \text{ ‘ } R \text{ } R' (+') (\cdot') \mathbf{0}' \mathbf{1}' \dots$

p 106, ll 26–27

locale *ideal-in-kernel* =
ring-homomorphism + *contained*: Ideal $I \text{ } R (+) (\cdot) \mathbf{0} \mathbf{1}$ **for** $I +$
assumes *subset*: $I \subseteq \text{additive.Ker}$
begin

p 106, ll 26–27

notation *contained.additive.quotient-composition* (**infixl** $\langle [+]\rangle$ 65)
notation *contained.multiplicative.quotient-composition* (**infixl** $\langle [\cdot]\rangle$ 70)

Provides *additive.induced*, which Jacobson calls $\bar{\eta}$.

p 106, l 30

sublocale *additive: normal-subgroup-in-kernel* $\eta \text{ } R (+) \mathbf{0} \text{ } R' (+') \mathbf{0}' \text{ } I$
rewrites *normal-subgroup.Congruence* $I \text{ } R \text{ addition zero} = \text{contained.Ring-Congruence}$
by *unfold-locale* (rule *subset contained.additive-congruence*) +

Only the multiplicative part needs some work.

p 106, ll 27–30

sublocale *induced: ring-homomorphism additive.induced* $R / (\text{subgroup-of-additive-group-of-ring.Ring-Congruence}$
 $I \text{ } R (+) \mathbf{0} ([+]) ([\cdot]) \text{ contained.additive.Class } \mathbf{0} \text{ contained.additive.Class } \mathbf{1}$
using *contained.multiplicative.Class-commutes-with-composition*
by *unfold-locale*
(auto elim!: *contained.additive.Left-CosetE simp:* *contained.additive.Class-is-Left-Coset*
multiplicative.commutes-with-composition multiplicative.commutes-with-unit)

p 106, l 30; p 107, ll 1–3

additive.induced denotes Jacobson’s $\bar{\eta}$. We have the commutativity of the diagram, where *additive.induced* is unique:

compose $R \text{ } \text{additive.induced} \text{ contained.additive.Class} = \eta$

$\llbracket ?\beta \in \text{contained.additive.Partition} \rightarrow_E R';$
compose $R \text{ } ?\beta \text{ contained.additive.Class} = \eta \rrbracket$
 $\implies ?\beta = \text{additive.induced}$

end

Fundamental Theorem of Homomorphisms of Rings

p 107, l 6

locale *ring-homomorphism-fundamental* = *ring-homomorphism* **begin**

p 107, l 6

notation *kernel.additive.quotient-composition* (**infixl** $\langle [+]\rangle$ 65)
notation *kernel.multiplicative.quotient-composition* (**infixl** $\langle [\cdot]\rangle$ 70)

p 107, l 6

sublocale *ideal-in-kernel* **where** $I = \text{additive.Ker}$ **by** *unfold-locates rule*

p 107, ll 8–9

sublocale *natural: ring-epimorphism*

where $\eta = \text{kernel.additive.Class}$ **and** $R' = R / (\text{subgroup-of-additive-group-of-ring.Ring-Congruence}$
 $\text{additive.Ker } R \ (+) \ 0)$
and $\text{addition}' = \text{kernel.additive.quotient-composition}$
and $\text{multiplication}' = \text{kernel.multiplicative.quotient-composition}$
and $\text{zero}' = \text{kernel.additive.Class } 0$ **and** $\text{unit}' = \text{kernel.additive.Class } 1$
..

p 107, l 9

sublocale *induced: ring-monomorphism*

where $\eta = \text{additive.induced}$ **and** $R = R / (\text{subgroup-of-additive-group-of-ring.Ring-Congruence}$
 $\text{additive.Ker } R \ (+) \ 0)$
and $\text{addition} = \text{kernel.additive.quotient-composition}$
and $\text{multiplication} = \text{kernel.multiplicative.quotient-composition}$
and $\text{zero} = \text{kernel.additive.Class } 0$ **and** $\text{unit} = \text{kernel.additive.Class } 1$
by *unfold-locates (simp add: additive.induced-inj-on)*

end

p 107, l 11

locale *ring-isomorphism* = *ring-homomorphism* + *bijective-map* η R R' **begin**

p 107, l 11

sublocale *ring-monomorphism* **..**

sublocale *ring-epimorphism* **..**

p 107, l 11

lemma *inverse-ring-isomorphism:*

ring-isomorphism (restrict (inv-into R η) R') R' $(+)$ $(\cdot)$ $0'$ $1'$ R $(+)$ $(\cdot)$ 0 1

proof

fix x y
assume $x \in R'$ **and** $y \in R'$
then obtain u v **where** $*$: η $u = x$ η $v = y$ $u \in R$ $v \in R$
using *surjective* **by** *blast*
then show $\text{restrict } (\text{inv-into } R \ \eta) \ R' \ (x \ +' \ y) = \text{restrict } (\text{inv-into } R \ \eta) \ R' \ x \ +$
 $\text{restrict } (\text{inv-into } R \ \eta) \ R' \ y$
by *(auto simp flip: additive.commutes-with-composition)*
show $\text{restrict } (\text{inv-into } R \ \eta) \ R' \ (x \ \cdot' \ y) = \text{restrict } (\text{inv-into } R \ \eta) \ R' \ x \ \cdot \ \text{restrict}$
 $(\text{inv-into } R \ \eta) \ R' \ y$
using $*$ **by** *(auto simp: inv-into-f-eq multiplicative.commutes-with-composition)*
next


```

show restrict (inv-into R η) R' 0' = 0
  using additive.commutes-with-unit by force
show restrict (inv-into R η) R' 1' = 1
  using multiplicative.commutes-with-unit by auto
show bij-betw (restrict (inv-into R η) R') R' R
  using bij-betw-inverse by blast
qed

```

end

p 107, l 11

```

definition isomorphic-as-rings (infixl <≅R> 50)
  where  $\mathcal{R} \cong_R \mathcal{R}' \iff (\text{let } (R, \text{addition}, \text{multiplication}, \text{zero}, \text{unit}) = \mathcal{R}; (R', \text{addition}', \text{multiplication}', \text{zero}', \text{unit}') = \mathcal{R}' \text{ in}$ 
     $(\exists \eta. \text{ring-isomorphism } \eta \ R \ \text{addition} \ \text{multiplication} \ \text{zero} \ \text{unit} \ R' \ \text{addition}' \ \text{multiplication}' \ \text{zero}' \ \text{unit}'))$ 

```

p 107, l 11

```

lemma isomorphic-as-rings-symmetric:
   $(R, \text{addition}, \text{multiplication}, \text{zero}, \text{unit}) \cong_R (R', \text{addition}', \text{multiplication}', \text{zero}', \text{unit}') \implies$ 
   $(R', \text{addition}', \text{multiplication}', \text{zero}', \text{unit}') \cong_R (R, \text{addition}, \text{multiplication}, \text{zero}, \text{unit})$ 
  by (simp add: isomorphic-as-rings-def) (meson ring-isomorphism.inverse-ring-isomorphism)

```

context ring-homomorphism begin

Corollary

p 107, ll 11–12

```

theorem image-is-isomorphic-to-quotient-ring:
   $\exists K \ \text{add} \ \text{mult} \ \text{zero} \ \text{one}. \text{Ideal } K \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \wedge (\eta \text{ ' } R, (+)', (\cdot)', \mathbf{0}', \mathbf{1}') \cong_R (R$ 
   $/ (\text{subgroup-of-additive-group-of-ring.Ring-Congruence } K \ R \ (+) \ \mathbf{0}), \text{add}, \text{mult}, \text{zero},$ 
   $\text{one})$ 

```

proof –

```

interpret image: ring-homomorphism-fundamental where R' = η ' R
  by unfold-locales (auto simp: target.additive.commutative additive.commutes-with-composition
    multiplicative.commutes-with-composition target.distributive multiplicative.commutes-with-unit)
  have ring-isomorphism image.additive.induced (R / (subgroup-of-additive-group-of-ring.Ring-Congruence
    additive.Ker R (+) 0)) ([+]) ([·]) (kernel.additive.Class 0) (kernel.additive.Class 1)
    (η ' R) (+') (·') 0' 1'
  by unfold-locales (simp add: image.additive.induced-image bij-betw-def)
  then have  $(\eta \text{ ' } R, (+)', (\cdot)', \mathbf{0}', \mathbf{1}') \cong_R (R / (\text{subgroup-of-additive-group-of-ring.Ring-Congruence}$ 
     $\text{additive.Ker } R \ (+) \ \mathbf{0}), ([+]), ([\cdot]), \text{kernel.additive.Class } \mathbf{0}, \text{kernel.additive.Class } \mathbf{1})$ 
  by (simp add: isomorphic-as-rings-def) (meson ring-isomorphism.inverse-ring-isomorphism)
  moreover have Ideal additive.Ker R (+) (·) 0 1 ..
  ultimately show ?thesis by blast
qed

```

end

30.5 Abstract type of rings

```

lemma trivial-Ring': Ring' trivial-monoid trivial-monoid
proof
qed (simp-all add: Monoid.mcarrier-monoid Monoid.munit-monoid trivial-Monoid
      trivial-monoid-def Monoid.mmult-monoid trivial-Monoid trivial-Monoid-invertible)

lemma trivial-Ring: Ring {undefined} ( $\lambda x y.$  undefined) ( $\lambda x y.$  undefined) undefined
undefined
proof –
  have commutative-monoid-axioms {undefined} ( $\lambda x y.$  undefined)
    by (simp add: commutative-monoid-axioms.intro)
  moreover have Ring-axioms {undefined} ( $\lambda x y.$  undefined) ( $\lambda x y.$  undefined)
    by (simp add: Ring-axioms.intro)
  ultimately show ?thesis
    by (auto simp: Ring-def Abelian-Group-def commutative-monoid-def trivial-Group
      trivial-Monoid)
qed

typedef 'a ring = {(addition::'a monoid, multiplication). Ring' addition multiplication}
  morphisms dest-ring ring
  using trivial-Ring' by blast

declare dest-ring-inverse [simp]

definition trivial-ring  $\equiv$  ring (trivial-monoid, trivial-monoid)

definition radd where radd m  $\equiv$  fst (dest-ring m)

definition rmult where rmult m  $\equiv$  snd (dest-ring m)

definition rcarrier where rcarrier m  $\equiv$  mcarrier (radd m)

lemma ring-is-Ring [iff]: Ring' (radd m) (rmult m)
  by (metis Product-Type.Collect-case-prodD dest-ring radd-def rmult-def)

lemma ring-collapse [simp]: ring (radd r, rmult r) = r
  by (simp add: radd-def rmult-def)

Allows reference to the current ring space within the locale as a value
definition (in Ring') Self  $\equiv$  ring (additive, multiplicative)

lemma (in Ring') radd-Self [simp]: radd Self = additive
  by (simp add: Ring'-axioms Self-def radd-def ring-inverse)

lemma (in Ring') rmult-Self [simp]: rmult Self = multiplicative
  by (simp add: Ring'-axioms Self-def ring-inverse rmult-def)

Further accessors, mirroring the 'a monoid type: the additive and multiplicative

```

operations and the two distinguished constants **0**, **1** recovered from the component monoids. (We keep *radd/rmult* returning whole '*a monoid*' values, as *Ring'* is built on those.)

definition *radd-op* **where** *radd-op* *m* $\equiv$ *mmult* (*radd* *m*)

definition *rmult-op* **where** *rmult-op* *m* $\equiv$ *mmult* (*rmult* *m*)

definition *rzero* **where** *rzero* *m* $\equiv$ *munit* (*radd* *m*)

definition *rone* **where** *rone* *m* $\equiv$ *munit* (*rmult* *m*)

The value really is a ring: the analogue of *monoid-is-Monoid*.

lemma *Ring-ring* [iff]: *Ring* (*rcarrier* *m*) (*radd-op* *m*) (*rmult-op* *m*) (*rzero* *m*) (*rone* *m*)

using *Ring'.Ring* [*OF ring-is-Ring* [*of m*]]

by (*simp* *add*: *rcarrier-def* *radd-op-def* *rmult-op-def* *rzero-def* *rone-def*)

Computing the accessors of a ring built from its component monoids.

lemma (**in** *Ring'*) *radd-ring* [*simp*]: *radd* (*ring* (*additive*, *multiplicative*)) = *additive*
by (*simp* *add*: *Ring'-axioms* *radd-def* *ring-inverse*)

lemma (**in** *Ring'*) *rmult-ring* [*simp*]: *rmult* (*ring* (*additive*, *multiplicative*)) = *multiplicative*
by (*simp* *add*: *Ring'-axioms* *rmult-def* *ring-inverse*)

lemma (**in** *Ring'*) *rcarrier-ring* [*simp*]: *rcarrier* (*ring* (*additive*, *multiplicative*)) = *mcarrier* *additive*
by (*simp* *add*: *rcarrier-def*)

lemma (**in** *Ring'*) *rzero-ring* [*simp*]: *rzero* (*ring* (*additive*, *multiplicative*)) = *munit* *additive*
by (*simp* *add*: *rzero-def*)

lemma (**in** *Ring'*) *rone-ring* [*simp*]: *rone* (*ring* (*additive*, *multiplicative*)) = *munit* *multiplicative*
by (*simp* *add*: *rone-def*)

lemma (**in** *Ring'*) *radd-op-ring* [*simp*]: *radd-op* (*ring* (*additive*, *multiplicative*)) = *mmult* *additive*
by (*simp* *add*: *radd-op-def*)

lemma (**in** *Ring'*) *rmult-op-ring* [*simp*]: *rmult-op* (*ring* (*additive*, *multiplicative*)) = *mmult* *multiplicative*
by (*simp* *add*: *rmult-op-def*)

Within the mathematical *Ring* locale, the component monoids form a *Ring'*, so the current ring can be referred to as a value.

lemma (**in** *Ring*) *Ring'-Self-components*:

$\text{Ring}' (\text{monoid } (R, (+), \mathbf{0})) (\text{monoid } (R, (\cdot), \mathbf{1}))$
proof –
have a : $\text{mcarrier } (\text{monoid } (R, (+), \mathbf{0})) = R \text{ mmult } (\text{monoid } (R, (+), \mathbf{0})) = (+)$
 $\text{munit } (\text{monoid } (R, (+), \mathbf{0})) = \mathbf{0}$
by (*simp-all add: additive.mcarrier-monoid additive.mmult-monoid additive.munit-monoid*)
have m : $\text{mcarrier } (\text{monoid } (R, (\cdot), \mathbf{1})) = R \text{ mmult } (\text{monoid } (R, (\cdot), \mathbf{1})) = (\cdot)$
 $\text{munit } (\text{monoid } (R, (\cdot), \mathbf{1})) = \mathbf{1}$
by (*simp-all add: multiplicative.mcarrier-monoid multiplicative.mmult-monoid multiplicative.munit-monoid*)
show ?thesis
by (*simp add: Ring'-def a m Ring-axioms*)
qed

definition (in *Ring*) $\text{Self} \equiv \text{ring } (\text{monoid } (R, (+), \mathbf{0}), \text{monoid } (R, (\cdot), \mathbf{1}))$

lemma (in *Ring*) $\text{rcarrier-Self} [\text{simp}]: \text{rcarrier Self} = R$
by (*simp add: Self-def Ring'.rcarrier-ring Ring'-Self-components additive.mcarrier-monoid*)

lemma (in *Ring*) $\text{rzero-Self} [\text{simp}]: \text{rzero Self} = \mathbf{0}$
by (*simp add: Self-def Ring'.rzero-ring Ring'-Self-components additive.munit-monoid*)

lemma (in *Ring*) $\text{rone-Self} [\text{simp}]: \text{rone Self} = \mathbf{1}$
by (*simp add: Self-def Ring'.rone-ring Ring'-Self-components multiplicative.munit-monoid*)

lemma (in *Ring*) $\text{radd-op-Self} [\text{simp}]: \text{radd-op Self} = (+)$
by (*simp add: Self-def Ring'.radd-op-ring Ring'-Self-components additive.mmult-monoid*)

lemma (in *Ring*) $\text{rmult-op-Self} [\text{simp}]: \text{rmult-op Self} = (\cdot)$
by (*simp add: Self-def Ring'.rmult-op-ring Ring'-Self-components multiplicative.mmult-monoid*)

The one abstract '*a ring*' type serves all the ring-like structures: which one a value belongs to is recorded by a predicate on its accessors. (We provide the field predicate, used in Galois theory; the others follow the same shape.)

definition *is-field* **where**
 $\text{is-field } r \equiv \text{Field } (\text{rcarrier } r) (\text{radd-op } r) (\text{rmult-op } r) (\text{rzero } r) (\text{rone } r)$

lemma *is-fieldI*:
 $\text{Field } (\text{rcarrier } r) (\text{radd-op } r) (\text{rmult-op } r) (\text{rzero } r) (\text{rone } r) \implies \text{is-field } r$
by (*simp add: is-field-def*)

lemma *is-fieldD*:
 $\text{is-field } r \implies \text{Field } (\text{rcarrier } r) (\text{radd-op } r) (\text{rmult-op } r) (\text{rzero } r) (\text{rone } r)$
by (*simp add: is-field-def*)

30.6 Linking the locale-based field theory to Isabelle's type classes

Following the *Met-TC* pattern of *HOL-Analysis-Abstract-Metric-Spaces* (where interpretation *Met-TC*: *Metric-space UNIV dist* feeds the type-class operations

into the locale), we interpret the locale-based *Field* at an arbitrary type-class field, with the whole type as carrier. This lets the locale-based constructions to come (polynomials, splitting fields, Galois groups) be applied to concrete type-class fields such as *real* and *complex*.

The additive-group, multiplicative-monoid and ring bridge lemmas need only the ring axioms, so we state them at the weakest suitable type class *comm-ring-1* rather than *field*. This lets the same bridge serve integral domains (e.g. *int*, in theory *IntRing*), not just fields.

lemma *add-group-TC*: *Group* (*UNIV* :: 'a :: group-add set) *plus 0*

proof (*rule GroupI*)

fix *u* :: 'a

show $\exists v \in \text{UNIV}. \text{plus } u \ v = 0 \wedge \text{plus } v \ u = 0$ **by** (*rule bezI* [**where** *x* = *uminus u*]) *simp-all*

qed (*auto simp: algebra-simps*)

lemma *add-abelian-TC*: *Abelian-Group* (*UNIV* :: 'a :: ab-group-add set) *plus 0*

proof (*rule Abelian-Group.intro* [*OF add-group-TC*])

show *commutative-monoid UNIV plus (0::'a)*

by *unfold-locales (simp-all add: ac-simps)*

qed

lemma *mult-monoid-TC*: *Monoid* (*UNIV* :: 'a :: monoid-mult set) *(*) 1*

by *unfold-locales (auto simp: ac-simps)*

lemma *mult-cmonoid-TC*: *commutative-monoid* (*UNIV* :: 'a :: comm-monoid-mult set) *(*) 1*

by *unfold-locales (auto simp: ac-simps)*

lemma *ring-TC*: *Ring* (*UNIV* :: 'a :: ring-1 set) *plus times 0 1*

proof (*intro Ring.intro add-abelian-TC*)

show *Monoid (UNIV::'a set) (*) 1*

using *mult-monoid-TC* **by** *blast*

show *Ring-axioms (UNIV::'a set) plus (*)* **by** *unfold-locales (auto simp: algebra-simps)*

qed

lemma *comm-ring-TC*: *commutative-ring* (*UNIV* :: 'a :: comm-ring-1 set) *plus (*) 0 1*

by (*rule commutative-ring.intro* [*OF ring-TC mult-cmonoid-TC*])

At an integral domain, the locale *integral-domain* holds: the type class provides nontriviality ($0 \neq 1$) and the absence of zero divisors ($((?a * ?b = 0) = (?a = 0 \vee ?b = 0))$).

lemma *idom-TC*: *integral-domain* (*UNIV* :: 'a :: idom set) *plus (*) 0 1*

proof –

interpret *R*: *commutative-ring UNIV :: 'a set plus (*) 0 1* **by** (*rule comm-ring-TC*)

show *?thesis*

proof (*unfold-locales*)

```

    show (1::'a) ≠ 0 by simp
  next
    fix a b :: 'a assume a * b = 0 then show a = 0 ∨ b = 0 by simp
  qed
qed

interpretation field-TC: Field UNIV :: 'a :: field set plus (*) 0 1
proof -
  interpret R: commutative-ring UNIV :: 'a set plus (*) 0 1 by (rule comm-ring-TC)
  show Field (UNIV :: 'a set) plus (*) 0 1
  proof
    fix a :: 'a assume a ≠ 0
    then have a * inverse a = 1 inverse a * a = 1 by simp-all
    then show R.multiplicative.invertible a
      by (intro R.multiplicative.invertibleI [where v = inverse a]) auto
  qed auto
qed

```

The interpretation gives, for every type-class field, the locale-based field structure on the whole type; e.g. its multiplicative inverse coincides with *inverse* on nonzero elements.

```

lemma TC-inverse-eq:
  fixes a :: 'a :: field
  assumes a ≠ 0
  shows field-TC.multiplicative.inverse a = inverse a
  using assms by (intro field-TC.multiplicative.inverse-equality) simp-all
end

```

31 Generic field automorphisms and fixed fields

```

theory Galois-Automorphism
  imports Group-Theory Subfield
begin

```

This theory supplies the representation-independent core of the Galois development. A field automorphism is a bijective endomorphism of the ambient type whose field laws hold on a subfield K and which fixes a base subset F . The definition therefore works for every type-class field; the complex-specific locale and conjugation lemmas remain in the complex-field extension theory.

31.1 Automorphisms are additive, multiplicative and fix the fixed field

```

definition field-auto ::
  'a :: field set ⇒ 'a set ⇒ ('a ⇒ 'a) set where
  field-auto K F =
    {σ. σ ∈ K →E K ∧ bij-betw σ K K

```

$$\begin{aligned}
& \wedge (\forall x \in K. \forall y \in K. \sigma (x + y) = \sigma x + \sigma y) \\
& \wedge (\forall x \in K. \forall y \in K. \sigma (x * y) = \sigma x * \sigma y) \\
& \wedge \sigma 1 = 1 \\
& \wedge (\forall x \in F. \sigma x = x) \}
\end{aligned}$$

lemma *field-auto-mem-iff*:

$\sigma \in \text{field-auto } K F \longleftrightarrow$
 $\sigma \in K \rightarrow_E K \wedge \text{bij-betw } \sigma K K$
 $\wedge (\forall x \in K. \forall y \in K. \sigma (x + y) = \sigma x + \sigma y)$
 $\wedge (\forall x \in K. \forall y \in K. \sigma (x * y) = \sigma x * \sigma y)$
 $\wedge \sigma 1 = 1$
 $\wedge (\forall x \in F. \sigma x = x)$
by (*simp add: field-auto-def*)

lemma *field-auto-zero*:

assumes *Ksub*: *Subfield K* **and** *s*: $\sigma \in \text{field-auto } K F$
shows $\sigma 0 = 0$
using *Ksub s* **by** (*metis Subfield.zero-closed add-cancel-right-right field-auto-mem-iff*)

lemma *field-auto-one*:

$\sigma \in \text{field-auto } K F \implies \sigma 1 = 1$
by (*simp add: field-auto-def*)

lemma *field-auto-closed*:

assumes $\sigma \in \text{field-auto } K F$ **and** $x \in K$
shows $\sigma x \in K$
using *assms bij-betwE* **by** (*fastforce simp: field-auto-def*)

lemma *field-auto-add*:

assumes *s*: $\sigma \in \text{field-auto } K F$ **and** $x: x \in K$ **and** $y: y \in K$
shows $\sigma (x + y) = \sigma x + \sigma y$
using *s x y* **by** (*auto simp: field-auto-def*)

lemma *field-auto-mult*:

assumes *s*: $\sigma \in \text{field-auto } K F$ **and** $x: x \in K$ **and** $y: y \in K$
shows $\sigma (x * y) = \sigma x * \sigma y$
using *s x y* **by** (*auto simp: field-auto-def*)

lemma *field-auto-uminus*:

assumes *Subfield K* $\sigma \in \text{field-auto } K F$ **and** $x: x \in K$
shows $\sigma (-x) = -\sigma x$
using *assms Subfield.uminus-closed add-eq-0-iff2 field-auto-add field-auto-zero x*
by *fastforce*

Additivity extends to finite sums, by induction on the index set.

lemma *field-auto-sum*:

assumes *K*: *Subfield K* **and** *s*: $\sigma \in \text{field-auto } K F$
and $x: \bigwedge j. j \in J \implies x j \in K$
shows $\sigma (\sum j \in J. x j) = (\sum j \in J. \sigma (x j))$
using *assms field-auto-zero*

by (induction J rule: infinite-finite-induct) (auto simp add: Subfield.sum-closed field-auto-add)

lemma *field-auto-as-hom*:
 assumes *Ksub*: Subfield *K* and *s*: $\sigma \in \text{field-auto } K \ F$
 shows *field-hom-on* *K* σ
proof –
 interpret *KS*: Subfield *K* by (rule *Ksub*)
 show ?thesis
proof
 show $\sigma \ 0 = 0$
 using *Ksub* field-auto-zero *s* by blast
 show $\sigma \ 1 = 1$
 using *Ksub* field-auto-one *s* by blast
 qed (use *s* in ⟨auto simp: field-auto-def⟩)
qed

lemma *field-auto-inverse-on*:
 assumes *Ksub*: Subfield *K* and *s*: $\sigma \in \text{field-auto } K \ F$ and *x*: $x \in K$
 shows $\sigma (\text{inverse } x) = \text{inverse } (\sigma \ x)$
 using field-auto-as-hom[OF *Ksub* *s*] *x* by (rule field-hom-on.hom-inverse-all)

lemma *field-auto-subset-Sym*:
 $\text{field-auto } K \ F \subseteq \text{transformations.Sym } K$
 by (auto simp: field-auto-def transformations.Units-bijective)

lemma *identity-apply*: $x \in S \implies \text{identity } S \ x = x$
 by (simp add: restrict-apply')

lemma *field-auto-identity*:
 assumes *Ksub*: Subfield *K* and *FK*: $F \subseteq K$
 shows *identity* $K \in \text{field-auto } K \ F$
proof –
 interpret *KS*: Subfield *K* by (rule *Ksub*)
 show ?thesis
 unfolding field-auto-mem-iff
proof (intro conjI ballI)
 show *identity* $K \in K \rightarrow_E K$
 by (auto simp: identity-apply PiE-iff)
 show *bij-betw* (*identity* *K*) *K* *K*
 by (simp add: bij-betw-def inj-on-def identity-apply)
 show $\bigwedge x. x \in F \implies \text{identity } K \ x = x$
 using *FK* by (auto simp: identity-apply)
 qed (auto simp: identity-apply *KS*.one-closed *KS*.add-closed *KS*.mult-closed)
qed

lemma *field-auto-compose*:
 assumes *Ksub*: Subfield *K* and *FK*: $F \subseteq K$
 and *s*: $\sigma \in \text{field-auto } K \ F$ and *t*: $\tau \in \text{field-auto } K \ F$
 shows *compose* *K* $\sigma \ \tau \in \text{field-auto } K \ F$


```

proof –
interpret KS: Subfield K by (rule Ksub)
from s have sbij: bij-betw  $\sigma$  K K
  and sext:  $\sigma \in K \rightarrow_E K$ 
  and sadd:  $\bigwedge x y. x \in K \implies y \in K \implies \sigma (x + y) = \sigma x + \sigma y$ 
  and smul:  $\bigwedge x y. x \in K \implies y \in K \implies \sigma (x * y) = \sigma x * \sigma y$ 
  and sone:  $\sigma 1 = 1$  and sfix:  $\bigwedge x. x \in F \implies \sigma x = x$ 
  by (auto simp: field-auto-def)
from t have tbij: bij-betw  $\tau$  K K
  and text2:  $\tau \in K \rightarrow_E K$ 
  and tadd:  $\bigwedge x y. x \in K \implies y \in K \implies \tau (x + y) = \tau x + \tau y$ 
  and tmul:  $\bigwedge x y. x \in K \implies y \in K \implies \tau (x * y) = \tau x * \tau y$ 
  and tone:  $\tau 1 = 1$  and tfix:  $\bigwedge x. x \in F \implies \tau x = x$ 
  by (auto simp: field-auto-def)
have tinK:  $\bigwedge x. x \in K \implies \tau x \in K$  using tbij bij-betwE by blast
let ?c = compose K  $\sigma$   $\tau$ 
show ?thesis
  unfolding field-auto-mem-iff
proof (intro conjI ballI)
  show  $?c \in K \rightarrow_E K$  using sext text2 by (simp add: compose-def PiE-iff)
  show bij-betw ?c K K using sbij tbij by (simp add: bij-betw-compose)
  show  $?c 1 = 1$  using KS.one-closed by (simp add: compose-eq tone sone)
  show  $\bigwedge x y. x \in K \implies y \in K \implies ?c (x + y) = ?c x + ?c y$ 
    by (simp add: compose-eq KS.add-closed tinK tadd sadd)
  show  $\bigwedge x y. x \in K \implies y \in K \implies ?c (x * y) = ?c x * ?c y$ 
    by (simp add: compose-eq KS.mult-closed tinK tmul smul)
  show  $\bigwedge x. x \in F \implies ?c x = x$ 
    using FK sfix tfix by (metis compose-eq in-mono)
qed
qed

lemma field-auto-inverse:
  assumes Ksub: Subfield K and s:  $\sigma \in \text{field-auto } K F$  and FK:  $F \subseteq K$ 
  shows restrict (inv-into K  $\sigma$ ) K  $\in \text{field-auto } K F$ 
proof –
interpret KS: Subfield K by (rule Ksub)
have bij: bij-betw  $\sigma$  K K
  and addh:  $\bigwedge x y. x \in K \implies y \in K \implies \sigma (x + y) = \sigma x + \sigma y$ 
  and mulh:  $\bigwedge x y. x \in K \implies y \in K \implies \sigma (x * y) = \sigma x * \sigma y$ 
  and one:  $\sigma 1 = 1$  and fixed:  $\bigwedge x. x \in F \implies \sigma x = x$ 
  using s by (auto simp: field-auto-def)
define t where t  $\equiv \text{restrict } (\text{inv-into } K \sigma) K$ 
have tin:  $\bigwedge y. y \in K \implies t y \in K$ 
  using bij by (simp add: t-def inv-into-into bij-betw-def)
have st:  $\bigwedge y. y \in K \implies \sigma (t y) = y$ 
  using bij by (simp add: t-def f-inv-into-f bij-betw-def)
have ts:  $\bigwedge x. x \in K \implies t (\sigma x) = x$ 
  using bij by (simp add: t-def bij-betwE inv-into-f-f bij-betw-def)
show ?thesis
  unfolding field-auto-mem-iff

```

```

proof (intro conjI strip)
  show bij-betw (restrict (inv-into K  $\sigma$ ) K) K K
    by (simp add: bij bij-betw-inv-into)
  show restrict (inv-into K  $\sigma$ ) K  $\in$  K  $\rightarrow_E$  K
    by (simp add: bij bij-betw-imp-funcset bij-betw-inv-into)
  show  $\bigwedge x y. x \in K \implies y \in K \implies$ 
    restrict (inv-into K  $\sigma$ ) K (x + y) = restrict (inv-into K  $\sigma$ ) K x + restrict
(inv-into K  $\sigma$ ) K y
    using KS.add-closed addh st t-def tin ts by metis
  show  $\bigwedge x y. x \in K \implies y \in K \implies$ 
    restrict (inv-into K  $\sigma$ ) K (x * y) = restrict (inv-into K  $\sigma$ ) K x * restrict
(inv-into K  $\sigma$ ) K y
    using KS.mult-closed mulh st t-def tin ts by metis
  show restrict (inv-into K  $\sigma$ ) K 1 = 1
    using one t-def ts by fastforce
  show  $\bigwedge x. x \in F \implies \text{restrict } (\text{inv-into } K \ \sigma) \ K \ x = x$ 
    using FK fixed t-def ts by (metis subsetD)
qed
qed

```

```

context
  fixes K F :: 'a :: field set
  assumes Ksub: Subfield K and FK: F  $\subseteq$  K
begin

```

interpretation *T*: *transformations* *K* .

```

lemma field-auto-restrict-inv-in-Sym:
  assumes a  $\in$  T.Sym
  shows restrict (inv-into K a) K  $\in$  T.Sym
proof –
  have bij-betw a K K using assms by (rule T.Units-bij-betwI)
  then show ?thesis
    by (simp add: bij-betw-imp-funcset bij-betw-inv-into)
qed

```

```

lemma field-auto-Sym-inverse-eq-inv-into:
  assumes a  $\in$  T.Sym
  shows T.symmetric.inverse a = restrict (inv-into K a) K
proof (intro T.symmetric.inverse-equality field-auto-restrict-inv-in-Sym assms)
  show compose K a (restrict (inv-into K a) K) = identity K
    by (simp add: assms bij-betw-imp-surj-on compose-id-inv-into)
  show compose K (restrict (inv-into K a) K) a = identity K
    by (simp add: assms compose-inv-into-id)
qed

```

```

theorem field-auto-subgroup:
  Subgroup (field-auto K F) (transformations.Sym K) (compose K) (identity K)
proof (intro T.symmetric.subgroupI field-auto-subset-Sym)
  show identity K  $\in$  field-auto K F using Ksub FK by (rule field-auto-identity)

```

```

next
  fix a b assume a ∈ field-auto K F b ∈ field-auto K F
  then show compose K a b ∈ field-auto K F
    using Ksub FK by (blast intro: field-auto-compose)
next
  fix a assume a: a ∈ field-auto K F
  then have aSym: a ∈ T.Sym using field-auto-subset-Sym by blast
  then show T.symmetric.inverse a ∈ field-auto K F
    by (simp add: FK Ksub field-auto-Sym-inverse-eq-inv-into a field-auto-inverse)
next
  fix g assume g ∈ field-auto K F
  then have g ∈ T.Sym using field-auto-subset-Sym by blast
  then show T.symmetric.invertible g by (simp add: T.mem-UnitsD)
qed

```

corollary *field-auto-group*:
 Group (field-auto K F) (compose K) (identity K)
 by (rule subgroup-imp-Group[OF field-auto-subgroup])

end

definition *fixed-field* ::
 'a :: field set $\Rightarrow$ ('a $\Rightarrow$ 'a) set $\Rightarrow$ 'a set **where**
fixed-field K H $\equiv$ {x ∈ K. $\forall \sigma \in H. \sigma x = x$ }

lemma *fixed-field-subset*: *fixed-field* K H $\subseteq$ K
 by (auto simp: fixed-field-def)

lemma *fixed-field-memI*:
 $\llbracket x \in K; \bigwedge \sigma. \sigma \in H \Longrightarrow \sigma x = x \rrbracket \Longrightarrow x \in \text{fixed-field } K H$
 by (simp add: fixed-field-def)

lemma *fixed-field-memD*:
 $x \in \text{fixed-field } K H \Longrightarrow x \in K \wedge (\forall \sigma \in H. \sigma x = x)$
 by (simp add: fixed-field-def)

lemma *fixed-field-antitone*:
 $H \subseteq H' \Longrightarrow \text{fixed-field } K H' \subseteq \text{fixed-field } K H$
 by (auto simp: fixed-field-def)

lemma *fixed-field-empty*: *fixed-field* K {} = K
 by (auto simp: fixed-field-def)

lemma *field-auto-fixed-set-subfield*:
 assumes Ksub: Subfield K **and** s: $\sigma \in \text{field-auto } K F$
 shows Subfield {x ∈ K. $\sigma x = x$ } (**is** Subfield ?KK)
proof –
 interpret KS: Subfield K **by** (rule Ksub)
 have add: $\bigwedge x y. x \in K \Longrightarrow y \in K \Longrightarrow \sigma (x + y) = \sigma x + \sigma y$
 and mul: $\bigwedge x y. x \in K \Longrightarrow y \in K \Longrightarrow \sigma (x * y) = \sigma x * \sigma y$

```

    and one:  $\sigma \ 1 = 1$ 
    using s by (auto simp: field-auto-def)
  have zero:  $\sigma \ 0 = 0$ 
    by (rule field-auto-zero[OF Ksub s])
  show ?thesis
proof
  show  $0 \in ?KK$  using KS.zero-closed zero by simp
  show  $1 \in ?KK$  using KS.one-closed one by simp
next
  fix x assume x:  $x \in ?KK$ 
  show  $-x \in ?KK$  inverse x  $\in ?KK$ 
    using x Ksub field-auto-uminus field-auto-inverse-on s by fastforce+
  fix y assume y:  $y \in ?KK$ 
  show  $x + y \in ?KK$   $x * y \in ?KK$ 
    using x y by (auto simp: add mul)
qed
qed

lemma field-auto-fixed-field-subfield:
  assumes Ksub: Subfield K and H:  $H \subseteq \text{field-auto } K \ F$ 
  shows Subfield (fixed-field K H)
proof -
  interpret KS: Subfield K by (rule Ksub)
  show ?thesis
proof
  show  $0 \in \text{fixed-field } K \ H$ 
    using H KS.zero-closed Ksub by (meson field-auto-zero fixed-field-memI subsetD)
  show  $1 \in \text{fixed-field } K \ H$ 
    using H KS.one-closed by (metis field-auto-mem-iff fixed-field-memI subsetD)
next
  fix x y assume x:  $x \in \text{fixed-field } K \ H$  and y:  $y \in \text{fixed-field } K \ H$ 
  have xK:  $x \in K$  using fixed-field-memD[OF x] by blast
  have yK:  $y \in K$  using fixed-field-memD[OF y] by blast
  show  $x + y \in \text{fixed-field } K \ H$ 
proof (rule fixed-field-memI)
  show  $x + y \in K$  by (rule KS.add-closed[OF xK yK])
  fix  $\sigma$  assume s:  $\sigma \in H$ 
  then have add:  $\sigma \ (x + y) = \sigma \ x + \sigma \ y$ 
    using H field-auto-add xK yK by blast
  show  $\sigma \ (x + y) = x + y$ 
    using add s x y by (metis fixed-field-memD)
qed
show  $x * y \in \text{fixed-field } K \ H$ 
proof (rule fixed-field-memI)
  show  $x * y \in K$  by (rule KS.mult-closed[OF xK yK])
  fix  $\sigma$  assume s:  $\sigma \in H$ 
  then have mul:  $\sigma \ (x * y) = \sigma \ x * \sigma \ y$ 
    using H field-auto-mult xK yK by blast
  then show  $\sigma \ (x * y) = x * y$ 
    using s x y by (metis fixed-field-memD)

```

```

    qed
  next
    fix x assume x: x ∈ fixed-field K H
    have xK: x ∈ K using fixed-field-memD[OF x] by blast
    show -x ∈ fixed-field K H
    proof (rule fixed-field-memI)
      show -x ∈ K by (rule KS.uminus-closed[OF xK])
      fix σ assume s: σ ∈ H
      show σ (-x) = -x
        using H Ksub field-auto-uminus fixed-field-memD s x by fastforce
    qed
    show inverse x ∈ fixed-field K H
    proof (rule fixed-field-memI)
      show inverse x ∈ K by (rule KS.inverse-closed[OF xK])
      fix σ assume s: σ ∈ H
      show σ (inverse x) = inverse x
        using H Ksub field-auto-inverse-on fixed-field-memD s x by fastforce
    qed
  qed
qed
qed

lemma field-auto-base-subset-fixed-field:
  assumes H: H ⊆ field-auto K F and FK: F ⊆ K
  shows F ⊆ fixed-field K H
  using assms by (auto simp: fixed-field-def field-auto-def)

lemma field-auto-faithful:
  assumes gen: K = generate-field (F ∪ X)
  and s: σ ∈ field-auto K F
  and fixX: ⋀x. x ∈ X ⟹ σ x = x
  and x ∈ K
  shows σ x = x
proof -
  have Ksub: Subfield K unfolding gen by (rule subfield-generate-field)
  have fixed-subfield: Subfield {x ∈ K. σ x = x}
    using Ksub s by (rule field-auto-fixed-set-subfield)
  have fixF: ⋀x. x ∈ F ⟹ σ x = x
    using s by (auto simp: field-auto-def)
  have FXsub: F ∪ X ⊆ {x ∈ K. σ x = x}
    using gen subset-generate-field fixF fixX by blast
  have generate-field (F ∪ X) ⊆ {x ∈ K. σ x = x}
    using fixed-subfield FXsub by (rule generate-field-least)
  with gen ⟨x ∈ K⟩ show ?thesis by auto
qed

end

```

32 Subfields of the Complex Numbers and their Automorphisms

```

theory Complex-Field-Extension
imports Ring-Theory Subfield Galois-Automorphism
         HOL-Computational-Algebra.Fundamental-Theorem-Algebra
begin

```

Working inside the algebraically closed field *complex*, we develop just enough field theory for the Galois-theoretic study of polynomials over $\mathbb{Q}$:

- a *subfield of $\mathbb{C}$* as a subset closed under the field operations, which is then a model of the locale-based *Field* (with the type-class operations);
- the *field automorphisms* of such a subfield fixing a base subfield, which form a subgroup of the symmetric group on the carrier — the Galois group;
- complex conjugation as such an automorphism.

This bridges the locale algebra to Isabelle’s type-class fields, and provides the group on which *Group-Action* and the symmetric-group results operate when the Galois group is realised on the (finite) set of roots.

Ring-Theory suppresses the HOL arithmetic notation; restore it locally.

```

notation plus (infixl  $\langle + \rangle$  65)
notation minus (infixl  $\langle - \rangle$  65)
unbundle uminus-syntax

```

32.1 Subfields of $\mathbb{C}$

A subfield of *complex*: a subset containing 0 and 1 and closed under addition, negation, multiplication and inversion of nonzero elements.

```

locale complex-subfield =
  fixes K :: complex set
  assumes zero-in:  $0 \in K$ 
  and one-in:  $1 \in K$ 
  and add-closed:  $\llbracket x \in K; y \in K \rrbracket \implies x + y \in K$ 
  and minus-closed:  $x \in K \implies -x \in K$ 
  and mult-closed:  $\llbracket x \in K; y \in K \rrbracket \implies x * y \in K$ 
  and inverse-closed:  $\llbracket x \in K; x \neq 0 \rrbracket \implies \text{inverse } x \in K$ 
begin

```

```

lemma diff-closed:  $\llbracket x \in K; y \in K \rrbracket \implies x - y \in K$ 
  using add-closed minus-closed by (metis diff-conv-add-uminus)

```

Built bottom-up, exactly as for the *GF2* and *Ring-Theory* witnesses.

```

lemma add-group: Group K  $(+)$  0

```

```

proof (intro GroupI zero-in)
  fix x assume x ∈ K
  then show ∃ y ∈ K. x + y = 0 ∧ y + x = 0
    using minus-closed by force
qed (auto simp: add-closed add.assoc)

lemma add-abelian: Abelian-Group K (+) 0
  by (rule Abelian-Group.intro [OF add-group]) (unfold-locales, simp-all add: zero-in
add-closed ac-simps)

lemma mult-cmonoid: commutative-monoid K (*) 1
  by unfold-locales (auto simp: one-in mult-closed ac-simps)

lemma is-ring: Ring K (+) (*) 0 1
proof (intro Ring.intro add-abelian)
  show Monoid K (*) 1 by (rule commutative-monoid.axioms(1) [OF mult-cmonoid])
  show Ring-axioms K (+) (*) by unfold-locales (auto simp: algebra-simps)
qed

lemma is-commutative-ring: commutative-ring K (+) (*) 0 1
  by (rule commutative-ring.intro [OF is-ring mult-cmonoid])

Every subfield of complex is a locale-based Field.

lemma is-field: Field K (+) (*) 0 1
proof –
  interpret R: commutative-ring K (+) (*) 0 1 by (rule is-commutative-ring)
  show ?thesis
  proof
    fix a :: complex assume a: a ∈ K a ≠ 0
    then have inv: inverse a ∈ K by (rule inverse-closed)
    have a * inverse a = 1 inverse a * a = 1 using a by simp-all
    then show R.multiplicative.invertible a
      using inv by blast
  qed auto
qed

end

The rationals and the reals are subfields of  $\mathbb{C}$ .

lemma complex-subfield-Reals: complex-subfield  $\mathbb{R}$ 
  by unfold-locales (auto simp: Reals-add Reals-minus Reals-mult Reals-inverse)

lemma complex-subfield-Rats: complex-subfield  $\mathbb{Q}$ 
  by unfold-locales (auto simp: Rats-add Rats-minus-iff Rats-mult Rats-inverse)

corollary field-Rats: Field  $\mathbb{Q}$  (+) (*) (0::complex) 1
  using complex-subfield.is-field [OF complex-subfield-Rats] by this

corollary field-Reals: Field  $\mathbb{R}$  (+) (*) (0::complex) 1
  using complex-subfield.is-field [OF complex-subfield-Reals] by this

```

32.2 Field automorphisms and the Galois group

The identity is an automorphism.

```

lemma identity-field-auto:
  assumes Ksub: complex-subfield K and FK:  $F \subseteq K$ 
  shows identity  $K \in \text{field-auto } K \ F$ 
proof –
  interpret KS: complex-subfield K by (rule Ksub)
  show ?thesis
    unfolding field-auto-mem-iff
  proof (intro conjI ballI)
    show identity  $K \in K \rightarrow_E K$ 
      by (auto simp: identity-apply PiE-iff)
    show bij-betw (identity K) K K
      by (simp add: bij-betw-def inj-on-def identity-apply)
    show  $\bigwedge x. x \in F \implies \text{identity } K \ x = x$ 
      using FK by (auto simp: identity-apply)
  qed (auto simp: identity-apply KS.one-in KS.add-closed KS.mult-closed)
qed

```

Automorphisms are closed under composition.

```

lemma compose-field-auto:
  assumes Ksub: complex-subfield K and FK:  $F \subseteq K$ 
  and s:  $\sigma \in \text{field-auto } K \ F$  and t:  $\tau \in \text{field-auto } K \ F$ 
  shows compose K  $\sigma \ \tau \in \text{field-auto } K \ F$ 
proof –
  interpret KS: complex-subfield K by (rule Ksub)
  from s have sbij: bij-betw  $\sigma \ K \ K$ 
    and sext:  $\sigma \in K \rightarrow_E K$ 
    and sadd:  $\bigwedge x \ y. x \in K \implies y \in K \implies \sigma \ (x + y) = \sigma \ x + \sigma \ y$ 
    and smul:  $\bigwedge x \ y. x \in K \implies y \in K \implies \sigma \ (x * y) = \sigma \ x * \sigma \ y$ 
    and sone:  $\sigma \ 1 = 1$  and sfix:  $\bigwedge x. x \in F \implies \sigma \ x = x$ 
    by (auto simp: field-auto-def)
  from t have tbij: bij-betw  $\tau \ K \ K$ 
    and text2:  $\tau \in K \rightarrow_E K$ 
    and tadd:  $\bigwedge x \ y. x \in K \implies y \in K \implies \tau \ (x + y) = \tau \ x + \tau \ y$ 
    and tmul:  $\bigwedge x \ y. x \in K \implies y \in K \implies \tau \ (x * y) = \tau \ x * \tau \ y$ 
    and tone:  $\tau \ 1 = 1$  and tfix:  $\bigwedge x. x \in F \implies \tau \ x = x$ 
    by (auto simp: field-auto-def)
  have tinK:  $\bigwedge x. x \in K \implies \tau \ x \in K$  using tbij bij-betwE by blast
  let ?c = compose K  $\sigma \ \tau$ 
  show ?thesis
    unfolding field-auto-mem-iff
  proof (intro conjI ballI)
    show ?c  $\in K \rightarrow_E K$  using sext text2 by (simp add: compose-def PiE-iff)
    show bij-betw ?c K K using sbij tbij by (simp add: bij-betw-compose)
    show ?c  $1 = 1$  using KS.one-in by (simp add: compose-eq tone sone)
    show  $\bigwedge x \ y. x \in K \implies y \in K \implies ?c \ (x + y) = ?c \ x + ?c \ y$ 
      by (simp add: compose-eq KS.add-closed tinK tadd sadd)

```



```

show  $\bigwedge x y. x \in K \implies y \in K \implies ?c (x * y) = ?c x * ?c y$ 
  by (simp add: compose-eq KS.mult-closed tinK tmul smul)
show  $\bigwedge x. x \in F \implies ?c x = x$ 
  by (metis FK compose-eq in-mono sfix tfix)
qed
qed

```

The inverse of an automorphism (extended by the identity outside K) is an automorphism: it is a ring homomorphism because σ is injective on K .

lemma *field-auto-inv*:

```

assumes Ksub: complex-subfield K and s:  $\sigma \in \text{field-auto } K F$  and FK:  $F \subseteq K$ 
shows restrict (inv-into K  $\sigma$ ) K  $\in \text{field-auto } K F$ 
proof -
interpret KS: complex-subfield K by (rule Ksub)
from s have bij: bij-betw  $\sigma$  K K
  and addh:  $\bigwedge x y. x \in K \implies y \in K \implies \sigma (x + y) = \sigma x + \sigma y$ 
  and mulh:  $\bigwedge x y. x \in K \implies y \in K \implies \sigma (x * y) = \sigma x * \sigma y$ 
  and one:  $\sigma 1 = 1$  and fixed:  $\bigwedge x. x \in F \implies \sigma x = x$ 
  by (auto simp: field-auto-def)
define t where t = restrict (inv-into K  $\sigma$ ) K
have tin:  $\bigwedge y. y \in K \implies t y \in K$  using bij by (simp add: t-def inv-into-into
bij-betw-def)
have st:  $\bigwedge y. y \in K \implies \sigma (t y) = y$  using bij by (simp add: t-def f-inv-into-f
bij-betw-def)
have ts:  $\bigwedge x. x \in K \implies t (\sigma x) = x$ 
  using bij by (simp add: t-def bij-betwE inv-into-f-f bij-betw-def)
have inj: inj-on  $\sigma$  K using bij by (simp add: bij-betw-def)
have tbij: bij-betw t K K
  using bij by (simp add: t-def bij-betw-restrict-eq bij-betw-inv-into)
have textl:  $t \in K \rightarrow_E K$  using tin by (auto simp: t-def PiE-iff)
have tadd:  $t (x + y) = t x + t y$  if  $x: x \in K$  and  $y: y \in K$  for  $x y$ 
  by (rule inj-onD[OF inj]) (simp-all add: st KS.add-closed addh tin x y)
have tmul:  $t (x * y) = t x * t y$  if  $x: x \in K$  and  $y: y \in K$  for  $x y$ 
  by (rule inj-onD[OF inj]) (simp-all add: st KS.mult-closed mulh tin x y)
have tone:  $t 1 = 1$ 
  using KS.one-in one ts by force
have tfix:  $t x = x$  if  $x \in F$  for  $x$ 
  by (metis FK fixed subsetD that ts)
have t  $\in \text{field-auto } K F$ 
  using textl tbij tadd tmul tone tfix by (simp add: field-auto-mem-iff)
then show ?thesis unfolding t-def .
qed

```

Helper lemmas relating the symmetric-group inverse to the functional inverse.

lemma *cnj-invol-bij*: $(\bigwedge x. x \in K \implies \text{cnj } x \in K) \implies \text{bij-betw } (\text{restrict cnj } K) K K$
 by (intro bij-betw-byWitness[where $f' = \text{restrict cnj } K$]) (auto simp: restrict-apply')

context

fixes $K F :: \text{complex set}$

assumes *Ksub*: complex-subfield *K* **and** *FK*: $F \subseteq K$
begin

interpretation *T*: transformations *K* .

lemma *restrict-inv-in-Sym*:
assumes *a*: $a \in T.Sym$
shows *restrict* (*inv-into* *K* *a*) $K \in T.Sym$
proof –
have *bij*: *bij-betw* *a* *K* *K* **using** *a* **by** (rule *T.Units-bij-betwI*)
have *restrict* (*inv-into* *K* *a*) $K \in K \rightarrow_E K$
using *bij* **by** (auto simp: *PiE-iff* *inv-into-into* *bij-betw-def*)
then show ?thesis
using *bij* *bij-betw-inv-into* *bij-betw-restrict-eq* **by** blast
qed

lemma *Sym-inverse-eq-inv-into*:
assumes *a* $\in T.Sym$
shows *T.symmetric.inverse* *a* = *restrict* (*inv-into* *K* *a*) *K*
proof (intro *T.symmetric.inverse-equality* *restrict-inv-in-Sym* *assms*)
have *img*: $a \cdot K = K$
by (simp add: *assms* *bij-betw-imp-surj-on*)
show *compose* *K* *a* (*restrict* (*inv-into* *K* *a*) *K*) = *identity* *K*
using *img* **by** (rule *compose-id-inv-into*)
show *compose* *K* (*restrict* (*inv-into* *K* *a*) *K*) *a* = *identity* *K*
by (simp add: *assms* *compose-inv-into-id*)
qed

The Galois group is a subgroup of the symmetric group on *K*.

theorem *Galois-group-subgroup*:
Subgroup (*field-auto* *K* *F*) (*transformations.Sym* *K*) (*compose* *K*) (*identity* *K*)
proof (intro *T.symmetric.subgroupI* *field-auto-subset-Sym*)
show *identity* *K* $\in$ *field-auto* *K* *F* **using** *Ksub* *FK* **by** (rule *identity-field-auto*)
next
fix *a* *b* **assume** *a* $\in$ *field-auto* *K* *F* *b* $\in$ *field-auto* *K* *F*
then show *compose* *K* *a* *b* $\in$ *field-auto* *K* *F* **using** *Ksub* *FK* **by** (blast intro: *compose-field-auto*)
next
fix *a* **assume** *a*: $a \in$ *field-auto* *K* *F*
then have *aSym*: $a \in T.Sym$ **using** *field-auto-subset-Sym* **by** blast
then show *T.symmetric.inverse* *a* $\in$ *field-auto* *K* *F*
by (simp add: *FK* *Ksub* *Sym-inverse-eq-inv-into* *a* *field-auto-inv*)
next
fix *g* **assume** *g* $\in$ *field-auto* *K* *F*
then have *g* $\in T.Sym$ **using** *field-auto-subset-Sym* **by** blast
then show *T.symmetric.invertible* *g* **by** (simp add: *T.mem-UnitsD*)
qed

Hence the Galois group is a group.

corollary *Galois-group-Group*:

```

Group (field-auto K F) (compose K) (identity K)
using Galois-group-subgroup by (simp add: Subgroup.axioms(2))

```

end

32.3 Complex conjugation as an automorphism

If K is closed under conjugation and the base F is real, then complex conjugation (restricted to K) is an element of the Galois group. This is the automorphism that, for a polynomial with real coefficients, transposes a pair of complex conjugate roots.

lemma *cnj-field-auto*:

assumes *Ksub*: complex-subfield K **and** *cnjK*: $\text{cnj} \restriction K = K$ **and** *FR*: $F \subseteq \mathbb{R}$ **and** *FK*: $F \subseteq K$

shows $\text{restrict } \text{cnj } K \in \text{field-auto } K F$

proof –

interpret *KS*: complex-subfield K **by** (rule *Ksub*)

have *capp*: $\bigwedge x. x \in K \implies \text{restrict } \text{cnj } K x = \text{cnj } x$ **by** (simp add: restrict-apply')

show ?thesis

unfolding field-auto-mem-iff

proof (intro conjI ballI)

show $\text{restrict } \text{cnj } K \in K \rightarrow_E K$ **using** *cnjK* **by** (auto simp: PiE-iff)

show *bij-betw* ($\text{restrict } \text{cnj } K$) $K K$ **by** (metis *cnjK cnj-invol-bij image-eqI*)

next

fix x **assume** $x \in F$

then have $x \in K$ $x \in \mathbb{R}$ **using** *FK FR* **by** auto

then show $\text{restrict } \text{cnj } K x = x$ **by** (simp add: capp Reals-cnj-iff)

qed (auto simp add: KS.one-in capp KS.add-closed KS.mult-closed)

qed

32.4 The subfield generated by a set; splitting subfields

The whole of *complex* and the intersection of a non-empty family of subfields are subfields; hence the *subfield generated by* a set of complex numbers is well defined as the intersection of all subfields containing it.

lemma *complex-subfield-UNIV*: complex-subfield ($UNIV :: \text{complex set}$)

by unfold-locale auto

lemma *complex-subfield-Inter*:

assumes $C \neq \{\}$ **and** [simp]: $\bigwedge K. K \in C \implies \text{complex-subfield } K$

shows complex-subfield $(\bigcap C)$

proof

fix x **assume** $x \in \bigcap C$ $x \neq 0$

then show $\text{inverse } x \in \bigcap C$ **using** *assms* **by** (auto simp: complex-subfield.inverse-closed)

qed (auto simp: complex-subfield.zero-in complex-subfield.one-in complex-subfield.add-closed

complex-subfield.minus-closed complex-subfield.mult-closed)

The subfield generated by X : the smallest subfield containing X .

definition *gen-subfield* :: *complex set* $\Rightarrow$ *complex set*
where *gen-subfield* $X = \bigcap \{K. \text{complex-subfield } K \wedge X \subseteq K\}$

lemma *gen-subfield-is-subfield*: *complex-subfield* (*gen-subfield* X)
using *complex-subfield-UNIV* **by** (*auto simp: complex-subfield-def gen-subfield-def*)

lemma *gen-subfield-subset*: $X \subseteq \text{gen-subfield } X$
unfolding *gen-subfield-def* **by** *auto*

lemma *gen-subfield-minimal*:
assumes *complex-subfield* K $X \subseteq K$
shows *gen-subfield* $X \subseteq K$
unfolding *gen-subfield-def* **using** *assms* **by** *auto*

The complex-specific and generic generated subfields are the same. Keeping this bridge next to *gen-subfield* makes it available to both the Galois and rational developments.

lemma *gen-subfield-eq-generate-field*: *gen-subfield* $X = \text{generate-field } X$
proof
have *sf*: *complex-subfield* (*generate-field* X)
proof –
have *Subfield* (*generate-field* X) **by** (*rule subfield-generate-field*)
then interpret *Subfield generate-field* X .
show *complex-subfield* (*generate-field* X)
by *unfold-locales (auto intro: add-closed uminus-closed mult-closed inverse-closed)*
qed
show *gen-subfield* $X \subseteq \text{generate-field } X$
using *sf subset-generate-field* **by** (*rule gen-subfield-minimal*)
have *sf*: *Subfield* (*gen-subfield* X)
proof –
have *complex-subfield* (*gen-subfield* X) **by** (*rule gen-subfield-is-subfield*)
then interpret *complex-subfield gen-subfield* X .
show *Subfield* (*gen-subfield* X)
using *complex-subfield-axioms complex-subfield-def Subfield.intro* **by** *fastforce*
qed
show *generate-field* $X \subseteq \text{gen-subfield } X$
using *sf gen-subfield-subset* **by** (*rule generate-field-least*)
qed

For the splitting subfield $\mathbb{Q}(X)$ we generate from $\mathbb{Q} \cup X$: it contains the rationals and the elements of X .

lemma *Rats-subset-gen-subfield*: $\mathbb{Q} \subseteq \text{gen-subfield } (\mathbb{Q} \cup X)$
using *gen-subfield-subset[of $\mathbb{Q} \cup X$]* **by** *blast*

lemma *roots-subset-gen-subfield*: $X \subseteq \text{gen-subfield } (\mathbb{Q} \cup X)$
using *gen-subfield-subset[of $\mathbb{Q} \cup X$]* **by** *blast*

Conjugation is a field automorphism of *complex*, so the image of a subfield under

conjugation is again a subfield.

lemma *cnj-image-eq*: $cnj \text{ ' } K = \{y. cnj \ y \in K\}$
by (*auto simp: image-iff*) (*metis complex-cnj-cnj*)

lemma *complex-subfield-cnj-image*:
assumes *complex-subfield* K
shows *complex-subfield* ($cnj \text{ ' } K$)
proof –
interpret KS : *complex-subfield* K **by** (*rule assms*)
show *?thesis*
unfolding *cnj-image-eq*
by *unfold-locale*
(auto simp: KS.add-closed KS.minus-closed KS.mult-closed KS.inverse-closed
KS.zero-in KS.one-in)
qed

A subfield generated by a conjugation-closed set is itself conjugation-closed.

lemma *gen-subfield-cnj-closed*:
assumes $cnj \text{ ' } Y = Y$
shows $cnj \text{ ' } gen\text{-subfield } Y = gen\text{-subfield } Y$
proof –
have $csub$: *complex-subfield* ($cnj \text{ ' } gen\text{-subfield } Y$)
by (*simp add: complex-subfield-cnj-image gen-subfield-is-subfield*)
have Yin : $Y \subseteq cnj \text{ ' } gen\text{-subfield } Y$
by (*metis assms gen-subfield-subset image-mono*)
have $gen\text{-subfield } Y \subseteq cnj \text{ ' } gen\text{-subfield } Y$
using $csub \ Yin$ **by** (*rule gen-subfield-minimal*)
then show *?thesis*
by *auto*
qed

Conjugation fixes the rationals.

lemma *cnj-of-rat*: $cnj \text{ (of-rat } q) = \text{of-rat } q$
proof (*induct q*)
case (*Fract a b*)
then show *?case* **by** (*simp add: of-rat-rat*)
qed

lemma *cnj-fixes-Rats*: $x \in \mathbb{Q} \implies cnj \ x = x$
by (*auto simp: Rats-def cnj-of-rat*)

lemma *cnj-Rats-eq*: $cnj \text{ ' } \mathbb{Q} = \mathbb{Q}$
using *cnj-fixes-Rats* **by** (*force simp: image-iff*)

Hence the splitting subfield $\mathbb{Q}(X)$ of a conjugation-closed set X (e.g. the set of roots of a real-coefficient polynomial) is conjugation-closed — so by *cnj-field-auto* complex conjugation restricts to an element of its Galois group over $\mathbb{Q}$.

lemma *splitting-subfield-cnj-closed*:

assumes $cnj \text{ ' } X = X$
shows $cnj \text{ ' } gen\text{-}subfield (\mathbb{Q} \cup X) = gen\text{-}subfield (\mathbb{Q} \cup X)$
by (*simp add: assms cnj-Rats-eq gen-subfield-cnj-closed image-Un*)

32.5 Faithfulness of the action on a generating set

The set of points fixed by an automorphism is itself a subfield (it contains 0 and 1, and is closed under the field operations, since the automorphism respects them).

lemma *fixed-set-subfield*:

assumes $Ksub$: *complex-subfield* K **and** s : $\sigma \in field\text{-}auto\ K\ F$
shows *complex-subfield* $\{x \in K. \sigma\ x = x\}$

proof —

interpret KS : *complex-subfield* K **by** (*rule* $Ksub$)

from s **have** bij : *bij-betw* $\sigma\ K\ K$

and add : $\bigwedge x\ y. x \in K \implies y \in K \implies \sigma\ (x + y) = \sigma\ x + \sigma\ y$

and mul : $\bigwedge x\ y. x \in K \implies y \in K \implies \sigma\ (x * y) = \sigma\ x * \sigma\ y$

and one : $\sigma\ 1 = 1$

by (*auto simp: field-auto-def*)

have $zero$: $\sigma\ 0 = 0$

using $KS.zero\text{-}in\ add$ **by** *fastforce*

have $\sigma\ (-\ x) = -\ x$ **if** $x \in K$ **and** $\sigma\ x = x$ **for** x

using *that* **by** (*metis* $KS.minus\text{-}closed\ add\ add\text{-}eq\ 0\text{-}iff\ zero$)

moreover **have** $\sigma\ (inverse\ x) = inverse\ x$

if $x \neq 0$ **and** $x \in K$ **and** $\sigma\ x = x$ **for** x

using *that* **by** (*metis* $KS.inverse\text{-}closed\ left\text{-}inverse\ mul\ mult\text{-}cancel\text{-}right\ one$)

ultimately show *?thesis*

by *unfold-locales (auto simp: KS.zero-in zero KS.one-in one add KS.add-closed KS.minus-closed mul KS.mult-closed KS.inverse-closed)*

qed

An automorphism of $K = F(X)$ fixing the base F and every generator in X is the identity on K : the fixed subfield contains $F \cup X$, hence all of the generated field K . Thus the Galois group acts *faithfully* on any generating set — in particular on the roots of a polynomial whose splitting field is K .

theorem *field-auto-faithful*:

assumes $Ksub$: *complex-subfield* K

and gen : $K = gen\text{-}subfield\ (F \cup X)$

and s : $\sigma \in field\text{-}auto\ K\ F$

and $fixX$: $\bigwedge x. x \in X \implies \sigma\ x = x$

shows $\forall x \in K. \sigma\ x = x$

proof —

have $fix\text{-}sub$: *complex-subfield* $\{x \in K. \sigma\ x = x\}$ **using** $Ksub\ s$ **by** (*rule* *fixed-set-subfield*)

have $fixF$: $\bigwedge x. x \in F \implies \sigma\ x = x$ **using** s **by** (*auto simp: field-auto-def*)

have $FXsub$: $F \cup X \subseteq \{x \in K. \sigma\ x = x\}$

using $gen\ gen\text{-}subfield\text{-}subset\ fixF\ fixX$ **by** *blast*

have $gen\text{-}subfield\ (F \cup X) \subseteq \{x \in K. \sigma\ x = x\}$

using $fix\text{-}sub\ FXsub$ **by** (*rule* *gen-subfield-minimal*)

with *gen* **show** *?thesis* **by** *auto*
qed

Specialised to the splitting field: two automorphisms agreeing on the generators agree everywhere, so the permutation action on the generators is faithful.

corollary *galois-faithful-on-generators*:
assumes *s*: $\sigma \in \text{field-auto } (\text{gen-subfield } (F \cup X)) \ F$
and *fixX*: $\bigwedge x. x \in X \implies \sigma \ x = x$
shows $\forall x \in \text{gen-subfield } (F \cup X). \sigma \ x = x$
using *field-auto-faithful*[*OF gen-subfield-is-subfield refl s fixX*] **by** *simp*
end

33 Iterating the Iso-Extension over a Finite Set of Algebraic Elements

theory *Iso-Extension-Tower*
imports *Iso-Extension Complex-Field-Extension*
begin

Starting from a homomorphism f defined on a subfield K , we extend it across a whole finite family of elements that are algebraic over K , reaching a homomorphism on the generated subfield *generate-field* $(K \cup A)$. Each extension step uses the single-element *field-hom-on.iso-extension*; the only external requirement is that the target field ' b ' contains a root for every non-constant polynomial, i.e. is “closed enough”. Over *complex* this is the fundamental theorem of algebra.

The minimal polynomial is monic, so its image under f keeps the same degree.

lemma (in *field-hom-on*) *map-poly-minpoly-degree*:
assumes *m*: *is-minpoly* $K \ a \ m$
shows $\text{degree } (\text{map-poly } f \ m) = \text{degree } m$
proof –
have *lead-coeff* $m = 1$ **using** *m* **by** (*simp add: is-minpoly-def*)
then have $f \ (\text{lead-coeff } m) = 1$ **by** *simp*
then show *?thesis* **by** (*intro map-poly-degree-eq*) *simp*
qed

A subfield is generated by itself.

lemma (in *Subfield*) *generate-field-self*: *generate-field* $K = K$
proof
show *generate-field* $K \subseteq K$ **using** *Subfield-axioms subset-refl* **by** (*rule generate-field-least*)
show $K \subseteq \text{generate-field } K$ **by** (*rule subset-generate-field*)
qed

The single-element extension, packaged with the closed-target hypothesis discharging the existence of a root of the mapped minimal polynomial.

```

lemma field-hom-on-extend-one:
  assumes f: field-hom-on K f
    and alg: algebraic-over K a
    and closed:  $\bigwedge q :: 'b :: \text{field poly. } 0 < \text{degree } q \implies \exists b. \text{poly } q \ b = 0$ 
  shows  $\exists g. \text{field-hom-on } (\text{generate-field } (K \cup \{a\})) \ g \wedge (\forall x \in K. \ g \ x = (f :: 'a :: \text{field} \Rightarrow 'b) \ x)$ 
proof –
  have sf: Subfield K using f by (rule field-hom-on.axioms(1))
  obtain m where m: is-minpoly K a m using Subfield.minpoly-exists[OF sf alg] by
    blast
  have degree (map-poly f m) > 0
  using m by (simp add: field-hom-on.map-poly-minpoly-degree[OF f] Subfield.minpoly-degree-pos[OF sf])
  then obtain b where b: poly (map-poly f m) b = 0 using closed by blast
  obtain g where g: field-hom-on (eval-img K a) g g a = b  $\forall x \in K. \ g \ x = f \ x$ 
    using field-hom-on.iso-extension[OF f alg m b] by blast
  have eval-img K a = generate-field (K  $\cup$  {a}) using Subfield.eval-img-eq-generate-field[OF sf alg] .
  then show ?thesis using g by auto
qed

```

Iterating over a finite set: *f* extends to the subfield generated by *K* together with finitely many elements algebraic over *K*.

```

theorem field-hom-on-extend-finite:
  assumes closed:  $\bigwedge q :: 'b :: \text{field poly. } 0 < \text{degree } q \implies \exists b. \text{poly } q \ b = 0$ 
    and finite A
    and field-hom-on K (f :: 'a :: field  $\Rightarrow$  'b')
    and  $\bigwedge a. \ a \in A \implies \text{algebraic-over } K \ a$ 
  shows  $\exists g. \text{field-hom-on } (\text{generate-field } (K \cup A)) \ g \wedge (\forall x \in K. \ g \ x = f \ x)$ 
using assms(2–4)
proof (induction A rule: finite-induct)
  case empty
  have sf: Subfield K using empty.prems(1) by (rule field-hom-on.axioms(1))
  have generate-field (K  $\cup$  {}) = K by (simp add: Subfield.generate-field-self[OF sf])
  then show ?case using empty.prems(1) by auto
next
  case (insert a A)
  define L where L = generate-field (K  $\cup$  A)
  obtain g1 where g1: field-hom-on L g1  $\forall x \in K. \ g1 \ x = f \ x$ 
    using insert.IH[OF insert.prems(1)] insert.prems(2) by (auto simp: L-def)
  have KsubL: K  $\subseteq$  L
  unfolding L-def using subset-generate-field by blast
  have algL: algebraic-over L a
    using insert.prems(2)[of a] KsubL by (auto intro: algebraic-over-mono)
  obtain g2 where g2: field-hom-on (generate-field (L  $\cup$  {a})) g2  $\forall x \in L. \ g2 \ x = g1 \ x$ 
    using field-hom-on-extend-one[OF g1(1) algL closed] by blast
  have generate-field (L  $\cup$  {a}) = generate-field ({a}  $\cup$  generate-field (K  $\cup$  A))
    unfolding L-def by (simp add: Un-commute)

```



```

also have ... = generate-field ({a} ∪ (K ∪ A))
  by (rule generate-field-Un-collapse2)
also have ... = generate-field (K ∪ insert a A)
  by (simp add: Un-commute Un-left-commute)
finally have generate-field (L ∪ {a}) = generate-field (K ∪ insert a A) .
moreover have ∀ x ∈ K. g2 x = f x
  using g2(2) g1(2) KsubL by auto
ultimately show ?case using g2(1) by auto
qed

```

The generic field-extension development works over an arbitrary ambient $'a$ via the *Subfield* and *field-hom-on* locales. The complex-specific Galois theory of *Complex-Field-Extension* instead fixes *complex* and packages automorphisms as the set *field-auto*. Now we connect the two.

The two subfield notions coincide on *complex*.

```

lemma complex-subfield-iff-subfield: complex-subfield K ⟷ Subfield K
proof
  assume complex-subfield K
  then interpret complex-subfield K .
  show Subfield K
    using complex-subfield-axioms complex-subfield-def Subfield.intro by fastforce
next
  assume Subfield K
  then interpret Subfield K .
  show complex-subfield K
    by unfold-locales (auto intro: add-closed uminus-closed mult-closed inverse-closed)
qed

```

A field homomorphism on K (in the generic sense) that maps K *onto* itself and fixes a base subfield F pointwise restricts to an element of the complex Galois group *field-auto* $K F$.

```

lemma field-hom-on-imp-field-auto:
  assumes hom: field-hom-on K σ
    and onto: σ ' K = K
    and fix-F: ∧ x. x ∈ F ⟹ σ x = x and FK: F ⊆ K
  shows restrict σ K ∈ field-auto K F
proof –
  interpret S: field-hom-on K σ by (rule hom)
  have bij: bij-betw (restrict σ K) K K
    by (simp add: S.inj-on bij-betw-def onto)
  show ?thesis
    unfolding field-auto-mem-iff
  proof (intro conjI ballI)
    show restrict σ K ∈ K →E K
      using onto by auto
    show bij-betw (restrict σ K) K K by (rule bij)
    show ∧ x. x ∈ F ⟹ restrict σ K x = x using fix-F FK by auto

```

```

    qed (use S.hom-1 S.one-closed S.hom-add S.add-closed S.hom-mult S.mult-closed
in auto)
qed
end

```

34 Transitivity of the Galois Action on the Roots

```

theory Galois-Transitivity
  imports Iso-Extension-Tower
begin

```

The capstone of the iso-extension development: for a polynomial Phi over a base subfield F whose root set R is finite and consists of elements algebraic over F , any two roots related by the minimal polynomial of one of them are connected by an F -fixing automorphism of the splitting field $K = generate_field (F \cup R)$. This is *transitivity* of the Galois action on the roots, the key input to identifying the Galois group of an irreducible quintic with S_5 .

The construction has three parts: a base isomorphism $F(a) \cong F(b)$ sending a to b (the single-step *field-hom-on.iso-extension*); its extension across the remaining roots (*field-hom-on-extend-finite*); and the observation that the resulting embedding permutes the finite root set and therefore maps the splitting field onto itself (*field-hom-on.image-generate-field-eq*), i.e. is an automorphism.

The identity is a homomorphism of any subfield — the starting point of the tower.

```

lemma field-hom-on-id:
  assumes Subfield K shows field-hom-on K ( $\lambda x. x$ )
proof -
  interpret Subfield K by fact
  show ?thesis by unfold-locales auto
qed

```

An endomorphism fixing F pointwise leaves every polynomial over F unchanged under *map-poly*.

```

lemma map-poly-fix:
  fixes f :: 'a :: field  $\Rightarrow$  'a
  assumes hom: field-hom-on K f and sfF: Subfield F
    and p:  $p \in poly-over F$  and fixF:  $\bigwedge x. x \in F \implies f x = x$ 
  shows map-poly f p = p
proof (rule poly-eqI)
  fix n
  have cF: coeff p n  $\in F$  using p by (rule Subfield.poly-over-coeff[OF sfF])
  have coeff (map-poly f p) n = f (coeff p n)
    by (rule field-hom-on.coeff-map-poly-f[OF hom])
  also have ... = coeff p n using cF by (rule fixF)
  finally show coeff (map-poly f p) n = coeff p n .

```

qed

Hence such an endomorphism sends a root of Phi to a root of Phi .

lemma *hom-preserves-roots:*

fixes $f :: 'a :: \text{field} \Rightarrow 'a$
assumes $\text{hom}: \text{field-hom-on } K \text{ } f$ **and** $\text{sfF}: \text{Subfield } F$ **and** $\text{FK}: F \subseteq K$
and $\text{fixF}: \bigwedge x. x \in F \implies f \ x = x$
and $\text{Phi}: \text{Phi} \in \text{poly-over } F$ **and** $r: r \in K$ **and** $\text{root}: \text{poly } \text{Phi } r = 0$
shows $\text{poly } \text{Phi } (f \ r) = 0$

proof –

have $\text{PhiK}: \text{Phi} \in \text{poly-over } K$
using $\text{Phi poly-over-mono}[OF \ \text{FK}]$ **by** *blast*
have $\text{poly } \text{Phi } (f \ r) = \text{poly } (\text{map-poly } f \ \text{Phi}) (f \ r)$
using $\text{map-poly-fix}[OF \ \text{hom } \text{sfF } \text{Phi } \text{fixF}]$ **by** *simp*
also have $\dots = f (\text{poly } \text{Phi } r)$
by $(\text{rule field-hom-on.poly-map-poly-hom}[OF \ \text{hom } \text{PhiK } r])$
also have $\dots = 0$ **using** $\text{root field-hom-on.hom-0}[OF \ \text{hom}]$ **by** *simp*
finally show *?thesis* .

qed

theorem *galois-transitive:*

fixes $F :: 'a :: \text{field set}$
assumes $\text{sfF}: \text{Subfield } F$
and $\text{finR}: \text{finite } R$
and $\text{algR}: \bigwedge r. r \in R \implies \text{algebraic-over } F \ r$
and $\text{Phi}: \text{Phi} \in \text{poly-over } F$
and $\text{rootsR}: R = \{r. \text{poly } \text{Phi } r = 0\}$
and $\text{closed}: \bigwedge q :: 'a \text{ poly. } 0 < \text{degree } q \implies \exists b. \text{poly } q \ b = 0$
and $a: a \in R$ **and** $b: b \in R$
and $m: \text{is-minpoly } F \ a \ m$ **and** $mb: \text{poly } m \ b = 0$
defines $K \equiv \text{generate-field } (F \cup R)$
shows $\exists \sigma. \text{field-hom-on } K \ \sigma \wedge \sigma \ a = b \wedge (\forall x \in F. \sigma \ x = x) \wedge \sigma \ ^\text{' } K = K$

proof –

— Step 1: the base isomorphism on $F(a)$ sending a to b .
interpret $F: \text{field-hom-on } F \ \lambda x. x$ **by** $(\text{rule field-hom-on-id}[OF \ \text{sfF}])$
have $\text{alga}: \text{algebraic-over } F \ a$ **using** a **by** (rule algR)
have $\text{mapm}: \text{map-poly } (\lambda x. x) \ m = m$ **by** (intro poly-eqI) $(\text{simp add: coeff-map-poly})$
have $\text{rootb}: \text{poly } (\text{map-poly } (\lambda x. x) \ m) \ b = 0$ **unfolding** mapm **by** (rule mb)
obtain $g0$ **where** $g0: \text{field-hom-on } (\text{eval-img } F \ a) \ g0 \ g0 \ a = b \ \forall x \in F. g0 \ x = x$
using $F.\text{iso-extension}[OF \ \text{alga } m \ \text{rootb}]$ **by** *blast*

— Step 2: extend the base isomorphism over the remaining roots.

define E **where** $E = \text{eval-img } F \ a$
have $\text{sfE}: \text{Subfield } E$ **unfolding** $E\text{-def}$ **by** $(\text{rule Subfield.subfield-eval-img}[OF \ \text{sfF } \text{alga}])$
have $F\text{subE}: F \subseteq E$ **unfolding** $E\text{-def}$ **using** $\text{Subfield.eval-img-base}[OF \ \text{sfF}]$ **by** *auto*
have $\text{algRE}: \bigwedge r. r \in R - \{a\} \implies \text{algebraic-over } E \ r$
using $\text{algR } F\text{subE}$ **by** $(\text{auto intro: algebraic-over-mono})$
have $\text{finRa}: \text{finite } (R - \{a\})$ **using** finR **by** *simp*
obtain g **where** $g: \text{field-hom-on } (\text{generate-field } (E \cup (R - \{a\}))) \ g \ \forall x \in E. g \ x =$

$g0\ x$
using *field-hom-on-extend-finite*[*OF closed finRa g0(1)[folded E-def] algRE*] **by**
blast

— Step 3: the extended domain is exactly the splitting field K .

have *Edef*: $E = \text{generate-field } (F \cup \{a\})$
unfolding *E-def* **by** (*rule Subfield.eval-img-eq-generate-field*[*OF sfF alga*])
have *domK*: $\text{generate-field } (E \cup (R - \{a\})) = K$
proof –
have $\text{generate-field } (E \cup (R - \{a\})) = \text{generate-field } (\text{generate-field } (F \cup \{a\}) \cup (R - \{a\}))$
by (*simp add: Edef*)
also have $\dots = \text{generate-field } ((F \cup \{a\}) \cup (R - \{a\}))$
by (*rule generate-field-Un-collapse1*)
also have $(F \cup \{a\}) \cup (R - \{a\}) = F \cup R$ **using** a **by** *auto*
finally show *?thesis* **by** (*simp add: K-def*)
qed
have *homg*: *field-hom-on* $K\ g$ **using** $g(1)\ \text{domK}$ **by** *simp*
interpret G : *field-hom-on* $K\ g$ **by** (*rule homg*)

— g fixes F and sends a to b .

have *gfixF*: $\bigwedge x. x \in F \implies g\ x = x$
using $g(2)\ g0(3)\ FsubE$ **by** *auto*
have *gab*: $g\ a = b$
proof –
have $a \in E$ **unfolding** *E-def* **using** *Subfield.eval-img-self*[*OF sfF*] **by** *auto*
then have $g\ a = g0\ a$ **using** $g(2)$ **by** *simp*
then show *?thesis* **using** $g0(2)$ **by** *simp*
qed

— Step 4: g permutes the root set, hence maps K onto itself.

have *RsubK*: $R \subseteq K$
unfolding *K-def* **using** *subset-generate-field* **by** *blast*
have *FsubK*: $F \subseteq K$
unfolding *K-def* **using** *subset-generate-field* **by** *blast*
have *gR-sub*: $g\ ' R \subseteq R$
proof –
fix y **assume** $y \in g\ ' R$
then obtain r **where** $r: r \in R\ y = g\ r$ **by** *blast*
have *rK*: $r \in K$ **using** $r(1)\ RsubK$ **by** *blast*
have *poly Phi r = 0* **using** $r(1)$ **unfolding** *rootsR* **by** *blast*
then have *poly Phi (g r) = 0*
using *hom-preserves-roots*[*OF homg sfF FsubK gfixF Phi rK*] **by** *blast*
then have $g\ r \in R$ **unfolding** *rootsR* **by** *blast*
then show $y \in R$ **using** $r(2)$ **by** *simp*
qed
have *gR-eq*: $g\ ' R = R$
proof –
have *inj-on* $g\ R$ **using** *G.inj-on RsubK* **by** (*rule inj-on-subset*)
then show *?thesis* **using** *gR-sub finR* **by** (*simp add: endo-inj-surj*)

```

qed
have gF-eq: g ' F = F
proof
  show g ' F  $\subseteq$  F using gfixF by auto
  show F  $\subseteq$  g ' F using gfixF by force
qed
have FRsubK: F  $\cup$  R  $\subseteq$  K using FsubK RsubK by blast
have g ' K = g ' generate-field (F  $\cup$  R) by (simp add: K-def)
also have ... = generate-field (g ' (F  $\cup$  R))
  by (rule G.image-generate-field-eq[OF FRsubK])
also have g ' (F  $\cup$  R) = F  $\cup$  R using gF-eq gR-eq by (simp add: image-Un)
also have generate-field (F  $\cup$  R) = K by (simp add: K-def)
finally have gK: g ' K = K .

```

```

  show ?thesis using homg gab gfixF gK by blast
qed

```

For an *irreducible* polynomial the minimal-polynomial hypothesis is automatic, so the Galois action is transitive on its roots: any two roots are related by an F -fixing automorphism of the splitting field.

corollary *galois-transitive-irreducible*:

```

fixes F :: 'a :: field set
assumes sfF: Subfield F
  and finR: finite R
  and algR:  $\bigwedge r. r \in R \implies \text{algebraic-over } F \ r$ 
  and Phi: Phi  $\in$  poly-over F
  and irr: irreducible-over F Phi
  and rootsR: R = {r. poly Phi r = 0}
  and closed:  $\bigwedge q :: 'a \text{ poly. } 0 < \text{degree } q \implies \exists b. \text{poly } q \ b = 0$ 
  and a: a  $\in$  R and b: b  $\in$  R
defines K  $\equiv$  generate-field (F  $\cup$  R)
shows  $\exists \sigma. \text{field-hom-on } K \ \sigma \wedge \sigma \ a = b \wedge (\forall x \in F. \sigma \ x = x) \wedge \sigma ' K = K$ 
proof -
  have alga: algebraic-over F a using a by (rule algR)
  obtain m where m: is-minpoly F a m using Subfield.minpoly-exists[OF sfF alga]
by blast
  have pa: poly Phi a = 0 using a unfolding rootsR by blast
  have pb: poly Phi b = 0 using b unfolding rootsR by blast
  have mb: poly m b = 0
    by (rule Subfield.irreducible-over-root-is-minpoly-root[OF sfF irr Phi m pa pb])
  show ?thesis
  unfolding K-def
  by (rule galois-transitive[OF sfF finR algR Phi rootsR closed a b m mb])
qed

```

34.1 Rational Subfields and Irreducibility over the Rationals

A complex polynomial all of whose coefficients are rational is the *of-rat*-image of a rational polynomial.

lemma *poly-over-Rats-imp-map-of-rat*:
assumes $p \in \text{poly-over } (\mathbb{Q} :: \text{complex set})$
shows $\exists r :: \text{rat poly. } p = \text{map-poly of-rat } r$
using *assms* **by** (*meson Rats-0 poly-over-iff-aux ratpolyE*)

lemma *map-poly-of-rat-mult*:
 $\text{map-poly of-rat } (a * b) = (\text{map-poly of-rat } a :: \text{complex poly}) * \text{map-poly of-rat } b$
by (*intro poly-eqI*) (*simp add: coeff-map-poly of-rat-mult of-rat-sum coeff-mult*)

lemma *map-poly-of-rat-inj*: $\text{inj } (\text{map-poly of-rat} :: \text{rat poly} \Rightarrow \text{complex poly})$
by (*intro injI poly-eqI*) (*metis coeff-map-poly of-rat-0 of-rat-eq-iff*)

lemma *degree-map-poly-of-rat*:
 $\text{degree } (\text{map-poly of-rat } r :: \text{complex poly}) = \text{degree } r$
by (*simp add: degree-map-poly*)

Irreducibility of a rational polynomial transfers to irreducibility *over* $\mathbb{Q}$ of its complex image: a factorisation over $\mathbb{Q}$ pulls back along the injective ring homomorphism *map-poly of-rat* to a factorisation of the rational polynomial.

lemma *irreducible-imp-irreducible-over-Rats*:
fixes $r :: \text{rat poly}$
assumes *irr*: *irreducible* r **and** *deg*: $\text{degree } r > 0$
shows *irreducible-over* $(\mathbb{Q} :: \text{complex set})$ $(\text{map-poly of-rat } r)$
unfolding *irreducible-over-def*
proof (*intro conjI allI impI*)
show $\text{map-poly of-rat } r \neq (0 :: \text{complex poly})$
by (*metis deg degree-0 degree-map-poly-of-rat not-less-zero*)
show $0 < \text{degree } (\text{map-poly of-rat } r :: \text{complex poly})$
using *deg* **by** (*simp add: degree-map-poly-of-rat*)
next
fix $b \ c :: \text{complex poly}$
assume *bover*: $b \in \text{poly-over } \mathbb{Q}$ **and** *cover*: $c \in \text{poly-over } \mathbb{Q}$
and *eq*: $\text{map-poly of-rat } r = b * c$
obtain b' **where** $b' = \text{map-poly of-rat } b'$ **using** *bover poly-over-Rats-imp-map-of-rat*
by *blast*
obtain c' **where** $c' = \text{map-poly of-rat } c'$ **using** *cover poly-over-Rats-imp-map-of-rat*
by *blast*
have $(\text{map-poly of-rat } r :: \text{complex poly}) = \text{map-poly of-rat } (b' * c')$
by (*simp add: b' c' eq map-poly-of-rat-mult*)
then have $rbc: r = b' * c'$ **using** *map-poly-of-rat-inj* **by** (*simp add: inj-eq*)
show $\text{degree } b = 0 \vee \text{degree } c = 0$
using *irreducibleD[OF irr rbc]* $b' \ c' \text{ degree-map-poly-of-rat poly-dvd-1}$ **by** *auto*
qed

end

35 The Galois Group Acts on the Roots

theory *Galois-Action*

imports *Galois-Transitivity Symmetric-Generation Sym-Not-Solvable*
begin

A field automorphism (an element of *field-auto*) is, in particular, a field homomorphism on its subfield domain.

lemma *field-auto-imp-field-hom-on*:

assumes *K*: Subfield *K* **and** *s*: $\sigma \in \text{field-auto } K \ F$
shows *field-hom-on* *K* σ
using *K s* **by** (rule *field-auto-as-hom*)

An automorphism fixing *F* maps a root in *K* of an *F*-polynomial to a root: it permutes the root set.

lemma *field-auto-maps-root*:

assumes *K*: Subfield *K* **and** *sfF*: Subfield *F* **and** *FK*: $F \subseteq K$
and *s*: $\sigma \in \text{field-auto } K \ F$ **and** *Phi*: *Phi* $\in \text{poly-over } F$
and *r*: $r \in K$ **and** *root*: $\text{poly } \text{Phi } r = 0$
shows $\text{poly } \text{Phi } (\sigma \ r) = 0$
using *hom-preserves-roots*[*OF field-auto-imp-field-hom-on sfF FK - Phi r root*]
using *K field-auto-mem-iff s* **by** *blast*

The Galois group acts on the root set by restriction. We package this as a *Group-Action* of *field-auto* *K F* on the finite root set *X*.

context

fixes *K F* :: 'a :: field set **and** *Phi* :: 'a poly **and** *X* :: 'a set
assumes *K*: Subfield *K* **and** *sfF*: Subfield *F* **and** *FK*: $F \subseteq K$
and *Phi*: *Phi* $\in \text{poly-over } F$
and *Xdef*: $X = \{r. \text{poly } \text{Phi } r = 0\}$ **and** *XK*: $X \subseteq K$ **and** *finX*: finite *X*

begin

lemma *field-auto-image-root-subset*:

assumes *s*: $\sigma \in \text{field-auto } K \ F$
shows $\sigma \ ` X \subseteq X$
using *FK K Phi XK Xdef field-auto-maps-root s sfF* **by** *blast*

lemma *field-auto-bij-on-root*:

assumes *s*: $\sigma \in \text{field-auto } K \ F$
shows *bij-betw* $\sigma \ X \ X$
unfolding *bij-betw-def*

proof

show *inj-on* $\sigma \ X$ **using** *XK*
by (meson *bij-betw-imp-inj-on field-auto-mem-iff inj-on-subset s*)
show $\sigma \ ` X = X$
by (meson $\langle \text{inj-on } \sigma \ X \rangle \text{endo-inj-surj field-auto-image-root-subset finX } s$)

qed

lemma *ga-sym*:

assumes $s: \sigma \in \text{field-auto } K \ F$

shows $\text{restrict } \sigma \ X \in \text{transformations.Sym } X$

by (*simp add: bij-betw-imp-funcset field-auto-bij-on-root s transformations.Units-bij-betwD*)

lemma *ga-compose*:

assumes $s: \sigma \in \text{field-auto } K \ F$ **and** $t: \tau \in \text{field-auto } K \ F$

shows $\text{restrict } (\text{compose } K \ \sigma \ \tau) \ X = \text{compose } X \ (\text{restrict } \sigma \ X) \ (\text{restrict } \tau \ X)$

using *assms XK field-auto-image-root-subset* **by** (*fastforce simp: compose-def*)

lemma *ga-unit*: $\text{restrict } (\text{identity } K) \ X = \text{identity } X$

using *XK* **by** *force*

lemma *galois-group-action*:

Group-Action (field-auto K F) (compose K) (identity K) ($\lambda \sigma. \text{restrict } \sigma \ X$) X

proof –

interpret *G*: *Group field-auto K F compose K identity K*

by (*rule field-auto-group[OF K FK]*)

interpret *T*: *transformations X* .

show *?thesis*

proof

fix σ **assume** $s: \sigma \in \text{field-auto } K \ F$

show $\text{restrict } \sigma \ X \in \text{transformations.Sym } X$

by (*rule ga-sym[OF s]*)

next

fix $\sigma \ \tau$

assume $s: \sigma \in \text{field-auto } K \ F$ **and** $t: \tau \in \text{field-auto } K \ F$

show $\text{restrict } (\text{compose } K \ \sigma \ \tau) \ X =$

$\text{compose } X \ (\text{restrict } \sigma \ X) \ (\text{restrict } \tau \ X)$

by (*rule ga-compose[OF s t]*)

next

show $\text{restrict } (\text{identity } K) \ X = \text{identity } X$

by (*rule ga-unit*)

qed

qed

When the ambient field is generated by the base and the root set, the root action is faithful: an automorphism fixing every root fixes the generators and is therefore the identity.

lemma *galois-action-faithful*:

assumes *Kgen*: $K = \text{generate-field } (F \cup X)$

shows $\forall \sigma \in \text{field-auto } K \ F.$

$(\forall x \in X. \text{restrict } \sigma \ X \ x = x) \longrightarrow \sigma = \text{identity } K$

proof (*intro ballI impI*)

fix σ

assume $s: \sigma \in \text{field-auto } K \ F$

and *triv*: $\forall x \in X. \text{restrict } \sigma \ X \ x = x$


```

have fixX:  $\bigwedge x. x \in X \implies \sigma x = x$ 
  using triv by (simp add: restrict-apply')
have allfix:  $\forall x \in K. \sigma x = x$ 
  — The theory prefix is needed: Complex-Field-Extension declares a complex-specific
theorem of the same name, which shadows the generic one here.
  using Galois-Automorphism.field-auto-faithful[OF Kgen s fixX] by force
have sext:  $\sigma \in K \rightarrow_E K$ 
  using s by (simp only: field-auto-mem-iff)
with allfix show  $\sigma = \text{identity } K$ 
  by (metis PiE-restrict restrict-ext)
qed

```

```

lemma finite-galois-group:
  assumes Kgen:  $K = \text{generate-field } (F \cup X)$ 
  shows finite (field-auto  $K F$ )
proof —
  interpret GA: Group-Action field-auto  $K F$  compose  $K$  identity  $K$   $\lambda \sigma. \text{restrict } \sigma X$ 
  X
  by (rule galois-group-action)
  — The action on the roots is faithful.
  have faith: GA.faithful
    unfolding GA.faithful-def
    by (rule galois-action-faithful[OF Kgen])
  — Faithfulness gives an injection into the finite symmetric group on  $X$ .
  have inj0: inj-on (restrict ( $\lambda \sigma. \text{restrict } \sigma X$ ) (field-auto  $K F$ )) (field-auto  $K F$ )
    using faith by (rule GA.faithful-inj)
  have inj: inj-on ( $\lambda \sigma. \text{restrict } \sigma X$ ) (field-auto  $K F$ )
    using inj0 inj-on-restrict-eq by blast
  have finSym: finite (transformations.Sym  $X$ )
    by (metis GA.mem-UnitsD finX finite-PiE finite-subset subsetI)
  show ?thesis using inj GA.hom-closed finSym inj-on-finite by blast
qed

```

end

Compatibility wrapper for the historical complex-only spelling. The generic theorem uses *generate-field*; *gen-subfield* is the same construction on *complex* and remains available to existing clients.

```

lemma finite-field-auto:
  fixes  $K F :: \text{complex set}$  and  $\Phi :: \text{complex poly}$  and  $X :: \text{complex set}$ 
  assumes K: Subfield  $K$  and sfF: Subfield  $F$  and FK:  $F \subseteq K$ 
    and  $\Phi: \Phi \in \text{poly-over } F$ 
    and Xdef:  $X = \{r. \text{poly } \Phi r = 0\}$  and XK:  $X \subseteq K$  and finX: finite  $X$ 
    and Kgen:  $K = \text{gen-subfield } (F \cup X)$ 
  shows finite (field-auto  $K F$ )
proof —
  have Kgen':  $K = \text{generate-field } (F \cup X)$ 
    using Kgen by (simp add: gen-subfield-eq-generate-field)
  show ?thesis

```

by (rule finite-galois-group[OF K sfF FK Phi Xdef XK finX Kgen'])
qed

Transitivity of the Galois action on the roots of an irreducible polynomial: any root is carried to any other by some F -automorphism (via *galois-transitive-irreducible* and the bridge *field-hom-on-imp-field-auto*).

lemma *galois-action-transitive*:

```

fixes F :: 'a :: field set and Phi :: 'a poly and X :: 'a set
assumes sfF: Subfield F
and finX: finite X
and Phi: Phi ∈ poly-over F
and irr: irreducible-over F Phi
and Xdef: X = {r. poly Phi r = 0}
and closed: ∧ q :: 'a poly. 0 < degree q ⇒ ∃ b. poly q b = 0
and a: a ∈ X and b: b ∈ X
defines K ≡ generate-field (F ∪ X)
shows ∃ σ ∈ field-auto K F. restrict σ X a = b
proof –
  have phinz: Phi ≠ 0 using irr by (simp add: irreducible-over-def)
  have algX: algebraic-over F r if r ∈ X for r
    using that Phi phinz Xdef unfolding algebraic-over-def by auto
  have XK: X ⊆ K unfolding K-def using subset-generate-field by blast
  have aK: a ∈ K using a XK by blast
  obtain σ where hom: field-hom-on K σ and sab: σ a = b
    and fixF: ∀ x ∈ F. σ x = x and onto: σ ` K = K
    using galois-transitive-irreducible[OF sfF finX algX Phi irr Xdef closed a b]
    unfolding K-def by blast
  have FK: F ⊆ K unfolding K-def using subset-generate-field by blast
  have aut: restrict σ K ∈ field-auto K F
    using field-hom-on-imp-field-auto[OF hom onto - FK] fixF by blast
  have restrict (restrict σ K) X a = b
    using aK a sab XK by simp
  then show ?thesis using aut by blast
qed

```

end

36 Embedding a Faithful Finite Action into a Concrete Symmetric Group

theory *Action-To-Sn*

imports *Symmetric-Group Group-Operations*

begin

36.1 Relabelling Symmetric Groups along a Bijection

Relabelling along a bijection e from $\{0..<n\}$ to a finite set X transports the symmetric group on X to S_n .

```

definition pull :: nat ⇒ (nat ⇒ 'a) ⇒ ('a ⇒ 'a) ⇒ (nat ⇒ nat)
  where pull n e a = (λi. if i < n then inv-into {0.. $n$ } e (a (e i)) else i)

context
  fixes n :: nat and e :: nat ⇒ 'a and X :: 'a set
  assumes ebij: bij-betw e {0.. $n$ } X
begin

lemma e-in-X: i < n ⇒ e i ∈ X
  using ebij by (auto simp: bij-betw-def)

lemma inv-e-lt: x ∈ X ⇒ inv-into {0.. $n$ } e x < n
  using ebij by (metis atLeastLessThan-iff bij-betw-def inv-into-into)

lemma inv-e-e: i < n ⇒ inv-into {0.. $n$ } e (e i) = i
  using ebij by (simp add: bij-betw-inv-into-left)

lemma e-inv-e: x ∈ X ⇒ e (inv-into {0.. $n$ } e x) = x
  using ebij by (simp add: bij-betw-inv-into-right)

lemma pull-permutes:
  assumes a: a ∈ transformations.Sym X
  shows pull n e a permutes {0.. $n$ }
proof (rule bij-imp-permutes)
  have abij: bij-betw a X X using a by (simp add: transformations.Units-bijective)
  show bij-betw (pull n e a) {0.. $n$ } {0.. $n$ }
  proof (rule bij-betwI)
    show pull n e a ∈ {0.. $n$ } → {0.. $n$ }
      using abij by (auto simp: pull-def e-in-X inv-e-lt bij-betwE)
    show (λi. inv-into {0.. $n$ } e (inv-into X a (e i))) ∈ {0.. $n$ } → {0.. $n$ }
      using abij by (simp add: bij-betw-def e-in-X inv-e-lt inv-into-into)
  next
    fix i assume i ∈ {0.. $n$ }
    then have iln: i < n by simp
    have eiX: e i ∈ X using iln by (rule e-in-X)
    have aeIX: a (e i) ∈ X using abij eiX by (auto simp: bij-betwE)
    have (λi. inv-into {0.. $n$ } e (inv-into X a (e i))) (pull n e a i)
      = inv-into {0.. $n$ } e (inv-into X a (e (inv-into {0.. $n$ } e (a (e i)))))
      using iln by (simp add: pull-def)
    also have inv-into X a (e (inv-into {0.. $n$ } e (a (e i)))) = e i
      using abij eiX by (simp add: aeIX bij-betw-inv-into-left e-inv-e)
    finally show (λi. inv-into {0.. $n$ } e (inv-into X a (e i))) (pull n e a i) = i
      by (simp add: iln inv-e-e)
    have inv-in: inv-into X a (e i) ∈ X
      using abij eiX by (metis bij-betw-inv-into bij-betwE)
    have pull n e a ((λi. inv-into {0.. $n$ } e (inv-into X a (e i))) i)
      = inv-into {0.. $n$ } e (a (e (inv-into {0.. $n$ } e (inv-into X a (e i)))))
      using iln inv-in by (simp add: pull-def inv-e-lt)
    also have a (e (inv-into {0.. $n$ } e (inv-into X a (e i)))) = e i
      by (metis abij bij-betw-inv-into-right e-inv-e eiX inv-in)

```

```

    finally show pull n e a ((λi. inv-into {0.. $n$ } e (inv-into X a (e i))) i) = i
      using iln inv-e-e by presburger
  qed
qed (simp add: pull-def)

```

```

lemma pull-compose:
  assumes a: a ∈ transformations.Sym X and b: b ∈ transformations.Sym X
  shows pull n e (compose X a b) = pull n e a ∘ pull n e b
proof (rule ext)
  fix i
  show pull n e (compose X a b) i = (pull n e a ∘ pull n e b) i
  proof (cases i < n)
    case True
    then have beiX: b (e i) ∈ X
      by (meson b bij-betwE e-in-X transformations.Units-bij-betwI)
    then show ?thesis
      using True inv-e-lt beiX by (simp add: True compose-eq e-in-X pull-def e-inv-e)
  next
    case False
    then show ?thesis by (simp add: pull-def compose-def)
  qed
qed

```

```

lemma pull-unit: pull n e (identity X) = id
  using e-in-X by (auto simp: pull-def inv-e-e)

```

```

lemma pull-in-Sn: a ∈ transformations.Sym X  $\implies$  pull n e a ∈ Sn n
  by (simp add: Sn-def pull-permutes)

```

end

```

context Group-Action
begin

```

Transport a faithful finite action into the concrete symmetric group S_n (card S), via an enumeration e of S .

```

lemma action-pull-hom:
  assumes ebij: bij-betw e {0.. $n$ } S
  shows group-homomorphism (restrict (λg. pull n e (φ g)) G) G (·) 1 (Sn n) (∘) id
proof -
  interpret T: Group Sn n (∘) id by (rule Group-Sn)
  have phiSym: φ g ∈ transformations.Sym S if g ∈ G for g
    using that by (rule hom-closed)
  show ?thesis
  proof
    show restrict (λg. pull n e (φ g)) G ∈ G  $\rightarrow_E$  Sn n
      using phiSym ebij by (simp add: pull-in-Sn)
  next

```

```

fix g h assume g: g ∈ G and h: h ∈ G
have pull n e (φ (g · h)) = pull n e (φ g) ∘ pull n e (φ h)
  by (simp add: ebij g h hom-compose pull-compose)
then show restrict (λg. pull n e (φ g)) G (g · h)
      = restrict (λg. pull n e (φ g)) G g ∘ restrict (λg. pull n e (φ g)) G h
    using g h by (simp add: grp.composition-closed)
next
show restrict (λg. pull n e (φ g)) G 1 = id
  using ebij grp.unit-closed by (simp add: hom-unit pull-unit)
qed
qed

```

Under faithfulness the transported homomorphism has trivial kernel, hence is injective.

lemma *action-pull-inj*:

```

assumes ebij: bij-betw e {0..<n} S and faith: faithful
shows inj-on (restrict (λg. pull n e (φ g)) G) G
proof (rule inj-onI)
  fix g h assume g: g ∈ G and h: h ∈ G
  and eq: restrict (λg. pull n e (φ g)) G g = restrict (λg. pull n e (φ g)) G h
  have pgh: pull n e (φ g) = pull n e (φ h) using eq g h by simp
  have agree: φ g s = φ h s if s: s ∈ S for s
  proof -
    define i where i ≡ inv-into {0..<n} e s
    have iln: i < n using ebij s unfolding i-def by (metis atLeastLessThan-iff
bij-betw-def inv-into-into)
    have ei: e i = s using ebij s unfolding i-def by (simp add: bij-betw-inv-into-right)
    have inv-into {0..<n} e (φ g s) = inv-into {0..<n} e (φ h s)
      by (metis ei iln pgh pull-def)
    then show φ g s = φ h s
      by (metis action-closed bij-betw-inv-into-right ebij g h that)
  qed
  have ginvg: grp.inverse g ∈ G using g by (simp add: grp.invertible-inverse-closed
grp.invertible)
  have ghG: grp.inverse g · h ∈ G using ginvg h by (rule grp.composition-closed)
  have triv: φ (grp.inverse g · h) s = s if s: s ∈ S for s
    by (metis action-mult agree g ginvg group-inv-rel h that)
  have invgh: grp.inverse g · h = 1
    using faith ghG triv unfolding faithful-def by blast
  show g = h
    using g grp.inverse-unique h invgh by auto
qed

```

Hence a faithful finite action embeds G as a subgroup of S_n n : the transported map is a group isomorphism onto its image.

theorem *action-embeds-Sn*:

```

assumes ebij: bij-betw e {0..<n} S and faith: faithful
shows group-isomorphism (restrict (λg. pull n e (φ g)) G) G (·) 1
      ((λg. pull n e (φ g)) ' G) (∘) id

```

```

proof —
  interpret H: group-homomorphism restrict (λg. pull n e (φ g)) G G (·) 1 Sn n (○)
  id
    by (rule action-pull-hom[OF ebij])
  have img: restrict (λg. pull n e (φ g)) G ‘ G = (λg. pull n e (φ g)) ‘ G by auto
  interpret Img: Subgroup (λg. pull n e (φ g)) ‘ G Sn n (○) id
    using H.image.Subgroup-axioms img by simp
  show ?thesis
  proof
    have inj: inj-on (restrict (λg. pull n e (φ g)) G) G
      by (rule action-pull-inj[OF ebij faith])
    with img show bij-betw (restrict (λg. pull n e (φ g)) G) G ((λg. pull n e (φ g))
      ‘ G)
      by (simp add: bij-betw-def)
    qed (use H.commutes-with-unit H.commutes-with-composition in auto)
  qed
end

end

```

37 The Symmetric Group on the Roots and Non-Solvability

```

theory Abel-Ruffini
  imports Galois-Action Action-To-Sn
begin

```

A subgroup of S_n containing some p -cycle (conjugate to n -cycle p) and some transposition is the whole group.

```

lemma subgroup-with-cycle-transpose-eq-Sn:
  fixes p :: nat and q :: nat ⇒ nat and i j :: nat
  assumes p: prime p and H: Subgroup H (Sn p) (○) id
    and q: q ∈ Sn p
    and c-in: q ○ ncycle p ○ inv q ∈ H
    and t-in: Transposition.transpose i j ∈ H
    and i: i < p and jj: j < p and ij: i ≠ j
  shows H = Sn p

```

```

proof —
  interpret G: Group Sn p (○) id by (rule Group-Sn)
  interpret H: Subgroup H Sn p (○) id by (rule H)
  have bq: bij q using q by (simp add: Sn-def permutes-bij)
  — Write the transposition as the same-conjugator form.
  define a where a = inv q i
  define b where b = inv q j
  have qa: q a = i unfolding a-def using bq by (simp add: surj-f-inv-f bij-is-surj)
  have qb: q b = j unfolding b-def using bq by (simp add: surj-f-inv-f bij-is-surj)
  have qperm: q permutes {0..

```

```

have ap: a < p
  using a-def i permutes-nat-inv-less qperm by presburger
have bp: b < p
  using b-def jj permutes-nat-less permutes-nat-inv-less qperm by blast
have ab: a ≠ b using ij qa qb by auto
have t-eq: Transposition.transpose i j = q ∘ Transposition.transpose a b ∘ inv q
  using bq qa qb by (simp add: transpose-conjug)
— Both generators lie in H, so the generated subgroup (= Sn p) is contained in H.
have gens-sub: {q ∘ ncycle p ∘ inv q, q ∘ Transposition.transpose a b ∘ inv q} ⊆ H
  using c-in t-in t-eq by auto
have Sn p = G.generate {q ∘ ncycle p ∘ inv q, q ∘ Transposition.transpose a b ∘
inv q}
  using Sn-generated-conj-cycle-transpose[OF p q ap bp ab] by simp
also have ... ⊆ H
  using G.generate-mono G.generate-subgroup-eq H gens-sub by blast
finally show ?thesis by blast
qed

```

The abstract group-theoretic heart of Abel–Ruffini: a finite group acting faithfully and transitively on a set of prime cardinality $(5::'a) \leq p$, whose image in $\text{Sn } p$ contains a transposition, is not solvable, because that image is then all of $\text{Sn } p$, which is not solvable.

theorem (in *Group-Action*) *not-solvable-of-transitive-with-transposition:*

```

fixes e :: nat ⇒ 'b and p :: nat
assumes ebij: bij-betw e {0..

```

```

then have pdvd:  $p \text{ dvd card } G$  using orb cardS by (metis dvd-triv-left)
— Cauchy: an order- $p$  element of  $G$ .
obtain g0 where  $g0 \in G$  and  $g0ord$ : grp.element-order  $g0 = p$ 
using grp.Cauchy[OF p finG pdvd] by blast
— Its image has the same order  $p$ , computed via the isomorphism (using the raw
Monoid.element-order bridge so the orders match by name), and so is a  $p$ -cycle.
have psg0-in:  $\Psi \ g0 \in I$  unfolding I-def  $\Psi$ -def using g0 by auto
have I-sub-Sn:  $I \subseteq S_n \ p$ 
using hom-closed ebij by (simp add: I-def image-subset-iff pull-in-Sn)
have finI: finite  $I$  using I-sub-Sn finite-Sn finite-subset by blast
have psg0-Sn:  $\Psi \ g0 \in S_n \ p$  using psg0-in I-sub-Sn by blast
have img-ord: Monoid.element-order  $(\circ) \ id \ (\Psi \ g0) = p$ 
using Iso.element-order-preserved'[OF finG finI g0]  $g0ord$  by simp
obtain  $q$  where  $q: q \in S_n \ p$  and cyc:  $\Psi \ g0 = q \circ \text{ncycle } p \circ \text{inv } q$ 
using order-p-elt-conj-ncycle-order[OF p psg0-Sn img-ord] by blast
—  $I$  is a subgroup of  $S_n \ p$  containing this  $p$ -cycle and (by hypothesis) a transposition.
have I-sub: Subgroup  $I \ (S_n \ p) \ (\circ) \ id$ 
using Hom.image.Subgroup-axioms unfolding  $\Psi$ -def I-def by simp
obtain  $i \ j$  where  $ij$ :  $i < p \ j < p \ i \neq j$  and t-in: Transposition.transpose  $i \ j \in I$ 
using transp I-def by blast
have c-in:  $q \circ \text{ncycle } p \circ \text{inv } q \in I$  using psg0-in cyc by simp
have  $I = S_n \ p$ 
using subgroup-with-cycle-transpose-eq-Sn[OF p I-sub q c-in t-in ij] .
— So  $G$  is isomorphic onto  $S_n \ p$ , which would make  $S_n \ p$  solvable — contradiction.
then have Group.solvable  $(S_n \ p) \ (\circ) \ id$ 
using Iso.solvable-image-epi[OF solv] by simp
with Sn-not-solvable p5 show False by blast
qed
end

```

38 Finite p -groups

```

theory P-Group
imports Sylow-Theorems
begin

```

```

context Group
begin

```

The class equation makes the center of a nontrivial finite p -group visible arithmetically: every noncentral conjugacy orbit has cardinality divisible by p .

```

theorem p-group-prime-dvd-card-center:
fixes  $p \ n :: \text{nat}$ 
assumes prime: prime  $p$  and fin: finite  $G$  and npos:  $n > 0$ 
and cardG:  $\text{card } G = p \wedge n$ 
shows  $p \text{ dvd card center}$ 
proof —
interpret conj: Group-Action  $G \ (\cdot) \ 1 \ \varphi\text{-conj } G$ 

```



```

    by (rule conjugation-action)
  have orbit-p: p dvd card (conj.orbit y) if yG: y ∈ G and ynz: y ∉ center for y
  proof -
    have orbit-dvd: card (conj.orbit y) dvd p ^ n
      using conj.orbit-stabilizer-card[OF yG fin] cardG by (metis dvd-triv-left)
    have orbit-ne: conj.orbit y ≠ {y}
      using conj.fixed-point-iff-orbit-singleton[OF yG] ynz center-eq-fixed-points by
blast
    have orbit-card-ne: card (conj.orbit y) ≠ 1
    proof
      assume card (conj.orbit y) = 1
      then obtain z where orbit-z: conj.orbit y = {z}
        by (metis card-1-singletonE)
      have y ∈ conj.orbit y by (rule conj.orbit-self[OF yG])
      then have z = y using orbit-z by simp
      then show False using orbit-ne orbit-z by simp
    qed
  show ?thesis
  proof (rule ccontr)
    assume ¬ p dvd card (conj.orbit y)
    then have card (conj.orbit y) = 1
      by (rule prime-power-dvd-not-prime-dvd[OF prime orbit-dvd])
    then show False using orbit-card-ne by contradiction
  qed
qed
have p-nonfixed: p dvd card (G - conj.fixed-points)
  by (rule conj.dvd-card-nonfixed-points[OF fin])
  (simp add: center-eq-fixed-points orbit-p)
have p-diff: p dvd card (G - center)
  using p-nonfixed by (simp add: center-eq-fixed-points)
have pG: p dvd card G
  proof -
    have p dvd p ^ n using npos by (cases n) simp-all
    then show ?thesis using cardG by simp
  qed
have center-le: card center ≤ card G
  by (rule card-mono[OF fin center-subset])
have fin-center: finite center
  by (rule finite-subset[OF center-subset fin])
have diff-card: card (G - center) = card G - card center
  by (rule card-Diff-subset[OF fin-center center-subset])
have p dvd card G - card center using p-diff diff-card by simp
then show ?thesis by (rule dvd-diffD1[OF - pG center-le])
qed

corollary p-group-center-nontrivial:
  fixes p n :: nat
  assumes prime: prime p and fin: finite G and npos: n > 0
  and cardG: card G = p ^ n
  shows ∃ z ∈ center. z ≠ 1

```

```

proof -
  have pdvd:  $p \text{ dvd } \text{card } \text{center}$ 
  by (rule p-group-prime-dvd-card-center[OF prime fin npos cardG])
  have unit-center:  $1 \in \text{center}$ 
  by (rule Submonoid.sub-unit-closed[OF Subgroup.axioms(1)[OF center-subgroup]])
  have card-ne:  $\text{card } \text{center} \neq 1$ 
  proof
    assume card_center = 1
    then have  $p \text{ dvd } 1$  using pdvd by simp
    then show False using prime by simp
  qed
  have center_ne_1:  $\text{center} \neq \{1\}$ 
  proof
    assume center = {1}
    then show False using card-ne by simp
  qed
  then show ?thesis using unit-center by blast
qed

```

A group of order p^2 is abelian. A hypothetical noncentral element has a centralizer strictly larger than the center; Lagrange then forces that centralizer to have the full group cardinality.

```

theorem card-prime-square-imp-abelian:
  fixes p :: nat
  assumes prime: prime p and fin: finite G and cardG:  $\text{card } G = p^2$ 
  shows Abelian-Group G ( $\cdot$ ) 1
proof -
  have pcenter:  $p \text{ dvd } \text{card } \text{center}$ 
  by (rule p-group-prime-dvd-card-center[OF prime fin, of 2]) (simp-all add: cardG)
  have unit-center:  $1 \in \text{center}$ 
  by (rule Submonoid.sub-unit-closed[OF Subgroup.axioms(1)[OF center-subgroup]])
  have fin-center: finite center by (rule finite-subset[OF center-subset fin])
  have center-ne:  $\text{center} \neq \{1\}$  using unit-center by blast
  have center-pos:  $0 < \text{card } \text{center}$ 
  by (rule iffD2[OF card-gt-0-iff]) (rule conjI[OF center-ne fin-center])
  have p-le-center:  $p \leq \text{card } \text{center}$ 
  by (rule dvd-imp-le[OF pcenter center-pos])
  have all-center:  $x \in \text{center}$  if  $xG$ :  $x \in G$  for  $x$ 
  proof (rule ccontr)
    assume xnot:  $x \notin \text{center}$ 
    interpret C: subgroup-of-group centralizer x G ( $\cdot$ ) 1
    using centralizer-subgroup[OF xG]
    by (simp add: Group-axioms subgroup-of-group-def)
    have center-C:  $\text{center} \subseteq \text{centralizer } x$ 
    proof
      fix z assume z:  $z \in \text{center}$ 
      have zG:  $z \in G$  using center-subset z by blast
      have  $z \cdot x = x \cdot z$  using center-mem-iff[OF zG] z xG by blast
      then show  $z \in \text{centralizer } x$  using centralizer-mem-iff[OF zG xG] by simp
    qed
  qed

```

```

qed
have xC:  $x \in \text{centralizer } x$ 
  using centralizer-mem-iff[OF xG xG] by simp
have center-ne-C:  $\text{center} \neq \text{centralizer } x$ 
proof
  assume center = centralizer x
  then have  $x \in \text{center}$  using xC by simp
  then show False using xnot by contradiction
qed
have proper:  $\text{center} \subset \text{centralizer } x$ 
  by (rule psubsetI[OF center-C center-ne-C])
have finC: finite (centralizer x)
  by (rule finite-subset[OF centralizer-subset fin])
have center-lt-C:  $\text{card center} < \text{card (centralizer } x)$ 
  by (rule psubset-card-mono[OF finC proper])
have lagrange:  $\text{card } G = \text{card (centralizer } x) * C.\text{index}$ 
  by (rule C.lagrange[OF fin])
have C-dvd:  $\text{card (centralizer } x) \text{ dvd } p \wedge 2$ 
  using lagrange cardG by (metis dvd-triv-left)
obtain j where jle:  $j \leq 2$  and cardC:  $\text{card (centralizer } x) = p \wedge j$ 
  using divides-primelow-nat[OF prime, of card (centralizer x) 2] C-dvd by blast
have p-lt-C:  $p < \text{card (centralizer } x)$ 
  using p-le-center center-lt-C by linarith
have j2:  $j = 2$ 
proof -
  have  $j = 0 \vee j = 1 \vee j = 2$  using jle by presburger
  then show ?thesis
  proof (elim disjE)
    assume j = 0
    then show ?thesis using cardC p-lt-C prime by (simp add: prime-ge-2-nat)
  next
    assume j = 1
    then show ?thesis using cardC p-lt-C by simp
  next
    assume j = 2
    then show ?thesis .
  qed
qed
have cardC-G:  $\text{card (centralizer } x) = \text{card } G$  using cardC cardG j2 by simp
have C-eq:  $\text{centralizer } x = G$ 
  by (rule card-subset-eq[OF fin centralizer-subset cardC-G])
have commute:  $x \cdot g = g \cdot x$  if gG:  $g \in G$  for g
proof -
  have  $g \in \text{centralizer } x$  using C-eq gG by simp
  then have  $g \cdot x = x \cdot g$ 
    using centralizer-mem-iff[OF gG xG] by simp
  then show ?thesis by (rule sym)
qed
have  $x \in \text{center}$  using center-mem-iff[OF xG] commute by blast
then show False using xnot by contradiction

```

```

qed
show ?thesis
proof
  fix x y
  assume xG: x ∈ G and yG: y ∈ G
  have x ∈ center by (rule all-center[OF xG])
  then show x · y = y · x using center-mem-iff[OF xG] yG by blast
qed
qed

end

end

```

39 The subring generated by a set

```

theory Subring-Generated
  imports Ring-Theory
begin

```

Mirroring the subgroup *generate* and the ideal generated by a set, the subring generated by a subset S of a ring is the intersection of all subrings containing $S \cap R$. We first show that an intersection of a nonempty family of subrings is a subring; the whole ring is a subring of itself, so the family intersected is always nonempty.

```

context Ring
begin

```

39.1 Intersections of subrings

The intersection of a nonempty family of subrings is a subring.

```

lemma subring-Inter:
  assumes ne:  $\mathcal{A} \neq \{\}$  and sr:  $\bigwedge S. S \in \mathcal{A} \implies \text{Subring } S \ R \ (+) \ (\cdot) \ 0 \ 1$ 
  shows  $\text{Subring } (\bigcap \mathcal{A}) \ R \ (+) \ (\cdot) \ 0 \ 1$ 
proof -
  obtain S0 where S0:  $S0 \in \mathcal{A}$  using ne by blast
  interpret S0:  $\text{Subring } S0 \ R \ (+) \ (\cdot) \ 0 \ 1$  by (rule sr[OF S0])
  — The additive part is a subgroup.
  have sg:  $\text{Subgroup } (\bigcap \mathcal{A}) \ R \ (+) \ 0$ 
  proof (rule additive.subgroupI)
    show  $\bigcap \mathcal{A} \subseteq R$  using S0 S0.additive.subset by blast
  next
    show  $0 \in \bigcap \mathcal{A}$ 
  proof
    fix S assume S ∈  $\mathcal{A}$ 
    then interpret S:  $\text{Subring } S \ R \ (+) \ (\cdot) \ 0 \ 1$  by (rule sr)
    show  $0 \in S$  by (rule S.additive.sub-unit-closed)
  qed
qed

```

```

next
  fix g h assume g: g ∈ ⋂ A and h: h ∈ ⋂ A
  show g + h ∈ ⋂ A
  proof
    fix S assume S: S ∈ A
    then interpret S: Subring S R (+) (·) 0 1 by (rule sr)
    show g + h ∈ S using g h S by (blast intro: S.additive.sub-composition-closed)
  qed
next
  fix g assume g ∈ ⋂ A
  then have g ∈ R using S0 S0.additive.subset by blast
  then show additive.invertible g by simp
next
  fix g assume g: g ∈ ⋂ A
  show additive.inverse g ∈ ⋂ A
  proof
    fix S assume S: S ∈ A
    then interpret S: Subring S R (+) (·) 0 1 by (rule sr)
    have g ∈ S using g S by blast
    then show additive.inverse g ∈ S
      using S.additive.submonoid-inverse-closed S.additive.sub.invertible by blast
  qed
qed
— The multiplicative part is a submonoid.
interpret sg: Subgroup ⋂ A R (+) 0 by (rule sg)
show ?thesis
proof
  show ⋂ A ⊆ R using sg.subset .
  show 1 ∈ ⋂ A
  proof
    fix S assume S ∈ A
    then interpret S: Subring S R (+) (·) 0 1 by (rule sr)
    show 1 ∈ S by (rule S.multiplicative.sub-unit-closed)
  qed
  fix a b assume a: a ∈ ⋂ A and b: b ∈ ⋂ A
  show a · b ∈ ⋂ A
  proof
    fix S assume S: S ∈ A
    then interpret S: Subring S R (+) (·) 0 1 by (rule sr)
    show a · b ∈ S using a b S by (blast intro: S.multiplicative.sub-composition-closed)
  qed
qed
qed

```

The whole ring is a subring of itself (the top subring).

```

lemma whole-subring: Subring R R (+) (·) 0 1
proof -
  have Subgroup R R (+) 0 by (rule additive.subgroupI) auto
  then interpret sg: Subgroup R R (+) 0 .
  show ?thesis by unfold-locale auto

```

qed

39.2 The generated subring

The subring generated by a subset S : the intersection of all subrings containing $S \cap R$ (intersecting with R in the guard, as with subgroups and ideals, so that elements outside the carrier are ignored).

definition *subring-generated* :: 'a set $\Rightarrow$ 'a set
where *subring-generated* $S = \bigcap \{T. \text{Subring } T \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \wedge S \cap R \subseteq T\}$

lemma *subring-generated-is-subring*: *Subring (subring-generated S) $R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}$*

proof –

have $R \in \{T. \text{Subring } T \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \wedge S \cap R \subseteq T\}$ **using** *whole-subring* **by** *auto*

then have $ne: \{T. \text{Subring } T \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \wedge S \cap R \subseteq T\} \neq \{\}$ **by** *blast*

show *?thesis*

unfolding *subring-generated-def* **by** (rule *subring-Inter[OF ne]*) *auto*

qed

lemma *subring-generated-incl*: $S \cap R \subseteq \text{subring-generated } S$

unfolding *subring-generated-def* **by** *blast*

lemma *subring-generated-minimal*:

assumes *Subring* $T \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \ S \cap R \subseteq T$

shows *subring-generated* $S \subseteq T$

unfolding *subring-generated-def* **using** *assms* **by** (blast *intro: Inter-lower*)

lemma *subring-generated-subset-carrier*: *subring-generated* $S \subseteq R$

proof –

interpret *gen*: *Subring* *subring-generated* $S \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}$

by (rule *subring-generated-is-subring*)

show *?thesis* **by** (rule *gen.additive.subset*)

qed

end

end

40 The characteristic of a ring; finite domains have prime characteristic

theory *Characteristic*

imports *Ring-Theory HOL-Computational-Algebra.Primes*

begin

The natural-number multiples $n \cdot \mathbf{1}$ of the unit form a ring homomorphism $\mathbb{N} \rightarrow R$. Its kernel determines the *characteristic* of the ring; for a finite integral domain the characteristic is a prime, and the multiples $n \cdot \mathbf{1}$ form the prime

subfield. This is the first step toward the structural theorem that a finite field has prime-power order.

context *Ring*
begin

The n -fold sum $\mathbf{1} + \dots + \mathbf{1}$ (n summands), i.e. the image of n under the canonical map $\mathbb{N} \rightarrow R$.

definition *natmult* :: *nat* $\Rightarrow$ '*a*
where *natmult* $n = (((+)$ $\mathbf{1}) \rightsquigarrow n)$ $\mathbf{0}$

lemma *natmult-0* [*simp*]: *natmult* $0 = \mathbf{0}$
by (*simp add: natmult-def*)

lemma *natmult-Suc* [*simp*]: *natmult* (*Suc* n) = $\mathbf{1} + \text{natmult } n$
by (*simp add: natmult-def*)

lemma *natmult-closed* [*simp*]: *natmult* $n \in R$
by (*induct n*) *auto*

lemma *natmult-1* [*simp*]: *natmult* $1 = \mathbf{1}$
by *simp*

Additivity: *natmult* ($m + n$) = *natmult* $m + \text{natmult } n$.

lemma *natmult-add*: *natmult* ($m + n$) = *natmult* $m + \text{natmult } n$
by (*induct m*) (*auto simp: additive.associative*)

Multiplicativity: *natmult* ($m * n$) = *natmult* $m \cdot \text{natmult } n$.

lemma *natmult-mult*: *natmult* ($m * n$) = *natmult* $m \cdot \text{natmult } n$
by (*induct m*) (*auto simp: natmult-add distributive*)

Cancellation across a difference: if two multiples agree, the multiple of their difference vanishes.

lemma *natmult-diff-zero*:
assumes $m \leq n$ **and** *natmult* $m = \text{natmult } n$
shows *natmult* ($n - m$) = $\mathbf{0}$
proof –
have *natmult* $m + \text{natmult } (n - m) = \text{natmult } n$
using *natmult-add*[*of* m $n - m$] *le-add-diff-inverse*[*OF* *assms*(1)] **by** *simp*
also have $\dots = \text{natmult } m + \mathbf{0}$ **using** *assms* **by** (*simp add: additive.right-unit*)
finally have *natmult* $m + \text{natmult } (n - m) = \text{natmult } m + \mathbf{0}$.
then show *?thesis*
using *additive.invertible-left-cancel*[*of* *natmult* m *natmult* ($n - m$) $\mathbf{0}$] **by** *simp*
qed

end

context *integral-domain*
begin

The *characteristic*: the least positive n with $n \cdot \mathbf{1} = \mathbf{0}$ (or 0 if no such n exists).

definition *characteristic* :: nat
where *characteristic* = (LEAST n . $0 < n \wedge \text{natmult } n = \mathbf{0}$)

In a *finite* integral domain the map *natmult* cannot be injective, so some positive multiple of $\mathbf{1}$ vanishes.

lemma *finite-natmult-not-inj*:
assumes *finite R* **shows** $\exists n. 0 < n \wedge \text{natmult } n = \mathbf{0}$
proof –
have $\neg \text{inj } \text{natmult}$
by (meson *assms image-subsetI infinite-iff-countable-subset natmult-closed*)
then obtain $i\ j$ **where** $ij: i \neq j$ **and** $eq: \text{natmult } i = \text{natmult } j$ **by** (auto *simp: inj-def*)
show ?thesis
proof (cases $i < j$)
case *True*
then have $\text{natmult } (j - i) = \mathbf{0}$ **using** eq **by** (simp *add: natmult-diff-zero*)
with *True* **show** ?thesis
using *zero-less-diff* **by** blast
next
case *False*
then have $\text{natmult } (i - j) = \mathbf{0}$
by (simp *add: eq natmult-diff-zero*)
then show ?thesis
by (meson *False ij linorder-cases zero-less-diff*)
qed
qed

lemma *characteristic-pos*:
assumes *finite R* **shows** $0 < \text{characteristic} \wedge \text{natmult } \text{characteristic} = \mathbf{0}$
unfolding *characteristic-def*
using *LeastI-ex[OF finite-natmult-not-inj[OF assms]]* .

lemma *characteristic-least*:
 $\llbracket 0 < n; \text{natmult } n = \mathbf{0} \rrbracket \implies \text{characteristic} \leq n$
unfolding *characteristic-def* **by** (rule *Least-le*) *simp*

The characteristic exceeds 1 : $1 \cdot \mathbf{1} = \mathbf{1} \neq \mathbf{0}$ by nontriviality.

lemma *characteristic-gt-1*:
assumes *finite R* **shows** $1 < \text{characteristic}$
by (metis *assms characteristic-pos nontrivial less-one nat-neq-iff natmult-1*)

The characteristic of a finite integral domain is prime. If it factored as *characteristic* = $a * b$ nontrivially, then $\text{natmult } a \cdot \text{natmult } b = \mathbf{0}$ would force one factor's multiple to vanish, contradicting minimality.

theorem *characteristic-prime*:
assumes *finR: finite R* **shows** *prime characteristic*
proof (unfold *prime-nat-iff*, intro *conjI allI impI*)


```

    show  $1 < \text{characteristic}$  using characteristic-gt-1[OF finR] .
next
  fix m assume mdvd: m dvd characteristic
  then obtain k where ch: characteristic = m * k by (rule dvdE)
  have chz: natmult characteristic = 0 using characteristic-pos[OF finR] by simp
  have natmult m · natmult k = 0 using chz ch by (simp add: natmult-mult)
  then have natmult m = 0  $\vee$  natmult k = 0 using no-zero-divisors by simp
  — Both m and k are positive (the product is positive).
  have cpos:  $0 < \text{characteristic}$  using characteristic-pos[OF finR] by simp
  have mpos:  $0 < m$  and kpos:  $0 < k$  using ch cpos by auto
  then show m = 1  $\vee$  m = characteristic
  proof (cases natmult m = 0)
    case True
    then show ?thesis
      using characteristic-least cpos dvd-imp-le le-antisym mdvd mpos by presburger
  next
    case False
    then have nk: natmult k = 0 using  $\langle \text{natmult } m = \mathbf{0} \vee \text{natmult } k = \mathbf{0} \rangle$  by simp
    have kdvd: k dvd characteristic using ch by auto
    have keq: k = characteristic
      by (simp add: characteristic-least cpos dvd-imp-le kdvd kpos le-antisym nk)
    then show ?thesis
      using ch cpos by force
  qed
qed
end
end

```

41 Product Operator for Commutative Monoids

```

theory Finite-Composite
  imports Group-Theory

```

```
begin
```

41.1 Products over Finite Sets

```
context commutative-monoid begin
```

```
definition M-ify x  $\equiv$  if x  $\in$  M then x else 1
```

```
definition fincomp f A  $\equiv$  if finite A then Finite-Set.fold ( $\lambda x y. f x \cdot M\text{-ify } y$ ) 1 A else 1
```

```
lemma fincomp-empty [simp]: fincomp f {} = 1
  by (simp add: fincomp-def)

```

lemma *fincomp-infinite*[simp]: $\text{infinite } A \implies \text{fincomp } f A = \mathbf{1}$
by (*simp add: fincomp-def*)

lemma *left-commute*: $\llbracket a \in M; b \in M; c \in M \rrbracket \implies b \cdot (a \cdot c) = a \cdot (b \cdot c)$
using *commutative* **by** *force*

lemma *comp-fun-commute-onI*:
assumes $f \in F \rightarrow M$
shows *comp-fun-commute-on* $F (\lambda x y. f x \cdot M\text{-ify } y)$
using *assms*
by (*auto simp add: comp-fun-commute-on-def Pi-iff M-ify-def left-commute*)

lemma *fincomp-closed* [simp]:
assumes $f \in F \rightarrow M$
shows *fincomp* $f F \in M$
proof –
interpret *comp-fun-commute-on* $F \lambda x y. f x \cdot M\text{-ify } y$
by (*simp add: assms comp-fun-commute-onI*)
show ?thesis
unfolding *fincomp-def*
by (*smt (verit, ccfu-threshold) M-ify-def Pi-iff fold-graph-fold assms composition-closed equalityE fold-graph-closed-lemma unit-closed*)
qed

lemma *fincomp-insert* [simp]:
assumes $F: \text{finite } F \text{ } a \notin F$ **and** $f: f \in F \rightarrow M \text{ } f a \in M$
shows *fincomp* $f (\text{insert } a F) = f a \cdot \text{fincomp } f F$
proof –
interpret *comp-fun-commute-on* *insert* $a F \lambda x y. f x \cdot M\text{-ify } y$
by (*simp add: comp-fun-commute-onI f*)
show ?thesis
using *assms fincomp-closed commutative-monoid.M-ify-def commutative-monoid-axioms*
by (*fastforce simp add: fincomp-def*)
qed

lemma *fincomp-unit-eqI*: $(\bigwedge x. x \in A \implies f x = \mathbf{1}) \implies \text{fincomp } f A = \mathbf{1}$
proof (*induct A rule: infinite-finite-induct*)
case *empty* **show** ?case **by** *simp*
next
case (*insert* $a A$)
have $(\lambda i. \mathbf{1}) \in A \rightarrow M$ **by** *auto*
with *insert* **show** ?case **by** *simp*
qed *simp*

lemma *fincomp-unit* [simp]: *fincomp* $(\lambda i. \mathbf{1}) A = \mathbf{1}$
by (*simp add: fincomp-unit-eqI*)

lemma *funcset-Int-left* [simp, intro]:
 $\llbracket f \in A \rightarrow C; f \in B \rightarrow C \rrbracket \implies f \in A \text{ Int } B \rightarrow C$

by fast

lemma *funcset-Un-left [iff]*:

$(f \in A \text{ Un } B \rightarrow C) = (f \in A \rightarrow C \wedge f \in B \rightarrow C)$

by fast

lemma *fincomp-Un-Int*:

$\llbracket \text{finite } A; \text{finite } B; g \in A \rightarrow M; g \in B \rightarrow M \rrbracket \implies$

$\text{fincomp } g (A \cup B) \cdot \text{fincomp } g (A \cap B) =$

$\text{fincomp } g A \cdot \text{fincomp } g B$

— The reversed orientation looks more natural, but LOOPS as a simplrule!

proof (*induct set: finite*)

case empty then show ?case by simp

next

case (insert a A)

then have $g a \in M$ $g \in A \rightarrow M$ by blast+

with insert show ?case

by (simp add: Int-insert-left associative insert-absorb left-commute)

qed

lemma *fincomp-Un-disjoint*:

$\llbracket \text{finite } A; \text{finite } B; A \cap B = \{\}; g \in A \rightarrow M; g \in B \rightarrow M \rrbracket$

$\implies \text{fincomp } g (A \cup B) = \text{fincomp } g A \cdot \text{fincomp } g B$

by (metis Pi-split-domain fincomp-Un-Int fincomp-closed fincomp-empty right-unit)

lemma *fincomp-comp*:

$\llbracket f \in A \rightarrow M; g \in A \rightarrow M \rrbracket \implies \text{fincomp } (\lambda x. f x \cdot g x) A = (\text{fincomp } f A \cdot \text{fincomp } g A)$

proof (*induct A rule: infinite-finite-induct*)

case empty show ?case by simp

next

case (insert a A)

then have $f a \in M$ $g \in A \rightarrow M$ $g a \in M$ $f \in A \rightarrow M$ $(\lambda x. f x \cdot g x) \in A \rightarrow M$

by blast+

then show ?case

by (simp add: insert associative left-commute)

qed simp

lemma *fincomp-cong'*:

assumes $A = B$ $g \in B \rightarrow M$ $\bigwedge i. i \in B \implies f i = g i$

shows $\text{fincomp } f A = \text{fincomp } g B$

proof (*cases finite B*)

case True

then have ?thesis

using assms

proof (*induct arbitrary: A*)

case empty thus ?case by simp

next

case (insert x B)

then have $\text{fincomp } f A = \text{fincomp } f (\text{insert } x B)$ by simp

```

    also from insert have ... = f x · fincomp f B
      by (simp add: Pi-iff)
    also from insert have ... = g x · fincomp g B by fastforce
    also from insert have ... = fincomp g (insert x B)
      by (intro fincomp-insert [THEN sym]) auto
    finally show ?case .
  qed
  with assms show ?thesis by simp
next
  case False with assms show ?thesis by simp
qed

```

```

lemma fincomp-cong:
  assumes A = B g ∈ B → M ∧ i. i ∈ B = simp => f i = g i
  shows fincomp f A = fincomp g B
  using assms unfolding simp-implies-def by (blast intro: fincomp-cong')

```

Usually, if this rule causes a failed congruence proof error, the reason is that the premise $g \in B \rightarrow M$ cannot be shown. Adding *Pi-def* to the simpset is often useful. For this reason, *fincomp-cong* is not added to the simpset by default.

```

lemma fincomp-0 [simp]:
  f ∈ {0::nat} → M ⇒ fincomp f {..0} = f 0
  by (simp add: Pi-def)

```

```

lemma fincomp-0': f ∈ {..n} → M ⇒ (f 0) · fincomp f {Suc 0..n} = fincomp f {..n}
  by (metis Pi-split-insert-domain Suc-n-not-le-n atLeastAtMost-iff atLeastAtMost-insertL atMost-atLeast0 finite-atLeastAtMost fincomp-insert le0)

```

```

lemma fincomp-Suc [simp]:
  f ∈ {..Suc n} → M ⇒ fincomp f {..Suc n} = (f (Suc n) · fincomp f {..n})
  by (simp add: Pi-def atMost-Suc)

```

```

lemma fincomp-Suc2:
  f ∈ {..Suc n} → M ⇒ fincomp f {..Suc n} = (fincomp (%i. f (Suc i)) {..n} · f 0)
proof (induct n)
  case 0 thus ?case by (simp add: Pi-def)
next
  case Suc thus ?case
    by (simp add: associative Pi-def)
qed

```

```

lemma fincomp-Suc3:
  assumes f ∈ {..n :: nat} → M
  shows fincomp f {.. n} = (f n) · fincomp f {..< n}
proof (cases n = 0)
  case True thus ?thesis
    using assms atMost-Suc by simp
next

```

case *False*
then obtain k **where** $n = \text{Suc } k$
using *not0-implies-Suc* **by** *blast*
thus *?thesis*
using *fincomp-Suc*[*of f k*] *assms atMost-Suc lessThan-Suc-atMost* **by** *simp*
qed

lemma *fincomp-reindex*:
 $f \in (h \text{ ' } A) \rightarrow M \implies$
 $\text{inj-on } h \ A \implies \text{fincomp } f \ (h \text{ ' } A) = \text{fincomp } (\lambda x. f \ (h \ x)) \ A$
proof (*induct A rule: infinite-finite-induct*)
case (*infinite A*)
hence $\neg \text{finite } (h \text{ ' } A)$
using *finite-imageD* **by** *blast*
with $\langle \neg \text{finite } A \rangle$ **show** *?case* **by** *simp*
qed (*auto simp add: Pi-def*)

lemma *fincomp-const*:
assumes $a \ [simp]: a \in M$
shows $\text{fincomp } (\lambda x. a) \ A = \text{rec-nat } \mathbf{1} \ (\lambda u. (\cdot) \ a) \ (\text{card } A)$
by (*induct A rule: infinite-finite-induct*) *auto*

lemma *fincomp-singleton*:
assumes *i-in-A*: $i \in A$ **and** *fin-A*: *finite A* **and** *f-Pi*: $f \in A \rightarrow M$
shows $\text{fincomp } (\lambda j. \text{if } i = j \text{ then } f \ j \text{ else } \mathbf{1}) \ A = f \ i$
using *i-in-A fincomp-insert* [*of A - {i} i*] ($\lambda j. \text{if } i = j \text{ then } f \ j \text{ else } \mathbf{1}$)
 $\text{fin-A } f\text{-Pi } \text{fincomp-unit}$ [*of A - {i}*]
 fincomp-cong [*of A - {i} A - {i}*] ($\lambda j. \text{if } i = j \text{ then } f \ j \text{ else } \mathbf{1}$) ($\lambda i. \mathbf{1}$)
unfolding *Pi-def simp-implies-def* **by** (*force simp add: insert-absorb*)

lemma *fincomp-singleton-swap*:
assumes *i-in-A*: $i \in A$ **and** *fin-A*: *finite A* **and** *f-Pi*: $f \in A \rightarrow M$
shows $\text{fincomp } (\lambda j. \text{if } j = i \text{ then } f \ j \text{ else } \mathbf{1}) \ A = f \ i$
using *fincomp-singleton* [*OF assms*] **by** (*simp add: eq-commute*)

lemma *fincomp-mono-neutral-cong-left*:
assumes *finite B*
and $A \subseteq B$
and $l: \bigwedge i. i \in B - A \implies h \ i = \mathbf{1}$
and $gh: \bigwedge x. x \in A \implies g \ x = h \ x$
and $h: h \in B \rightarrow M$
shows $\text{fincomp } g \ A = \text{fincomp } h \ B$
proof –
have $eq: A \cup (B - A) = B$ **using** $\langle A \subseteq B \rangle$ **by** *blast*
have $d: A \cap (B - A) = \{\}$ **using** $\langle A \subseteq B \rangle$ **by** *blast*
from $\langle \text{finite } B \rangle \langle A \subseteq B \rangle$ **have** $f: \text{finite } A \ \text{finite } (B - A)$
by (*auto intro: finite-subset*)
have $h \in A \rightarrow M \ h \in B - A \rightarrow M$
using *assms* **by** (*auto simp: image-subset-iff-funcset*)
moreover have $\text{fincomp } g \ A = \text{fincomp } h \ A \cdot \text{fincomp } h \ (B - A)$

```

proof –
  have fincomp h (B – A) = 1
    using 1 fincomp-unit-eqI by blast
  moreover have fincomp g A = fincomp h A
    using  $\langle h \in A \rightarrow M \rangle$  fincomp-cong' gh by blast
  ultimately show ?thesis
    by (simp add:  $\langle h \in A \rightarrow M \rangle$ )
qed
ultimately show ?thesis
  by (simp add: fincomp-Un-disjoint [OF f d, unfolded eq])
qed

lemma fincomp-mono-neutral-cong-right:
  assumes finite B
    and  $A \subseteq B \wedge i. i \in B - A \implies g\ i = \mathbf{1} \wedge x. x \in A \implies g\ x = h\ x\ g \in B \rightarrow M$ 
  shows fincomp g B = fincomp h A
  using assms by (auto intro!: fincomp-mono-neutral-cong-left [symmetric])

lemma fincomp-mono-neutral-cong:
  assumes [simp]: finite B finite A
    and *:  $\wedge i. i \in B - A \implies h\ i = \mathbf{1} \wedge i. i \in A - B \implies g\ i = \mathbf{1}$ 
    and gh:  $\wedge x. x \in A \cap B \implies g\ x = h\ x$ 
    and g:  $g \in A \rightarrow M$ 
    and h:  $h \in B \rightarrow M$ 
  shows fincomp g A = fincomp h B
proof –
  have fincomp g A = fincomp g (A  $\cap$  B)
    by (rule fincomp-mono-neutral-cong-right) (use assms in auto)
  also have ... = fincomp h (A  $\cap$  B)
    by (rule fincomp-cong) (use assms in auto)
  also have ... = fincomp h B
    by (rule fincomp-mono-neutral-cong-left) (use assms in auto)
  finally show ?thesis .
qed

lemma fincomp-UN-disjoint:
  assumes
    finite I  $\wedge i. i \in I \implies \text{finite } (A\ i) \text{ pairwise } (\lambda i\ j. \text{disjnt } (A\ i) (A\ j))\ I$ 
     $\wedge i\ x. i \in I \implies x \in A\ i \implies g\ x \in M$ 
  shows fincomp g ( $\bigcup (A\ ` I)$ ) = fincomp ( $\lambda i. \text{fincomp } g\ (A\ i)$ ) I
  using assms
proof (induction set: finite)
  case empty
  then show ?case
    by force
next
  case (insert i I)
  then show ?case
    unfolding pairwise-def disjnt-def

```

```

    apply clarsimp
    apply (subst fincomp-Un-disjoint)
      apply (fastforce intro!: funcsetI fincomp-closed)+
    done
qed

```

```

lemma fincomp-Union-disjoint:
  
$$\llbracket \text{finite } C; \bigwedge A. A \in C \implies \text{finite } A \wedge (\forall x \in A. f\ x \in M); \text{pairwise disjoint } C \rrbracket \implies$$


$$\text{fincomp } f\ (\bigcup C) = \text{fincomp } (\text{fincomp } f)\ C$$

  by (frule fincomp-UN-disjoint [of C id f]) auto

end

```

41.2 Results for Abelian Groups

context *Abelian-Group* **begin**

```

lemma fincomp-inverse:
  
$$f \in A \rightarrow G \implies \text{fincomp } (\lambda x. \text{inverse } (f\ x))\ A = \text{inverse } (\text{fincomp } f\ A)$$

proof (induct A rule: infinite-finite-induct)
  case empty show ?case by simp
next
  case (insert a A)
  then have  $f\ a \in G$   $f \in A \rightarrow G$   $(\lambda x. \text{inverse } (f\ x)) \in A \rightarrow G$ 
    by blast+
  with insert show ?case
    by (simp add: commutative inverse-composition-commute)
qed simp

```

Jeremy Avigad. This should be generalized to arbitrary groups, not just Abelian ones, using Lagrange's theorem.

```

lemma power-order-eq-one:
  assumes  $\text{fin } [\text{simp}]: \text{finite } G$ 
  and  $a\ [\text{simp}]: a \in G$ 
  shows  $\text{rec-nat } 1\ (\lambda u. (\cdot)\ a)\ (\text{card } G) = 1$ 
proof -
  have  $\text{rec-G}: \text{rec-nat } 1\ (\lambda u. (\cdot)\ a)\ (\text{card } G) \in G$ 
    by (metis Pi-I' a fincomp-closed fincomp-const)
  have  $\bigwedge x. x \in G \implies x \in (\cdot)\ a \cdot G$ 
    by (metis a composition-closed imageI invertible invertible-inverse-closed invertible-right-inverse2)
  with a have  $(\cdot)\ a \cdot G = G$  by blast
  then have  $1 \cdot \text{fincomp } (\lambda x. x)\ G = \text{fincomp } (\lambda x. x)\ ((\cdot)\ a \cdot G)$ 
    by simp
  also have  $\dots = \text{fincomp } (\lambda x. a \cdot x)\ G$ 
    using fincomp-reindex
    by (subst (2) fincomp-reindex [symmetric]) (auto simp: inj-on-def)
  also have  $\dots = \text{fincomp } (\lambda x. a)\ G \cdot \text{fincomp } (\lambda x. x)\ G$ 
    by (simp add: fincomp-comp)
  also have  $\text{fincomp } (\lambda x. a)\ G = \text{rec-nat } 1\ (\lambda u. (\cdot)\ a)\ (\text{card } G)$ 

```

```

    by (simp add: fincomp-const)
  finally show ?thesis
    by (metis commutative fincomp-closed funcset-id invertible invertible-left-cancel
rec-G unit-closed)
qed

end

```

41.3 Transfer between two commutative monoids

commutative-monoid.fincomp depends on the *carrier* as well as on the operation, through *commutative-monoid.M-ify*, so two structures whose operations agree do not automatically have the same finite composites, and *commutative-monoid.fincomp-cong'* — which stays within one monoid — does not apply. What is true is the following: if the smaller carrier is contained in the larger, the operations agree there, the unit is shared, and the function composed takes its values in the smaller carrier, then the composites agree. This is what allows a sum computed inside a substructure to be read in the ambient structure, and in particular what makes evaluation of a polynomial independent of which of two nested structures it is performed in.

The other direction of the same gap: a finite composite may be pushed *along a homomorphism*. HOL-Algebra gets the corresponding fact for polynomial evaluation free from *ring-hom-ring.eval-hom'*; the carrier-set layer has no homomorphism apparatus at this level, so it is proved here directly, by the same finite induction as the transfer lemma below.

lemma *fincomp-hom*:

```

assumes M: commutative-monoid M f e and N: commutative-monoid N g d
and hM:  $\bigwedge x. x \in M \implies h\ x \in N$ 
and he:  $h\ e = d$ 
and hf:  $\bigwedge x\ y. \llbracket x \in M; y \in M \rrbracket \implies h\ (f\ x\ y) = g\ (h\ x)\ (h\ y)$ 
and k:  $k \in A \rightarrow M$ 
shows  $h\ (commutative-monoid.fincomp\ M\ f\ e\ k\ A)$ 
    =  $commutative-monoid.fincomp\ N\ g\ d\ (\lambda i. h\ (k\ i))\ A$ 
proof (cases finite A)
  case False
  then show ?thesis
    using he by (simp add: commutative-monoid.fincomp-infinite [OF M]
commutative-monoid.fincomp-infinite [OF N])
next
  case True
  case True
  then show ?thesis using k
proof (induction A rule: finite-induct)
  case empty
  then show ?case
    using he by (simp add: commutative-monoid.fincomp-empty [OF M]
commutative-monoid.fincomp-empty [OF N])
next

```



```

case (insert a A)
have kA: k ∈ A → M and ka: k a ∈ M using insert.premis by auto
have hKA: (λi. h (k i)) ∈ A → N using kA hM by auto
have hka: h (k a) ∈ N using ka by (rule hM)
have cLM: commutative-monoid.fincomp M f e k A ∈ M
  by (rule commutative-monoid.fincomp-closed [OF M kA])
have h (commutative-monoid.fincomp M f e k (insert a A))
  = h (f (k a) (commutative-monoid.fincomp M f e k A))
  by (simp add: commutative-monoid.fincomp-insert
    [OF M insert.hyps(1) insert.hyps(2) kA ka])
also have ... = g (h (k a)) (h (commutative-monoid.fincomp M f e k A))
  by (rule hf [OF ka cLM])
also have ... = g (h (k a)) (commutative-monoid.fincomp N g d (λi. h (k i)) A)
  using insert.IH [OF kA] by simp
also have ... = commutative-monoid.fincomp N g d (λi. h (k i)) (insert a A)
  by (rule commutative-monoid.fincomp-insert
    [OF N insert.hyps(1) insert.hyps(2) hKA hka, symmetric])
finally show ?case .
qed
qed

lemma fincomp-transfer:
assumes M: commutative-monoid M f e and N: commutative-monoid N g e
and sub: M ⊆ N
and agree: ⋀x y. [ x ∈ M; y ∈ M ] ⇒ f x y = g x y
and h: h ∈ A → M
shows commutative-monoid.fincomp M f e h A = commutative-monoid.fincomp N
g e h A
proof (cases finite A)
case False
then show ?thesis
  by (simp add: commutative-monoid.fincomp-infinite [OF M]
    commutative-monoid.fincomp-infinite [OF N])
next
case True
then show ?thesis using h
proof (induction A rule: finite-induct)
case empty
then show ?case
  by (simp add: commutative-monoid.fincomp-empty [OF M]
    commutative-monoid.fincomp-empty [OF N])
next
case (insert a A)
have hA: h ∈ A → M and ha: h a ∈ M using insert.premis by auto
then have hAN: h ∈ A → N and hAN: h a ∈ N using sub by auto
have cLM: commutative-monoid.fincomp M f e h A ∈ M
  by (rule commutative-monoid.fincomp-closed [OF M hA])
have commutative-monoid.fincomp M f e h (insert a A)
  = f (h a) (commutative-monoid.fincomp M f e h A)
  by (rule commutative-monoid.fincomp-insert [OF M insert.hyps(1) insert.hyps(2)

```

```

hA ha])
  also have ... = g (h a) (commutative-monoid.fincomp M f e h A)
  using agree [OF ha clM] by simp
  also have ... = g (h a) (commutative-monoid.fincomp N g e h A)
  using insert.IH [OF hA] by simp
  also have ... = commutative-monoid.fincomp N g e h (insert a A)
  by (rule commutative-monoid.fincomp-insert
      [OF N insert.hyps(1) insert.hyps(2) hAN haN, symmetric])
  finally show ?case .
qed
qed
end

```

42 The Polynomial Ring $R[X]$ (Carrier-Set Construction)

```

theory Poly-Ring
  imports Ring-Theory Finite-Composite
begin

```

Note on notation. *Ring-Theory* frees $+/-$ for the ring operations, while *FiniteProduct* (via *Main*) keeps HOL's arithmetic notation; the theory merge admits *both*. This is exactly what the convolution needs — ring operations on coefficients (type $'a$) and ordinary arithmetic on the *nat* indices $k - i$.

We construct the polynomial ring over a carrier-set ring R using the *finite-support function* representation: a polynomial is a function $\text{nat} \Rightarrow 'a$ that is eventually zero, with all coefficients drawn from the carrier R . This mirrors HOL's own $'a \text{ poly}$ type (which is *Abs-poly* over a finite-support function), so the bridge to the type-class world is near definitional, and there is no leading-zero normalisation invariant to maintain.

Coefficient i of a polynomial p is simply $p\ i$. Addition is pointwise; multiplication is the convolution $(p \otimes q)\ k = (\sum i \leq k. p\ i \cdot q\ (k - i))$, computed with the additive *fincomp* operator of R .

```

context Ring
begin

```

42.1 Carrier, coefficients, and support

The set of polynomials over R : finite support, coefficients in R .

definition *poly-carrier* :: $(\text{nat} \Rightarrow 'a) \text{ set}$
where *poly-carrier* $\equiv \{p. \text{finite } \{i. p\ i \neq 0\} \wedge (\forall i. p\ i \in R)\}$

lemma *poly-carrierI*:
 $\llbracket \text{finite } \{i. p\ i \neq 0\}; \bigwedge i. p\ i \in R \rrbracket \implies p \in \text{poly-carrier}$

unfolding *poly-carrier-def* **by** *blast*

lemma *poly-carrier-finite*: $p \in \text{poly-carrier} \implies \text{finite } \{i. p\ i \neq \mathbf{0}\}$
unfolding *poly-carrier-def* **by** *blast*

lemma *poly-carrier-coeff-closed*: $p \in \text{poly-carrier} \implies p\ i \in R$
unfolding *poly-carrier-def* **by** *blast*

42.2 Zero and one

definition *poly-zero* :: $\text{nat} \Rightarrow 'a$ ($\langle \mathbf{0}_P \rangle$)
where *poly-zero* $\equiv (\lambda i. \mathbf{0})$

definition *poly-one* :: $\text{nat} \Rightarrow 'a$ ($\langle \mathbf{1}_P \rangle$)
where *poly-one* $\equiv (\lambda i. \text{if } i = 0 \text{ then } \mathbf{1} \text{ else } \mathbf{0})$

42.3 Addition

definition *poly-add* :: $(\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a)$ (**infixl** $\langle \oplus_P \rangle$ 65)
where *poly-add* $p\ q \equiv (\lambda i. p\ i + q\ i)$

definition *poly-neg* :: $(\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a)$ ($\langle \ominus_P \rightarrow [66] \ 65 \rangle$)
where *poly-neg* $p \equiv (\lambda i. -\ p\ i)$

42.4 Multiplication (convolution)

definition *poly-mult* :: $(\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a)$ (**infixl** $\langle \otimes_P \rangle$ 70)
where *poly-mult* $p\ q \equiv (\lambda k. \text{additive.fincomp } (\lambda i. p\ i \cdot q\ (k - i))\ \{..k\})$

A polynomial is a *unit* when it has a two-sided multiplicative inverse in the polynomial ring. (Stated directly on the carrier-set operations, since the ambient *multiplicative* monoid acts on the coefficient type, not on polynomials.)

definition *poly-unit* :: $(\text{nat} \Rightarrow 'a) \Rightarrow \text{bool}$
where *poly-unit* $p \equiv p \in \text{poly-carrier}$
 $\wedge (\exists q. q \in \text{poly-carrier} \wedge p \otimes_P q = \mathbf{1}_P \wedge q \otimes_P p = \mathbf{1}_P)$

lemma *poly-unitI*:
 $\llbracket p \in \text{poly-carrier}; q \in \text{poly-carrier}; p \otimes_P q = \mathbf{1}_P; q \otimes_P p = \mathbf{1}_P \rrbracket \implies \text{poly-unit } p$
unfolding *poly-unit-def* **by** *blast*

lemma *poly-unit-closed*: $\text{poly-unit } p \implies p \in \text{poly-carrier}$
unfolding *poly-unit-def* **by** *blast*

42.5 Closure under the operations

lemma *poly-zero-closed*: $\mathbf{0}_P \in \text{poly-carrier}$
by (*rule poly-carrierI*) (*auto simp: poly-zero-def*)

lemma *poly-one-closed*: $\mathbf{1}_P \in \text{poly-carrier}$

by (simp add: poly-carrierI poly-one-def)

lemma *poly-add-closed*:

assumes $p \in \text{poly-carrier}$ $q \in \text{poly-carrier}$

shows $p \oplus_P q \in \text{poly-carrier}$

proof (rule *poly-carrierI*)

have $\{i. (p \oplus_P q) i \neq 0\} \subseteq \{i. p i \neq 0\} \cup \{i. q i \neq 0\}$

using *additive.right-unit poly-add-def* **by** *force*

moreover have *finite* $(\{i. p i \neq 0\} \cup \{i. q i \neq 0\})$

using *assms* **by** (simp add: *poly-carrier-finite*)

ultimately show *finite* $\{i. (p \oplus_P q) i \neq 0\}$ **by** (rule *finite-subset*)

show $\bigwedge i. (p \oplus_P q) i \in R$

using *assms* **by** (simp add: *poly-add-def poly-carrier-coeff-closed*)

qed

Each coefficient of a product is a finite sum of products of carrier elements.

lemma *poly-mult-coeff-closed*:

assumes $p \in \text{poly-carrier}$ $q \in \text{poly-carrier}$

shows $(p \otimes_P q) k \in R$

by (simp add: *assms(1,2) poly-carrier-coeff-closed poly-mult-def*)

A product coefficient vanishes beyond the sum of the two top degrees, so the product has finite support.

lemma *poly-mult-support-bound*:

assumes $p: p \in \text{poly-carrier}$ **and** $q: q \in \text{poly-carrier}$

and $k: k > (\text{if } \{i. p i \neq 0\} = \{\} \text{ then } 0 \text{ else } \text{Max } \{i. p i \neq 0\})$
 $+ (\text{if } \{i. q i \neq 0\} = \{\} \text{ then } 0 \text{ else } \text{Max } \{i. q i \neq 0\})$

shows $(p \otimes_P q) k = 0$

proof –

define Np **where** $Np = (\text{if } \{i. p i \neq 0\} = \{\} \text{ then } 0 \text{ else } \text{Max } \{i. p i \neq 0\})$

define Nq **where** $Nq = (\text{if } \{i. q i \neq 0\} = \{\} \text{ then } 0 \text{ else } \text{Max } \{i. q i \neq 0\})$

have *finP*: *finite* $\{i. p i \neq 0\}$ **using** p **by** (rule *poly-carrier-finite*)

have *finQ*: *finite* $\{i. q i \neq 0\}$ **using** q **by** (rule *poly-carrier-finite*)

have $pNp: \bigwedge i. p i \neq 0 \implies i \leq Np$

using *finP* **unfolding** *Np-def* **by** (auto simp: *Max-ge*)

have $qNq: \bigwedge i. q i \neq 0 \implies i \leq Nq$

using *finQ* **unfolding** *Nq-def* **by** (auto simp: *Max-ge*)

have *kgt*: $k > Np + Nq$ **using** k **unfolding** *Np-def Nq-def* **by** *simp*

have $p i \cdot q (k - i) = 0$ **if** $i \in \{..k\}$ **for** i

proof (cases $p i = 0$)

case *True* **then show** *?thesis* **using** $p q$ **by** (simp add: *poly-carrier-coeff-closed*)

next

case *False*

then show *?thesis* **using** $p q$ *kgt pNp qNq*

using *poly-carrier-coeff-closed Ring.right-zero Ring-axioms order.strict-trans*

by *fastforce*

qed

then show *?thesis*

unfolding *poly-mult-def* **by** (rule *additive.fincomp-unit-eqI*)

qed

lemma *poly-mult-closed*:

assumes $p \in \text{poly-carrier}$ $q \in \text{poly-carrier}$

shows $p \otimes_P q \in \text{poly-carrier}$

proof (rule *poly-carrierI*)

let $?B = (\text{if } \{i. p\ i \neq 0\} = \{\} \text{ then } 0 \text{ else } \text{Max } \{i. p\ i \neq 0\})$
 $+ (\text{if } \{i. q\ i \neq 0\} = \{\} \text{ then } 0 \text{ else } \text{Max } \{i. q\ i \neq 0\})$

have $\{i. (p \otimes_P q)\ i \neq 0\} \subseteq \{..?B\}$

by (fastforce simp: intro: *poly-mult-support-bound*[OF *assms*])

then show *finite* $\{i. (p \otimes_P q)\ i \neq 0\}$ **by** (rule *finite-subset*) simp

show $\bigwedge i. (p \otimes_P q)\ i \in R$ **using** *assms* **by** (rule *poly-mult-coeff-closed*)

qed

42.6 Distribution of multiplication over finite additive sums

A fixed factor distributes over an additive finite sum (left and right versions).

lemma *fincomp-mult-distrib-left*:

assumes $a \in R$ **and** $g \in A \rightarrow R$

shows $a \cdot \text{additive.fincomp } g\ A = \text{additive.fincomp } (\lambda i. a \cdot g\ i)\ A$

proof (cases *finite A*)

case *True*

then show *?thesis* **using** *assms*

proof (induct *A* rule: *finite-induct*)

case *empty* **then show** *?case* **by** simp

next

case (insert *x A*)

then have $a \cdot \text{additive.fincomp } g\ (\text{insert } x\ A) = a \cdot g\ x + \text{additive.fincomp } (\lambda i. a \cdot g\ i)\ A$

by (simp add: *distributive*)

also have $\dots = \text{additive.fincomp } (\lambda i. a \cdot g\ i)\ (\text{insert } x\ A)$

using *insert assms* **by** (simp add: *additive.fincomp-insert Pi-iff*)

finally show *?case* .

qed

next

case *False*

then show *?thesis* **using** *assms* **by** (simp add: *additive.fincomp-infinite*)

qed

lemma *fincomp-mult-distrib-right*:

assumes $a \in R$ **and** $g \in A \rightarrow R$

shows $\text{additive.fincomp } g\ A \cdot a = \text{additive.fincomp } (\lambda i. g\ i \cdot a)\ A$

proof (cases *finite A*)

case *True*

then show *?thesis* **using** *assms*

proof (induct *A* rule: *finite-induct*)

case *empty* **then show** *?case* **by** simp

next

case (insert *x A*)

```

    then have additive.fincomp g (insert x A) · a = g x · a + additive.fincomp (λi. g
i · a) A
    by (simp add: distributive)
    also have ... = additive.fincomp (λi. g i · a) (insert x A)
    using insert assms by (simp add: Pi-iff additive.fincomp-insert)
    finally show ?case .
qed
next
case False
then show ?thesis using assms by (simp add: additive.fincomp-infinite)
qed

```

42.7 Pointwise additive structure

lemma *poly-add-assoc*:

```

[[ p ∈ poly-carrier; q ∈ poly-carrier; r ∈ poly-carrier ]]
  ⇒ (p ⊕P q) ⊕P r = p ⊕P (q ⊕P r)
by (auto simp: poly-add-def additive.associative poly-carrier-coeff-closed)

```

lemma *poly-add-comm*:

```

[[ p ∈ poly-carrier; q ∈ poly-carrier ]] ⇒ p ⊕P q = q ⊕P p
by (auto simp: poly-add-def additive.commutative poly-carrier-coeff-closed)

```

lemma *poly-add-zero*: $p \in \text{poly-carrier} \implies \mathbf{0}_P \oplus_P p = p$

by (auto simp: poly-add-def poly-zero-def poly-carrier-coeff-closed)

lemma *poly-neg-closed*: $p \in \text{poly-carrier} \implies \ominus_P p \in \text{poly-carrier}$

proof (rule *poly-carrierI*)

assume $p: p \in \text{poly-carrier}$

have $\{i. (\ominus_P p) i \neq \mathbf{0}\} \subseteq \{i. p i \neq \mathbf{0}\}$

using *additive.inverse-unit poly-neg-def* by force

then show finite $\{i. (\ominus_P p) i \neq \mathbf{0}\}$

using p *poly-carrier-finite finite-subset* by blast

show $\bigwedge i. (\ominus_P p) i \in R$ using p by (simp add: *poly-neg-def poly-carrier-coeff-closed*)

qed

lemma *poly-add-neg*: $p \in \text{poly-carrier} \implies (\ominus_P p) \oplus_P p = \mathbf{0}_P$

by (auto simp: *poly-add-def poly-neg-def poly-zero-def poly-carrier-coeff-closed*)

lemma *poly-add-neg-right*: $p \in \text{poly-carrier} \implies p \oplus_P (\ominus_P p) = \mathbf{0}_P$

using *poly-add-comm poly-add-neg poly-neg-closed* by force

Additive cancellation: $x \oplus (a \oplus \ominus x) = a$ for carrier polynomials.

lemma *poly-add-minus-cancel*:

assumes $x \in \text{poly-carrier}$ $a \in \text{poly-carrier}$

shows $x \oplus_P (a \oplus_P (\ominus_P x)) = a$

by (metis assms *poly-add-assoc poly-add-comm poly-add-neg-right poly-add-zero poly-neg-closed*)

Left cancellation for polynomial addition.

lemma *poly-add-left-cancel*:

assumes $x: x \in \text{poly-carrier}$ **and** $a: a \in \text{poly-carrier}$ **and** $b: b \in \text{poly-carrier}$
and $\text{eq}: x \oplus_P a = x \oplus_P b$
shows $a = b$
by (*metis a b eq poly-add-assoc poly-add-minus-cancel poly-neg-closed x*)

The negation is the unique additive inverse.

lemma *poly-minus-unique*:
assumes $x: x \in \text{poly-carrier}$ **and** $y: y \in \text{poly-carrier}$ **and** $z: x \oplus_P y = \mathbf{0}_P$
shows $y = \ominus_P x$
using *poly-add-left-cancel poly-add-neg-right poly-neg-closed x y z* **by** *presburger*

Double negation.

lemma *poly-neg-neg*:
assumes $x: x \in \text{poly-carrier}$ **shows** $\ominus_P (\ominus_P x) = x$
using *poly-add-neg poly-minus-unique poly-neg-closed x* **by** *presburger*

Reassociation: $(u \oplus a) \oplus c = (u \oplus c) \oplus a$ in the abelian group.

lemma *poly-add-swap-right*:
assumes $u: u \in \text{poly-carrier}$ **and** $a: a \in \text{poly-carrier}$ **and** $c: c \in \text{poly-carrier}$
shows $(u \oplus_P a) \oplus_P c = (u \oplus_P c) \oplus_P a$
by (*metis a c poly-add-assoc poly-add-comm u*)

Subtracting one equation from another: if $u \oplus a = v \oplus b$ then $u \oplus \ominus v = b \oplus \ominus a$.

lemma *poly-sub-shift*:
assumes $u: u \in \text{poly-carrier}$ **and** $v: v \in \text{poly-carrier}$
and $a: a \in \text{poly-carrier}$ **and** $b: b \in \text{poly-carrier}$
and $\text{eq}: u \oplus_P a = v \oplus_P b$
shows $u \oplus_P (\ominus_P v) = b \oplus_P (\ominus_P a)$
proof —
have $nv: \ominus_P v \in \text{poly-carrier}$ **and** $na: \ominus_P a \in \text{poly-carrier}$
using $u v a b$ **by** (*auto simp: poly-neg-closed*)
— Add $\ominus v$ and $\ominus a$ to both sides of $u \oplus a = v \oplus b$.
have $(u \oplus_P a) \oplus_P ((\ominus_P v) \oplus_P (\ominus_P a)) = (v \oplus_P b) \oplus_P ((\ominus_P v) \oplus_P (\ominus_P a))$
using eq **by** *simp*
— LHS reduces to $u \oplus \ominus v$, RHS to $b \oplus \ominus a$.
moreover have $(u \oplus_P a) \oplus_P ((\ominus_P v) \oplus_P (\ominus_P a)) = u \oplus_P (\ominus_P v)$
by (*simp add: a na nv poly-add-assoc poly-add-closed poly-add-minus-cancel u*)
moreover have $(v \oplus_P b) \oplus_P ((\ominus_P v) \oplus_P (\ominus_P a)) = b \oplus_P (\ominus_P a)$
by (*metis b na nv poly-add-assoc poly-add-closed poly-add-minus-cancel v*)
ultimately show *?thesis* **by** *simp*
qed

If $x \oplus \ominus y = \mathbf{0}$ then $x = y$.

lemma *poly-diff-zero-eq*:
assumes $x \in \text{poly-carrier}$ **and** $y \in \text{poly-carrier}$ **and** $x \oplus_P (\ominus_P y) = \mathbf{0}_P$
shows $x = y$
by (*metis poly-add-comm poly-add-minus-cancel poly-add-zero poly-zero-closed assms*)

42.8 Multiplicative unit

lemma *poly-mult-one-left*:
assumes $p: p \in \text{poly-carrier}$
shows $\mathbf{1}_P \otimes_P p = p$
proof
fix k
have $(\mathbf{1}_P \otimes_P p) k = \text{additive.fincomp } (\lambda i. \mathbf{1}_P i \cdot p (k - i)) \{..k\}$
by (*simp add: poly-mult-def*)
also have $\dots = \text{additive.fincomp } (\lambda j. \text{if } j = 0 \text{ then } (\lambda -. p k) j \text{ else } \mathbf{0}) \{..k\}$
using p **by** (*intro additive.fincomp-cong'*) (*auto simp: poly-one-def poly-carrier-coeff-closed*)
also have $\dots = p k$
using p **by** (*simp add: additive.fincomp-singleton-swap poly-carrier-coeff-closed*)
finally show $(\mathbf{1}_P \otimes_P p) k = p k$.
qed

lemma *poly-mult-one-right*:
assumes $p: p \in \text{poly-carrier}$
shows $p \otimes_P \mathbf{1}_P = p$
proof
fix k
have $(p \otimes_P \mathbf{1}_P) k = \text{additive.fincomp } (\lambda i. p i \cdot \mathbf{1}_P (k - i)) \{..k\}$
by (*simp add: poly-mult-def*)
also have $\dots = \text{additive.fincomp } (\lambda j. \text{if } j = k \text{ then } (\lambda -. p k) j \text{ else } \mathbf{0}) \{..k\}$
using p **by** (*intro additive.fincomp-cong'*) (*auto simp: poly-one-def poly-carrier-coeff-closed*)
also have $\dots = p k$
using p **by** (*simp add: additive.fincomp-singleton-swap poly-carrier-coeff-closed*)
finally show $(p \otimes_P \mathbf{1}_P) k = p k$.
qed

42.9 Distributivity

lemma *poly-mult-add-distrib-left*:
assumes $p: p \in \text{poly-carrier}$ **and** $q: q \in \text{poly-carrier}$ **and** $r: r \in \text{poly-carrier}$
shows $p \otimes_P (q \oplus_P r) = (p \otimes_P q) \oplus_P (p \otimes_P r)$
proof
fix k
have $pi: \bigwedge i. p i \in R$ **and** $qi: \bigwedge i. q i \in R$ **and** $ri: \bigwedge i. r i \in R$
using $p q r$ **by** (*auto simp: poly-carrier-coeff-closed*)
have $(p \otimes_P (q \oplus_P r)) k = \text{additive.fincomp } (\lambda i. p i \cdot (q (k - i) + r (k - i))) \{..k\}$
by (*simp add: poly-mult-def poly-add-def*)
also have $\dots = \text{additive.fincomp } (\lambda i. p i \cdot q (k - i) + p i \cdot r (k - i)) \{..k\}$
using *distributive(1)* $pi qi ri$ **by** *presburger*
also have $\dots = \text{additive.fincomp } (\lambda i. p i \cdot q (k - i)) \{..k\}$
 $\quad + \text{additive.fincomp } (\lambda i. p i \cdot r (k - i)) \{..k\}$
by (*rule additive.fincomp-comp*) (*use pi qi ri in auto*)
also have $\dots = ((p \otimes_P q) \oplus_P (p \otimes_P r)) k$
by (*simp add: poly-mult-def poly-add-def*)
finally show $(p \otimes_P (q \oplus_P r)) k = ((p \otimes_P q) \oplus_P (p \otimes_P r)) k$.
qed

lemma *poly-mult-add-distrib-right*:
assumes $p: p \in \text{poly-carrier}$ **and** $q: q \in \text{poly-carrier}$ **and** $r: r \in \text{poly-carrier}$
shows $(q \oplus_P r) \otimes_P p = (q \otimes_P p) \oplus_P (r \otimes_P p)$
proof
fix k
have $pi: \bigwedge i. p\ i \in R$ **and** $qi: \bigwedge i. q\ i \in R$ **and** $ri: \bigwedge i. r\ i \in R$
using $p\ q\ r$ **by** (*auto simp: poly-carrier-coeff-closed*)
have $((q \oplus_P r) \otimes_P p)\ k = \text{additive.fincomp } (\lambda i. (q\ i + r\ i) \cdot p\ (k - i))\ \{..k\}$
by (*simp add: poly-mult-def poly-add-def*)
also have $\dots = \text{additive.fincomp } (\lambda i. q\ i \cdot p\ (k - i) + r\ i \cdot p\ (k - i))\ \{..k\}$
using *distributive(2)* $pi\ qi\ ri$ **by** *presburger*
also have $\dots = \text{additive.fincomp } (\lambda i. q\ i \cdot p\ (k - i))\ \{..k\}$
 $\quad + \text{additive.fincomp } (\lambda i. r\ i \cdot p\ (k - i))\ \{..k\}$
by (*rule additive.fincomp-comp*) (*use pi qi ri in auto*)
also have $\dots = ((q \otimes_P p) \oplus_P (r \otimes_P p))\ k$
by (*simp add: poly-mult-def poly-add-def*)
finally show $((q \oplus_P r) \otimes_P p)\ k = ((q \otimes_P p) \oplus_P (r \otimes_P p))\ k$.
qed

42.10 A Fubini law for the additive finite sum

A nested additive sum equals the sum over the dependent product (Sigma) of the index sets: $\sum_{i \in I}. \sum_{j \in A\ i}. g\ i\ j = \sum_{(i,j) \in \text{Sigma } I\ A}. g\ i\ j$.

lemma *fincomp-Sigma*:
assumes $I: \text{finite } I$ **and** $A: \bigwedge i. i \in I \implies \text{finite } (A\ i)$
and $g: \bigwedge j. i \in I \implies j \in A\ i \implies g\ i\ j \in R$
shows $\text{additive.fincomp } (\lambda i. \text{additive.fincomp } (g\ i)\ (A\ i))\ I$
 $= \text{additive.fincomp } (\lambda p. g\ (\text{fst } p)\ (\text{snd } p))\ (\text{Sigma } I\ A)$
proof –
have *disj: pairwise* $(\lambda i\ j. \text{disjnt } (\{i\} \times A\ i)\ (\{j\} \times A\ j))\ I$
by (*auto simp: pairwise-def disjnt-def*)
have $\text{additive.fincomp } (\lambda p. g\ (\text{fst } p)\ (\text{snd } p))\ (\text{Sigma } I\ A)$
 $= \text{additive.fincomp } (\lambda p. g\ (\text{fst } p)\ (\text{snd } p))\ (\bigcup i \in I. \{i\} \times A\ i)$
by (*simp add: Sigma-def*)
also have $\dots = \text{additive.fincomp } (\lambda i. \text{additive.fincomp } (\lambda p. g\ (\text{fst } p)\ (\text{snd } p))\ (\{i\} \times A\ i))\ I$
using $A\ I$ **by** (*intro additive.fincomp-UN-disjoint disj g*) *auto*
also have $\dots = \text{additive.fincomp } (\lambda i. \text{additive.fincomp } (g\ i)\ (A\ i))\ I$
proof (*rule additive.fincomp-cong'*)
fix i **assume** $i: i \in I$
have $\{i\} \times A\ i = \text{Pair } i\ \text{' } A\ i$ **by** *auto*
then have $\text{additive.fincomp } (\lambda p. g\ (\text{fst } p)\ (\text{snd } p))\ (\{i\} \times A\ i)$
 $= \text{additive.fincomp } (\lambda p. g\ (\text{fst } p)\ (\text{snd } p))\ (\text{Pair } i\ \text{' } A\ i)$
by *simp*
also have $\dots = \text{additive.fincomp } (\lambda j. g\ (\text{fst } (\text{Pair } i\ j))\ (\text{snd } (\text{Pair } i\ j)))\ (A\ i)$
proof (*rule additive.fincomp-reindex*)
show $(\lambda p. g\ (\text{fst } p)\ (\text{snd } p)) \in \text{Pair } i\ \text{' } A\ i \rightarrow R$ **using** $i\ g$ **by** *auto*
show *inj-on* $(\text{Pair } i)\ (A\ i)$ **by** (*simp add: inj-on-def*)

```

qed
also have ... = additive.fincomp (g i) (A i) by simp
finally show additive.fincomp (λp. g (fst p) (snd p)) ({i} × A i)
    = additive.fincomp (g i) (A i) .
qed (use I A g additive.fincomp-closed in auto)
finally show ?thesis ..
qed

```

42.11 Associativity of convolution

lemma *poly-mult-assoc*:

```

assumes p: p ∈ poly-carrier and q: q ∈ poly-carrier and r: r ∈ poly-carrier
shows (p ⊗P q) ⊗P r = p ⊗P (q ⊗P r)
proof
  fix n
  have pi: ∧i. p i ∈ R and qi: ∧i. q i ∈ R and ri: ∧i. r i ∈ R
    using p q r by (auto simp: poly-carrier-coeff-closed)
  — LHS as a sum over the triangle {(j,i). i ≤ j ≤ n}.
  have ((p ⊗P q) ⊗P r) n
    = additive.fincomp (λj. additive.fincomp (λi. p i · q (j - i)) {..j} · r (n - j))
      {..n}
    by (simp add: poly-mult-def)
  also have ... = additive.fincomp
    (λj. additive.fincomp (λi. (p i · q (j - i)) · r (n - j)) {..j}) {..n}
    using pi qi ri
    by (intro additive.fincomp-cong' fincomp-mult-distrib-right) auto
  also have ... = additive.fincomp (λx. (p (snd x) · q (fst x - snd x)) · r (n - fst
x))
    (Sigma {..n} (λj. {..j})))
    by (rule fincomp-Sigma) (use pi qi ri in auto)
  finally have LHS: ((p ⊗P q) ⊗P r) n
    = additive.fincomp (λx. (p (snd x) · q (fst x - snd x)) · r (n - fst x))
      (Sigma {..n} (λj. {..j}))) .
  — RHS as a sum over the triangle {(i,j). j ≤ n - i}.
  have (p ⊗P (q ⊗P r)) n
    = additive.fincomp (λi. p i · additive.fincomp (λj. q j · r (n - i - j)) {..n -
i}) {..n}
    by (simp add: poly-mult-def)
  also have ... = additive.fincomp
    (λi. additive.fincomp (λj. p i · (q j · r (n - i - j))) {..n - i}) {..n}
    using pi qi ri
    by (intro additive.fincomp-cong' fincomp-mult-distrib-left) auto
  also have ... = additive.fincomp (λx. p (fst x) · (q (snd x) · r (n - fst x - snd
x)))
    (Sigma {..n} (λi. {..n - i})))
    by (rule fincomp-Sigma) (use pi qi ri in auto)
  finally have RHS: (p ⊗P (q ⊗P r)) n
    = additive.fincomp (λx. p (fst x) · (q (snd x) · r (n - fst x - snd x)))
      (Sigma {..n} (λi. {..n - i}))) .
  — Reindex the LHS triangle to the RHS triangle by (j,i) ↦ (i, j - i).

```

```

let ?h =  $\lambda x::nat \times nat. (snd\ x, fst\ x - snd\ x)$ 
let ?h' =  $\lambda x::nat \times nat. (fst\ x + snd\ x, fst\ x)$ 
have bij: bij-betw ?h (Sigma {.. $n$ } ( $\lambda j. \{..j\}$ )) (Sigma {.. $n$ } ( $\lambda i. \{..n - i\}$ ))
  by (rule bij-betw-byWitness[where f' = ?h]) auto
let ?F =  $\lambda x. p\ (fst\ x) \cdot (q\ (snd\ x) \cdot r\ (n - fst\ x - snd\ x))$ 
have additive.fincomp ?F (Sigma {.. $n$ } ( $\lambda i. \{..n - i\}$ ))
  = additive.fincomp ?F (?h 'Sigma {.. $n$ } ( $\lambda j. \{..j\}$ ))
  using bij bij-betw-imp-surj-on by fastforce
also have ... = additive.fincomp ( $\lambda x. ?F\ (?h\ x)$ ) (Sigma {.. $n$ } ( $\lambda j. \{..j\}$ ))
proof (rule additive.fincomp-reindex)
  show ?F  $\in$  ?h 'Sigma {.. $n$ } ( $\lambda j. \{..j\}$ )  $\rightarrow R$  using pi qi ri by auto
  show inj-on ?h (Sigma {.. $n$ } ( $\lambda j. \{..j\}$ ))
    using bij bij-betw-imp-inj-on by blast
qed
also have ... = additive.fincomp ( $\lambda x. (p\ (snd\ x) \cdot q\ (fst\ x - snd\ x)) \cdot r\ (n - fst\ x)$ )
  (Sigma {.. $n$ } ( $\lambda j. \{..j\}$ ))
  using pi qi ri multiplicative.associative
  by (intro additive.fincomp-cong') auto
finally show (( $p \otimes_P q$ )  $\otimes_P r$ )  $n = (p \otimes_P (q \otimes_P r))\ n$ 
  using LHS RHS by simp
qed

```

42.12 Coefficients, degree, monomials, constants, and the variable

The coefficient map is just function application (kept as a named operation to mirror the HOL-Algebra polynomial interface).

definition *coeff* :: ($nat \Rightarrow 'a$) $\Rightarrow nat \Rightarrow 'a$
 where *coeff* $p = p$

lemma *coeff-eq* [*simp*]: *coeff* $p\ i = p\ i$
 by (*simp* add: *coeff-def*)

The degree is the greatest index with a nonzero coefficient (and 0 for the zero polynomial, as is conventional here).

definition *degree* :: ($nat \Rightarrow 'a$) $\Rightarrow nat$
 where *degree* $p \equiv (if\ \{i. p\ i \neq 0\} = \{\}\ then\ 0\ else\ Max\ \{i. p\ i \neq 0\})$

lemma *degree-zero* [*simp*]: *degree* $0_P = 0$
 by (*simp* add: *degree-def* *poly-zero-def*)

lemma *coeff-gt-degree*:
 assumes $p \in poly\text{-}carrier$ and *degree* $p < i$
 shows $p\ i = 0$
proof (rule *ccontr*)
 assume $p\ i \neq 0$
 then have $\S: \{i. p\ i \neq 0\} \neq \{\}$ by blast

moreover have *finite* $\{i. p \ i \neq \mathbf{0}\}$ **using** *assms(1)* **by** (*rule poly-carrier-finite*)
ultimately have $i \leq \text{Max } \{i. p \ i \neq \mathbf{0}\}$ **using** $\langle p \ i \neq \mathbf{0} \rangle$ **by** *simp*
then show *False* **using** *assms(2)*
by (*metis § degree-def linorder-not-le*)
qed

A degree bound: $\text{degree } p \leq n$ exactly when all coefficients above n vanish (for a polynomial, whose support is finite).

lemma *degree-le-iff*:
assumes $p \in \text{poly-carrier}$
shows $\text{degree } p \leq n \longleftrightarrow (\forall i > n. p \ i = \mathbf{0})$
proof
assume $\text{degree } p \leq n$
then show $\forall i > n. p \ i = \mathbf{0}$ **using** *coeff-gt-degree[OF assms]* **by** *force*
next
assume $*: \forall i > n. p \ i = \mathbf{0}$
show $\text{degree } p \leq n$
using $* \text{ assms degree-def linorder-le-less-linear poly-carrier-finite}$ **by** *force*
qed

lemma *degree-leI*:
 $\llbracket p \in \text{poly-carrier}; \bigwedge i. i > n \implies p \ i = \mathbf{0} \rrbracket \implies \text{degree } p \leq n$
by (*simp add: degree-le-iff*)

Negation preserves the support, hence the degree (for carrier polynomials).

lemma *degree-neg*:
assumes $p \in \text{poly-carrier}$ **shows** $\text{degree } (\ominus_P p) = \text{degree } p$
proof –
have $\bigwedge x. (\ominus_P p) \ x = \mathbf{0} \implies p \ x = \mathbf{0}$
by (*metis additive.inverse-unit p poly-neg-def poly-neg-neg*)
then have $\{i. (\ominus_P p) \ i \neq \mathbf{0}\} = \{i. p \ i \neq \mathbf{0}\}$
by (*metis additive.inverse-unit poly-neg-def*)
then show *?thesis* **by** (*simp add: degree-def*)
qed

42.13 Polynomials of bounded degree

The polynomials supported on $\{..<n\}$ — those of degree below n , together with the zero polynomial. These are exactly the carrier polynomials whose coefficients vanish from index n on. When the coefficient ring is finite, there are $|R|^n$ of them; they serve as the canonical representatives of $F[X]/(b)$ for b of degree n .

definition *low-poly* $:: \text{nat} \Rightarrow (\text{nat} \Rightarrow 'a) \text{ set}$
where $\text{low-poly } n \equiv \{p. p \in \text{poly-carrier} \wedge (\forall i \geq n. p \ i = \mathbf{0})\}$

lemma *low-polyI*:
 $\llbracket p \in \text{poly-carrier}; \bigwedge i. i \geq n \implies p \ i = \mathbf{0} \rrbracket \implies p \in \text{low-poly } n$
unfolding *low-poly-def* **by** *blast*

lemma *low-poly-closed*: $p \in \text{low-poly } n \implies p \in \text{poly-carrier}$
unfolding *low-poly-def* **by** *blast*

lemma *low-poly-coeff*: $\llbracket p \in \text{low-poly } n; i \geq n \rrbracket \implies p \ i = 0$
unfolding *low-poly-def* **by** *blast*
— *low-poly-coeff* and *low-poly-closed* must *not* go into a simpset.

The low-degree polynomials form a subgroup of the additive group, and this is immediate *coefficientwise*: the operations are pointwise, so no reasoning about degrees is needed.

lemma *low-poly-zero* [*simp*]: $0_P \in \text{low-poly } n$
using *low-polyI* *poly-zero-closed* *poly-zero-def* **by** *force*

lemma *low-poly-add-closed*:
assumes $p: p \in \text{low-poly } n$ **and** $q: q \in \text{low-poly } n$ **shows** $p \oplus_P q \in \text{low-poly } n$
proof (*rule low-polyI*)
have $pc: p \in \text{poly-carrier}$ **and** $qc: q \in \text{poly-carrier}$
using $p \ q$ **by** (*blast intro: low-poly-closed*)
show $p \oplus_P q \in \text{poly-carrier}$ **using** $pc \ qc$ **by** (*rule poly-add-closed*)
next
fix i **assume** $i: n \leq i$
have $p \ i = 0$ **and** $q \ i = 0$
using *low-poly-coeff* [*OF* $p \ i$] *low-poly-coeff* [*OF* $q \ i$] **by** *blast*
then show $(p \oplus_P q) \ i = 0$
by (*simp add: poly-add-def additive.left-unit*)
qed

lemma *low-poly-neg-closed*:
assumes $p: p \in \text{low-poly } n$ **shows** $\ominus_P p \in \text{low-poly } n$
using *additive.inverse-unit* *low-polyI* *low-poly-closed* *low-poly-coeff* p *poly-neg-closed* *poly-neg-def*
by *metis*

lemma *low-poly-one*:
assumes $n: 0 < n$ **shows** $1_P \in \text{low-poly } n$
using *low-polyI* n *poly-one-closed* *poly-one-def* **by** *force*

Membership of *low-poly* in terms of degree: a carrier polynomial lies in *low-poly* n iff it is zero or has degree below n .

lemma *low-poly-iff-degree*:
assumes $p: p \in \text{poly-carrier}$
shows $p \in \text{low-poly } n \longleftrightarrow (p = 0_P \vee \text{degree } p < n)$
proof
assume $p \in \text{low-poly } n$
then have $z: \bigwedge i. i \geq n \implies p \ i = 0$ **unfolding** *low-poly-def* **by** *blast*
show $p = 0_P \vee \text{degree } p < n$
proof (*cases* $p = 0_P$)
case *False*
— The support is finite and nonempty, so its maximum *degree* p lies in it.

```

    have fin: finite {i. p i ≠ 0} using p by (rule poly-carrier-finite)
    have ne: {i. p i ≠ 0} ≠ {} using False by (auto simp: poly-zero-def fun-eq-iff)
    then show ?thesis
      by (metis (mono-tags, lifting) Max-in degree-def fin leI mem-Collect-eq z)
  qed simp
next
  assume p = 0_P ∨ degree p < n
  then show p ∈ low-poly n
    using coeff-gt-degree low-polyI poly-zero-def p by auto
  qed

```

Restricting a low-degree polynomial to $\{..<n\}$ is a bijection onto the coefficient vectors $\{..<n\} \rightarrow_E R$; the inverse extends a vector by zero.

```

lemma bij-low-poly-PiE:
  bij-betw (λp. restrict p {..<n}) (low-poly n) ({..<n} →E R)
proof (rule bij-betw-byWitness[where f' = λf i. if i < n then f i else 0])
  show ∀ p ∈ low-poly n. (λi. if i < n then restrict p {..<n} i else 0) = p
    by (auto simp: low-poly-def not-less)
  show ∀ f ∈ {..<n} →E R. restrict (λi. if i < n then f i else 0) {..<n} = f
    by (auto simp: restrict-def PiE-def extensional-def fun-eq-iff)
  show (λp. restrict p {..<n}) ' low-poly n ⊆ {..<n} →E R
    using low-poly-closed poly-carrier-coeff-closed by force
  show (λf i. if i < n then f i else 0) ' ({..<n} →E R) ⊆ low-poly n
proof
  fix p assume p ∈ (λf i. if i < n then f i else 0) ' ({..<n} →E R)
  then obtain f where f: f ∈ {..<n} →E R and pf: p = (λi. if i < n then f i else
0) by blast
  have p ∈ poly-carrier
    using f by (intro poly-carrierI) (auto simp: pf)
  then show p ∈ low-poly n using pf by (auto intro!: low-polyI)
  qed
qed

```

Hence, over a finite coefficient ring, there are exactly $|R|^n$ polynomials of degree below n .

```

theorem card-low-poly:
  assumes finite R shows card (low-poly n) = card R ^ n
  using bij-betw-same-card [OF bij-low-poly-PiE] by (simp add: card-PiE)

```

The constant polynomial with value a .

```

definition poly-const :: 'a ⇒ (nat ⇒ 'a)
  where poly-const a = (λi. if i = 0 then a else 0)

```

```

lemma poly-const-closed: a ∈ R ⇒ poly-const a ∈ poly-carrier
  by (simp add: poly-carrier-def poly-const-def)

```

```

lemma poly-const-zero [simp]: poly-const 0 = 0_P
  by (simp add: poly-const-def poly-zero-def)

```

lemma *poly-const-one* [simp]: *poly-const* **1** = **1_P**
by (simp add: *poly-const-def poly-one-def*)

The monomial $a X^n$: the coefficient a at degree n .

definition *monom* :: 'a $\Rightarrow$ nat $\Rightarrow$ (nat $\Rightarrow$ 'a)
where *monom* $a\ n = (\lambda i. \text{if } i = n \text{ then } a \text{ else } \mathbf{0})$

lemma *monom-closed*: $a \in R \implies \text{monom } a\ n \in \text{poly-carrier}$
by (simp add: *monom-def poly-carrier-def*)

lemma *monom-0-eq-const*: *monom* $a\ 0 = \text{poly-const } a$
by (simp add: *monom-def poly-const-def*)

The variable X is the monomial **1** X^1 .

definition *var* :: nat $\Rightarrow$ 'a ($\langle X_P \rangle$)
where *var* = *monom* **1** **1**

lemma *var-closed*: $X_P \in \text{poly-carrier}$
by (simp add: *var-def monom-closed*)

lemma *coeff-var*: $X_P\ i = (\text{if } i = 1 \text{ then } \mathbf{1} \text{ else } \mathbf{0})$
by (simp add: *var-def monom-def*)

Multiplying by a monomial *monom* $c\ k$ shifts coefficients up by k and scales by c : (*monom* $c\ k \otimes b$) $j = (\text{if } k \leq j \text{ then } c \cdot b\ (j - k) \text{ else } \mathbf{0})$.

lemma *coeff-monom-mult*:
assumes $c: c \in R$ **and** $b: b \in \text{poly-carrier}$
shows (*monom* $c\ k \otimes_P b$) $j = (\text{if } k \leq j \text{ then } c \cdot b\ (j - k) \text{ else } \mathbf{0})$
proof –
have $bi: \bigwedge i. b\ i \in R$ **using** b **by** (simp add: *poly-carrier-coeff-closed*)
have (*monom* $c\ k \otimes_P b$) $j = \text{additive.fincomp } (\lambda i. \text{monom } c\ k\ i \cdot b\ (j - i))\ \{..j\}$
by (simp add: *poly-mult-def*)
also have $\dots = \text{additive.fincomp } (\lambda i. \text{if } i = k \text{ then } c \cdot b\ (j - k) \text{ else } \mathbf{0})\ \{..j\}$
by (rule *additive.fincomp-cong*) (use $c\ bi$ **in** $\langle \text{auto simp: monom-def} \rangle$)
finally have $\text{eq: } (\text{monom } c\ k \otimes_P b)\ j$
 $= \text{additive.fincomp } (\lambda i. \text{if } i = k \text{ then } c \cdot b\ (j - k) \text{ else } \mathbf{0})\ \{..j\} .$
show ?thesis
proof (cases $k \leq j$)
case *True*
then have $k \in \{..j\}$ **by** simp
have $\text{additive.fincomp } (\lambda i. \text{if } i = k \text{ then } (\lambda -. c \cdot b\ (j - k))\ i \text{ else } \mathbf{0})\ \{..j\}$
 $= (\lambda -. c \cdot b\ (j - k))\ k$
by (rule *additive.fincomp-singleton-swap*[OF $\langle k \in \{..j\} \rangle$]) (use $c\ bi$ **in** *auto*)
then show ?thesis **using** *eq True* **by** simp
next
case *False*
then have $\bigwedge i. i \in \{..j\} \implies (\text{if } i = k \text{ then } c \cdot b\ (j - k) \text{ else } \mathbf{0}) = \mathbf{0}$ **by** *auto*
then show ?thesis **using** *eq False*
by (smt (verit) *additive.fincomp-unit-eqI*)

qed
qed

Degrees of the basic polynomials

Multiplying a monomial by the variable raises its exponent, and multiplying by a constant scales its coefficient.

lemma *var-mult-monom*:
assumes $a \in R$ **shows** $X_P \otimes_P \text{monom } a \ k = \text{monom } a \ (Suc \ k)$
proof
fix j
have $(X_P \otimes_P \text{monom } a \ k) \ j = (\text{if } Suc \ 0 \leq j \text{ then } \mathbf{1} \cdot \text{monom } a \ k \ (j - Suc \ 0) \text{ else } 0)$
by (*simp add: var-def coeff-monom-mult [OF - monom-closed [OF a]]*)
then show $(X_P \otimes_P \text{monom } a \ k) \ j = \text{monom } a \ (Suc \ k) \ j$
using a by (*auto simp: monom-def*)
qed

A monomial is zero exactly when its coefficient is zero.

lemma *monom-eq-zero-iff* [*simp*]: $\text{monom } a \ n = \mathbf{0}_P \longleftrightarrow a = \mathbf{0}$
by (*auto simp: monom-def poly-zero-def fun-eq-iff*)

lemma *degree-monom*:
assumes $a \neq \mathbf{0}$
shows $\text{degree } (\text{monom } a \ n) = n$
using *assms* **by** (*auto simp: monom-def degree-def*)

lemma *degree-monom-le*: $\text{degree } (\text{monom } a \ n) \leq n$
using *degree-monom degree-zero monom-eq-zero-iff* **by** (*metis dual-order.refl le0*)

lemma *degree-poly-const*:
assumes $a \neq \mathbf{0}$
shows $\text{degree } (\text{poly-const } a) = 0$
using *degree-monom [OF assms, of 0]* **by** (*simp add: monom-0-eq-const*)

lemma *degree-const-le*: $\text{degree } (\text{poly-const } a) \leq 0$
using *degree-monom-le [of a 0]* **by** (*simp add: monom-0-eq-const*)

lemma *degree-one* [*simp*]: $\text{degree } \mathbf{1}_P = 0$
using *degree-const-le [of 1]* **by** *simp*

In a nontrivial coefficient ring, the variable has degree one. The explicit hypothesis keeps this lemma available in the more general *Ring* context above.

lemma *degree-var*:
assumes $\mathbf{1} \neq \mathbf{0}$
shows $\text{degree } X_P = 1$
by (*simp add: assms degree-monom var-def*)

The product of two constants is the constant of the product.


```

lemma poly-const-add:
  assumes  $a: a \in R$  and  $b: b \in R$ 
  shows  $\text{poly-const } a \oplus_P \text{poly-const } b = \text{poly-const } (a + b)$ 
  using poly-add-def poly-const-def by fastforce

lemma poly-const-mult:
  assumes  $a: a \in R$  and  $b: b \in R$ 
  shows  $\text{poly-const } a \otimes_P \text{poly-const } b = \text{poly-const } (a \cdot b)$ 
proof
  fix  $j$ 
  show  $(\text{poly-const } a \otimes_P \text{poly-const } b) j = \text{poly-const } (a \cdot b) j$ 
    using coeff-monom-mult[OF a poly-const-closed[OF b], of 0] a
    by (simp add: monom-0-eq-const poly-const-def)
qed

```

A polynomial of degree 0 is the constant given by its zeroth coefficient.

```

lemma degree-zero-imp-const:
  assumes  $p \in \text{poly-carrier}$   $\text{degree } p = 0$ 
  shows  $p = \text{poly-const } (p \ 0)$ 
  using assms coeff-gt-degree poly-const-def by auto

```

```

lemma degree-add-le:
  assumes  $p \in \text{poly-carrier}$   $q \in \text{poly-carrier}$ 
  shows  $\text{degree } (p \oplus_P q) \leq \max (\text{degree } p) (\text{degree } q)$ 
proof (intro degree-leI poly-add-closed assms)
qed (simp add: assms coeff-gt-degree poly-add-def)

```

Multiplication by the zero polynomial.

```

lemma
  assumes  $p \in \text{poly-carrier}$ 
  shows poly-mult-zero-right:  $p \otimes_P \mathbf{0}_P = \mathbf{0}_P$ 
    and poly-mult-zero-left:  $\mathbf{0}_P \otimes_P p = \mathbf{0}_P$ 
  using assms by (simp-all add: poly-mult-def poly-zero-def poly-carrier-coeff-closed)

```

The leading coefficient.

```

definition lead-coeff ::  $(\text{nat} \Rightarrow 'a) \Rightarrow 'a$ 
  where lead-coeff  $p = p (\text{degree } p)$ 

```

The coefficient at the degree is nonzero for a nonzero polynomial.

```

lemma coeff-degree-nonzero:
  assumes  $p \in \text{poly-carrier}$  and  $p \neq \mathbf{0}_P$ 
  shows  $p (\text{degree } p) \neq \mathbf{0}$ 
  by (metis antisym assms coeff-gt-degree less-not-refl linorder-not-le low-polyI
    low-poly-iff-degree)

```

```

lemma lead-coeff-nonzero:
   $\llbracket p \in \text{poly-carrier}; p \neq \mathbf{0}_P \rrbracket \implies \text{lead-coeff } p \neq \mathbf{0}$ 
  unfolding lead-coeff-def by (rule coeff-degree-nonzero)

```

The coefficient of a product at the sum of the two degrees is the product of the leading coefficients: every other term of the convolution vanishes for degree reasons.

```

lemma coeff-mult-degree-add:
  assumes  $p: p \in \text{poly-carrier}$  and  $q: q \in \text{poly-carrier}$ 
  shows  $(p \otimes_P q) (\text{degree } p + \text{degree } q) = p (\text{degree } p) \cdot q (\text{degree } q)$ 
proof -
  let  $?m = \text{degree } p$  and  $?n = \text{degree } q$ 
  have  $pi: \bigwedge i. p \ i \in R$  and  $qi: \bigwedge i. q \ i \in R$ 
    using  $p \ q$  by (auto simp: poly-carrier-coeff-closed)
  have  $(p \otimes_P q) (?m + ?n) = \text{additive.fincomp } (\lambda i. p \ i \cdot q \ (?m + ?n - i)) \ \{..?m + ?n\}$ 
    by (simp add: poly-mult-def)
  also have  $\dots = \text{additive.fincomp } (\lambda i. \text{if } i = ?m \text{ then } p \ ?m \cdot q \ ?n \text{ else } 0) \ \{..?m + ?n\}$ 
proof (rule additive.fincomp-cong')
    fix  $i$  assume  $i \in \{..?m + ?n\}$ 
    show  $p \ i \cdot q \ (?m + ?n - i) = (\text{if } i = ?m \text{ then } p \ ?m \cdot q \ ?n \text{ else } 0)$ 
    proof (cases i = ?m)
      case False
        have  $p \ i \cdot q \ (?m + ?n - i) = 0$ 
        proof (cases i > ?m)
          case False
            with  $pi$  coeff-gt-degree[OF  $q$ ]  $\langle i \neq ?m \rangle$  show ?thesis by simp
          qed (simp add: coeff-gt-degree p qi)
            then show ?thesis using False by simp
        qed auto
      qed (use pi qi in auto)
    also have  $\dots = p \ ?m \cdot q \ ?n$ 
      using additive.fincomp-singleton-swap[of  $?m \ \{..?m + ?n\} \ \lambda -. p \ ?m \cdot q \ ?n$ ]
      using  $pi \ qi$  by simp
    finally show ?thesis .
qed

```

The degree of a product is at most the sum of the degrees.

```

lemma degree-mult-le:
  assumes  $p: p \in \text{poly-carrier}$  and  $q: q \in \text{poly-carrier}$ 
  shows  $\text{degree } (p \otimes_P q) \leq \text{degree } p + \text{degree } q$ 
proof (rule degree-leI)
  show  $p \otimes_P q \in \text{poly-carrier}$  using assms by (rule poly-mult-closed)
  fix  $i$  assume  $i: i > \text{degree } p + \text{degree } q$ 
  have  $p \ j \cdot q \ (i - j) = 0$  if  $j \in \{..i\}$  for  $j$ 
    using  $i \ p \ q$  coeff-gt-degree
  proof (cases j > degree p)
    case True
      then show ?thesis using  $q$  coeff-gt-degree[OF  $p$ ]
        by (simp add: poly-carrier-coeff-closed)
    next
      case False

```

```

    then show ?thesis using p i coeff-gt-degree[OF q]
      by (simp add: poly-carrier-coeff-closed)
  qed
  then show (p  $\otimes_P$  q) i = 0
    unfolding poly-mult-def by (rule additive.fincomp-unit-eqI)
  qed

```

42.14 The ring structure

```

theorem poly-ring: Ring poly-carrier ( $\oplus_P$ ) ( $\otimes_P$ ) 0_P 1_P
proof -
  have add-grp: Group poly-carrier ( $\oplus_P$ ) 0_P
  proof (rule GroupI)
    show  $\bigwedge p. p \in \text{poly-carrier} \implies p \oplus_P 0_P = p$ 
      using poly-add-comm poly-add-zero poly-zero-closed by simp
    show  $\bigwedge p. p \in \text{poly-carrier} \implies \exists q \in \text{poly-carrier}. p \oplus_P q = 0_P \wedge q \oplus_P p = 0_P$ 
      using poly-add-neg poly-add-neg-right poly-neg-closed by blast
  qed (auto simp: poly-add-closed poly-zero-closed poly-add-assoc poly-add-zero)
  interpret add: Group poly-carrier ( $\oplus_P$ ) 0_P by (rule add-grp)
  interpret add: Abelian-Group poly-carrier ( $\oplus_P$ ) 0_P
    by unfold-locales (rule poly-add-comm)
  interpret mult: Monoid poly-carrier ( $\otimes_P$ ) 1_P
  proof
    qed (auto simp: poly-mult-one-left poly-mult-one-right poly-mult-assoc poly-one-closed
poly-mult-closed)
    show ?thesis
  proof qed (use poly-mult-add-distrib-left poly-mult-add-distrib-right in auto)
  qed

```

42.15 Evaluation at a point

Powers in the (multiplicative monoid of the) ring. We define them locally rather than importing *Group-Action*'s *Monoid.power*, to keep the polynomial theory's dependencies minimal.

```

definition rpow :: 'a  $\Rightarrow$  nat  $\Rightarrow$  'a
  where rpow a n = ((( $\cdot$ ) a)  $\frown$  n) 1

```

```

lemma rpow-0 [simp]: rpow a 0 = 1
  by (simp add: rpow-def)

```

```

lemma rpow-Suc [simp]: rpow a (Suc n) = a  $\cdot$  rpow a n
  by (simp add: rpow-def)

```

```

lemma rpow-closed [simp]: a  $\in R \implies$  rpow a n  $\in R$ 
  by (induct n) auto

```

Evaluation of a polynomial at a point a : the finite sum $\sum_{i \leq \text{degree } p} p\ i \cdot a^i$.

```

definition eval :: 'a  $\Rightarrow$  (nat  $\Rightarrow$  'a)  $\Rightarrow$  'a
  where eval a p = additive.fincomp ( $\lambda i. p\ i \cdot$  rpow a i) {.. $\text{degree } p$ }

```

Each summand lies in R , and the sum is closed.

lemma *eval-summand-closed*: $\llbracket p \in \text{poly-carrier}; a \in R \rrbracket \implies p \cdot i \cdot \text{rpow } a \cdot i \in R$
by (*simp add: poly-carrier-coeff-closed*)

lemma *eval-closed* [*simp*]:
assumes $p: p \in \text{poly-carrier}$ **and** $a: a \in R$ **shows** $\text{eval } a \cdot p \in R$
unfolding *eval-def* **using** $p \cdot a$ **by** (*auto intro!: additive.fincomp-closed eval-summand-closed*)

The summation range may be enlarged past the degree: the extra coefficients vanish.

lemma *eval-fincomp-le*:
assumes $p: p \in \text{poly-carrier}$ **and** $a: a \in R$ **and** $n: \text{degree } p \leq n$
shows $\text{eval } a \cdot p = \text{additive.fincomp } (\lambda i. p \cdot i \cdot \text{rpow } a \cdot i) \{..n\}$
proof –
have $\text{fin: } \bigwedge i. p \cdot i \cdot \text{rpow } a \cdot i \in R$ **using** $p \cdot a$ **by** (*rule eval-summand-closed*)
have $\text{split: } \{..n\} = \{..\text{degree } p\} \cup \{\text{Suc } (\text{degree } p) .. n\}$ **using** n **by** *auto*
have $\text{disj: } \{..\text{degree } p\} \cap \{\text{Suc } (\text{degree } p) .. n\} = \{\}$ **by** *auto*
have $\text{additive.fincomp } (\lambda i. p \cdot i \cdot \text{rpow } a \cdot i) \{..n\}$
 $= \text{additive.fincomp } (\lambda i. p \cdot i \cdot \text{rpow } a \cdot i) \{..\text{degree } p\}$
 $+ \text{additive.fincomp } (\lambda i. p \cdot i \cdot \text{rpow } a \cdot i) \{\text{Suc } (\text{degree } p) .. n\}$
using fin **by** (*simp add: split additive.fincomp-Un-disjoint[OF - - disj]*)
also have $\text{additive.fincomp } (\lambda i. p \cdot i \cdot \text{rpow } a \cdot i) \{\text{Suc } (\text{degree } p) .. n\} = \mathbf{0}$
using a *coeff-gt-degree*[*OF* p] **by** (*auto intro: additive.fincomp-unit-eqI*)
finally show *?thesis*
unfolding *eval-def* **using** fin **by** (*simp add: additive.fincomp-closed*)
qed

Evaluation of the basic polynomials.

lemma *eval-zero* [*simp*]: $a \in R \implies \text{eval } a \cdot \mathbf{0}_P = \mathbf{0}$
by (*simp add: eval-def poly-zero-def*)

lemma *eval-const* [*simp*]:
assumes $a: a \in R$ **and** $c: c \in R$ **shows** $\text{eval } a \cdot (\text{poly-const } c) = c$
unfolding *eval-def*
by (*simp add: degree-def poly-const-def*) (*use* c **in** *simp add: poly-const-def*)

lemma *eval-one* [*simp*]: $a \in R \implies \text{eval } a \cdot \mathbf{1}_P = \mathbf{1}$
using *eval-const*[*of* $a \cdot \mathbf{1}$] **by** *simp*

Evaluation is additive.

lemma *eval-add*:
assumes $p: p \in \text{poly-carrier}$ **and** $q: q \in \text{poly-carrier}$ **and** $a: a \in R$
shows $\text{eval } a \cdot (p \oplus_P q) = \text{eval } a \cdot p + \text{eval } a \cdot q$
proof –
define n **where** $n \equiv \max (\text{degree } p) (\text{degree } q)$
have $pq: p \oplus_P q \in \text{poly-carrier}$ **using** $p \cdot q$ **by** (*rule poly-add-closed*)
have $dpg: \text{degree } (p \oplus_P q) \leq n$ **unfolding** *n-def* **using** *degree-add-le*[*OF* $p \cdot q$].
have $pi: \bigwedge i. p \cdot i \in R$ **and** $qi: \bigwedge i. q \cdot i \in R$ **using** $p \cdot q$ **by** (*auto simp: poly-carrier-coeff-closed*)

```

have eval a (p ⊕P q) = additive.fincomp (λi. (p i + q i) · rpow a i) {..n}
using eval-fincomp-le[OF pq a dpq] by (simp add: poly-add-closed poly-add-def)
also have ... = additive.fincomp (λi. p i · rpow a i) {..n}
      + additive.fincomp (λi. q i · rpow a i) {..n}
by (simp add: a additive.fincomp-comp distributive(2) pi qi)
also have ... = eval a p + eval a q
by (metis (lifting) a eval-fincomp-le max.cobounded1 max.cobounded2 n-def p q)
finally show ?thesis .
qed

```

Evaluation of a monomial $\text{monom } c \ k$ is $c \cdot a^k$.

```

lemma eval-monom:
  assumes c: c ∈ R and a: a ∈ R
  shows eval a (monom c k) = c · rpow a k
proof –
  have eval a (monom c k) = additive.fincomp (λi. monom c k i · rpow a i) {..k}
  using a c degree-monom-le eval-fincomp-le monom-closed by blast
  also have ... = additive.fincomp (λi. if i = k then c · rpow a k else 0) {..k}
  by (rule additive.fincomp-cong') (use c a in (auto simp: monom-def))
  also have ... = c · rpow a k
  using additive.fincomp-singleton-swap[of k {..k} λi. c · rpow a k] c a by simp
  finally show ?thesis .
qed

```

Evaluation of the variable X_P returns the point itself.

```

lemma poly-rpow-var: Ring.rpow (⊗P) 1P XP k = monom 1 k
proof (induct k)
  case 0
  show ?case
  using Ring.rpow-0 [OF poly-ring, of XP] monom-0-eq-const [of 1] by simp
next
  case (Suc k) then show ?case
  using multiplicative.unit-closed poly-ring var-mult-monom by (metis Ring.rpow-Suc)
qed

```

```

lemma poly-const-mult-monom:
  assumes c: c ∈ R shows poly-const c ⊗P monom 1 k = monom c k
proof
  fix j
  have (poly-const c ⊗P monom 1 k) j = c · monom 1 k j
  using coeff-monom-mult [OF c monom-closed] by (simp flip: monom-0-eq-const)
  then show (poly-const c ⊗P monom 1 k) j = monom c k j
  using c by (simp add: monom-def)
qed

```

Expanding a polynomial into its monomials. Sums in the polynomial ring are pointwise, so a finite composite may be read coefficientwise, and then only the monomial whose exponent matches the coefficient index survives. This is the univariate counterpart of the closed form *ieval-ivar* in *Indexed-Poly*, and it is

what identifies evaluation of a polynomial's constant lift at the variable with the polynomial itself.

lemma *poly-add-cmonoid: commutative-monoid poly-carrier $(\oplus_P) \mathbf{0}_P$*
using *poly-ring by (meson Abelian-Group-def Ring-def)*

lemma *poly-fincomp-apply:*
assumes *f: $\bigwedge k. f\ k \in \text{poly-carrier}$*
shows *commutative-monoid.fincomp poly-carrier $(\oplus_P) \mathbf{0}_P f A j$*
 $= \text{additive.fincomp } (\lambda k. f\ k\ j)\ A$
proof (cases finite A)
case *False then show ?thesis*
using *commutative-monoid.fincomp-infinite poly-add-cmonoid poly-zero-def by fastforce*
next
case *True*
then show ?thesis
proof (induction A rule: finite-induct)
case *empty*
then show ?case
using *additive.fincomp-empty poly-add-cmonoid poly-zero-def by (metis commutative-monoid.fincomp-empty)*
next
case (insert a A)
have *fA: $f \in A \rightarrow \text{poly-carrier}$ and fa: $f\ a \in \text{poly-carrier}$ using f by auto*
have *commutative-monoid.fincomp poly-carrier $(\oplus_P) \mathbf{0}_P f (\text{insert } a\ A)\ j$*
 $= f\ a\ j + \text{commutative-monoid.fincomp poly-carrier } (\oplus_P) \mathbf{0}_P f\ A\ j$
by (simp add: commutative-monoid.fincomp-insert f insert.hyps poly-add-cmonoid poly-add-def)
also have $\dots = f\ a\ j + \text{additive.fincomp } (\lambda k. f\ k\ j)\ A$ **using** *insert.IH* **by** *simp*
also have $\dots = \text{additive.fincomp } (\lambda k. f\ k\ j)\ (\text{insert } a\ A)$
by (simp add: f insert.hyps poly-carrier-coeff-closed)
finally show ?case .
qed
qed

lemma *poly-sum-monom:*
assumes *q: $q \in \text{poly-carrier}$ and n: $\text{degree } q \leq n$*
shows *commutative-monoid.fincomp poly-carrier $(\oplus_P) \mathbf{0}_P (\lambda k. \text{monom } (q\ k)\ k) \{..n\}$*
 $\{..n\} = q$
proof
fix *j*
have *cl: $\text{monom } (q\ k)\ k \in \text{poly-carrier}$ for k*
using *q by (simp add: monom-closed poly-carrier-coeff-closed)*
have *qjR: $q\ j \in R$ by (rule poly-carrier-coeff-closed [OF q])*
have *commutative-monoid.fincomp poly-carrier $(\oplus_P) \mathbf{0}_P (\lambda k. \text{monom } (q\ k)\ k) \{..n\}$*
 j
 $= \text{additive.fincomp } (\lambda k. \text{monom } (q\ k)\ k\ j)\ \{..n\}$
by (rule poly-fincomp-apply [OF cl])
also have $\dots = \text{additive.fincomp } (\lambda k. \text{if } k = j \text{ then } (\lambda -. q\ j)\ k \text{ else } \mathbf{0})\ \{..n\}$

```

    using qjR monom-def by (intro additive.fincomp-cong') auto
  also have ... = q j
  proof (cases j ≤ n)
    case True
      with qjR show ?thesis
        by (intro additive.fincomp-singleton-swap) auto
    next
      case False
        have (if k = j then (λ-. q j) k else 0) = 0 if k ∈ {...n} for k
          using False that by auto
        moreover have q j = 0 using False n by (intro coeff-gt-degree [OF q]) simp
        ultimately show ?thesis
          using additive.fincomp-unit-eqI by fastforce
      qed
    finally show commutative-monoid.fincomp poly-carrier (⊕P) 0P (λk. monom (q k)
k) {...n} j
      = q j .
  qed

```

A polynomial is the value of its own constant lift at the variable. Evaluating in the polynomial ring itself, with each coefficient embedded as a constant and the point taken to be X_P , returns the polynomial. Composing this with the reduction homomorphism is what identifies the class of q with the value of q at the adjoined root.

```

lemma eval-const-lift-at-var:
  assumes q: q ∈ poly-carrier
  shows Ring.eval poly-carrier (⊕P) (⊗P) 0P 1P XP (λk. poly-const (q k)) = q
proof -
  have coeffR: q k ∈ R for k using q by (rule poly-carrier-coeff-closed)
  have supp: {k. poly-const (q k) ≠ 0P} = {k. q k ≠ 0}
    by (auto simp: poly-const-def poly-zero-def fun-eq-iff)
  have lift: (λk. poly-const (q k)) ∈ Ring.poly-carrier poly-carrier 0P
    using poly-carrier-finite [OF q] supp
    using coeffR poly-const-closed poly-ring by (metis Ring.poly-carrierI)
  have deg: Ring.degree 0P (λk. poly-const (q k)) = degree q
    using Ring.degree-def [OF poly-ring, of λk. poly-const (q k)] supp
    by (simp add: degree-def)
  have poly-const (q k) ⊗P Ring.rpow (⊗P) 1P XP k = monom (q k) k for k
    using poly-rpow-var [of k] poly-const-mult-monom [OF coeffR, of k] by simp
  then show ?thesis
    using Ring.eval-def [OF poly-ring] poly-sum-monom [OF q order-refl]
    by (simp add: deg)
  qed

```

```

lemma eval-var [simp]: a ∈ R ⟹ eval a XP = a
  using eval-monom[of 1 a 1] by (simp add: var-def)

```

Powers add exponents.

```

lemma rpow-add: a ∈ R ⟹ rpow a (i + j) = rpow a i · rpow a j

```

by (induct i) (auto simp: multiplicative.associative)

end

42.16 Evaluation along a ring homomorphism

Evaluating in the target, at the image of the point and with the images of the coefficients, gives the image of the evaluation. Powers transport by induction on the exponent and the sum by *fincomp-hom*; the summation range is the only thing that does not come for free, so the equality of the two degrees is taken as a hypothesis..

lemma *eval-hom*:

```

fixes A :: 'b set and add mult :: 'b  $\Rightarrow$  'b  $\Rightarrow$  'b and z u :: 'b
and B :: 'c set and badd bmult :: 'c  $\Rightarrow$  'c  $\Rightarrow$  'c and bz bu :: 'c
and h :: 'b  $\Rightarrow$  'c
assumes RA: Ring A add mult z u and RB: Ring B badd bmult bz bu
and hB:  $\bigwedge y. y \in A \implies h\ y \in B$ 
and hz:  $h\ z = bz$  and hu:  $h\ u = bu$ 
and hadd:  $\bigwedge y\ y'. \llbracket y \in A; y' \in A \rrbracket \implies h\ (add\ y\ y') = badd\ (h\ y)\ (h\ y')$ 
and hmult:  $\bigwedge y\ y'. \llbracket y \in A; y' \in A \rrbracket \implies h\ (mult\ y\ y') = bmult\ (h\ y)\ (h\ y')$ 
and x:  $x \in A$  and p:  $\bigwedge k. p\ k \in A$ 
and deg:  $Ring.degree\ bz\ (\lambda k. h\ (p\ k)) = Ring.degree\ z\ p$ 
shows  $Ring.eval\ B\ badd\ bmult\ bz\ bu\ (h\ x)\ (\lambda k. h\ (p\ k)) = h\ (Ring.eval\ A\ add\ mult\ z\ u\ x\ p)$ 
proof -
  have cmA: commutative-monoid A add z using RA by (simp add: Ring-def Abelian-Group-def)
  have cmB: commutative-monoid B badd bz using RB by (simp add: Ring-def Abelian-Group-def)
  have MonM: Monoid A mult u using RA by (simp add: Ring-def)
  have rp:  $Ring.rpow\ bmult\ bu\ (h\ x)\ n = h\ (Ring.rpow\ mult\ u\ x\ n)$  for n
  proof (induct n)
    case 0
    show ?case using hu by (simp add: Ring.rpow-0 [OF RA] Ring.rpow-0 [OF RB])
  next
    case (Suc n)
    have cl:  $Ring.rpow\ mult\ u\ x\ n \in A$  by (rule Ring.rpow-closed [OF RA x])
    then show ?case
      using Ring.rpow-Suc [OF RB] RA Suc hmult x by (metis Ring.rpow-Suc)
  qed
  define kf where  $kf \equiv (\lambda i. mult\ (p\ i)\ (Ring.rpow\ mult\ u\ x\ i))$ 
  have kA:  $kf\ i \in A$  for i
  unfolding kf-def
  using p Ring.rpow-closed [OF RA x] Monoid.composition-closed [OF MonM] by
blast
  have summand:  $bmult\ (h\ (p\ i))\ (Ring.rpow\ bmult\ bu\ (h\ x)\ i) = h\ (kf\ i)$  for i
  unfolding kf-def using rp hmult [OF p Ring.rpow-closed [OF RA x]] by simp
  have commutative-monoid.fincomp B badd bz  $(\lambda i. bmult\ (h\ (p\ i))\ (Ring.rpow\ bmult\ bu\ (h\ x)\ i))$ 
    {.. $Ring.degree\ bz\ (\lambda k. h\ (p\ k))$ }

```



```

    = commutative-monoid.fincomp B badd bz (λi. h (kf i)) {..Ring.degree z p}
  using summand deg by simp
  also have ... = h (commutative-monoid.fincomp A add z kf {..Ring.degree z p})
  proof (rule fincomp-hom [OF cmA cmB, symmetric])
    show ∧y. y ∈ A ⇒ h y ∈ B by (rule hB)
    show h z = bz by (rule hz)
    show ∧y y'. [ y ∈ A; y' ∈ A ] ⇒ h (add y y') = badd (h y) (h y') by (rule hadd)
    show kf ∈ {..Ring.degree z p} → A using kA by blast
  qed
  also have ... = h (Ring.eval A add mult z u x p)
    unfolding kf-def by (simp add: Ring.eval-def [OF RA, symmetric])
  finally show ?thesis
    using Ring.eval-def [OF RB] by simp
  qed

```

When the map is injective on the source and the target is exactly the image, the degree hypothesis is automatic: an injection sends the zero to the zero and nothing else there, so the two coefficient supports are the same set.

corollary *eval-transport*:

```

  fixes A :: 'b set and add mult :: 'b ⇒ 'b ⇒ 'b and z u :: 'b
    and h :: 'b ⇒ 'c and hadd hmult :: 'c ⇒ 'c ⇒ 'c
  assumes RA: Ring A add mult z u
    and RK: Ring (h ` A) hadd hmult (h z) (h u)
    and ha: ∧y y'. [ y ∈ A; y' ∈ A ] ⇒ h (add y y') = hadd (h y) (h y')
    and hm: ∧y y'. [ y ∈ A; y' ∈ A ] ⇒ h (mult y y') = hmult (h y) (h y')
    and inj: inj-on h A
    and x: x ∈ A and p: ∧k. p k ∈ A
  shows Ring.eval (h ` A) hadd hmult (h z) (h u) (h x) (λk. h (p k))
    = h (Ring.eval A add mult z u x p)
  proof (rule eval-hom [OF RA RK])
    show ∧y. y ∈ A ⇒ h y ∈ h ` A by blast
    show h z = h z by rule
    show h u = h u by rule
    show ∧y y'. [ y ∈ A; y' ∈ A ] ⇒ h (add y y') = hadd (h y) (h y') by (rule ha)
    show ∧y y'. [ y ∈ A; y' ∈ A ] ⇒ h (mult y y') = hmult (h y) (h y') by (rule hm)
    show x ∈ A by (rule x)
    show ∧k. p k ∈ A by (rule p)
  have cmA: commutative-monoid A add z using RA by (simp add: Ring-def Abelian-Group-def)
  have MonA: Monoid A add z using cmA by (simp add: commutative-monoid-def)
  have zA: z ∈ A by (rule Monoid.unit-closed [OF MonA])
  have supp: {i. h (p i) ≠ h z} = {i. p i ≠ z}
    using inj-on-eq-iff [OF inj p zA] by blast
  show Ring.degree (h z) (λk. h (p k)) = Ring.degree z p
    using Ring.degree-def [OF RK, of λk. h (p k)] Ring.degree-def [OF RA, of p] supp
    by simp
  qed

```

43 Polynomials over a Commutative Ring (continued): evaluation is multiplicative

context *commutative-ring*
begin

Evaluation is multiplicative. This needs commutativity (the point and the coefficients must commute). The proof mirrors *poly-mult-assoc*: expand both sides to sums over a triangle of index pairs and reindex by $(k, i) \mapsto (i, k - i)$.

lemma *eval-mult*:

assumes $p: p \in \text{poly-carrier}$ **and** $q: q \in \text{poly-carrier}$ **and** $a: a \in R$
shows $\text{eval } a \ (p \otimes_P q) = \text{eval } a \ p \cdot \text{eval } a \ q$

proof —

have $pi: \bigwedge i. p \ i \in R$ **and** $qi: \bigwedge i. q \ i \in R$ **using** $p \ q$ **by** (*auto simp: poly-carrier-coeff-closed*)
have $pq: p \otimes_P q \in \text{poly-carrier}$ **using** $p \ q$ **by** (*rule poly-mult-closed*)
define N **where** $N = \text{degree } p + \text{degree } q$
have $dpq: \text{degree } (p \otimes_P q) \leq N$ **unfolding** $N\text{-def}$ **using** *degree-mult-le[OF p q]* .
— LHS: distribute a^k into the convolution and view it over the triangle.
have $\text{eval } a \ (p \otimes_P q) = \text{additive.fincomp } (\lambda k. (p \otimes_P q) \ k \cdot \text{rpow } a \ k) \ \{..N\}$
using *eval-fincomp-le[OF pq a dpq]* .
also have $\dots = \text{additive.fincomp}$
 $(\lambda k. \text{additive.fincomp } (\lambda i. (p \ i \cdot q \ (k - i)) \cdot \text{rpow } a \ k) \ \{..k\}) \ \{..N\}$
unfolding *poly-mult-def*
using $pi \ qi \ a$ **by** (*auto intro!: additive.fincomp-cong' fincomp-mult-distrib-right*)
also have $\dots = \text{additive.fincomp } (\lambda x. (p \ (\text{snd } x) \cdot q \ (\text{fst } x - \text{snd } x)) \cdot \text{rpow } a \ (\text{fst } x))$
 $(\text{Sigma } \{..N\} \ (\lambda k. \{..k\}))$
by (*rule fincomp-Sigma*) (*use pi qi a in auto*)
finally have $LHS: \text{eval } a \ (p \otimes_P q)$
 $= \text{additive.fincomp } (\lambda x. (p \ (\text{snd } x) \cdot q \ (\text{fst } x - \text{snd } x)) \cdot \text{rpow } a \ (\text{fst } x))$
 $(\text{Sigma } \{..N\} \ (\lambda k. \{..k\}))$.
— RHS: expand the product of the two evaluation sums into a double sum.
have $\text{eval } a \ q \in R$ **using** $q \ a$ **by** (*rule eval-closed*)
then have $\text{eval } a \ p \cdot \text{eval } a \ q = \text{additive.fincomp } (\lambda i. (p \ i \cdot \text{rpow } a \ i) \cdot \text{eval } a \ q)$
 $\{..\text{degree } p\}$
using $pi \ a$ **unfolding** *eval-def* **by** (*intro fincomp-mult-distrib-right*) *auto*
also have $\dots = \text{additive.fincomp}$
 $(\lambda i. \text{additive.fincomp } (\lambda j. (p \ i \cdot \text{rpow } a \ i) \cdot (q \ j \cdot \text{rpow } a \ j)) \ \{..\text{degree}$
 $q\})$
 $\{..\text{degree } p\}$
unfolding *eval-def* **using** $pi \ qi \ a$ *additive.fincomp-closed*
by (*auto simp: intro!: additive.fincomp-cong' fincomp-mult-distrib-left*)
also have $\dots = \text{additive.fincomp}$
 $(\lambda x. (p \ (\text{fst } x) \cdot \text{rpow } a \ (\text{fst } x)) \cdot (q \ (\text{snd } x) \cdot \text{rpow } a \ (\text{snd } x)))$
 $(\text{Sigma } \{..\text{degree } p\} \ (\lambda i. \{..\text{degree } q\}))$
by (*intro fincomp-Sigma*) (*use pi qi a in auto*)
finally have $RHS: \text{eval } a \ p \cdot \text{eval } a \ q$
 $= \text{additive.fincomp } (\lambda x. (p \ (\text{fst } x) \cdot \text{rpow } a \ (\text{fst } x)) \cdot (q \ (\text{snd } x) \cdot \text{rpow } a \ (\text{snd } x)))$
 $(\text{Sigma } \{..\text{degree } p\} \ (\lambda i. \{..\text{degree } q\}))$
 $x)))$

$(\text{Sigma } \{..degree\ p\} (\lambda i. \{..degree\ q\})) .$
 — Both sides are sums of $p\ i \cdot q\ j \cdot a^{i+j}$; identify the index sets.
let $?T = \text{Sigma } \{..N\} (\lambda k. \{..k\})$
let $?S = \text{Sigma } \{..degree\ p\} (\lambda i. \{..degree\ q\})$
let $?G = \lambda i\ j. (p\ i \cdot q\ j) \cdot \text{rpow } a\ (i + j)$
 — Rewrite the LHS summand into the symmetric form $?G\ i\ j$ with $(i, j) = (\text{snd } x,$
 $\text{fst } x - \text{snd } x)$.
have $\text{LHS}' : \text{additive.fincomp } (\lambda x. (p\ (\text{snd } x) \cdot q\ (\text{fst } x - \text{snd } x)) \cdot \text{rpow } a\ (\text{fst } x))$
 $?T$
 $= \text{additive.fincomp } (\lambda x. ?G\ (\text{snd } x)\ (\text{fst } x - \text{snd } x))\ ?T$
by $(\text{intro additive.fincomp-cong}')\ (\text{auto simp: } a\ \text{pi } qi)$
 — And the RHS summand likewise, with $(i, j) = (\text{fst } x, \text{snd } x)$.
have $\text{RHS}' : \text{additive.fincomp } (\lambda x. (p\ (\text{fst } x) \cdot \text{rpow } a\ (\text{fst } x)) \cdot (q\ (\text{snd } x) \cdot \text{rpow } a\ (\text{snd } x)))\ ?S$
 $= \text{additive.fincomp } (\lambda x. ?G\ (\text{fst } x)\ (\text{snd } x))\ ?S$
by $(\text{simp add: } a\ \text{multiplicative.associative multiplicative.left-commute pi } qi\ \text{rpow-add})$
 — The rectangle $?S$ sits inside the larger triangle $?U$; the extra summands of $?U$
 vanish, since $p\ i = \mathbf{0}$ for $i > \text{degree } p$ and likewise q . So the sum over $?S$ equals the
 sum over $?U$.
let $?U = \text{Sigma } \{..N\} (\lambda i. \{..N - i\})$
have $GR : \bigwedge i\ j. ?G\ i\ j \in R$ **using** $pi\ qi\ a$ **by** simp
have $G\text{-eq-}S : \text{additive.fincomp } (\lambda x. ?G\ (\text{fst } x)\ (\text{snd } x))\ ?S = \text{additive.fincomp } (\lambda x.$
 $?G\ (\text{fst } x)\ (\text{snd } x))\ ?U$
proof —
have $?G\ (\text{fst } x)\ (\text{snd } x) = \mathbf{0}$ **if** $x \in ?U - ?S$ **for** x
proof —
have $p\ (\text{fst } x) = \mathbf{0} \vee q\ (\text{snd } x) = \mathbf{0}$
using $\text{that coeff-gt-degree}[OF\ p]\ \text{coeff-gt-degree}[OF\ q]$ **by** $(\text{auto simp: mem-Times-iff})$
then show $?thesis$
using $a\ \text{left-zero pi } qi\ \text{right-zero rpow-closed}$ **by** presburger
qed
with $GR\ N\text{-def}$ **show** $?thesis$
by $(\text{intro additive.fincomp-mono-neutral-cong-left})\ \text{auto}$
qed
 — Reindex the LHS triangle $\text{Sigma } \{..N\}$ *atMost* to $\text{SIGMA } i : \{..N\}. \{..N - i\}$ by
 $(k, i) \mapsto (i, k - i)$ — the same bijection used for associativity of convolution.
let $?h = \lambda x :: \text{nat} \times \text{nat}. (\text{snd } x, \text{fst } x - \text{snd } x)$
let $?h' = \lambda x :: \text{nat} \times \text{nat}. (\text{fst } x + \text{snd } x, \text{fst } x)$
have $\text{bij} : \text{bij-betw } ?h\ ?T\ ?U$
by $(\text{rule bij-betw-byWitness}[\text{where } f' = ?h])\ (\text{auto simp: } N\text{-def})$
have $\text{additive.fincomp } (\lambda x. ?G\ (\text{fst } x)\ (\text{snd } x))\ ?U$
 $= \text{additive.fincomp } (\lambda x. ?G\ (\text{fst } x)\ (\text{snd } x))\ (?h\ ' ?T)$
using $\text{bij bij-betw-imp-surj-on}$ **by** fastforce
also have $\dots = \text{additive.fincomp } (\lambda x. ?G\ (\text{fst } (?h\ x))\ (\text{snd } (?h\ x)))\ ?T$
using $\text{bij bij-betw-def } GR$ **by** $(\text{intro additive.fincomp-reindex})\ \text{auto}$
finally have $\text{additive.fincomp } (\lambda x. ?G\ (\text{fst } x)\ (\text{snd } x))\ ?S$
 $= \text{additive.fincomp } (\lambda x. ?G\ (\text{snd } x)\ (\text{fst } x - \text{snd } x))\ ?T$
using $G\text{-eq-}S$ **by** simp
then show $?thesis$ **using** $\text{LHS RHS LHS}'\ \text{RHS}'$ **by** simp

qed

Evaluation respects negation, hence is a ring homomorphism $F[X] \rightarrow F$.

lemma *eval-neg*:

assumes $p \in \text{poly-carrier}$ **and** $a \in R$

shows $\text{eval } a (\ominus_P p) = - \text{eval } a p$

proof –

have $\text{eval } a p + \text{eval } a (\ominus_P p) = 0$

using *assms eval-add poly-add-neg-right poly-neg-closed* **by** *fastforce*

then show *?thesis*

by (*simp add: assms additive.commutative additive.inverse-equality poly-neg-closed*)

qed

The linear polynomial $X - a$ (a root factor).

definition *root-factor* :: $'a \Rightarrow (\text{nat} \Rightarrow 'a)$

where $\text{root-factor } a \equiv X_P \oplus_P (\ominus_P \text{poly-const } a)$

lemma *root-factor-closed*: $a \in R \implies \text{root-factor } a \in \text{poly-carrier}$

unfolding *root-factor-def* **using** *var-closed poly-const-closed poly-neg-closed* **by**
(*simp add: poly-add-closed*)

root-factor a evaluates to 0 at *a*.

lemma *eval-root-factor* [*simp*]:

assumes *a*: $a \in R$ **shows** $\text{eval } a (\text{root-factor } a) = 0$

using *a poly-const-closed poly-neg-closed*

by (*simp add: eval-add eval-neg poly-const-closed root-factor-def var-closed*)

end

44 Polynomials over a Commutative Ring

context *commutative-ring*

begin

When the coefficient ring is commutative, so is the polynomial ring: the convolution $\sum_{i \leq k} p_i \cdot q_{(k-i)}$ is symmetric under the reindexing $i \mapsto k - i$.

lemma *poly-mult-comm*:

assumes *p*: $p \in \text{poly-carrier}$ **and** *q*: $q \in \text{poly-carrier}$

shows $p \otimes_P q = q \otimes_P p$

proof

fix *k*

have *inj*: $\text{inj-on } (\lambda i. k - i) \{..k::\text{nat}\}$

unfolding *atMost-iff image-def Ball-def set-eq-iff inj-on-def*

by *presburger*

have *inj*: $\text{inj-on } (\lambda i. k - i) \{..k::\text{nat}\}$ **and** *img*: $(\lambda i. k - i) ' \{..k\} = \{..k::\text{nat}\}$

unfolding *inj-on-def atMost-iff image-def Bex-def Ball-def set-eq-iff mem-Collect-eq*

by *presburger+*

```

have pi:  $\bigwedge i. p\ i \in R$  and qi:  $\bigwedge i. q\ i \in R$ 
  using p q by (auto simp: poly-carrier-coeff-closed)
have (p  $\otimes_P$  q) k = additive.fincomp ( $\lambda i. q\ (k - i) \cdot p\ i$ ) (( $\lambda i. k - i$ ) ‘ {.. $k$ })
  using img multiplicative.commutative pi poly-mult-def qi by presburger
also have ... = additive.fincomp ( $\lambda i. q\ (k - (k - i)) \cdot p\ (k - i)$ ) {.. $k$ }
  using additive.fincomp-reindex[where h =  $\lambda i. k - i$  and A = {.. $k$ }] pi qi inj by
auto
also have ... = additive.fincomp ( $\lambda i. q\ i \cdot p\ (k - i)$ ) {.. $k$ }
  by (rule additive.fincomp-cong') (use pi qi in auto)
also have ... = (q  $\otimes_P$  p) k by (simp add: poly-mult-def)
finally show (p  $\otimes_P$  q) k = (q  $\otimes_P$  p) k .
qed

```

Hence the polynomial ring over a commutative ring is itself commutative.

```

theorem poly-commutative-ring:
  commutative-ring poly-carrier ( $\oplus_P$ ) ( $\otimes_P$ )  $\mathbf{0}_P$   $\mathbf{1}_P$ 
proof -
  interpret R: Ring poly-carrier ( $\oplus_P$ ) ( $\otimes_P$ )  $\mathbf{0}_P$   $\mathbf{1}_P$  by (rule poly-ring)
  show ?thesis
  proof qed (simp add: poly-mult-comm)
qed
end

```

45 Polynomials over an Integral Domain

```

context integral-domain
begin

```

The leading coefficients are nonzero, and a domain has no zero divisors, so the product’s coefficient at *degree* $p + \text{degree } q$ is nonzero.

```

lemma degree-mult:
  assumes p:  $p \in \text{poly-carrier}$  and q:  $q \in \text{poly-carrier}$ 
    and pnz:  $p \neq \mathbf{0}_P$  and qnz:  $q \neq \mathbf{0}_P$ 
  shows degree (p  $\otimes_P$  q) = degree p + degree q
proof (rule antisym)
  have p (degree p)  $\cdot$  q (degree q)  $\neq \mathbf{0}$ 
    by (metis coeff-degree-nonzero local.no-zero-divisors poly-carrier-coeff-closed assms)
  then have (p  $\otimes_P$  q) (degree p + degree q)  $\neq \mathbf{0}$ 
    using coeff-mult-degree-add[OF p q] by simp
  then show degree p + degree q  $\leq$  degree (p  $\otimes_P$  q)
    by (meson coeff-gt-degree linorder-not-le p poly-mult-closed q)
qed (simp add: degree-mult-le p q)

```

Multiplication by a nonzero monomial shifts the degree by its exponent.

```

lemma degree-monom-mult:
  assumes c:  $c \in R$  and b:  $b \in \text{poly-carrier}$ 
    and cnz:  $c \neq \mathbf{0}$  and bnz:  $b \neq \mathbf{0}_P$ 

```

shows $\text{degree } (\text{monom } c \ n \ \otimes_P \ b) = n + \text{degree } b$
by (*simp add: b bnz c cnz degree-monom degree-mult monom-closed*)

Hence the polynomial ring over an integral domain has no zero divisors.

lemma *poly-no-zero-divisors:*

assumes $p \in \text{poly-carrier}$ **and** $q \in \text{poly-carrier}$ **and** $p \otimes_P q = \mathbf{0}_P$

shows $p = \mathbf{0}_P \vee q = \mathbf{0}_P$

proof (*rule ccontr*)

assume $\neg (p = \mathbf{0}_P \vee q = \mathbf{0}_P)$

then obtain $p \ (\text{degree } p) \neq \mathbf{0}$ **and** $q \ (\text{degree } q) \neq \mathbf{0}$

using *assms coeff-degree-nonzero* **by** *blast*

then have $(p \otimes_P q) \ (\text{degree } p + \text{degree } q) \neq \mathbf{0}$

by (*metis coeff-mult-degree-add no-zero-divisors poly-carrier-coeff-closed assms*)

then show *False* **using** $\langle p \otimes_P q = \mathbf{0}_P \rangle$ **by** (*simp add: poly-zero-def*)

qed

Hence $R[X]$ is itself an integral domain. .

theorem *poly-integral-domain:*

integral-domain poly-carrier $(\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$

proof –

interpret *comm: commutative-ring poly-carrier* $(\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$

by (*rule poly-commutative-ring*)

show *?thesis*

proof

show *nt: $\mathbf{1}_P \neq \mathbf{0}_P$*

by (*metis nontrivial poly-one-def poly-zero-def*)

qed (*simp add: poly-no-zero-divisors*)

qed

end

46 Polynomial Division over a Field

context *Field*

begin

46.1 Units of $F[X]$

Over a field, multiplication of nonzero polynomials adds degrees, so a polynomial can only be a multiplicative unit if it has degree 0: the units of $F[X]$ are exactly the nonzero constants. This pins down what it means for an irreducible polynomial to admit only trivial factorisations.

A nonzero constant is a unit, with the constant of the inverse coefficient as inverse.

lemma *poly-const-unit:*

assumes $a: a \in R$ **and** $\text{anz: } a \neq \mathbf{0}$

shows *poly-unit* (*poly-const* a)

by (*metis poly-unitI field-inverse poly-const-mult assms multiplicative.invertibleE poly-const-closed poly-const-one*)

Conversely, every unit of $F[X]$ is a nonzero constant. An invertible polynomial is nonzero and has degree 0 (degrees add under multiplication), so it is the constant of its own (nonzero) leading coefficient.

lemma *poly-unit-imp-const*:

assumes *inv*: *poly-unit* *p*

shows $p \neq \mathbf{0}_P \wedge \text{degree } p = 0 \wedge p = \text{poly-const } (p \ 0) \wedge p \ 0 \neq \mathbf{0}$

proof –

have *p*: $p \in \text{poly-carrier}$ **using** *inv* **by** (*rule poly-unit-closed*)

obtain *q* **where** *q*: $q \in \text{poly-carrier}$ **and** *pq*: $p \otimes_P q = \mathbf{1}_P$

using *inv* **unfolding** *poly-unit-def* **by** *blast*

have *pnz*: $p \neq \mathbf{0}_P$

using *q poly-mult-zero-left pq eval-zero nontrivial* **by** *force*

have *qnz*: $q \neq \mathbf{0}_P$

using *p pnz poly-mult-one-right poly-mult-zero-right pq* **by** *force*

— Degrees add and the product has degree 0, so *p* has degree 0.

have *d0*: *degree* *p* = 0

by (*metis add-eq-0-iff-both-eq-0 degree-mult degree-one p pnz pq q qnz*)

then show *?thesis* **using** *pnz d0*

by (*metis coeff-degree-nonzero degree-zero-imp-const p*)

qed

A polynomial is *irreducible* when it is a non-unit, nonzero, and every factorisation has a unit factor. Over a field this excludes the units (the nonzero constants) and the zero polynomial, matching the usual notion for $F[X]$.

definition *poly-irreducible* :: (*nat* $\Rightarrow$ 'a) $\Rightarrow$ *bool*

where *poly-irreducible* *p* $\equiv$

$p \in \text{poly-carrier} \wedge p \neq \mathbf{0}_P \wedge \neg \text{poly-unit } p$

$\wedge (\forall a \ b. a \in \text{poly-carrier} \longrightarrow b \in \text{poly-carrier} \longrightarrow p = a \otimes_P b$
 $\longrightarrow \text{poly-unit } a \vee \text{poly-unit } b)$

lemma *poly-irreducibleD-carrier*: *poly-irreducible* *p* $\implies p \in \text{poly-carrier}$

unfolding *poly-irreducible-def* **by** *blast*

lemma *poly-irreducibleD-nonzero*: *poly-irreducible* *p* $\implies p \neq \mathbf{0}_P$

unfolding *poly-irreducible-def* **by** *blast*

lemma *poly-irreducibleD-nonunit*: *poly-irreducible* *p* $\implies \neg \text{poly-unit } p$

unfolding *poly-irreducible-def* **by** *blast*

lemma *poly-irreducibleD-factor*:

$\llbracket \text{poly-irreducible } p; a \in \text{poly-carrier}; b \in \text{poly-carrier}; p = a \otimes_P b \rrbracket$

$\implies \text{poly-unit } a \vee \text{poly-unit } b$

unfolding *poly-irreducible-def* **by** *blast*

An irreducible polynomial is nonconstant: a nonzero constant is a unit, and the zero polynomial is excluded outright. Needed because the remainders modulo

p contain $\mathbf{1}_P$ only when p is nonconstant.

lemma *poly-irreducible-degree-pos*:

assumes *irr*: *poly-irreducible* p **shows** $0 < \text{degree } p$

proof (rule *ccontr*)

assume *non*: $\neg 0 < \text{degree } p$

have *pc*: $p \in \text{poly-carrier}$ **using** *irr* **by** (rule *poly-irreducibleD-carrier*)

have *const*: $p = \text{poly-const } (p \ 0)$

using *non degree-zero-imp-const pc* **by** *blast*

have *nz*: $p \ 0 \neq \mathbf{0}$

using *const irr poly-const-zero poly-irreducibleD-nonzero* **by** *force*

have *poly-unit* p

using *const poly-const-unit [OF poly-carrier-coeff-closed [OF pc] nz]* **by** *simp*

with *poly-irreducibleD-nonunit [OF irr]* **show** *False* ..

qed

interpretation *Ring-R*: *Ring* R *addition* $(\cdot) \ \mathbf{0} \ \mathbf{1} \ ..$

The reduction step of the division algorithm: when $\text{degree } b \leq \text{degree } a$ and both are nonzero, subtracting the appropriate monomial multiple of b cancels the leading term of a , strictly decreasing its degree.

lemma *div-step*:

assumes *a*: $a \in \text{poly-carrier}$ **and** *b*: $b \in \text{poly-carrier}$

and *anz*: $a \neq \mathbf{0}_P$ **and** *bnz*: $b \neq \mathbf{0}_P$ **and** *ge*: $\text{degree } b \leq \text{degree } a$

obtains m **where** $m \in \text{poly-carrier}$ $\text{degree } m = \text{degree } a - \text{degree } b$

and $a \oplus_P (\ominus_P (m \otimes_P b)) = \mathbf{0}_P \vee \text{degree } (a \oplus_P (\ominus_P (m \otimes_P b))) < \text{degree } a$

proof –

define *c* **where** $c \equiv \text{lead-coeff } a \cdot \text{multiplicative.inverse } (\text{lead-coeff } b)$

define *k* **where** $k \equiv \text{degree } a - \text{degree } b$

have *laR*: $\text{lead-coeff } a \in R$ **and** *lbR*: $\text{lead-coeff } b \in R$

using $a \ b$ **by** (*simp-all add: lead-coeff-def poly-carrier-coeff-closed*)

have *invb*: $\text{multiplicative.invertible } (\text{lead-coeff } b)$

by (*simp add: b bnz field-inverse lbR lead-coeff-nonzero*)

have *ibR*: $\text{multiplicative.inverse } (\text{lead-coeff } b) \in R$ **using** *lbR invb* **by** *simp*

have *ib-nz*: $\text{multiplicative.inverse } (\text{lead-coeff } b) \neq \mathbf{0}$

using *invb lbR multiplicative.invertible-right-inverse nontrivial right-zero* **by** *moura*

have *cnz*: $c \neq \mathbf{0}$

using $a \ anz \ c\text{-def } ibR \ ib\text{-nz } laR \ \text{lead-coeff-nonzero } \text{local.no-zero-divisors}$ **by** *auto*

define m **where** $m = \text{monom } c \ k$

have *mc*: $m \in \text{poly-carrier}$

by (*simp add: c-def ibR laR m-def monom-closed*)

have *mk*: $m \ k \neq \mathbf{0}$ **unfolding** $m\text{-def monom-def}$ **using** *cnz* **by** *simp*

— m has degree exactly k .

have *degm*: $\text{degree } m = k$

using *coeff-gt-degree degree-monom-le le-neq-implies-less m-def mc mk* **by** *blast*

have *mbc*: $m \otimes_P b \in \text{poly-carrier}$ **using** *mc b* **by** (rule *poly-mult-closed*)

— The product $m \otimes_P b$ has degree $\text{degree } a$ and matching leading term.

have *degmb*: $\text{degree } (m \otimes_P b) = \text{degree } a$

using $b \ bnz \ cnz \ degm \ \text{degree-mult } ge \ k\text{-def } m\text{-def } mc$ **by** *fastforce*

have *lead-mb*: $(m \otimes_P b) (\text{degree } a) = \text{lead-coeff } a$


```

    using coeff-mult-degree-add[OF mc b] invb degm ge k-def laR lbR m-def
    by (force simp: c-def multiplicative.associative lead-coeff-def monom-def)
  define d where  $d \equiv a \oplus_P (\ominus_P (m \otimes_P b))$ 
  have dc:  $d \in \text{poly-carrier}$  unfolding d-def using a mbc poly-neg-closed poly-add-closed
  by simp
  — The difference is zero or has strictly smaller degree, since its top coefficient cancels.
  have  $d = \mathbf{0}_P \vee \text{degree } d < \text{degree } a$ 
  proof (cases  $d = \mathbf{0}_P$ )
    case True then show ?thesis ..
  next
    case False
    have  $\text{degree } d \leq \text{degree } a$ 
    by (intro degree-leI[OF dc]) (simp add: a d-def poly-add-def poly-neg-def coeff-gt-degree degmb mbc)
    moreover have  $d (\text{degree } a) = \mathbf{0}$ 
    using Ring.lead-coeff-def Ring-axioms d-def laR lead-mb poly-add-def poly-neg-def
    by fastforce
    ultimately show ?thesis
    using False by (metis coeff-degree-nonzero dc le-neq-implies-less)
  qed
  then show ?thesis using mc degm d-def by (metis k-def that)
  qed

```

The division algorithm. For any dividend a and nonzero divisor b there exist a quotient q and remainder r with $a = b \otimes q \oplus r$ and the remainder either zero or of degree below that of b . Proved by strong induction on $\text{degree } a$, repeatedly applying *div-step*.

```

theorem poly-divide:
  assumes  $a: a \in \text{poly-carrier}$  and  $b: b \in \text{poly-carrier}$  and bnz:  $b \neq \mathbf{0}_P$ 
  shows  $\exists q r. q \in \text{poly-carrier} \wedge r \in \text{poly-carrier} \wedge a = (b \otimes_P q) \oplus_P r$ 
     $\wedge (r = \mathbf{0}_P \vee \text{degree } r < \text{degree } b)$ 
  using a
  proof (induct degree a arbitrary: a rule: less-induct)
    case (less a)
    show ?case
    proof (cases  $a = \mathbf{0}_P \vee \text{degree } a < \text{degree } b$ )
      case True then show ?thesis
      — Base case: the quotient is zero and  $a$  is the remainder.
      by (metis b less.premis poly-zero-closed poly-add-zero poly-mult-zero-right)
    next
      case False
      then have anz:  $a \neq \mathbf{0}_P$  and ge:  $\text{degree } b \leq \text{degree } a$  by auto
      — Reduce the leading term, then divide the smaller remainder by the induction hypothesis.
      obtain m where  $m: m \in \text{poly-carrier}$ 
        and  $d: a \oplus_P (\ominus_P (m \otimes_P b)) = \mathbf{0}_P \vee \text{degree } (a \oplus_P (\ominus_P (m \otimes_P b))) < \text{degree } a$ 
        using div-step[OF less.premis b anz bnz ge] by blast
      define dd where  $dd = a \oplus_P (\ominus_P (m \otimes_P b))$ 
      have ddc:  $dd \in \text{poly-carrier}$ 

```

unfolding *dd-def* **using** *less.premis* *b m poly-neg-closed poly-mult-closed poly-add-closed*
by *simp*
— *a* is recovered: $a = (m \otimes b) \oplus dd$, by additive-group cancellation.
have *mbc*: $m \otimes_P b \in \text{poly-carrier}$ **using** *m b* **by** (rule *poly-mult-closed*)
have *a-eq*: $a = (m \otimes_P b) \oplus_P dd$
unfolding *dd-def* **using** *poly-add-minus-cancel[OF mbc less.premis]* **by** *simp*
have *dd-div*: $\exists q r. q \in \text{poly-carrier} \wedge r \in \text{poly-carrier} \wedge dd = (b \otimes_P q) \oplus_P r$
 $\wedge (r = \mathbf{0}_P \vee \text{degree } r < \text{degree } b)$
by (*metis b d dd-def ddc less.hyps poly-add-zero poly-mult-zero-right*)
then obtain *q r* **where** *q*: $q \in \text{poly-carrier}$ **and** *r*: $r \in \text{poly-carrier}$
and *dd-eq*: $dd = (b \otimes_P q) \oplus_P r$ **and** *rdeg*: $r = \mathbf{0}_P \vee \text{degree } r < \text{degree } b$ **by**
blast
— Assemble: $a = (m \otimes b) \oplus (b \otimes q) \oplus r = b \otimes (m \oplus q) \oplus r$.
have $a = (m \otimes_P b) \oplus_P ((b \otimes_P q) \oplus_P r)$ **using** *a-eq dd-eq* **by** *simp*
also have $\dots = (b \otimes_P (m \oplus_P q)) \oplus_P r$
using *poly-mult-add-distrib-left[OF b m q]*
by (*simp add: poly-add-assoc poly-mult-closed poly-mult-comm q r*)
finally have $a = (b \otimes_P (m \oplus_P q)) \oplus_P r$.
then show *?thesis*
by (*metis r rdeg m poly-add-closed q*)
qed
qed

Uniqueness of quotient and remainder. If *b* is nonzero and two quotient/remainder pairs both work — each remainder of degree below *b* — then they coincide. For if the quotients differed, $b \otimes (q_1 \oplus \ominus q_2)$ would have degree $\geq \text{degree } b$ (degrees add over a domain), yet it equals $r_2 \oplus \ominus r_1$, of degree below *b*.

Left cancellation of a nonzero factor: $a \otimes x = a \otimes y \implies x = y$ (the ring of polynomials over a field is a domain).

lemma *poly-mult-left-cancel*:

assumes *a*: $a \in \text{poly-carrier}$ **and** *anz*: $a \neq \mathbf{0}_P$
and *x*: $x \in \text{poly-carrier}$ **and** *y*: $y \in \text{poly-carrier}$
and *eq*: $a \otimes_P x = a \otimes_P y$
shows $x = y$

proof —

have *ny*: $\ominus_P y \in \text{poly-carrier}$ **using** *y* **by** (rule *poly-neg-closed*)
have *dxyc*: $x \oplus_P (\ominus_P y) \in \text{poly-carrier}$ **using** *x ny* **by** (rule *poly-add-closed*)
— $a \otimes \ominus y = \ominus(a \otimes y)$: the two sum to $a \otimes (y \oplus \ominus y) = \mathbf{0}$.
have $(a \otimes_P y) \oplus_P (a \otimes_P (\ominus_P y)) = a \otimes_P (y \oplus_P (\ominus_P y))$
using *poly-mult-add-distrib-left[OF a y ny]* **by** *simp*
also have $\dots = \mathbf{0}_P$ **using** *a*
by (*simp add: poly-add-neg-right poly-mult-zero-right y*)
finally have $(a \otimes_P y) \oplus_P (a \otimes_P (\ominus_P y)) = \mathbf{0}_P$.
then have $a \otimes_P (x \oplus_P (\ominus_P y)) = \mathbf{0}_P$
— No zero divisors and $a \neq \mathbf{0}_P$ force $x \oplus \ominus y = \mathbf{0}_P$, i.e. $x = y$.
by (*simp add: a eq ny poly-mult-add-distrib-left x*)
then show *?thesis*
by (*meson a anz dxyc poly-diff-zero-eq poly-no-zero-divisors x y*)
qed

theorem *poly-divide-unique*:

assumes $b: b \in \text{poly-carrier}$ **and** $\text{bnz}: b \neq \mathbf{0}_P$
and $q1: q1 \in \text{poly-carrier}$ **and** $r1: r1 \in \text{poly-carrier}$
and $q2: q2 \in \text{poly-carrier}$ **and** $r2: r2 \in \text{poly-carrier}$
and $e1: a = (b \otimes_P q1) \oplus_P r1$ **and** $d1: r1 = \mathbf{0}_P \vee \text{degree } r1 < \text{degree } b$
and $e2: a = (b \otimes_P q2) \oplus_P r2$ **and** $d2: r2 = \mathbf{0}_P \vee \text{degree } r2 < \text{degree } b$
shows $q1 = q2 \wedge r1 = r2$

proof

have $bq1: b \otimes_P q1 \in \text{poly-carrier}$ **using** $b \ q1$ **by** (rule *poly-mult-closed*)
have $bq2: b \otimes_P q2 \in \text{poly-carrier}$ **using** $b \ q2$ **by** (rule *poly-mult-closed*)
define dq **where** $dq = q1 \oplus_P (\ominus_P q2)$
define dr **where** $dr = r2 \oplus_P (\ominus_P r1)$
have $dqc: dq \in \text{poly-carrier}$ **unfolding** $dq\text{-def}$ **using** $q1 \ q2$ *poly-neg-closed* **by** (*simp* *add: poly-add-closed*)

have $drc: dr \in \text{poly-carrier}$ **unfolding** $dr\text{-def}$ **using** $r1 \ r2$ *poly-neg-closed* **by** (*simp* *add: poly-add-closed*)

— The two equations give $b \otimes_P dq = dr$.

have *key*: $b \otimes_P dq = dr$

proof —

have $b \otimes_P dq = (b \otimes_P q1) \oplus_P (b \otimes_P (\ominus_P q2))$

unfolding $dq\text{-def}$ **using** *poly-mult-add-distrib-left*[*OF* $b \ q1$ *poly-neg-closed*[*OF*

$q2$]] .

also have $b \otimes_P (\ominus_P q2) = \ominus_P (b \otimes_P q2)$

using *poly-mult-add-distrib-left*[*OF* $b \ q2$ *poly-neg-closed*[*OF* $q2$]]

by (*simp* *add: b poly-add-neg-right poly-minus-unique poly-mult-closed poly-mult-zero-right poly-neg-closed* $q2$)

finally have $b \otimes_P dq = (b \otimes_P q1) \oplus_P (\ominus_P (b \otimes_P q2))$.

— And $(b \otimes q1) \oplus \ominus(b \otimes q2) = r2 \oplus \ominus r1 = dr$ from the two division equations.

then show *?thesis*

using $bq1 \ bq2 \ dr\text{-def} \ e1 \ e2$ *poly-sub-shift* $r1 \ r2$ **by** *force*

qed

— If dq were nonzero, $b \otimes_P dq$ would have $\text{degree} \geq \text{degree } b$.

have $dqz: dq = \mathbf{0}_P$

proof (rule *ccontr*)

assume $dqnz: dq \neq \mathbf{0}_P$

have $ge: \text{degree } b \leq \text{degree } dr$ **using** *key*

using $b \ \text{bnz} \ \text{degree-mult} \ dqc \ dqnz$ **by** *force*

— But dr has degree below b .

have $dr = \mathbf{0}_P \vee \text{degree } dr < \text{degree } b$

proof (*cases* $dr = \mathbf{0}_P$)

case *True* **then show** *?thesis* ..

next

case *False*

— dr is nonzero, so $0 < \text{degree } b$ (else both remainders vanish).

have $bpos: \text{degree } b > 0$

using *False* $d1 \ d2 \ dr\text{-def} \ \text{poly-add-neg-right} \ r2$ **by** *force*

have $r1b: \text{degree } r1 < \text{degree } b$ **using** $d1 \ bpos$ **by** *auto*

have $r2b: \text{degree } r2 < \text{degree } b$ **using** $d2 \ bpos$ **by** *auto*

have $\text{degree } dr \leq \max (\text{degree } r2) (\text{degree } (\ominus_P r1))$

```

      unfolding dr-def by (rule degree-add-le[OF r2 poly-neg-closed[OF r1]])
    then show ?thesis
      using degree-neg r1 r1b r2b by fastforce
  qed
then show False
  using b bnz dqz dqnz ge key poly-no-zero-divisors by fastforce
qed
— So  $q1 = q2$ , and then  $r1 = r2$  from either equation.
show qeq:  $q1 = q2$ 
  using dq-def dqz poly-diff-zero-eq q1 q2 by blast
show  $r1 = r2$ 
  using bq1 e1 e2 poly-add-left-cancel qeq r1 r2 by blast
qed

```

46.2 Remainder modulo a polynomial

The remainder of a on division by b : the canonical representative of a modulo b , of degree below that of b . Defined by choice from the division algorithm, and pinned down uniquely by *poly-divide-unique*.

definition $\text{pmod} :: (\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a)$
where $\text{pmod } a \ b = (\text{SOME } r. \exists q. q \in \text{poly-carrier} \wedge r \in \text{poly-carrier}$
 $\wedge a = (b \otimes_P q) \oplus_P r \wedge (r = \mathbf{0}_P \vee \text{degree } r < \text{degree } b))$

The defining property of pmod : there is a quotient realising it.

lemma *pmod-spec*:

assumes $a: a \in \text{poly-carrier}$ **and** $b: b \in \text{poly-carrier}$ **and** $\text{bnz}: b \neq \mathbf{0}_P$
shows $\exists q. q \in \text{poly-carrier} \wedge \text{pmod } a \ b \in \text{poly-carrier}$
 $\wedge a = (b \otimes_P q) \oplus_P \text{pmod } a \ b \wedge (\text{pmod } a \ b = \mathbf{0}_P \vee \text{degree } (\text{pmod } a \ b) <$
 $\text{degree } b)$

proof —

let $?P = \lambda r. \exists q. q \in \text{poly-carrier} \wedge r \in \text{poly-carrier}$
 $\wedge a = (b \otimes_P q) \oplus_P r \wedge (r = \mathbf{0}_P \vee \text{degree } r < \text{degree } b)$
have $\exists r. ?P \ r$ **using** *poly-divide[OF a b bnz]* **by** *blast*
then have $?P \ (\text{SOME } r. ?P \ r)$ **by** *(rule someI-ex)*
then show *?thesis* **unfolding** *pmod-def* .

qed

lemma *pmod-closed*:

$\llbracket a \in \text{poly-carrier}; b \in \text{poly-carrier}; b \neq \mathbf{0}_P \rrbracket \implies \text{pmod } a \ b \in \text{poly-carrier}$
using *pmod-spec* **by** *blast*

lemma *pmod-degree*:

$\llbracket a \in \text{poly-carrier}; b \in \text{poly-carrier}; b \neq \mathbf{0}_P \rrbracket$
 $\implies \text{pmod } a \ b = \mathbf{0}_P \vee \text{degree } (\text{pmod } a \ b) < \text{degree } b$
using *pmod-spec* **by** *blast*

pmod is the unique remainder: any low-degree r congruent to a modulo b equals $\text{pmod } a \ b$.

lemma *pmod-unique*:

assumes $a: a \in \text{poly-carrier}$ **and** $b: b \in \text{poly-carrier}$ **and** $\text{bnz}: b \neq \mathbf{0}_P$
and $q: q \in \text{poly-carrier}$ **and** $r: r \in \text{poly-carrier}$
and $\text{eq}: a = (b \otimes_P q) \oplus_P r$ **and** $\text{rd}: r = \mathbf{0}_P \vee \text{degree } r < \text{degree } b$
shows $\text{pmod } a \ b = r$
using $\text{pmod-spec}[OF \ a \ b \ \text{bnz}]$ $\text{poly-divide-unique}[OF \ b \ \text{bnz}]$
by (*metis eq q r rd*)

A remainder is a low-degree polynomial — the form in which the remainders are used as a carrier.

lemma *pmod-low-poly*:

assumes $a: a \in \text{poly-carrier}$ **and** $b: b \in \text{poly-carrier}$ **and** $\text{bnz}: b \neq \mathbf{0}_P$
shows $\text{pmod } a \ b \in \text{low-poly } (\text{degree } b)$
using $\text{pmod-closed } [OF \ a \ b \ \text{bnz}]$ $\text{pmod-degree } [OF \ a \ b \ \text{bnz}]$
by (*simp add: low-poly-iff-degree*)

A polynomial of degree below b is its own remainder.

lemma *pmod-small*:

assumes $b: b \in \text{poly-carrier}$ **and** $\text{bnz}: b \neq \mathbf{0}_P$
and $r: r \in \text{poly-carrier}$ **and** $\text{rd}: r = \mathbf{0}_P \vee \text{degree } r < \text{degree } b$
shows $\text{pmod } r \ b = r$
using $\text{pmod-unique}[OF \ r \ b \ \text{bnz}]$ $\text{poly-zero-closed } r \ - \ \text{rd}$
using $b \ \text{poly-add-zero}$ $\text{poly-mult-zero-right } r$ **by** *force*

Changing a by a multiple of b does not change the remainder. Everything below is a consequence of this and *pmod-unique*.

lemma *pmod-shift*:

assumes $y: y \in \text{poly-carrier}$ **and** $c: c \in \text{poly-carrier}$
and $b: b \in \text{poly-carrier}$ **and** $\text{bnz}: b \neq \mathbf{0}_P$
shows $\text{pmod } ((b \otimes_P c) \oplus_P y) \ b = \text{pmod } y \ b$

proof –

obtain q **where** $q: q \in \text{poly-carrier}$ **and** $\text{yeq}: y = (b \otimes_P q) \oplus_P \text{pmod } y \ b$
using $\text{pmod-spec } [OF \ y \ b \ \text{bnz}]$ **by** *blast*
have $r: \text{pmod } y \ b \in \text{poly-carrier}$ **by** (*rule pmod-closed [OF y b bnz]*)
have $(b \otimes_P c) \oplus_P y = (b \otimes_P (c \oplus_P q)) \oplus_P \text{pmod } y \ b$
using $\text{yeq } b \ c \ q \ r$
by (*simp add: poly-mult-add-distrib-left poly-add-assoc poly-mult-closed*)
moreover have $(b \otimes_P c) \oplus_P y \in \text{poly-carrier}$
using $b \ c \ y$ **by** (*simp add: poly-add-closed poly-mult-closed*)
ultimately show *?thesis*
using $\text{pmod-unique } [OF \ - \ b \ \text{bnz} \ - \ r \ - \ \text{pmod-degree } [OF \ y \ b \ \text{bnz}]] \ c \ q$
by (*simp add: poly-add-closed*)

qed

The remainder is additive outright: a sum of two low-degree remainders is already low-degree, so no second reduction is needed.

lemma *pmod-add*:

assumes $a: a \in \text{poly-carrier}$ **and** $a': a' \in \text{poly-carrier}$

and $b: b \in \text{poly-carrier}$ and $\text{bnz}: b \neq \mathbf{0}_P$
 shows $\text{pmod } (a \oplus_P a') b = \text{pmod } a b \oplus_P \text{pmod } a' b$
 proof –
 obtain q where $q: q \in \text{poly-carrier}$ and $\text{aeq}: a = (b \otimes_P q) \oplus_P \text{pmod } a b$
 using $\text{pmod-spec } [OF \ a \ b \ \text{bnz}]$ by *blast*
 obtain $q': q' \in \text{poly-carrier}$ and $\text{a'eq}: a' = (b \otimes_P q') \oplus_P \text{pmod } a' b$
 using $\text{pmod-spec } [OF \ a' \ b \ \text{bnz}]$ by *blast*
 have $r: \text{pmod } a b \in \text{poly-carrier}$ and $r': \text{pmod } a' b \in \text{poly-carrier}$
 using $\text{pmod-closed } [OF \ a \ b \ \text{bnz}] \ \text{pmod-closed } [OF \ a' \ b \ \text{bnz}]$ by *blast+*
 have $rr': \text{pmod } a b \oplus_P \text{pmod } a' b \in \text{poly-carrier}$ using $r \ r'$ by (rule *poly-add-closed*)
 — The sum of the two remainders is itself low-degree.
 have $\text{low}: \text{pmod } a b \oplus_P \text{pmod } a' b = \mathbf{0}_P \vee \text{degree } (\text{pmod } a b \oplus_P \text{pmod } a' b) < \text{degree } b$
 proof (cases $\text{pmod } a b \oplus_P \text{pmod } a' b = \mathbf{0}_P$)
 case *False*
 — Not both remainders vanish, so b has positive degree, and then a vanishing remainder has degree below it as well.
 have $\text{dpos}: 0 < \text{degree } b$
 using $\text{pmod-degree } [OF \ - \ b \ \text{bnz}] \ \text{False } a \ a' \ \text{poly-add-zero } \text{poly-zero-closed}$ by *fastforce*
 have $\text{dr}: \text{degree } (\text{pmod } a b) < \text{degree } b$ and $\text{dr}': \text{degree } (\text{pmod } a' b) < \text{degree } b$
 using $a \ a' \ b \ \text{dpos } \text{pmod-spec}$ by *fastforce+*
 then show *?thesis*
 using *low-poly-add-closed low-poly-iff-degree* $r \ r' \ rr'$ by *meson*
 qed *simp*
 — Reduce the left summand, then the right, as *pmod-mult* below does for a product; each step only moves a multiple of b . The sum of the remainders is then its own remainder.
 have *step1*: $a \oplus_P a' = (b \otimes_P q) \oplus_P (\text{pmod } a b \oplus_P a')$
 using $a' \ \text{aeq } b \ \text{poly-add-assoc } \text{poly-mult-closed } q \ r$ by *metis*
 have $b \otimes_P q' \in \text{poly-carrier}$ using $b \ q'$ by (rule *poly-mult-closed*)
 then have *step2*: $\text{pmod } a b \oplus_P a' = (b \otimes_P q') \oplus_P (\text{pmod } a b \oplus_P \text{pmod } a' b)$
 using $a' \ \text{eq } \text{poly-add-assoc } \text{poly-add-comm } r \ r'$ by *metis*
 have $\text{pmod } (a \oplus_P a') b = \text{pmod } (\text{pmod } a b \oplus_P a') b$
 using *step1 pmod-shift* $[OF \ \text{poly-add-closed } [OF \ r \ a'] \ q \ b \ \text{bnz}]$ by *simp*
 also have $\dots = \text{pmod } (\text{pmod } a b \oplus_P \text{pmod } a' b) b$
 using *step2 pmod-shift* $[OF \ rr' \ q' \ b \ \text{bnz}]$ by *simp*
 also have $\dots = \text{pmod } a b \oplus_P \text{pmod } a' b$ by (rule *pmod-small* $[OF \ b \ \text{bnz } rr' \ \text{low}]$)
 finally show *?thesis* .
 qed

The remainder is multiplicative *after a further reduction*: a product of two low-degree remainders need not be low-degree, so the outer *pmod* cannot be dropped here as it was for the sum. This is the congruence that makes the remainders a ring.

lemma *pmod-mult*:

assumes $a: a \in \text{poly-carrier}$ and $a': a' \in \text{poly-carrier}$
 and $b: b \in \text{poly-carrier}$ and $\text{bnz}: b \neq \mathbf{0}_P$
 shows $\text{pmod } (a \otimes_P a') b = \text{pmod } (\text{pmod } a b \otimes_P \text{pmod } a' b) b$

proof —

obtain q **where** $q: q \in \text{poly-carrier}$ **and** $\text{aeq}: a = (b \otimes_P q) \oplus_P \text{pmod } a \ b$
using $\text{pmod-spec } [OF \ a \ b \ \text{bnz}]$ **by** blast
obtain q' **where** $q': q' \in \text{poly-carrier}$ **and** $\text{a'eq}: a' = (b \otimes_P q') \oplus_P \text{pmod } a' \ b$
using $\text{pmod-spec } [OF \ a' \ b \ \text{bnz}]$ **by** blast
have $r: \text{pmod } a \ b \in \text{poly-carrier}$ **and** $r': \text{pmod } a' \ b \in \text{poly-carrier}$
using $\text{pmod-closed } [OF \ - \ b \ \text{bnz}] \ a \ a'$ **by** blast+
— Reduce the left factor, then the right; each step only moves a multiple of b .
have $a \otimes_P a' = ((b \otimes_P q) \otimes_P a') \oplus_P (\text{pmod } a \ b \otimes_P a')$
using $\text{aeq poly-mult-add-distrib-right } [OF \ a' \ \text{poly-mult-closed } [OF \ b \ q] \ r]$ **by** simp
then have $\text{step1}: a \otimes_P a' = (b \otimes_P (q \otimes_P a')) \oplus_P (\text{pmod } a \ b \otimes_P a')$
by $(\text{simp add: } a' \ b \ \text{poly-mult-assoc } q)$
have $\text{pmod } a \ b \otimes_P a' = (\text{pmod } a \ b \otimes_P (b \otimes_P q')) \oplus_P (\text{pmod } a \ b \otimes_P \text{pmod } a' \ b)$
using $\text{a'eq poly-mult-add-distrib-left } [OF \ r \ \text{poly-mult-closed } [OF \ b \ q'] \ r']$ **by** simp
then have $\text{step2}: \text{pmod } a \ b \otimes_P a' = (b \otimes_P (\text{pmod } a \ b \otimes_P q')) \oplus_P (\text{pmod } a \ b \otimes_P \text{pmod } a' \ b)$
using $b \ \text{poly-mult-assoc poly-mult-comm } q' \ r$ **by** metis
show $?thesis$
using $\text{step1 step2 } a' \ b \ \text{bnz pmod-shift poly-mult-closed } q \ q' \ r \ r'$ **by** metis
qed

46.3 The factor theorem

The root factor $\text{root-factor } a$ has degree exactly 1: its degree-1 coefficient is 1, which is nonzero in a (nontrivial) field.

lemma $\text{degree-root-factor}$:

assumes $a: a \in R$ **shows** $\text{degree } (\text{root-factor } a) = 1$

proof (rule antisym)

have $\text{rf}: \text{root-factor } a \in \text{poly-carrier}$ **using** a **by** ($\text{rule root-factor-closed}$)

— Coefficients above 1 vanish.

show $\text{degree } (\text{root-factor } a) \leq 1$

using $a \ \text{rf unfolding root-factor-def poly-const-def}$

by (intro degree-leI) ($\text{auto simp: poly-add-def poly-neg-def coeff-var poly-const-def}$)

have $\text{root-factor } a \ 1 = 1$

using $a \ \text{poly-const-def coeff-var poly-add-def poly-neg-def root-factor-def}$ **by** fastforce

force

then show $1 \leq \text{degree } (\text{root-factor } a)$

using $\text{degree-le-iff nontrivial rf}$ **by** fastforce

qed

lemma $\text{root-factor-nonzero}$:

assumes $a: a \in R$ **shows** $\text{root-factor } a \neq 0_P$

using $\text{assms degree-root-factor}$ **by** force

The factor theorem. A field element a is a root of p iff the linear polynomial $\text{root-factor } a$ divides p .

theorem factor-theorem :

assumes $p: p \in \text{poly-carrier}$ **and** $a: a \in R$

shows $\text{eval } a \ p = 0 \iff (\exists q. q \in \text{poly-carrier} \wedge p = \text{root-factor } a \otimes_P q)$

proof

assume $\exists q. q \in \text{poly-carrier} \wedge p = \text{root-factor } a \otimes_P q$ **then show** $\text{eval } a \ p = 0$
using $a \text{ eval-mult root-factor-closed}$ **by** force

next

assume $\text{root: eval } a \ p = 0$

obtain $\text{rf: root-factor } a \in \text{poly-carrier}$ **and** $\text{rfnz: root-factor } a \neq 0_P$

using a **by** $(\text{simp add: root-factor-closed root-factor-nonzero})$

— Divide by the degree-1 factor; the remainder has degree < 1 , so is a constant.

obtain $q \ r$ **where** $q: q \in \text{poly-carrier}$ **and** $r: r \in \text{poly-carrier}$

and $\text{peq: } p = (\text{root-factor } a \otimes_P q) \oplus_P r$

and $\text{rdeg: } r = 0_P \vee \text{degree } r < \text{degree } (\text{root-factor } a)$

using $\text{poly-divide}[OF \ p \ \text{rf} \ \text{rfnz}]$ **by** blast

have $\text{rconst: } r = \text{poly-const } (r \ 0)$

using $a \text{ degree-root-factor degree-zero-imp-const } r \ \text{rdeg}$ **by** fastforce

— Evaluate at a : the $\text{root-factor } a$ term vanishes, leaving the constant.

have $\text{rqc: root-factor } a \otimes_P q \in \text{poly-carrier}$ **using** $\text{rf } q$

by $(\text{rule poly-mult-closed})$

have $\text{r0: } r \ 0 \in R$

using r **by** $(\text{simp add: poly-carrier-coeff-closed})$

have $0 = 0 + r \ 0$

using $\text{Ring.eval-const Ring-axioms } a \ \text{eval-add eval-mult peq } q \ r \ \text{r0 rconst rf root}$

rqc

by fastforce

then have $\text{rz: } r = 0_P$ **using** rconst

using $\text{poly-const-zero } r$ **by** $(\text{simp add: poly-carrier-coeff-closed})$

— So p is exactly the multiple $\text{root-factor } a \otimes_P q$.

then show $\exists q. q \in \text{poly-carrier} \wedge p = \text{root-factor } a \otimes_P q$

using $\text{peq poly-add-comm poly-add-zero poly-zero-closed } q \ \text{rqc}$ **by** auto

qed

46.4 A nonzero polynomial has at most *degree* roots

Evaluation of the linear factor $\text{root-factor } a$ at an arbitrary point b is $b - a$; in particular it is nonzero unless $b = a$.

lemma $\text{eval-root-factor-apply}$:

assumes $a: a \in R$ **and** $b: b \in R$ **shows** $\text{eval } b \ (\text{root-factor } a) = b - a$

proof —

have $\text{nc: } \ominus_P \text{ poly-const } a \in \text{poly-carrier}$

using $a \text{ poly-const-closed poly-neg-closed}$ **by** simp

have $\text{eval } b \ (\text{root-factor } a) = \text{eval } b \ X_P + \text{eval } b \ (\ominus_P \text{ poly-const } a)$

unfolding root-factor-def **using** $\text{var-closed nc } b$ **by** (rule eval-add)

also have $\dots = b - a$

by $(\text{simp add: } a \ b \ \text{eval-neg poly-const-closed})$

finally show $?thesis$.

qed

The root set $\{a \in R. \text{eval } a \ p = 0\}$ of a nonzero polynomial over a field is finite and has at most *degree* p elements. By induction on the degree: a root a lets the factor theorem write $p = \text{root-factor } a \otimes_P q$ with $\text{degree } q = \text{degree } p - 1$

(*degree-mult*, *degree-root-factor*); as a field has no zero divisors, every root of p is a or a root of q .

theorem *card-roots-le-degree*:

$p \in \text{poly-carrier} \implies p \neq \mathbf{0}_P$

$\implies \text{finite } \{a \in R. \text{ eval } a \ p = \mathbf{0}\} \wedge \text{card } \{a \in R. \text{ eval } a \ p = \mathbf{0}\} \leq \text{degree } p$

proof (*induct degree p arbitrary: p rule: less-induct*)

case *less*

note $p = \langle p \in \text{poly-carrier} \rangle$ **and** $\text{pnz} = \langle p \neq \mathbf{0}_P \rangle$

show *?case*

proof (*cases $\exists a \in R. \text{ eval } a \ p = \mathbf{0}$*)

case *False*

then have $e: \{a \in R. \text{ eval } a \ p = \mathbf{0}\} = \{\}$ **by** *blast*

show *?thesis*

unfolding e **by** *simp*

next

case *True*

then obtain a **where** $aR: a \in R$ **and** $\text{aroot}: \text{eval } a \ p = \mathbf{0}$ **by** *blast*

obtain q **where** $q: q \in \text{poly-carrier}$ **and** $\text{peq}: p = \text{root-factor } a \otimes_P q$ **and** $\text{qnz}: q \neq \mathbf{0}_P$

using *factor-theorem[OF p] aroot aR pnz poly-mult-zero-right root-factor-closed*

by *metis*

have $\text{rf}: \text{root-factor } a \in \text{poly-carrier}$ **using** aR **by** (*rule root-factor-closed*)

have $\text{qnz}: q \neq \mathbf{0}_P$ **using** peq pnz **by** (*metis poly-mult-zero-right rf*)

have $\text{degp}: \text{degree } p = 1 + \text{degree } q$

by (*simp add: aR degree-mult degree-root-factor peq q qnz root-factor-closed root-factor-nonzero*)

have $\text{IH}: \text{finite } \{b \in R. \text{ eval } b \ q = \mathbf{0}\} \wedge \text{card } \{b \in R. \text{ eval } b \ q = \mathbf{0}\} \leq \text{degree } q$

using $\text{degp } q \text{ qnz}$ **by** (*intro less*) *simp*

have $\text{subset}: \{b \in R. \text{ eval } b \ p = \mathbf{0}\} \subseteq \text{insert } a \ \{b \in R. \text{ eval } b \ q = \mathbf{0}\}$

proof

fix b **assume** $b \in \{b \in R. \text{ eval } b \ p = \mathbf{0}\}$

then have $bR: b \in R$ **and** $\text{broot}: \text{eval } b \ p = \mathbf{0}$ **by** *auto*

have $\text{eval } b \ (\text{root-factor } a) \cdot \text{eval } b \ q = \mathbf{0}$

using $\text{peq eval-mult[OF rf q bR]}$ broot **by** *simp*

then have $\text{eval } b \ (\text{root-factor } a) = \mathbf{0} \vee \text{eval } b \ q = \mathbf{0}$

using *eval-closed[OF rf bR]* **by** (*simp add: bR local.no-zero-divisors q*)

then show $b \in \text{insert } a \ \{b \in R. \text{ eval } b \ q = \mathbf{0}\}$

proof

assume $\text{eval } b \ (\text{root-factor } a) = \mathbf{0}$

then have $z: b + (-a) = \mathbf{0}$ **using** *eval-root-factor-apply[OF aR bR]* **by** *simp*

then show *?thesis*

using aR *additive.inverse-unique additive.invertible-left-inverse bR* **by** *blast*

qed (*auto simp: bR*)

qed

have $\text{card } \{a \in R. \text{ eval } a \ p = \mathbf{0}\} \leq \text{card } (\text{insert } a \ \{b \in R. \text{ eval } b \ q = \mathbf{0}\})$

by (*simp add: IH card-mono subset*)

also have $\dots \leq \text{degree } p$

using IH degp **by** (*intro card-insert-le-m1*) *auto*

finally show *?thesis*

```

    using IH finite-subset subset by auto
  qed
qed

```

46.5 Low-degree irreducibility

Evaluation of a polynomial of degree ≤ 1 : $eval\ \alpha\ c = c\ 0 + c\ 1 \cdot \alpha$.

```

lemma eval-le1:
  assumes c: c ∈ poly-carrier and al: α ∈ R and d: degree c ≤ 1
  shows eval α c = c 0 + c 1 · α
proof -
  have ci: ∧i. c i ∈ R using c by (simp add: poly-carrier-coeff-closed)
  have additive.fincomp (λi. c i · rpow α i) {0, 1} = c 0 · rpow α 0 + c 1 · rpow α 1
  1
    using ci al by simp
  then show ?thesis
    unfolding eval-fincomp-le[OF c al d]
    by (simp add: al atMost-Suc ci insert-commute)
qed

```

Every polynomial of degree 1 over a field has a root: $-(c\ 0) \cdot (c\ 1)^{-1}$.

```

lemma degree1-has-root:
  assumes c: c ∈ poly-carrier and d: degree c = 1
  shows ∃α∈R. eval α c = 0
proof -
  have ci: ∧i. c i ∈ R using c by (simp add: poly-carrier-coeff-closed)
  have inv: multiplicative.invertible (c 1)
    using c ci coeff-degree-nonzero d field-inverse by force
  then have i1: multiplicative.inverse (c 1) ∈ R using ci by simp
  have nc0: - c 0 ∈ R using ci by simp
  define α where α ≡ (- c 0) · multiplicative.inverse (c 1)
  have aR: α ∈ R unfolding α-def using nc0 i1 by simp
  have c 1 · α = (c 1 · multiplicative.inverse (c 1)) · (- c 0)
    using α-def ci i1 multiplicative.associative multiplicative.commutative by auto
  then have eval α c = 0
    using aR c ci d eval-le1 inv by force
  then show ?thesis using aR by blast
qed

```

```

lemma degree-nz:
  assumes a: a ∈ poly-carrier and nua: ¬ poly-unit a and anz: a ≠ 0_P
  shows degree a ≠ 0
  using degree-zero-imp-const[OF a] coeff-degree-nonzero[OF a anz]
  by (metis a nua poly-carrier-coeff-closed poly-const-unit)

```

A degree-1 factor of p supplies a root of p : if $p = a \otimes_P b$ with a (or b) of degree 1, evaluation at the factor's root kills p .

```

lemma degree1-factor-root:

```

assumes $a: a \in \text{poly-carrier}$ **and** $b \in \text{poly-carrier}$ $p = a \otimes_P b$ $\text{degree } a = 1$
shows $\exists \alpha \in R. \text{eval } \alpha \ p = \mathbf{0}$
using $\text{degree1-has-root}[OF \ a]$ eval-mult assms **by** force

Degree-2, 3 irreducibility criterion. A cubic with no root is irreducible: a non-trivial factorisation $p = a \otimes_P b$ has $\text{degree } a + \text{degree } b = 2$ or 3 with both factors of degree ≥ 1 , so one of them has degree 1 and hence a root of p .

theorem $\text{degree23-no-root-irreducible}$:

assumes $p: p \in \text{poly-carrier}$ **and** $d: \text{degree } p \in \{2,3\}$
and $\text{noroot}: \bigwedge \alpha. \alpha \in R \implies \text{eval } \alpha \ p \neq \mathbf{0}$
shows $\text{poly-irreducible } p$

proof –

have $\text{pnu}: \neg \text{poly-unit } p$

using $\text{assms}(2)$ $\text{poly-unit-imp-const}$ **by** fastforce

have False

if $a: a \in \text{poly-carrier} \neg \text{poly-unit } a$ **and** $b: b \in \text{poly-carrier} \neg \text{poly-unit } b$
and $\text{pab}: p = a \otimes_P b$ **for** $a \ b$

proof –

have $\text{anz}: a \neq \mathbf{0}_P$ **and** $\text{bnz}: b \neq \mathbf{0}_P$

using $\text{pab } d \ a \ b \ \text{poly-mult-zero-left poly-mult-zero-right}$ **by** auto

— Degrees add to 3 ; neither factor is a unit, so neither has degree 0 .

then have $\text{dsum}: \text{degree } a + \text{degree } b \in \{2,3\}$

using $d \ \text{degree-mult } \text{pab that}$ **by** force

obtain $\text{degree } a \neq 0$ **and** $\text{degree } b \neq 0$

using $a \ \text{anz } b \ \text{bnz } \text{degree-nz}$ **by** presburger

then have $\text{degree } a = 1 \vee \text{degree } b = 1$ **using** dsum **by** auto

with $\text{noroot } a \ b \ \text{degree1-factor-root } \text{pab } \text{poly-mult-comm}$ **show** False **by** metis

qed

then show $?thesis$ **unfolding** $\text{poly-irreducible-def}$ **using** $p \ d \ \text{pnu}$ **by** auto

qed

46.6 Factorisation into irreducibles

The product of a list of polynomials (right fold, empty list giving $\mathbf{1}_P$).

definition $\text{poly-prod} :: (\text{nat} \Rightarrow 'a) \text{ list} \Rightarrow (\text{nat} \Rightarrow 'a)$

where $\text{poly-prod } fs \equiv \text{foldr } (\otimes_P) \ fs \ \mathbf{1}_P$

lemma poly-prod-Nil $[simp]$: $\text{poly-prod } [] = \mathbf{1}_P$

by $(\text{simp add: poly-prod-def})$

lemma poly-prod-Cons $[simp]$: $\text{poly-prod } (f \# fs) = f \otimes_P \text{poly-prod } fs$

by $(\text{simp add: poly-prod-def})$

lemma poly-prod-closed :

$(\bigwedge f. f \in \text{set } fs \implies f \in \text{poly-carrier}) \implies \text{poly-prod } fs \in \text{poly-carrier}$

by $(\text{induct } fs) \ (\text{auto simp: poly-one-closed poly-mult-closed})$

lemma poly-prod-append :

assumes $fs: \bigwedge f. f \in \text{set } fs \implies f \in \text{poly-carrier}$

and $gs: \bigwedge g. g \in \text{set } gs \implies g \in \text{poly-carrier}$
shows $\text{poly-prod } (fs @ gs) = \text{poly-prod } fs \otimes_P \text{poly-prod } gs$
using fs **by** $(\text{induct } fs) (\text{simp-all add: } gs \text{ poly-mult-one-left poly-mult-assoc poly-prod-closed})$

46.7 Splitting into linear factors

A polynomial *splits* when it is a constant times a product of root factors, all of whose roots lie in the field. Equivalently it is a product of linear factors: the constant carries the leading coefficient. Taking that constant to range over the whole field, rather than the nonzero elements, lets $\mathbf{0}_P$ split as well (empty list, zero constant), so no polynomial is excluded.

definition $\text{splits} :: (\text{nat} \Rightarrow 'a) \Rightarrow \text{bool}$
where $\text{splits } p \equiv (\exists c \text{ as. } c \in R \wedge \text{set as} \subseteq R$
 $\quad \wedge p = \text{poly-const } c \otimes_P \text{poly-prod } (\text{List.map root-factor as}))$

lemma splitsI :
 $\llbracket c \in R; \text{set as} \subseteq R; p = \text{poly-const } c \otimes_P \text{poly-prod } (\text{List.map root-factor as}) \rrbracket$
 $\implies \text{splits } p$
unfolding splits-def **by** blast

lemma splitsE :
assumes $\text{splits } p$
obtains $c \text{ as where } c \in R \text{ set as} \subseteq R$
and $p = \text{poly-const } c \otimes_P \text{poly-prod } (\text{List.map root-factor as})$
using assms **unfolding** splits-def **by** blast

lemma splits-closed : $\text{splits } p \implies p \in \text{poly-carrier}$
proof (elim splitsE)
fix $c \text{ as assume } c: c \in R \text{ and as: set as} \subseteq R$
and $\text{peq: } p = \text{poly-const } c \otimes_P \text{poly-prod } (\text{List.map root-factor as})$
have $\text{poly-prod } (\text{List.map root-factor as}) \in \text{poly-carrier}$
using $\text{as by (intro poly-prod-closed) (auto intro: root-factor-closed)}$
then show $p \in \text{poly-carrier}$ **using** $\text{peq poly-const-closed [OF c] by (simp add: poly-mult-closed)}$
qed

lemma splits-const :
assumes $c: c \in R$ **shows** $\text{splits } (\text{poly-const } c)$
proof (rule splitsI)
show $\text{poly-const } c = \text{poly-const } c \otimes_P \text{poly-prod } (\text{List.map root-factor []})$
using $\text{poly-mult-one-right [OF poly-const-closed [OF c]] by simp}$
qed (use c in auto)

lemma $\text{splits-zero [simp]}$: $\text{splits } \mathbf{0}_P$
using $\text{splits-const [of 0] by simp}$

lemma $\text{splits-root-factor}$:
assumes $a: a \in R$ **shows** $\text{splits } (\text{root-factor } a)$
proof (rule splitsI)

```

show root-factor  $a = \text{poly-const } 1 \otimes_P \text{poly-prod } (\text{List.map root-factor } [a])$ 
by (simp add: assms poly-mult-one-left poly-mult-one-right root-factor-closed)
qed (use a in auto)

```

Splitting is multiplicative: concatenate the root lists and multiply the constants.

lemma *splits-mult*:

```

assumes  $p$ : splits  $p$  and  $q$ : splits  $q$  shows splits  $(p \otimes_P q)$ 
proof –
obtain  $c$   $as$  where  $c: c \in R$  and  $as$ : set  $as \subseteq R$ 
and  $peq$ :  $p = \text{poly-const } c \otimes_P \text{poly-prod } (\text{List.map root-factor } as)$ 
using  $p$  by (rule splitsE)
obtain  $d$   $bs$  where  $d: d \in R$  and  $bs$ : set  $bs \subseteq R$ 
and  $qeq$ :  $q = \text{poly-const } d \otimes_P \text{poly-prod } (\text{List.map root-factor } bs)$ 
using  $q$  by (rule splitsE)
have  $pab$ :  $\text{poly-prod } (\text{List.map root-factor } as) \in \text{poly-carrier}$ 
poly-prod  $(\text{List.map root-factor } bs) \in \text{poly-carrier}$ 
using  $as$   $bs$  by (auto intro!: poly-prod-closed root-factor-closed)
have  $prod\text{-app}$ :  $\text{poly-prod } (\text{List.map root-factor } (as @ bs))$ 
=  $\text{poly-prod } (\text{List.map root-factor } as) \otimes_P \text{poly-prod } (\text{List.map root-factor } bs)$ 
bs)
unfolding List.map-append
using  $as$   $bs$  by (force intro: intro: poly-prod-append root-factor-closed)
show ?thesis
proof (rule splitsI)
show  $c \cdot d \in R$  using  $c$   $d$  by simp
show set  $(as @ bs) \subseteq R$  using  $as$   $bs$  by simp
show  $p \otimes_P q = \text{poly-const } (c \cdot d) \otimes_P \text{poly-prod } (\text{List.map root-factor } (as @ bs))$ 
proof –
have  $cc$ :  $\text{poly-const } c \in \text{poly-carrier}$  using  $c$  by (rule poly-const-closed)
have  $dc$ :  $\text{poly-const } d \in \text{poly-carrier}$  using  $d$  by (rule poly-const-closed)
have  $p \otimes_P q$ 
=  $(\text{poly-const } c \otimes_P \text{poly-prod } (\text{List.map root-factor } as))$ 
 $\otimes_P$   $(\text{poly-const } d \otimes_P \text{poly-prod } (\text{List.map root-factor } bs))$ 
using  $peq$   $qeq$  by simp
also have  $\dots = (\text{poly-const } c \otimes_P \text{poly-const } d)$ 
 $\otimes_P$   $(\text{poly-prod } (\text{List.map root-factor } as) \otimes_P \text{poly-prod } (\text{List.map root-factor } bs))$ 
using  $cc$  commutative-ring.mult-ac(3)  $dc$   $pab$  poly-commutative-ring poly-mult-assoc
poly-mult-closed by fastforce
also have  $\dots = \text{poly-const } (c \cdot d) \otimes_P \text{poly-prod } (\text{List.map root-factor } (as @ bs))$ 
using poly-const-mult [OF  $c$   $d$ ]  $prod\text{-app}$  by simp
finally show ?thesis .
qed
qed
qed

```

A polynomial of degree one splits. Divide by the leading coefficient: the result is root-factor a for a the negated ratio of the two coefficients. This is the base case that makes a factorisation into linear factors out of a factorisation into irreducibles all of whose factors have degree one.

```

lemma degree-one-splits:
  assumes  $q: q \in \text{poly-carrier}$  and  $d: \text{degree } q = 1$ 
  shows  $\text{splits } q$ 
proof -
  have  $qnz: q \neq 0_P$  using  $d$  by auto
  have  $c: q \ 1 \in R$  and  $q0: q \ 0 \in R$  using  $q$  by (blast intro: poly-carrier-coeff-closed)+
  have  $cnz: q \ 1 \neq 0$ 
    using lead-coeff-nonzero [OF  $q \ nz$ ]  $d$  by (simp add: lead-coeff-def)
  have  $inv: \text{multiplicative.invertible } (q \ 1)$  by (rule field-inverse [OF  $c \ cnz$ ])
  define  $e$  where  $e \equiv \text{multiplicative.inverse } (q \ 1)$ 
  define  $a$  where  $a \equiv -(e \cdot q \ 0)$ 
  obtain  $eR: e \in R$  and  $ce: q \ 1 \cdot e = 1$  using  $c \ e\text{-def } inv$  by force
  obtain  $eq0: e \cdot q \ 0 \in R$  and  $aR: a \in R$  and  $nega: -a = e \cdot q \ 0$ 
    using  $eR \ q0 \ a\text{-def}$  by simp
  have  $rf: \text{root-factor } a \in \text{poly-carrier}$  using  $aR$  by (rule root-factor-closed)
  — Multiplying by a constant scales each coefficient.
  have  $scale: (\text{poly-const } (q \ 1) \otimes_P \text{root-factor } a) \ j = q \ 1 \cdot \text{root-factor } a \ j$  for  $j$ 
    using coeff-monom-mult [OF  $c \ rf, of \ 0 \ j$ ] by (simp add: monom-0-eq-const)
  — The three coefficients of  $\text{root-factor } a$ .
  have  $rf0: \text{root-factor } a \ 0 = -a$ 
    by (simp add: eq0 monom-def nega poly-add-def poly-const-def poly-neg-def root-factor-def
    var-def)
  have  $rf1: \text{root-factor } a \ 1 = 1$ 
    by (simp add: root-factor-def var-def monom-def poly-const-def poly-add-def poly-neg-def)
  have  $rfj: \text{root-factor } a \ j = 0$  if  $j \neq 0 \ j \neq 1$  for  $j$ 
    using  $aR \ \text{coeff-gt-degree } \text{degree-root-factor } rf$  that by auto
  have  $qeq: q = \text{poly-const } (q \ 1) \otimes_P \text{root-factor } a$ 
  proof (rule ext)
    fix  $j$ 
    show  $q \ j = (\text{poly-const } (q \ 1) \otimes_P \text{root-factor } a) \ j$ 
  proof (cases  $j = 0$ )
    case True
      then show ?thesis using True scale rf0
        using  $c \ ce \ eR \ \text{multiplicative.associative } \text{multiplicative.left-unit } nega \ q0$  by force
    next
      case False
      then show ?thesis
        proof (cases  $j = 1$ )
          case True
            then show ?thesis using scale rf1  $c$  by simp
          next
            case False
            then show ?thesis
              using  $\langle j \neq 0 \rangle \ d \ scale \ rfj \ \text{coeff-gt-degree } [OF \ q] \ c$  by simp
        qed
      qed
    qed
  show ?thesis
    using  $aR \ c \ qeq \ \text{splits-const } \text{splits-mult } \text{splits-root-factor}$  by metis
  qed

```

lemma *splits-poly-prod*:
assumes $\bigwedge f. f \in \text{set } fs \implies \text{splits } f$ **shows** $\text{splits } (\text{poly-prod } fs)$
using *assms*
by (*induct* *fs*; *use splits-const splits-mult in force*)

Existence of an irreducible factorisation. Every nonzero non-unit polynomial over a field is a product of a nonempty list of irreducibles. By strong induction on the degree: an irreducible p is its own singleton factorisation; otherwise $p = a \otimes_P b$ with both factors non-units — hence, over a field, of strictly smaller degree — and the induction hypothesis factors each.

theorem *factorization-into-irreducibles*:

assumes $p \in \text{poly-carrier}$ **and** $p \neq \mathbf{0}_P$ **and** $\neg \text{poly-unit } p$
shows $\exists fs. fs \neq [] \wedge (\forall f \in \text{set } fs. \text{poly-irreducible } f) \wedge p = \text{poly-prod } fs$
using *assms*

proof (*induct degree p arbitrary: p rule: less-induct*)

case (*less p*)

show *?case*

proof (*cases poly-irreducible p*)

case *True*

— p is its own factorisation.

have $p = \text{poly-prod } [p]$ **using** *less.prem* **by** (*simp add: poly-mult-one-right*)

moreover have $[p] \neq [] \wedge (\forall f \in \text{set } [p]. \text{poly-irreducible } f)$ **using** *True* **by** *simp*

ultimately show *?thesis* **by** *blast*

next

case *False*

— A nontrivial factorisation.

from *less.prem* *False* **obtain** $a \ b$ **where** $a: a \in \text{poly-carrier}$ **and** $b: b \in \text{poly-carrier}$

and $\text{pab}: p = a \otimes_P b$ **and** $\text{nua}: \neg \text{poly-unit } a$ **and** $\text{nub}: \neg \text{poly-unit } b$

unfolding *poly-irreducible-def* **by** *blast*

have $\text{anz}: a \neq \mathbf{0}_P$ **and** $\text{bnz}: b \neq \mathbf{0}_P$

using *pab less.prem(2)* $a \ b$ *poly-mult-zero-left poly-mult-zero-right* **by** *auto*

have $\text{dsum}: \text{degree } a + \text{degree } b = \text{degree } p$

using *degree-mult[OF a b anz bnz]* *pab* **by** *simp*

— Neither factor has degree 0, so each has degree strictly below *degree p*.

obtain $\text{da}: \text{degree } a < \text{degree } p$ **and** $\text{db}: \text{degree } b < \text{degree } p$

using $a \ \text{anz} \ b \ \text{bnz} \ \text{degree-nz} \ \text{nua} \ \text{nub} \ \text{dsum}$ **by** *fastforce*

then have $\text{da}: \text{degree } a < \text{degree } p$ **and** $\text{db}: \text{degree } b < \text{degree } p$ **using** *dsum* **by** *auto*

— Factor each smaller factor by the induction hypothesis, then concatenate.

obtain fa **where** $\text{fa}: \text{fa} \neq [] \wedge \forall f \in \text{set } \text{fa}. \text{poly-irreducible } f$ $a = \text{poly-prod } \text{fa}$

using *less.hyps[OF da a anz nua]* **by** *blast*

obtain fb **where** $\text{fb}: \text{fb} \neq [] \wedge \forall f \in \text{set } \text{fb}. \text{poly-irreducible } f$ $b = \text{poly-prod } \text{fb}$

using *less.hyps[OF db b bnz nub]* **by** *blast*

have $p = \text{poly-prod } (\text{fa} @ \text{fb})$

proof —

have $\text{poly-prod } (\text{fa} @ \text{fb}) = \text{poly-prod } \text{fa} \otimes_P \text{poly-prod } \text{fb}$

using *fa(2) fb(2)*

```

proof (induct fa)
  case Nil
  then show ?case
    using b fb(3) poly-mult-one-left by auto
next
  case (Cons g gs)
  then obtain gc:  $g \in \text{poly-carrier}$  and gsc:  $\text{poly-prod } gs \in \text{poly-carrier}$ 
    by (simp add: Cons.prem(1) poly-irreducibleD-carrier poly-prod-closed)
  then show ?case
    using b fb(3) by (simp add: Cons.hyps Cons.prem gc gsc poly-mult-assoc)
qed
then show ?thesis using pab fa(3) fb(3) by simp
qed
moreover have  $\forall f \in \text{set } (fa @ fb). \text{poly-irreducible } f$  using fa(2) fb(2) by auto
ultimately show ?thesis
  using fa(1) by blast
qed
qed

```

46.8 Monic polynomials and normalisation

A polynomial is *monic* when it is nonzero with leading coefficient **1**.

definition *poly-monic* :: $(\text{nat} \Rightarrow 'a) \Rightarrow \text{bool}$
where *poly-monic* $p \equiv p \neq \mathbf{0}_P \wedge \text{lead-coeff } p = \mathbf{1}$

Scaling a nonzero polynomial by a nonzero constant preserves its degree and scales its leading coefficient.

lemma *degree-poly-const-mult*:
assumes $c: c \in R$ **and** $\text{cnz}: c \neq \mathbf{0}$ **and** $p: p \in \text{poly-carrier}$ **and** $\text{pnz}: p \neq \mathbf{0}_P$
shows $\text{degree } (\text{poly-const } c \otimes_P p) = \text{degree } p$
by (simp add: c cnz degree-mult p pnz poly-const-closed poly-const-unit poly-unit-imp-const)

lemma *lead-coeff-poly-const-mult*:
assumes $c: c \in R$ **and** $\text{cnz}: c \neq \mathbf{0}$ **and** $p: p \in \text{poly-carrier}$ **and** $\text{pnz}: p \neq \mathbf{0}_P$
shows $\text{lead-coeff } (\text{poly-const } c \otimes_P p) = c \cdot \text{lead-coeff } p$
proof –
have $\text{cc}: \text{poly-const } c \in \text{poly-carrier}$ **using** c **by** (rule poly-const-closed)
have $\text{dc0}: \text{degree } (\text{poly-const } c) = 0$ **using** degree-const-le **by** simp
then have $\text{lead-coeff } (\text{poly-const } c \otimes_P p) = \text{poly-const } c (\text{degree } (\text{poly-const } c)) \cdot p$
 (degree p)
using coeff-mult-degree-add c cc cnz degree-poly-const-mult lead-coeff-def p pnz **by**
 fastforce
then show ?thesis
using dc0 lead-coeff-def poly-const-def **by** force
qed

Every nonzero polynomial has a monic associate, obtained by scaling by the inverse of its leading coefficient.

theorem *monic-associate-exists*:


```

assumes  $p: p \in \text{poly-carrier}$  and  $\text{pnz}: p \neq \mathbf{0}_P$ 
shows  $\exists u \ m. u \in R \wedge u \neq \mathbf{0} \wedge m \in \text{poly-carrier} \wedge \text{poly-monic } m \wedge p = \text{poly-const}$ 
 $u \otimes_P m$ 
proof —
  define  $c$  where  $c \equiv \text{multiplicative.inverse } (\text{lead-coeff } p)$ 
  have  $\text{lc}: \text{lead-coeff } p \in R$ 
    using  $p$  by ( $\text{simp add: lead-coeff-def poly-carrier-coeff-closed}$ )
  have  $\text{inv}: \text{multiplicative.invertible } (\text{lead-coeff } p)$ 
    by ( $\text{simp add: field-inverse lc lead-coeff-nonzero } p \ \text{pnz}$ )
  have  $cR: c \in R$  unfolding  $c\text{-def}$  using  $\text{lc inv}$  by  $\text{simp}$ 
  have  $\text{cnz}: c \neq \mathbf{0}$ 
    using  $c\text{-def inv lc multiplicative.invertible-inverse-invertible nontrivial}$ 
    by  $\text{fastforce}$ 
  define  $m$  where  $m \equiv \text{poly-const } c \otimes_P p$ 
  have  $\text{mc}: m \in \text{poly-carrier}$  unfolding  $m\text{-def}$ 
    by ( $\text{simp add: } cR \ p \ \text{poly-const-closed poly-mult-closed}$ )
  —  $m$  is monic: its leading coefficient is  $c \cdot \text{lead-coeff } p = \mathbf{1}$ .
  have  $\text{cnzp}: \text{poly-const } c \neq \mathbf{0}_P$  using  $\text{cnz}$  by ( $\text{auto simp: poly-const-def poly-zero-def}$ 
 $\text{fun-eq-iff}$ )
  have  $\text{mnz}: m \neq \mathbf{0}_P$  unfolding  $m\text{-def}$ 
    using  $\text{poly-no-zero-divisors}[OF - p] \ \text{cnzp pnz } cR \ \text{poly-const-closed}$  by  $\text{force}$ 
  have  $\text{eq1}: \text{multiplicative.inverse } (\text{lead-coeff } p) \cdot \text{lead-coeff } p = \mathbf{1}$ 
    using  $\text{inv lc}$  by  $\text{simp}$ 
  have  $\text{monic-}m: \text{poly-monic } m$ 
    using  $cR \ c\text{-def cnz eq1 lead-coeff-poly-const-mult } m\text{-def mnz } p \ \text{pnz poly-monic-def}$ 
by  $\text{metis}$ 
  — Recover  $p$  by scaling  $m$  by  $\text{lead-coeff } p$ .
  have  $\text{poly-const } (\text{lead-coeff } p) \otimes_P m = \text{poly-const } \mathbf{1} \otimes_P p$ 
    using  $cR \ c\text{-def eq1 lc } m\text{-def } p \ \text{poly-const-closed poly-const-mult poly-mult-assoc}$ 
 $\text{poly-mult-comm}$ 
    by  $\text{metis}$ 
  also have  $\dots = p$ 
    using  $p$  by ( $\text{simp add: poly-mult-one-left}$ )
  finally have  $p = \text{poly-const } (\text{lead-coeff } p) \otimes_P m$  ..
  then show  $?thesis$  using  $\text{lc lead-coeff-nonzero mc monic-}m \ p \ \text{pnz}$  by  $\text{blast}$ 
qed

```

46.9 The polynomial ring over a field is a principal ideal domain

Every ideal of $F[X]$ is principal. Given an ideal J , the zero ideal is generated by $\mathbf{0}_P$; otherwise pick a nonzero b of *least degree* in J . Dividing any $a \in J$ by b (the division algorithm *poly-divide*) leaves a remainder that again lies in J but has degree below that of b , so by minimality it must be zero: hence a is a multiple of b . This is the structural fact behind Kronecker's construction — it makes (p) maximal for irreducible p , and so $F[X]/(p)$ a field.

theorem *poly-PID*:

assumes J : Ideal J *poly-carrier* $(\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$

```

shows  $\exists b. b \in \text{poly-carrier} \wedge J = \{r \otimes_P b \mid r. r \in \text{poly-carrier}\}$ 
proof -
interpret J: Ideal J poly-carrier  $(\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$  by (rule J)
show ?thesis
proof (cases  $J = \{\mathbf{0}_P\}$ )
  case True
  then show ?thesis using poly-zero-closed by force
next
  case False
  — Otherwise J contains a nonzero element, hence one of least degree.
  define hasdeg where  $\text{hasdeg } n \longleftrightarrow (\exists a. a \in J \wedge a \neq \mathbf{0}_P \wedge \text{degree } a = n)$  for n
  have  $\exists n. \text{hasdeg } n$ 
  using False unfolding hasdeg-def by blast
  then obtain b where bJ:  $b \in J$  and bnz:  $b \neq \mathbf{0}_P$ 
  and bLeast:  $\text{degree } b = (\text{LEAST } n. \text{hasdeg } n)$ 
  using hasdeg-def by (metis LeastI-ex)
  have bP:  $b \in \text{poly-carrier}$  using bJ J.additive.subset by blast
  — b has least degree among the nonzero elements of J.
  have minimal:  $\text{degree } b \leq \text{degree } c$  if  $c \in J$   $c \neq \mathbf{0}_P$  for c
  by (metis Least-le bLeast hasdeg-def that)
  — J is exactly the principal ideal generated by b.
  have  $J = \{r \otimes_P b \mid r. r \in \text{poly-carrier}\}$ 
  proof (intro antisym subsetI)
    fix a assume aJ:  $a \in J$ 
    have aP:  $a \in \text{poly-carrier}$  using aJ J.additive.subset by blast
    obtain q rem where q:  $q \in \text{poly-carrier}$  and rem:  $\text{rem} \in \text{poly-carrier}$ 
    and aeq:  $a = (b \otimes_P q) \oplus_P \text{rem}$ 
    and remd:  $\text{rem} = \mathbf{0}_P \vee \text{degree } \text{rem} < \text{degree } b$ 
    using poly-divide[OF aP bP bnz] by blast
    have bqc:  $b \otimes_P q \in \text{poly-carrier}$  using bP q by (rule poly-mult-closed)
    — The multiple  $b \otimes_P q$  and its negative both lie in J.
    have J.additive.inverse  $(b \otimes_P q) \in J$ 
    by (simp add: J.Ideal(2) bJ q)
    then have negbqJ:  $\ominus_P (b \otimes_P q) \in J$ 
    using J.additive.inverse-equality bqc poly-add-neg poly-add-neg-right poly-neg-closed

    by metis
    have rem-eq:  $\text{rem} = (\ominus_P (b \otimes_P q)) \oplus_P a$ 
    by (metis aeq bqc poly-add-comm poly-add-minus-cancel poly-neg-closed poly-neg-neg
rem)
    then have remJ:  $\text{rem} \in J$ 
    using negbqJ aJ by blast
    have rem =  $\mathbf{0}_P$ 
    using leD minimal remJ remd by auto
    then show  $a \in \{r \otimes_P b \mid r. r \in \text{poly-carrier}\}$ 
    using q aeq bP poly-mult-comm by auto
  qed (auto simp: J.Ideal bJ)
  then show ?thesis using bP by blast
qed
qed

```

end

47 Pushing a Polynomial along a Ring Homomorphism

Coefficientwise application of a ring homomorphism commutes with every operation of the polynomial ring..

context

```

fixes A :: 'b set and add mult :: 'b  $\Rightarrow$  'b  $\Rightarrow$  'b and z u :: 'b
and B :: 'c set and badd bmult :: 'c  $\Rightarrow$  'c  $\Rightarrow$  'c and bz bu :: 'c
and h :: 'b  $\Rightarrow$  'c
assumes RA: Ring A add mult z u and RB: Ring B badd bmult bz bu
and hB:  $\bigwedge y. y \in A \implies h y \in B$ 
and hz:  $h z = bz$  and hu:  $h u = bu$ 
and hadd:  $\bigwedge y y'. \llbracket y \in A; y' \in A \rrbracket \implies h (add y y') = badd (h y) (h y')$ 
and hmult:  $\bigwedge y y'. \llbracket y \in A; y' \in A \rrbracket \implies h (mult y y') = bmult (h y) (h y')$ 

```

begin

```

lemma cmA: commutative-monoid A add z
using RA by (simp add: Ring-def Abelian-Group-def)

```

```

lemma cmB: commutative-monoid B badd bz
using RB by (simp add: Ring-def Abelian-Group-def)

```

```

lemma gpA: Abelian-Group A add z
using RA by (simp add: Ring-def)

```

```

lemma gpB: Abelian-Group B badd bz
using RB by (simp add: Ring-def)

```

```

lemma monA: Monoid A mult u
using RA by (simp add: Ring-def)

```

```

lemma poly-zero-hom:  $(\lambda k. h (Ring.poly-zero z k)) = Ring.poly-zero bz$ 
by (simp add: Ring.poly-zero-def [OF RA] Ring.poly-zero-def [OF RB] hz)

```

```

lemma poly-one-hom:  $(\lambda k. h (Ring.poly-one z u k)) = Ring.poly-one bz bu$ 
by (auto simp add: Ring.poly-one-def [OF RA] Ring.poly-one-def [OF RB] hz hu)

```

```

lemma poly-const-hom:  $(\lambda k. h (Ring.poly-const z c k)) = Ring.poly-const bz (h c)$ 
by (auto simp: Ring.poly-const-def [OF RA] Ring.poly-const-def [OF RB] hz)

```

```

lemma var-hom:  $(\lambda k. h (Ring.var z u k)) = Ring.var bz bu$ 
using RA RB Ring.coeff-var[of A add mult z u] Ring.coeff-var[of B badd bmult bz
bu] hu hz
by auto

```

Coefficients of the polynomials named above, needed as closure side conditions below.

lemma *zA*: $z \in A$
using *cmA* **by** (*metis Monoid-def commutative-monoid-def*)

lemma *uA*: $u \in A$
using *monA* **by** (*metis Monoid-def*)

lemma *poly-const-coeff-closed*: $c \in A \implies \text{Ring.poly-const } z \ c \ k \in A$
using *zA* **by** (*simp add: Ring.poly-const-def [OF RA]*)

lemma *var-coeff-closed*: $\text{Ring.var } z \ u \ k \in A$
using *zA uA* **by** (*simp add: Ring.var-def [OF RA] Ring.monom-def [OF RA]*)

lemma *poly-add-hom*:
assumes $p: \bigwedge k. p \ k \in A$ **and** $q: \bigwedge k. q \ k \in A$
shows $(\lambda k. h \ (\text{Ring.poly-add } \text{add } p \ q \ k)) = \text{Ring.poly-add } \text{badd } (\lambda k. h \ (p \ k)) \ (\lambda k. h \ (q \ k))$
by (*simp add: Ring.poly-add-def [OF RA] Ring.poly-add-def [OF RB] hadd p q*)

The additive inverse is pinned by the equation it solves, so a homomorphism of the additive groups carries it along.

lemma *inverse-hom*:
assumes $y \in A$
shows $h \ (\text{Monoid.inverse } A \ \text{add } z \ y) = \text{Monoid.inverse } B \ \text{badd } bz \ (h \ y)$
proof –
interpret *GA*: *Abelian-Group* $A \ \text{add } z$ **by** (*rule gpA*)
interpret *GB*: *Abelian-Group* $B \ \text{badd } bz$ **by** (*rule gpB*)
have *inv*: $\text{Monoid.inverse } A \ \text{add } z \ y \in A$ **using** *assms* **by** *simp*
have *1*: $\text{badd } (h \ y) \ (h \ (\text{Monoid.inverse } A \ \text{add } z \ y)) = bz$
using *GA.invertible GA.invertible-right-inverse hadd hz inv assms* **by** *metis*
then have *2*: $\text{badd } (h \ (\text{Monoid.inverse } A \ \text{add } z \ y)) \ (h \ y) = bz$
by (*simp add: GB.commutative hB assms*)
show *?thesis*
using *1 2 GB.inverse-equality hB inv assms* **by** *presburger*
qed

lemma *poly-neg-hom*:
assumes $p: \bigwedge k. p \ k \in A$
shows $(\lambda k. h \ (\text{Ring.poly-neg } A \ \text{add } z \ p \ k)) = \text{Ring.poly-neg } B \ \text{badd } bz \ (\lambda k. h \ (p \ k))$
by (*simp add: Ring.poly-neg-def [OF RA] Ring.poly-neg-def [OF RB] inverse-hom p*)

lemma *poly-neg-coeff-closed*:
assumes $p: \bigwedge k. p \ k \in A$
shows $\text{Ring.poly-neg } A \ \text{add } z \ p \ k \in A$
proof –
interpret *GA*: *Abelian-Group* $A \ \text{add } z$ **by** (*rule gpA*)
show *?thesis* **using** *p* **by** (*simp add: Ring.poly-neg-def [OF RA]*)

qed

Convolution is a finite composite of products, so it pushes along by *fincomp-hom*. Only the two monoid premises are supplied: resolving the homomorphism premise inside *OF* sends unification into higher-order search, as the corresponding step of *eval-hom* also avoids.

lemma *poly-mult-hom*:

assumes $p: \bigwedge k. p\ k \in A$ **and** $q: \bigwedge k. q\ k \in A$
shows $(\lambda k. h\ (Ring.poly-mult\ A\ add\ mult\ z\ p\ q\ k))$
 $= Ring.poly-mult\ B\ badd\ bmult\ bz\ (\lambda k. h\ (p\ k))\ (\lambda k. h\ (q\ k))$

proof

fix k
have $cl: (\lambda i. mult\ (p\ i)\ (q\ (k - i))) \in \{..k\} \rightarrow A$
using *Monoid.composition-closed monA* $p\ q$ **by** *fastforce*
have $h\ (Ring.poly-mult\ A\ add\ mult\ z\ p\ q\ k)$
 $= commutative-monoid.fincomp\ B\ badd\ bz\ (\lambda i. h\ (mult\ (p\ i)\ (q\ (k - i))))\ \{..k\}$
by (*simp add: Ring.poly-mult-def [OF RA] fincomp-hom [OF cmA cmB] hB hz*
hadd cl)
also have $\dots = Ring.poly-mult\ B\ badd\ bmult\ bz\ (\lambda k. h\ (p\ k))\ (\lambda k. h\ (q\ k))\ k$
by (*simp add: hmult p q Ring.poly-mult-def [OF RB]*)
finally show $h\ (Ring.poly-mult\ A\ add\ mult\ z\ p\ q\ k)$
 $= Ring.poly-mult\ B\ badd\ bmult\ bz\ (\lambda k. h\ (p\ k))\ (\lambda k. h\ (q\ k))\ k$.

qed

lemma *poly-mult-coeff-closed*:

assumes $p: \bigwedge k. p\ k \in A$ **and** $q: \bigwedge k. q\ k \in A$
shows $Ring.poly-mult\ A\ add\ mult\ z\ p\ q\ k \in A$

proof –

have $(\lambda i. mult\ (p\ i)\ (q\ (k - i))) \in \{..k\} \rightarrow A$
using *Monoid.composition-closed monA* $p\ q$ **by** *fastforce*
then show *?thesis*
by (*simp add: Ring.poly-mult-def [OF RA] commutative-monoid.fincomp-closed [OF cmA]*)

qed

lemma *poly-one-coeff-closed*: $Ring.poly-one\ z\ u\ k \in A$

using $zA\ uA$ **by** (*simp add: Ring.poly-one-def [OF RA]*)

The linear factor $X - a$, the list product, and hence the property of splitting. The stronger structure is taken as an ordinary assumption of each lemma, so that the ring-level facts above keep their generality.

lemma *root-factor-coeff-closed*:

assumes *commutative-ring* $A\ add\ mult\ z\ u\ a \in A$
shows *commutative-ring.root-factor* $A\ add\ z\ u\ a\ k \in A$
by (*meson assms RA Ring.poly-carrier-coeff-closed commutative-ring.root-factor-closed*)

lemma *root-factor-hom*:

assumes $CA: commutative-ring\ A\ add\ mult\ z\ u$ **and** $CB: commutative-ring\ B\ badd\ bmult\ bz\ bu$

```

    and a: a ∈ A
  shows (λk. h (commutative-ring.root-factor A add z u a k)) = commutative-ring.root-factor
    B badd bz bu (h a)
proof -
  have cn: ∧k. Ring.poly-const z a k ∈ A
    using a by (rule poly-const-coeff-closed)
  have (λk. h (commutative-ring.root-factor A add z u a k))
    = Ring.poly-add badd (λk. h (Ring.var z u k))
      (λk. h (Ring.poly-neg A add z (Ring.poly-const z a k)))
    using CA cn poly-add-hom poly-neg-coeff-closed var-coeff-closed
    by (metis commutative-ring.root-factor-def)
  also have ... = commutative-ring.root-factor B badd bz bu (h a)
    by (simp add: var-hom poly-neg-hom [OF cn] poly-const-hom commutative-ring.root-factor-def
      [OF CB])
  finally show ?thesis .
qed

```

```

lemma poly-prod-coeff-closed:
  assumes FA: Field A add mult z u
    and fs: ∧f. f ∈ set fs ⟹ (∀k. f k ∈ A)
  shows Field.poly-prod A add mult z u fs k ∈ A
  using fs
proof (induct fs arbitrary: k)
  case Nil
  show ?case
    using poly-one-coeff-closed by (simp add: Field.poly-prod-Nil [OF FA])
next
  case (Cons g gs)
  then show ?case
    by (simp add: FA Field.poly-prod-Cons poly-mult-coeff-closed)
qed

```

```

lemma poly-prod-hom:
  assumes FA: Field A add mult z u and FB: Field B badd bmult bz bu
    and fs: ∧f. f ∈ set fs ⟹ (∀k. f k ∈ A)
  shows (λk. h (Field.poly-prod A add mult z u fs k))
    = Field.poly-prod B badd bmult bz bu (List.map (λf k. h (f k)) fs)
  using fs
proof (induct fs)
  case Nil
  show ?case
    by (simp add: Field.poly-prod-Nil [OF FA] Field.poly-prod-Nil [OF FB] poly-one-hom)
next
  case (Cons g gs)
  then have g: ∧k. g k ∈ A and gs: ∧f. f ∈ set gs ⟹ (∀k. f k ∈ A)
    by auto
  have pp: ∧k. Field.poly-prod A add mult z u gs k ∈ A
    using FA gs poly-prod-coeff-closed by blast
  have (λk. h (Field.poly-prod A add mult z u (g # gs) k))
    = Ring.poly-mult B badd bmult bz (λk. h (g k)) (λk. h (Field.poly-prod A add

```

```

mult z u gs k))
  by (simp add: Field.poly-prod-Cons [OF FA] poly-mult-hom [OF g pp])
  also have ... = Field.poly-prod B badd bmult bz bu (List.map (λf k. h (f k)) (g #
gs))
  by (simp add: Field.poly-prod-Cons [OF FB] Cons.hyps [OF gs])
  finally show ?case .
qed

```

Splitting is preserved: the witnessing constant and root list are simply carried across. The steps use *simp only* throughout, to keep *map-map* from rewriting the two lists into composed form and so out of reach of the transport equations.

theorem *splits-hom*:

```

assumes FA: Field A add mult z u and FB: Field B badd bmult bz bu
  and sp: Field.splits A add mult z u p
shows Field.splits B badd bmult bz bu (λk. h (p k))
proof -
  have CA: commutative-ring A add mult z u using FA by (simp add: Field-def)
  have CB: commutative-ring B badd bmult bz bu using FB by (simp add: Field-def)
  interpret B: commutative-ring B badd bmult bz bu
  using CB by auto
  from sp obtain c as where c: c ∈ A and as: set as ⊆ A
  and peq: p = Ring.poly-mult A add mult z
    (Ring.poly-const z c)
    (Field.poly-prod A add mult z u
      (List.map (commutative-ring.root-factor A add z u) as))
  using Field.splits-def [OF FA] by blast
  have rf: ⋀f. f ∈ set (List.map (commutative-ring.root-factor A add z u) as) ⇒
(∀k. f k ∈ A)
  using as by (auto simp: root-factor-coeff-closed [OF CA])
  then have pp: ⋀k. Field.poly-prod A add mult z u (List.map (commutative-ring.root-factor
A add z u) as) k ∈ A
  using FA poly-prod-coeff-closed by blast
  have mm: List.map (λf k. h (f k)) (List.map (commutative-ring.root-factor A add z
u) as)
    = List.map (commutative-ring.root-factor B badd bz bu) (List.map h as)
  using as by (induct as) (simp-all add: root-factor-hom [OF CA CB])
  have (λk. h (p k))
    = Ring.poly-mult B badd bmult bz (Ring.poly-const bz (h c))
      (Field.poly-prod B badd bmult bz bu
        (List.map (λf k. h (f k))
          (List.map (commutative-ring.root-factor A add z u) as)))
  using FA FB c poly-const-coeff-closed peq poly-const-hom poly-mult-hom poly-prod-hom
pp rf
  by presburger
  moreover have set (List.map h as) ⊆ B using as hB by auto
  ultimately show ?thesis
  using Field.splits-def [OF FB] mm c hB by metis
qed

```

end

end

48 Prime and Maximal Ideals

```
theory Ideal-Theory
  imports Ring-Theory HOL.Zorn
begin
```

Phase B: prime and maximal ideals, and the quotient characterisations.

```
context Ideal
begin
```

Prime and maximal ideals: proper ideals with the usual closure / maximality property.

```
definition prime-ideal :: bool
  where prime-ideal  $\longleftrightarrow I \neq R \wedge (\forall a \in R. \forall b \in R. a \cdot b \in I \longrightarrow a \in I \vee b \in I)$ 
```

```
definition maximal-ideal :: bool
  where maximal-ideal  $\longleftrightarrow I \neq R \wedge (\forall J. \text{Ideal } J \wedge R \cap J \subseteq I \longrightarrow I \subseteq J \longrightarrow J = I \vee J = R)$ 
```

The coset of an element and the set of all cosets (the quotient R/I), given clean names so they can be referred to from interpretations at the top level (the underlying *additive.Class/additive.Partition* live behind several sublocale hops and have no stable qualified name).

```
definition coset :: 'a  $\Rightarrow$  'a set
  where coset a = additive.Class a
```

```
definition quotient-set :: 'a set set
  where quotient-set = additive.Partition
```

```
lemma coset-eq-iff:
  assumes a  $\in R$  b  $\in R$  shows coset a = coset b  $\longleftrightarrow a + (- b) \in I$ 
proof -
  have coset a = coset b  $\longleftrightarrow (a, b) \in \text{Ring-Congruence}$ 
    unfolding coset-def
    using assms by (simp add: additive.Class-equivalence additive-congruence)
  also have ...  $\longleftrightarrow a + (- b) \in I$  using assms by (auto simp: Ring-Congruence-def)
  finally show ?thesis .
qed
```

```
lemma coset-in-quotient-set: a  $\in R \implies$  coset a  $\in$  quotient-set
  unfolding coset-def quotient-set-def by (rule additive.Class-in-Partition)
```

```
lemma quotient-set-repr: C  $\in$  quotient-set  $\implies \exists a \in R. C = \text{coset } a$ 
  unfolding coset-def quotient-set-def using additive.representant-exists by blast
```


Membership criterion: a coset is the zero coset iff its representative is in the ideal.

```

lemma Class-eq-zero-iff:
  assumes  $a \in R$ 
  shows  $\text{additive.Class } a = \text{additive.Class } \mathbf{0} \longleftrightarrow a \in I$ 
proof –
  have  $\text{additive.Class } a = \text{additive.Class } \mathbf{0} \longleftrightarrow (a, \mathbf{0}) \in \text{Ring-Congruence}$ 
    using assms by (simp add: additive.Class-equivalence additive-congruence)
  also have  $\dots \longleftrightarrow a \in I$ 
    using assms by (auto simp: Ring-Congruence-def)
  finally show ?thesis .
qed

end

```

An ideal of a commutative ring: add only the commutativity axiom on top of *Ideal* (so the *Ring/multiplicative* structure is shared, avoiding a locale-merge clash).

```

locale ideal-in-comm-ring = Ideal + commutative-ring
begin

```

```

lemmas comm = multiplicative.commutative

```

The quotient multiplication is commutative.

```

lemma quot-mult-comm:
  assumes  $a \in R$   $b \in R$ 
  shows  $\text{additive.Class } a \cdot [\cdot] \text{additive.Class } b = \text{additive.Class } b \cdot [\cdot] \text{additive.Class } a$ 
proof –
  have  $\text{additive.Class } a \cdot [\cdot] \text{additive.Class } b = \text{additive.Class } (a \cdot b)$ 
    by (rule multiplicative.Class-commutes-with-composition[OF assms])
  also have  $\dots = \text{additive.Class } (b \cdot a)$  using comm[OF assms] by simp
  also have  $\dots = \text{additive.Class } b \cdot [\cdot] \text{additive.Class } a$ 
    by (rule multiplicative.Class-commutes-with-composition[OF assms(2,1), symmetric])
  finally show ?thesis .
qed

```

Quotient multiplication in terms of representatives.

```

lemma quot-mult-Class:
  assumes  $a \in R$   $b \in R$ 
  shows  $\text{additive.Class } a \cdot [\cdot] \text{additive.Class } b = \text{additive.Class } (a \cdot b)$ 
  by (rule multiplicative.Class-commutes-with-composition[OF assms])

```

B1: the ideal is prime iff the quotient has no zero divisors (is an integral domain, given nontriviality $I \neq R$).

```

theorem prime-ideal-iff-quotient-no-zero-divisors:
   $\text{prime-ideal} \longleftrightarrow I \neq R \wedge$ 
     $(\forall a \in R. \forall b \in R. \text{additive.Class } a \cdot [\cdot] \text{additive.Class } b = \text{additive.Class } \mathbf{0})$ 

```

$\longrightarrow \text{additive.Class } a = \text{additive.Class } \mathbf{0} \vee \text{additive.Class } b = \text{additive.Class } \mathbf{0}$
proof –
have $(\forall a \in R. \forall b \in R. \text{additive.Class } a [\cdot] \text{additive.Class } b = \text{additive.Class } \mathbf{0} \longrightarrow \text{additive.Class } a = \text{additive.Class } \mathbf{0} \vee \text{additive.Class } b = \text{additive.Class } \mathbf{0})$
 $= (\forall a \in R. \forall b \in R. a \cdot b \in I \longrightarrow a \in I \vee b \in I)$
proof (*intro ball-cong refl iff-allI*)
fix $a \ b$ **assume** $ab: a \in R \ b \in R$
have $abR: a \cdot b \in R$ **using** ab **by** *simp*
show $(\text{additive.Class } a [\cdot] \text{additive.Class } b = \text{additive.Class } \mathbf{0} \longrightarrow \text{additive.Class } a = \text{additive.Class } \mathbf{0} \vee \text{additive.Class } b = \text{additive.Class } \mathbf{0})$
 $= (a \cdot b \in I \longrightarrow a \in I \vee b \in I)$
by (*simp add: quot-mult-Class[OF ab] Class-eq-zero-iff abR Class-eq-zero-iff[OF ab(I)]*)
 $\text{Class-eq-zero-iff}[OF\ ab(2)]$
qed
then show *?thesis* **unfolding** *prime-ideal-def* **by** *blast*
qed

The ideal generated by I together with one element a , i.e. $I + (a)$.

definition *ext-ideal* :: $'a \Rightarrow 'a \text{ set}$
where $\text{ext-ideal } a = \{m + r \cdot a \mid m \in I \wedge r \in R\}$

lemma *ext-ideal-memI*: $\llbracket m \in I; r \in R \rrbracket \Longrightarrow m + r \cdot a \in \text{ext-ideal } a$
unfolding *ext-ideal-def* **by** *blast*

lemma *ext-ideal-contains-I*:
assumes $a \in R$
shows $I \subseteq \text{ext-ideal } a$
proof
fix m **assume** $m: m \in I$
then have $mR: m \in R$ **using** *additive.subset* **by** *blast*
have $m + \mathbf{0} \cdot a = m$ **using** *assms mR* **by** (*simp add: additive.right-unit*)
moreover have $m + \mathbf{0} \cdot a \in \text{ext-ideal } a$ **using** m *ext-ideal-memI* [*of m 0*] **by** *simp*
ultimately show $m \in \text{ext-ideal } a$ **by** *simp*
qed

lemma *ext-ideal-contains-elt*:
assumes $a \in R$
shows $a \in \text{ext-ideal } a$
proof –
have $\mathbf{0} + \mathbf{1} \cdot a = a$ **using** *assms* **by** (*simp add: additive.left-unit*)
moreover have $\mathbf{0} + \mathbf{1} \cdot a \in \text{ext-ideal } a$ **using** *ext-ideal-memI* [*of 0 1*] **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

ext-ideal a is closed under addition and additive inverses, hence an additive subgroup of R .

```

lemma ext-ideal-subgroup:
  assumes a:  $a \in R$ 
  shows Subgroup (ext-ideal a) R (+) 0
proof (rule additive.subgroupI)
  show ext-ideal a  $\subseteq R$ 
  proof
    fix x assume x  $\in$  ext-ideal a
    then obtain m r where  $m \in I$   $r \in R$   $x = m + r \cdot a$ 
      unfolding ext-ideal-def by blast
    then show  $x \in R$  using a additive.subset by auto
  qed
next
  have 0  $\in I$  by (rule additive.sub-unit-closed)
  then have 0 + 0  $\cdot a \in$  ext-ideal a using a ext-ideal-memI[of 0 0] by blast
  moreover have 0 + 0  $\cdot a = 0$  using a by simp
  ultimately show 0  $\in$  ext-ideal a by simp
next
  fix g h assume g  $\in$  ext-ideal a h  $\in$  ext-ideal a
  then obtain m1 r1 m2 r2 where
    g:  $m1 \in I$   $r1 \in R$   $g = m1 + r1 \cdot a$  and
    h:  $m2 \in I$   $r2 \in R$   $h = m2 + r2 \cdot a$ 
    unfolding ext-ideal-def by blast
  have m1R:  $m1 \in R$  and m2R:  $m2 \in R$  using g h additive.subset by auto
  have ra1:  $r1 \cdot a \in R$  and ra2:  $r2 \cdot a \in R$  using g h a by auto
  — Rearrange  $(m1 + x) + (m2 + y) = (m1 + m2) + (x + y)$  in the abelian group.
  have rearr:  $(p + x) + (q + y) = (p + q) + (x + y)$ 
    if  $p \in R$   $x \in R$   $q \in R$   $y \in R$  for  $p$   $x$   $q$   $y$ 
  proof —
    have  $(p + x) + (q + y) = p + (x + (q + y))$ 
      using that by (simp add: additive.associative)
    also have  $\dots = p + ((x + q) + y)$ 
      using that by (simp add: additive.associative)
    also have  $\dots = p + ((q + x) + y)$ 
      using that by (simp add: additive.commutative)
    also have  $\dots = p + (q + (x + y))$ 
      using that by (simp add: additive.associative)
    also have  $\dots = (p + q) + (x + y)$ 
      using that by (simp add: additive.associative)
    finally show ?thesis .
  qed
  have  $g + h = (m1 + r1 \cdot a) + (m2 + r2 \cdot a)$  using g h by simp
  also have  $\dots = (m1 + m2) + (r1 \cdot a + r2 \cdot a)$ 
    using rearr[OF m1R ra1 m2R ra2] .
  also have  $\dots = (m1 + m2) + (r1 + r2) \cdot a$ 
    using g h a by (simp add: distributive)
  finally have  $g + h = (m1 + m2) + (r1 + r2) \cdot a$  .
  moreover have  $m1 + m2 \in I$  using g h by (simp add: additive.sub-composition-closed)
  ultimately show  $g + h \in$  ext-ideal a using g h ext-ideal-memI[of m1+m2 r1+r2]
  by simp
next

```

```

fix g assume g ∈ ext-ideal a
then obtain m r where m ∈ I r ∈ R g = m + r · a unfolding ext-ideal-def by
blast
then have g ∈ R using a additive.subset by auto
then show additive.invertible g by simp
next
fix g assume g ∈ ext-ideal a
then obtain m r where g: m ∈ I r ∈ R g = m + r · a
  unfolding ext-ideal-def by blast
have mR: m ∈ R using g additive.subset by auto
have additive.inverse g = (additive.inverse m) + (additive.inverse r) · a
  using g mR a by (simp add: additive.inverse-composition-commute additive.commutative
left-minus)
moreover have additive.inverse m ∈ I using g by simp
moreover have additive.inverse r ∈ R using g by simp
ultimately show additive.inverse g ∈ ext-ideal a
  using ext-ideal-memI[of additive.inverse m additive.inverse r] by simp
qed

```

Hence *ext-ideal a* is an ideal of *R*: the absorption laws follow from *I*'s absorption and commutativity of multiplication.

lemma *ext-ideal-is-ideal*:

```

assumes a: a ∈ R
shows Ideal (ext-ideal a) R (+) (·) 0 1

```

proof –

```

interpret sub: Subgroup ext-ideal a R (+) 0 by (rule ext-ideal-subgroup[OF a])

```

```

show ?thesis

```

proof (*unfold-locales*)

```

fix x y assume x: x ∈ R and y: y ∈ ext-ideal a

```

```

obtain m r where mr: m ∈ I r ∈ R y = m + r · a

```

```

  using y unfolding ext-ideal-def by blast

```

```

have mR: m ∈ R using mr additive.subset by auto

```

```

show x · y ∈ ext-ideal a

```

proof –

```

have x · y = x · m + (x · r) · a

```

```

  using x mr mR a by (simp add: distributive multiplicative.associative)

```

```

moreover have x · m ∈ I using x mr by (simp add: Ideal)

```

```

moreover have x · r ∈ R using x mr by simp

```

```

ultimately show ?thesis using ext-ideal-memI[of x · m x · r] by simp

```

qed

```

show y · x ∈ ext-ideal a

```

proof –

```

have rax: (r · a) · x = (r · x) · a

```

proof –

```

  have (r · a) · x = r · (a · x) using mr a x by (simp add: multiplica-
tive.associative)

```

```

  also have ... = r · (x · a) using a x by (simp add: comm)

```

```

  also have ... = (r · x) · a using mr a x by (simp add: multiplicative.associative)

```

```

  finally show ?thesis .

```

```

qed
have  $y \cdot x = m \cdot x + (r \cdot a) \cdot x$ 
  using  $x \text{ } mR \text{ } a$  by (simp add: distributive)
also have  $\dots = m \cdot x + (r \cdot x) \cdot a$  using  $rax$  by simp
finally have  $y \cdot x = m \cdot x + (r \cdot x) \cdot a$  .
moreover have  $m \cdot x \in I$  using  $x \text{ } mR$  by (simp add: Ideal)
moreover have  $r \cdot x \in R$  using  $x \text{ } mR$  by simp
ultimately show ?thesis using ext-ideal-memI[of  $m \cdot x \text{ } r \cdot x$ ] by simp
qed
qed
qed

```

For a maximal ideal, any element outside I generates (with I) the whole ring, giving a representation $\mathbf{1} = m + r \cdot a$ with $m \in I$.

```

lemma maximal-ext-ideal-eq-whole:
  assumes  $max$ : maximal-ideal and  $a$ :  $a \in R$  and  $aI$ :  $a \notin I$ 
  shows ext-ideal  $a = R$ 
proof -
  have Ideal (ext-ideal  $a$ )  $R$  (+) ( $\cdot$ )  $\mathbf{0}$   $\mathbf{1}$  by (rule ext-ideal-is-ideal[OF  $a$ ])
  moreover have  $I \subseteq \text{ext-ideal } a$  by (rule ext-ideal-contains-I[OF  $a$ ])
  moreover have ext-ideal  $a \neq I$ 
    using ext-ideal-contains-elt[OF  $a$ ]  $aI$  by blast
  ultimately show ?thesis using  $max$  unfolding maximal-ideal-def by blast
qed

```

A maximal ideal is prime. If $a \cdot b \in I$ but $a \notin I$, then adjoining a generates the whole ring, giving $\mathbf{1} = m + r \cdot a$ with $m \in I$; hence $b = m \cdot b + r \cdot (a \cdot b) \in I$.

```

theorem maximal-imp-prime-ideal:
  assumes  $max$ : maximal-ideal shows prime-ideal
proof (unfold prime-ideal-def, intro conjI ballI impI)
  show  $I \neq R$  using  $max$  unfolding maximal-ideal-def by simp
next
  fix  $a \text{ } b$  assume  $a$ :  $a \in R$  and  $b$ :  $b \in R$  and  $abI$ :  $a \cdot b \in I$ 
  show  $a \in I \vee b \in I$ 
  proof (cases  $a \in I$ )
    case True then show ?thesis ..
  next
    case False
    —  $a$  generates the whole ring together with  $I$ :  $\mathbf{1} = m + r \cdot a$ .
    have  $\mathbf{1} \in \text{ext-ideal } a$  using maximal-ext-ideal-eq-whole[OF  $max \text{ } a \text{ } False$ ] by simp
    then obtain  $m \text{ } r$  where  $m$ :  $m \in I$  and  $r$ :  $r \in R$  and  $one$ :  $\mathbf{1} = m + r \cdot a$ 
      unfolding ext-ideal-def by blast
    have  $mR$ :  $m \in R$  using  $m$  additive.subset by auto
    — Then  $b = m \cdot b + r \cdot (a \cdot b)$ , both summands in  $I$ .
    have  $b = \mathbf{1} \cdot b$  using  $b$  by simp
    also have  $\dots = (m + r \cdot a) \cdot b$  using  $one$  by simp
    also have  $\dots = m \cdot b + (r \cdot a) \cdot b$  using  $mR \text{ } r \text{ } a \text{ } b$  by (simp add: distributive)
  qed

```

also have $(r \cdot a) \cdot b = r \cdot (a \cdot b)$ **using** $r \ a \ b$ **by** (*simp add: multiplicative.associative*)
finally have $beq: b = m \cdot b + r \cdot (a \cdot b)$.
have $m \cdot b \in I$ **using** $Ideal(2)[OF \ b \ m]$.
moreover have $r \cdot (a \cdot b) \in I$ **using** $Ideal(1)[OF \ r \ abI]$.
ultimately have $m \cdot b + r \cdot (a \cdot b) \in I$ **by** (*rule additive.sub-composition-closed*)
then have $b \in I$ **using** beq **by** *simp*
then show *?thesis* ..
qed
qed

lemma *maximal-quotient-invertible:*

assumes *max: maximal-ideal* **and** $a: a \in R$ **and** $aI: a \notin I$
shows $\exists r \in R. \text{additive.Class } r \ [\cdot] \text{additive.Class } a = \text{additive.Class } 1$
proof –
have $1 \in \text{ext-ideal } a$ **using** *maximal-ext-ideal-eq-whole* $[OF \ max \ a \ aI]$ **by** *simp*
then obtain $m \ r$ **where** $m: m \in I$ **and** $r: r \in R$ **and** *one*: $1 = m + r \cdot a$
unfolding *ext-ideal-def* **by** *blast*
have $mR: m \in R$ **using** m *additive.subset* **by** *auto*
have $raR: r \cdot a \in R$ **using** $r \ a$ **by** *simp*
have $\text{additive.Class } 1 = \text{additive.Class } (m + r \cdot a)$ **using** *one* **by** *simp*
also have $\dots = \text{additive.Class } m \ [+]\ \text{additive.Class } (r \cdot a)$
using $mR \ raR$ **by** (*simp add: additive.Class-commutes-with-composition*)
also have $\text{additive.Class } m = \text{additive.Class } 0$ **using** m **by** (*simp add: Class-eq-zero-iff mR*)
finally have $\text{additive.Class } 1 = \text{additive.Class } 0 \ [+]\ \text{additive.Class } (r \cdot a)$.
also have $\dots = \text{additive.Class } (r \cdot a)$
using raR **by** (*simp add: additive.Class-commutes-with-composition[symmetric]*
additive.left-unit)
also have $\text{additive.Class } (r \cdot a) = \text{additive.Class } r \ [\cdot] \text{additive.Class } a$
using $r \ a$ **by** (*simp add: quot-mult-Class*)
finally have $\text{additive.Class } 1 = \text{additive.Class } r \ [\cdot] \text{additive.Class } a$.
then show *?thesis* **using** r **by** *auto*
qed

If two ring elements have equal cosets, their difference lies in I .

lemma *Class-eq-imp-diff-in-I:*

assumes $a \in R \ b \in R$ $\text{additive.Class } a = \text{additive.Class } b$
shows $a - b \in I$
proof –
have $(a, b) \in \text{Ring-Congruence}$
using *assms* **by** (*simp add: additive.Class-equivalence additive-congruence*)
then show *?thesis* **by** (*rule Ring-CongruenceD*)
qed

theorem *maximal-ideal-iff-quotient-field:*

$\text{maximal-ideal} \iff I \neq R \wedge$
 $(\forall a \in R. \text{additive.Class } a \neq \text{additive.Class } 0$
 $\implies (\exists r \in R. \text{additive.Class } r \ [\cdot] \text{additive.Class } a = \text{additive.Class } 1))$
proof

```

assume max: maximal-ideal
have I ≠ R using max unfolding maximal-ideal-def by simp
moreover have ∃ r ∈ R. additive.Class r [·] additive.Class a = additive.Class 1
  if aR: a ∈ R and ane: additive.Class a ≠ additive.Class 0 for a
proof -
  have a ∉ I using aR ane by (simp add: Class-eq-zero-iff)
  then show ?thesis by (rule maximal-quotient-invertible[OF max aR])
qed
ultimately show I ≠ R ∧
  (∀ a ∈ R. additive.Class a ≠ additive.Class 0
    → (∃ r ∈ R. additive.Class r [·] additive.Class a = additive.Class 1)) by
blast
next
assume rhs: I ≠ R ∧
  (∀ a ∈ R. additive.Class a ≠ additive.Class 0
    → (∃ r ∈ R. additive.Class r [·] additive.Class a = additive.Class 1))
show maximal-ideal
  unfolding maximal-ideal-def
proof (intro conjI allI impI)
  show I ≠ R using rhs by simp
  fix J assume J: Ideal J R (+) (·) 0 1 and IJ: I ⊆ J
  show J = I ∨ J = R
  proof (rule ccontr)
    assume ¬ (J = I ∨ J = R)
    then have JnI: J ≠ I and JnR: J ≠ R by auto
    interpret J: Ideal J R (+) (·) 0 1 by (rule J)
    from JnI IJ obtain a where aJ: a ∈ J and aI: a ∉ I by blast
    have aR: a ∈ R using aJ J.additive.subset by auto
    have ane: additive.Class a ≠ additive.Class 0 using aR aI by (simp add:
      Class-eq-zero-iff)
    then obtain r where r: r ∈ R and inv: additive.Class r [·] additive.Class a =
      additive.Class 1
    using rhs aR by blast
    have raJ: r · a ∈ J using r aJ by (simp add: J.Ideal)
    have raR: r · a ∈ R using r aR by simp
    have ClassEq: additive.Class (r · a) = additive.Class 1
      using r aR inv by (simp add: quot-mult-Class)
    — So r · a - 1 ∈ I, hence 1 - r · a ∈ I ⊆ J, giving 1 ∈ J.
    have r · a - 1 ∈ I using ClassEq-imp-diff-in-I[OF raR - ClassEq] by simp
    then have negI: -(r · a - 1) ∈ I by simp
    have -(r · a - 1) = 1 - r · a
    using raR by (simp add: additive.inverse-composition-commute additive.commutative)
    then have oneI: 1 - r · a ∈ I using negI by simp
    have oneJ: 1 - r · a ∈ J using oneI IJ by blast
    have (1 - r · a) + r · a ∈ J using oneJ raJ by (simp add: J.additive.sub-composition-closed)
    moreover have (1 - r · a) + r · a = 1
    proof -
      have (1 - r · a) + r · a = 1 + ((- r · a) + r · a)
      using raR by (simp add: additive.associative)
      also have ... = 1 using raR by simp

```

```

    finally show ?thesis .
  qed
  ultimately have  $1 \in J$  by simp
  then have  $J = R$  using  $J.additive.subset$ 
    by (metis  $J.Ideal(1)$   $multiplicative.right-unit$   $subsetI$   $subset-antisym$ )
  then show False using  $JnR$  by simp
qed
qed
qed

end

```

49 Principal Ideals

The principal ideal generated by a single element of a commutative ring: the set of all multiples $r \cdot a$. This is the smallest ideal containing a , and it is the building block for principal ideal domains (e.g. $F[X]$ over a field).

49.1 Existence of maximal ideals

Every proper ideal is contained in a maximal one. This is the standard application of Zorn's lemma (*Zorn-Lemma2*) to the family of proper ideals containing a given one: the union of a chain of such ideals is again one, so the family has a maximal element.

The only delicate point is *properness* of the union. An ideal is proper exactly when it omits the unit 1 : if 1 lies in an ideal then so does $a \cdot 1 = a$ for every a , making it the whole ring. Since every member of the chain omits 1 , so does the union — which is why the chain condition can be checked at all, and why the argument needs a ring with a unit.

```

context Ring
begin

```

An ideal is proper if and only if it does not contain 1 .

```

lemma ideal-proper-iff-one-notin:

```

```

  assumes  $I: Ideal\ I\ R\ (+)\ (\cdot)\ 0\ 1$ 

```

```

  shows  $I \neq R \longleftrightarrow 1 \notin I$ 

```

```

proof -

```

```

  interpret  $I: Ideal\ I\ R\ (+)\ (\cdot)\ 0\ 1$  by (rule  $I$ )

```

```

  show ?thesis

```

```

  proof

```

```

    assume  $I \neq R$ 

```

```

    show  $1 \notin I$ 

```

```

  proof

```

```

    assume one:  $1 \in I$ 

```

```

    have  $R \subseteq I$ 

```

```

  proof

```



```

    fix a assume a: a ∈ R
    have a · 1 ∈ I using a one by (rule I.Ideal(1))
    then show a ∈ I using a by simp
qed
with I.additive.subset have I = R by blast
with ⟨I ≠ R⟩ show False by simp
qed
next
  assume 1 ∉ I
  then show I ≠ R using multiplicative.unit-closed by blast
qed
qed

```

The union of a nonempty chain of ideals is an ideal. Directedness is what makes the additive closure work: two elements of the union lie in a common member of the chain.

lemma *ideal-Union-chain*:

```

  assumes ne: C ≠ {}
  and id: ⋀ I. I ∈ C ⟹ Ideal I R (+) (·) 0 1
  and chain: ⋀ I J. [ I ∈ C; J ∈ C ] ⟹ I ⊆ J ∨ J ⊆ I
  shows Ideal (⋃ C) R (+) (·) 0 1

```

proof —

```

  obtain I0 where I0: I0 ∈ C using ne by blast
  interpret I0: Ideal I0 R (+) (·) 0 1 by (rule id[OF I0])
  have sub: ⋃ C ⊆ R
  proof
    fix x assume x ∈ ⋃ C
    then obtain I where I: I ∈ C and x: x ∈ I by blast
    interpret I: Ideal I R (+) (·) 0 1 by (rule id[OF I])
    show x ∈ R using x I.additive.subset by blast
  qed

```

have sg: Subgroup (⋃ C) R (+) 0

proof (rule additive.subgroupI)

show ⋃ C ⊆ R by (rule sub)

show 0 ∈ ⋃ C using I0 I0.additive.sub-unit-closed by blast

next

— Directedness: pick the larger of the two members containing g and h .

fix g h assume g ∈ ⋃ C and h ∈ ⋃ C

then obtain I J where I: I ∈ C g ∈ I and J: J ∈ C h ∈ J by blast

show g + h ∈ ⋃ C

proof (cases I ⊆ J)

case True

interpret J: Ideal J R (+) (·) 0 1 by (rule id[OF J(1)])

have g + h ∈ J using True I(2) J(2) by (blast intro: J.additive.sub-composition-closed)

then show ?thesis using J(1) by blast

next

case False

then have J ⊆ I using chain[OF I(1) J(1)] by blast

interpret I: Ideal I R (+) (·) 0 1 by (rule id[OF I(1)])

```

have  $g + h \in I$  using  $\langle J \subseteq I \rangle I(2) J(2)$  by (blast intro:  $I.additive.sub-composition-closed$ )
then show ?thesis using  $I(1)$  by blast
qed
next
fix  $g$  assume  $g \in \bigcup C$ 
then show  $additive.invertible\ g$  using  $sub$  by blast
next
fix  $g$  assume  $g \in \bigcup C$ 
then obtain  $I$  where  $I: I \in C$  and  $g: g \in I$  by blast
interpret  $I: Ideal\ I\ R\ (+)\ (\cdot)\ 0\ 1$  by (rule  $id[OF\ I]$ )
have  $additive.inverse\ g \in I$ 
using  $g\ I.additive.submonoid-inverse-closed\ I.additive.sub.invertible$  by blast
then show  $additive.inverse\ g \in \bigcup C$  using  $I$  by blast
qed
interpret  $S: Subgroup\ \bigcup C\ R\ (+)\ 0$  by (rule  $sg$ )
show ?thesis
proof unfold-locales
fix  $a\ b$  assume  $a: a \in R$  and  $b: b \in \bigcup C$ 
from  $b$  obtain  $I$  where  $I: I \in C$  and  $bI: b \in I$  by blast
interpret  $I: Ideal\ I\ R\ (+)\ (\cdot)\ 0\ 1$  by (rule  $id[OF\ I]$ )
show  $a \cdot b \in \bigcup C$  using  $I\ a\ bI$  by (blast intro:  $I.Ideal(1)$ )
show  $b \cdot a \in \bigcup C$  using  $I\ a\ bI$  by (blast intro:  $I.Ideal(2)$ )
qed
qed

```

Krull's theorem. Every proper ideal extends to a maximal one. Apply *Zorn-Lemma2* to the family of proper ideals containing I : a nonempty chain's union is an ideal by *ideal-Union-chain*, is proper because no member contains $\mathbf{1}$, and contains I ; the empty chain is bounded by I itself.

theorem *maximal-ideal-exists*:

assumes $I: Ideal\ I\ R\ (+)\ (\cdot)\ 0\ 1$ and *proper*: $I \neq R$

shows $\exists M. Ideal\ M\ R\ (+)\ (\cdot)\ 0\ 1 \wedge I \subseteq M \wedge Ideal.maximal-ideal\ M\ R\ (+)\ (\cdot)\ 0\ 1$

proof —

interpret $I: Ideal\ I\ R\ (+)\ (\cdot)\ 0\ 1$ by (rule I)

have $Ione: \mathbf{1} \notin I$ using $I\ proper$ by (simp add: *ideal-proper-iff-one-notin*)

— The family: proper ideals containing I .

define $\mathcal{A}$ where $\mathcal{A} = \{J. Ideal\ J\ R\ (+)\ (\cdot)\ 0\ 1 \wedge I \subseteq J \wedge \mathbf{1} \notin J\}$

have $IA: I \in \mathcal{A}$ unfolding $\mathcal{A}\text{-def}$ using $I\ Ione$ by blast

— Every chain in the family has an upper bound in the family. We use *Zorn-Lemma2* rather than *Zorn-Lemma* precisely because of the empty chain: the latter would demand $\bigcup \{\} = \{\} \in \mathcal{A}$, which is false, whereas an upper bound for the empty chain is supplied by I itself.

have *bound*: $\exists U \in \mathcal{A}. \forall X \in \mathcal{C}. X \subseteq U$ if $\mathcal{C}: \mathcal{C} \in chains\ \mathcal{A}$ for $\mathcal{C}$

proof (cases $\mathcal{C} = \{\}$)

case *True*

then show ?thesis using IA by blast

next

case *False*

have *sub*: $\mathcal{C} \subseteq \mathcal{A}$ and *ch*: $chain_{\subseteq} \mathcal{C}$ using $\mathcal{C}$ by (auto simp: *chains-def*)

```

have id:  $\bigwedge J. J \in \mathcal{C} \implies \text{Ideal } J \text{ } R \text{ } (+) \text{ } (\cdot) \text{ } 0 \text{ } 1$  using sub by (auto simp:  $\mathcal{A}$ -def)
have le:  $\bigwedge J \text{ } K. \llbracket J \in \mathcal{C}; K \in \mathcal{C} \rrbracket \implies J \subseteq K \vee K \subseteq J$ 
  using ch by (auto simp: chain-subset-def)
have Ideal ( $\bigcup \mathcal{C}$ )  $R \text{ } (+) \text{ } (\cdot) \text{ } 0 \text{ } 1$  using False id le by (rule ideal-Union-chain)
moreover have  $I \subseteq \bigcup \mathcal{C}$  using False sub by (auto simp:  $\mathcal{A}$ -def)
moreover have  $1 \notin \bigcup \mathcal{C}$  using sub by (auto simp:  $\mathcal{A}$ -def)
ultimately have  $\bigcup \mathcal{C} \in \mathcal{A}$  unfolding  $\mathcal{A}$ -def by blast
then show ?thesis by blast
qed
— Zorn gives a maximal member of the family; maximality among proper ideals over
 $I$  upgrades to maximality among all ideals, since a strictly larger ideal either contains
 $1$  — hence is  $R$  — or would still be in the family.
— Zorn's premise is discharged as an explicit  $\forall$ -statement first: handed to blast as
the conditional rule bound, the set-valued unknown sends it into a search that does
not terminate.
have prem:  $\forall \mathcal{C} \in \text{chains } \mathcal{A}. \exists U \in \mathcal{A}. \forall X \in \mathcal{C}. X \subseteq U$  using bound by blast
obtain  $M$  where  $M: M \in \mathcal{A}$  and max:  $\bigwedge X. X \in \mathcal{A} \implies M \subseteq X \implies X = M$ 
  using Zorn-Lemma2[OF prem] by blast
from  $M$  have Mid:  $\text{Ideal } M \text{ } R \text{ } (+) \text{ } (\cdot) \text{ } 0 \text{ } 1$  and IM:  $I \subseteq M$  and Mone:  $1 \notin M$ 
  by (auto simp:  $\mathcal{A}$ -def)
interpret  $M: \text{Ideal } M \text{ } R \text{ } (+) \text{ } (\cdot) \text{ } 0 \text{ } 1$  by (rule Mid)
have Ideal.maximal-ideal  $M \text{ } R \text{ } (+) \text{ } (\cdot) \text{ } 0 \text{ } 1$ 
  unfolding  $M$ .maximal-ideal-def
proof (intro conjI allI impI)
  show  $M \neq R$  using Mid Mone by (simp add: ideal-proper-iff-one-notin)
  fix  $J$  assume  $J: \text{Ideal } J \text{ } R \text{ } (+) \text{ } (\cdot) \text{ } 0 \text{ } 1$  and MJ:  $M \subseteq J$ 
  show  $J = M \vee J = R$ 
  proof (cases  $1 \in J$ )
    case True
    — ideal-proper-iff-one-notin read contrapositively: containing  $1$  forces  $J = R$ .
    then have  $J = R$  using ideal-proper-iff-one-notin[OF J] by blast
    then show ?thesis by simp
  next
    case False
    then have  $J \in \mathcal{A}$  unfolding  $\mathcal{A}$ -def using J IM MJ by blast
    then show ?thesis using max MJ by blast
  qed
qed
then show ?thesis using Mid IM by blast
qed

```

end

context commutative-ring
begin

definition principal-ideal :: ' $a \Rightarrow 'a$ set
where principal-ideal $a = \{r \cdot a \mid r. r \in R\}$

lemma *principal-ideal-memI*: $r \in R \implies r \cdot a \in \text{principal-ideal } a$
unfolding *principal-ideal-def* **by** *blast*

lemma *principal-ideal-subset*:
assumes $a \in R$ **shows** $\text{principal-ideal } a \subseteq R$
proof
fix x **assume** $x \in \text{principal-ideal } a$
then obtain r **where** $r \in R$ $x = r \cdot a$ **unfolding** *principal-ideal-def* **by** *blast*
then show $x \in R$ **using** *assms* **by** *auto*
qed

a itself lies in *principal-ideal* a (take $r = 1$).

lemma *principal-ideal-contains*:
assumes $a \in R$ **shows** $a \in \text{principal-ideal } a$
proof $-$
have $1 \cdot a = a$ **using** *assms* **by** *simp*
moreover have $1 \cdot a \in \text{principal-ideal } a$ **using** *principal-ideal-memI*[*of* 1 a] **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

principal-ideal a is an additive subgroup: closed under $+$ via distributivity, and under inverses via *left-minus*.

lemma *principal-ideal-subgroup*:
assumes $a: a \in R$ **shows** *Subgroup* (*principal-ideal* a) R $(+)$ 0
proof (*rule additive.subgroupI*)
show $\text{principal-ideal } a \subseteq R$ **using** a **by** (*rule principal-ideal-subset*)
next
have $0 \cdot a = 0$ **using** a **by** *simp*
moreover have $0 \cdot a \in \text{principal-ideal } a$ **using** *principal-ideal-memI*[*of* 0 a] **by** *simp*
ultimately show $0 \in \text{principal-ideal } a$ **by** *simp*
next
fix g h **assume** $g \in \text{principal-ideal } a$ $h \in \text{principal-ideal } a$
then obtain $r1$ $r2$ **where** $g: r1 \in R$ $g = r1 \cdot a$ **and** $h: r2 \in R$ $h = r2 \cdot a$
unfolding *principal-ideal-def* **by** *blast*
have $g + h = r1 \cdot a + r2 \cdot a$ **using** g h **by** *simp*
also have $\dots = (r1 + r2) \cdot a$ **using** g h a **by** (*simp add: distributive*)
finally have $g + h = (r1 + r2) \cdot a$.
moreover have $r1 + r2 \in R$ **using** g h **by** *simp*
ultimately show $g + h \in \text{principal-ideal } a$ **using** *principal-ideal-memI*[*of* $r1 + r2$ a] **by** *simp*
next
fix g **assume** $g \in \text{principal-ideal } a$
then obtain r **where** $r \in R$ $g = r \cdot a$ **unfolding** *principal-ideal-def* **by** *blast*
then have $g \in R$ **using** a **by** *auto*
then show *additive.invertible* g **by** *simp*
next

```

fix g assume g ∈ principal-ideal a
then obtain r where g: r ∈ R g = r · a unfolding principal-ideal-def by blast
have additive.inverse g = (additive.inverse r) · a using g a by (simp add: left-minus)
moreover have additive.inverse r ∈ R using g by simp
ultimately show additive.inverse g ∈ principal-ideal a
  using principal-ideal-memI[of additive.inverse r a] by simp
qed

```

Hence *principal-ideal a* is an ideal: the absorption laws follow from associativity and commutativity of multiplication.

lemma *principal-ideal-is-ideal*:

```

  assumes a: a ∈ R shows Ideal (principal-ideal a) R (+) (·) 0 1
proof -
  interpret sub: Subgroup principal-ideal a R (+) 0 by (rule principal-ideal-subgroup[OF a])
  show ?thesis
  proof unfold-locales
    fix x y assume x: x ∈ R and y: y ∈ principal-ideal a
    obtain r where r: r ∈ R y = r · a using y unfolding principal-ideal-def by blast
    have yR: y ∈ R using r a by simp
    show x · y ∈ principal-ideal a
    proof -
      have x · y = (x · r) · a using x r a by (simp add: multiplicative.associative)
      moreover have x · r ∈ R using x r by simp
      ultimately show ?thesis using principal-ideal-memI[of x · r a] by simp
    qed
    show y · x ∈ principal-ideal a
    proof -
      have y · x = x · y using multiplicative.commutative[OF yR x] .
      also have ... = (x · r) · a using x r a by (simp add: multiplicative.associative)
      finally have y · x = (x · r) · a .
      moreover have x · r ∈ R using x r by simp
      ultimately show ?thesis using principal-ideal-memI[of x · r a] by simp
    qed
  qed
qed
qed
end
end

```

50 Ideals of the Polynomial Ring; Kronecker's Construction

theory *Poly-Ideal*

```

  imports Poly-Ring Ideal-Theory HOL-Library.Multiset
begin

```

This theory joins the carrier-set polynomial ring *Poly-Ring* to the ideal theory of *Ideal-Theory*. The payoff is Kronecker's lemma: over a field, the ideal generated by an irreducible polynomial is maximal, so the quotient $F[X]/(p)$ is a field. This is the engine that builds field extensions (and, ultimately, finite fields) by adjoining roots of irreducible polynomials.

context *Field*
begin

Inside a field the polynomial ring is a commutative ring, so its principal ideals are available through *commutative-ring.principal-ideal*. We abbreviate the principal ideal (p) of $F[X]$.

The principal ideal generated by p in $F[X]$, written out concretely.

definition *poly-pideal* :: $(nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a)$ set
where *poly-pideal* $p = \{r \otimes_P p \mid r. r \in \text{poly-carrier}\}$

lemma *poly-pideal-memI*: $r \in \text{poly-carrier} \implies r \otimes_P p \in \text{poly-pideal } p$
unfolding *poly-pideal-def* **by** *blast*

lemma *poly-pideal-contains*:
assumes $p \in \text{poly-carrier}$ **shows** $p \in \text{poly-pideal } p$
by $(metis \text{ assms } \text{poly-mult-one-left } \text{poly-one-closed } \text{poly-pideal-memI})$

poly-pideal agrees with the generic *commutative-ring.principal-ideal* of the polynomial ring, and is therefore an ideal of $F[X]$.

lemma *poly-pideal-is-ideal*:
assumes $p \in \text{poly-carrier}$
shows *Ideal* $(\text{poly-pideal } p)$ *poly-carrier* $(\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$
proof –
interpret *comm*: *commutative-ring poly-carrier* $(\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$
by $(rule \text{ poly-commutative-ring})$
show *?thesis*
using *comm.principal-ideal-is-ideal p*
unfolding *comm.principal-ideal-def poly-pideal-def* **by** *presburger*
qed

Kronecker's lemma. For an irreducible p , the principal ideal (p) is maximal in $F[X]$. Properness holds because $\mathbf{1}_P$ would force p to be a unit; maximality uses that $F[X]$ is a PID (*poly-PID*): any ideal J between (p) and $F[X]$ is principal, $J = \text{poly-pideal } b$, and $p = c \otimes_P b$ forces (by irreducibility) either b a unit, so $J = \text{poly-carrier}$, or c a unit, so $J \subseteq \text{poly-pideal } p$.

theorem *poly-irreducible-maximal*:
assumes *irr*: *poly-irreducible p*
shows *Ideal.maximal-ideal* $(\text{poly-pideal } p)$ *poly-carrier* $(\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$
proof –
have $p \in \text{poly-carrier}$ **using** *irr* **by** $(rule \text{ poly-irreducibleD-carrier})$
have *pnu*: $\neg \text{poly-unit } p$ **using** *irr* **by** $(rule \text{ poly-irreducibleD-nonunit})$
interpret *I*: *Ideal poly-pideal p poly-carrier* $(\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$

by (rule poly-pideal-is-ideal[OF p])
 — Properness: the ideal omits 1_P , else p would be a unit.
 have proper: poly-pideal $p \neq$ poly-carrier
 proof
 assume poly-pideal $p =$ poly-carrier
 then obtain r where $r: r \in$ poly-carrier and $rp: 1_P = r \otimes_P p$
 unfolding poly-pideal-def by blast
 then have $pr1: p \otimes_P r = 1_P$
 by (simp add: p poly-mult-comm)
 with pnu show False
 using p poly-unitI r rp by force
 qed
 — Maximality: any larger ideal is principal and collapses to (p) or all of $F[X]$.
 have $\forall J. \text{Ideal } J \text{ poly-carrier } (\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P \longrightarrow \text{poly-pideal } p \subseteq J$
 $\longrightarrow J = \text{poly-pideal } p \vee J = \text{poly-carrier}$
 proof (intro allI impI)
 fix J assume $J: \text{Ideal } J \text{ poly-carrier } (\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$ and sub: poly-pideal $p \subseteq$
 J
 interpret $J: \text{Ideal } J \text{ poly-carrier } (\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$ by (rule J)
 obtain b where $b: b \in$ poly-carrier and $Jeq: J = \{r \otimes_P b \mid r. r \in \text{poly-carrier}\}$
 using poly-PID[OF J] by blast
 have $JeqP: J = \text{poly-pideal } b$ unfolding poly-pideal-def by (rule Jeq)
 — p lies in J , so p is a multiple of b .
 have $p \in J$ using sub poly-pideal-contains[OF p] by blast
 then obtain c where $c: c \in$ poly-carrier and $pcb: p = c \otimes_P b$
 using $JeqP$ unfolding poly-pideal-def by blast
 from poly-irreducibleD-factor[OF irr c b pcb] show $J = \text{poly-pideal } p \vee J =$
 poly-carrier
 proof
 — If b is a unit then J is all of $F[X]$.
 assume poly-unit b
 then obtain b' where $b': b' \in$ poly-carrier and $b' \otimes_P b = 1_P$
 unfolding poly-unit-def by blast
 — $1_P = b' \otimes_P b$ is a multiple of b , hence in $J = \text{poly-pideal } b$.
 have $b' \otimes_P b \in \text{poly-pideal } b$ using b' by (rule poly-pideal-memI)
 then have $1_P \in J$ using $\langle b' \otimes_P b = 1_P \rangle JeqP$ by simp
 then have poly-carrier $\subseteq J$
 using $J.\text{Ideal}(1) \langle 1_P \in J \rangle$ poly-mult-one-right by fastforce
 then have $J = \text{poly-carrier}$ using $J.\text{additive.subset}$ by blast
 then show ?thesis ..
 next
 — If c is a unit then b is a multiple of p , so $J \subseteq (p)$.
 assume poly-unit c
 then obtain c' where $c': c' \in$ poly-carrier and $c \otimes_P c' = 1_P$ and $c': c' \otimes_P$
 $c = 1_P$
 unfolding poly-unit-def by blast
 — From $p = c \otimes_P b: b = c' \otimes_P p$.
 have $c' \otimes_P p = c' \otimes_P (c \otimes_P b)$ using pcb by simp
 also have $\dots = b$ using b
 by (metis c c' $c'c$ poly-mult-assoc poly-mult-one-left)

```

finally have beq:  $b = c' \otimes_P p$  ..
— Hence every multiple of  $b$  is a multiple of  $p$ :  $J \subseteq (p)$ .
have  $J \subseteq \text{poly-pideal } p$ 
proof
  fix  $x$  assume  $x \in J$ 
  then obtain  $r$  where  $r \in \text{poly-carrier}$   $x = r \otimes_P b$ 
    using  $\text{JeqP}$  unfolding  $\text{poly-pideal-def}$  by  $\text{blast}$ 
  then show  $x \in \text{poly-pideal } p$  using  $\text{poly-pideal-memI}$ 
    using  $I.\text{Ideal}(1)$   $\text{beq } c'$  by  $\text{presburger}$ 
qed
then have  $J = \text{poly-pideal } p$  using  $\text{sub}$  by  $\text{blast}$ 
then show  $?thesis$  ..
qed
qed
then show  $?thesis$  using  $\text{proper}$  unfolding  $I.\text{maximal-ideal-def}$  by  $\text{blast}$ 
qed

```

The principal ideal (p) is an ideal of the *commutative* polynomial ring, so the quotient theory of *ideal-in-comm-ring* applies to it.

lemma *poly-pideal-in-comm-ring*:

```

assumes  $p: p \in \text{poly-carrier}$ 
shows  $\text{ideal-in-comm-ring } (\text{poly-pideal } p) \text{ poly-carrier } (\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$ 
by ( $\text{simp add: ideal-in-comm-ring.intro } p \text{ poly-commutative-ring poly-pideal-is-ideal}$ )

```

Kronecker's theorem. For an irreducible p , the principal ideal (p) is a maximal ideal of the *commutative* polynomial ring $F[X]$. By the maximal-ideal characterisation *ideal-in-comm-ring.maximal-ideal-iff-quotient-field*, this is exactly the statement that the quotient $F[X]/(p)$ is a field: it is nontrivial and every nonzero coset is invertible.

theorem *poly-quotient-field*:

```

assumes  $\text{irr: poly-irreducible } p$ 
shows  $\text{ideal-in-comm-ring } (\text{poly-pideal } p) \text{ poly-carrier } (\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$ 
   $\wedge \text{Ideal.maximal-ideal } (\text{poly-pideal } p) \text{ poly-carrier } (\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$ 
by ( $\text{simp add: irr poly-irreducibleD-carrier poly-irreducible-maximal poly-pideal-in-comm-ring}$ )

```

50.1 The size of the quotient ring

Membership in the principal ideal (p) is detected by the remainder: x is a multiple of p iff $p \text{ mod } x \ p = \mathbf{0}_P$.

lemma *poly-pideal-iff-pmod*:

```

assumes  $p: p \in \text{poly-carrier}$  and  $\text{pnz: } p \neq \mathbf{0}_P$  and  $x: x \in \text{poly-carrier}$ 
shows  $x \in \text{poly-pideal } p \iff p \text{ mod } x \ p = \mathbf{0}_P$ 

```

proof

```

assume  $x \in \text{poly-pideal } p$ 
then obtain  $r$  where  $r: r \in \text{poly-carrier}$  and  $xr: x = r \otimes_P p$ 
  unfolding  $\text{poly-pideal-def}$  by  $\text{blast}$ 
—  $x = (p \otimes r) \oplus \mathbf{0}$ , so its remainder is  $\mathbf{0}_P$ .

```



```

have  $x = (p \otimes_P r) \oplus_P \mathbf{0}_P$ 
  by (metis  $p$  poly-add-comm poly-add-zero poly-mult-comm poly-zero-closed  $r$   $x$   $xr$ )
then show  $p \bmod x = \mathbf{0}_P$ 
  using pmod-unique[OF  $x$   $p$   $p \bmod x$   $r$  poly-zero-closed] by simp
next
assume  $z$ :  $p \bmod x = \mathbf{0}_P$ 
obtain  $q$  where  $q \in \text{poly-carrier}$  and  $xeq$ :  $x = (p \otimes_P q) \oplus_P p \bmod x$ 
  using pmod-spec[OF  $x$   $p$   $p \bmod x$ ] by blast
have  $x = (p \otimes_P q) \oplus_P \mathbf{0}_P$ 
  using  $xeq$   $z$  by simp
also have  $\dots = q \otimes_P p$ 
  using  $p$   $q$  by (simp add: poly-add-comm poly-add-zero poly-mult-comm poly-zero-closed
poly-mult-closed)
finally have  $x = q \otimes_P p$  .
then show  $x \in \text{poly-pideal } p$  using  $q$  by (auto simp: poly-pideal-def)
qed

```

The coset map $r \mapsto \text{coset } r$ restricted to *low-poly* (degree p) is a bijection onto the set of cosets: every coset has a unique representative of degree below p . Injectivity is uniqueness of $p \bmod$ (equal cosets have equal remainders, and a low-degree polynomial is its own remainder).

theorem *bij-betw-low-poly-cosets*:

```

assumes  $p$ :  $p \in \text{poly-carrier}$  and  $p \neq \mathbf{0}_P$ 
shows bij-betw ( $\text{Ideal.coset } (\text{poly-pideal } p)$   $\text{poly-carrier } (\oplus_P) \mathbf{0}_P$ )
  (low-poly (degree  $p$ ))
  ( $\text{Ideal.quotient-set } (\text{poly-pideal } p)$   $\text{poly-carrier } (\oplus_P) \mathbf{0}_P$ )

```

proof –

```

interpret  $I$ : ideal-in-comm-ring  $\text{poly-pideal } p$   $\text{poly-carrier } (\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$ 
  by (rule poly-pideal-in-comm-ring[OF  $p$ ])

```

— Two carrier polynomials are in the same coset iff their difference is a multiple of p .

— The additive-group inverse in $F[X]$ is the polynomial negation $\ominus_P y$.

```

have inv-eq:  $I.\text{additive.inverse } y = \ominus_P y$  if  $y$ :  $y \in \text{poly-carrier}$  for  $y$ 
  by (simp add: poly-diff-zero-eq poly-neg-closed poly-neg-neg  $y$ )

```

```

have same-coset:  $I.\text{coset } x = I.\text{coset } y \iff x \oplus_P (\ominus_P y) \in \text{poly-pideal } p$ 
  if  $x$ :  $x \in \text{poly-carrier}$  and  $y$ :  $y \in \text{poly-carrier}$  for  $x$   $y$ 
  using  $I.\text{coset-eq-iff}$ [OF  $x$   $y$ ] by (simp add: inv-eq[OF  $y$ ])

```

— A polynomial of degree below p that is a multiple of p must be zero.

```

have low-mult-zero:  $d = \mathbf{0}_P$ 
  if  $dc$ :  $d \in \text{poly-carrier}$   $d = \mathbf{0}_P \vee \text{degree } d < \text{degree } p$   $d \in \text{poly-pideal } p$  for  $d$ 
  by (metis that  $p$  pmod-small  $p \bmod x$  poly-pideal-iff-pmod)

```

show *?thesis*

proof (rule *bij-betw-imageI*)

— Injective: equal cosets give equal remainders, and low-degree polys are their own remainders.

```

show inj-on ( $I.\text{coset}$ ) (low-poly (degree  $p$ ))

```

proof (rule *inj-onI*)

```

fix  $r1$   $r2$  assume  $r1$ :  $r1 \in \text{low-poly } (\text{degree } p)$  and  $r2$ :  $r2 \in \text{low-poly } (\text{degree } p)$ 
  and  $eq$ :  $I.\text{coset } r1 = I.\text{coset } r2$ 

```

have $r1c$: $r1 \in \text{poly-carrier}$ **using** $r1$ **by** (rule *low-poly-closed*)
have $r2c$: $r2 \in \text{poly-carrier}$ **using** $r2$ **by** (rule *low-poly-closed*)
have $d1$: $r1 = \mathbf{0}_P \vee \text{degree } r1 < \text{degree } p$ **using** $r1$ $r1c$ **by** (simp add:
low-poly-iff-degree)
have $d2$: $r2 = \mathbf{0}_P \vee \text{degree } r2 < \text{degree } p$ **using** $r2$ $r2c$ **by** (simp add:
low-poly-iff-degree)
— The difference is a low-degree multiple of p , hence zero.
have diffc : $r1 \oplus_P (\ominus_P r2) \in \text{poly-carrier}$
using $r1c$ $r2c$ *poly-neg-closed* **by** (simp add: *poly-add-closed*)
have $r1 \oplus_P (\ominus_P r2) \in \text{poly-pideal } p$ **using** *same-coset[OF r1c r2c]* **eq** **by** *simp*
moreover **have** $r1 \oplus_P (\ominus_P r2) = \mathbf{0}_P \vee \text{degree } (r1 \oplus_P (\ominus_P r2)) < \text{degree } p$
proof (cases $r1 \oplus_P (\ominus_P r2) = \mathbf{0}_P$)
case *False*
— The difference is nonzero, so $0 < \text{degree } p$ (else $r1, r2$ both zero).
have dp : $\text{degree } p > 0$
using $d1$ $d2$ *poly-add-neg-right* *False* **by** *force*
have $\text{degree } (r1 \oplus_P (\ominus_P r2)) \leq \max (\text{degree } r1) (\text{degree } r2)$
by (*metis degree-add-le degree-neg poly-neg-closed r1c r2c*)
then show *?thesis* **using** $d1$ $d2$ dp **by** *auto*
qed *simp*
ultimately have $r1 \oplus_P (\ominus_P r2) = \mathbf{0}_P$ **using** *low-mult-zero[OF diffc]* **by** *blast*
then show $r1 = r2$ **using** *poly-diff-zero-eq[OF r1c r2c]* **by** *simp*
qed
— Image is exactly the set of cosets.
show $(I.\text{coset}) \text{ ' } \text{low-poly } (\text{degree } p) = I.\text{quotient-set}$
proof
show $(I.\text{coset}) \text{ ' } \text{low-poly } (\text{degree } p) \subseteq I.\text{quotient-set}$
by (*meson I.coset-in-quotient-set image-subsetI low-poly-closed*)
show $I.\text{quotient-set} \subseteq (I.\text{coset}) \text{ ' } \text{low-poly } (\text{degree } p)$
proof
fix C **assume** $C \in I.\text{quotient-set}$
then obtain a **where** $a: a \in \text{poly-carrier}$ **and** $Ca: C = I.\text{coset } a$
using *I.quotient-set-repr* **by** *blast*
— The remainder of a is a low-degree representative of the same coset.
have mc : $\text{pmod } a \text{ } p \in \text{poly-carrier}$ **using** *pmod-closed[OF a p pnz]* .
have pmc : $\text{pmod } a \text{ } p \in \text{low-poly } (\text{degree } p)$
using mc *pmod-degree[OF a p pnz]* **by** (simp add: *low-poly-iff-degree*)
— $a \oplus \ominus(\text{pmod } a \text{ } p) = p \otimes q \in (p)$, so a and $\text{pmod } a \text{ } p$ share a coset.
obtain q **where** $q: q \in \text{poly-carrier}$ **and** $aeq: a = (p \otimes_P q) \oplus_P \text{pmod } a \text{ } p$
using *pmod-spec[OF a p pnz]* **by** *blast*
have pqc : $p \otimes_P q \in \text{poly-carrier}$ **using** p q **by** (rule *poly-mult-closed*)
— Cancel $\text{pmod } a \text{ } p$: $a \oplus \ominus(\text{pmod } a \text{ } p) = p \otimes q$.
have $a \oplus_P (\ominus_P \text{pmod } a \text{ } p) = p \otimes_P q$
by (*metis I.additive commute-iff-inverse aeq inv-eq mc poly-add-comm pqc*)
then have $a \oplus_P (\ominus_P \text{pmod } a \text{ } p) = q \otimes_P p$
using *poly-mult-comm[OF p q]* **by** *simp*
then have $\text{mem}: a \oplus_P (\ominus_P \text{pmod } a \text{ } p) \in \text{poly-pideal } p$
using q **by** (simp add: *poly-pideal-memI*)
have $I.\text{coset } a = I.\text{coset } (\text{pmod } a \text{ } p)$
by (rule *iffD2[OF same-coset[OF a mc] mem]*)

```

    then show  $C \in (I.\text{coset})$  ‘ low-poly (degree  $p$ )
    by (simp add: Ca pmc)
  qed
qed
qed
qed

```

The quotient $F[X]/(p)$ has $|F|^{\text{deg } p}$ elements when the field F is finite: the cosets are in bijection with the $(\text{card } R)^{\text{deg } p}$ polynomials of degree below p .

theorem *card-poly-quotient*:

```

  assumes finite  $R$   $p \in \text{poly-carrier}$   $p \neq \mathbf{0}_P$ 
  shows  $\text{card } (\text{Ideal.quotient-set } (\text{poly-pideal } p) \text{ poly-carrier } (\oplus_P) \mathbf{0}_P) = \text{card } R \wedge^{\text{deg } p}$ 
  by (metis bij-betw-low-poly-cosets bij-betw-same-card card-low-poly assms)

```

50.2 Divisibility and Euclid’s lemma

Divisibility in $F[X]$: d divides x when x is a multiple of d . This is exactly membership of x in the principal ideal *poly-pideal* d .

definition *poly-dvd* :: $(\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow \text{bool}$
where *poly-dvd* $d \ x \longleftrightarrow (\exists q. q \in \text{poly-carrier} \wedge x = q \otimes_P d)$

lemma *poly-dvd-iff-mem-pideal*: *poly-dvd* $d \ x \longleftrightarrow x \in \text{poly-pideal } d$

unfolding *poly-dvd-def poly-pideal-def* **by** blast

A concrete divisibility witness with a polynomial divisor necessarily has a polynomial dividend. This recovers the carrier condition after working with the ambient function type.

lemma *poly-dvd-closed*:

```

  assumes dvd: poly-dvd  $d \ x$  and  $d: d \in \text{poly-carrier}$ 
  shows  $x \in \text{poly-carrier}$ 

```

proof –

```

  obtain  $q$  where  $q: q \in \text{poly-carrier}$  and  $xq: x = q \otimes_P d$ 
  using dvd unfolding poly-dvd-def by blast
  then show ?thesis using  $d$  by (simp add: poly-mult-closed)

```

qed

For a nonzero divisor, concrete polynomial divisibility is equivalent to having zero remainder under the polynomial division algorithm.

lemma *poly-dvd-iff-pmod-zero*:

```

  assumes  $d: d \in \text{poly-carrier}$  and  $\text{dnz}: d \neq \mathbf{0}_P$ 
  and  $x: x \in \text{poly-carrier}$ 
  shows poly-dvd  $d \ x \longleftrightarrow \text{pmod } x \ d = \mathbf{0}_P$ 
  using poly-dvd-iff-mem-pideal poly-pideal-iff-pmod[OF  $d \ \text{dnz } x$ ] by blast

```

Euclid’s lemma / irreducibles are prime. For an irreducible p , if p divides a product $a \otimes_P b$ then it divides one of the factors. This is primality of the maximal ideal *poly-pideal* p (*ideal-in-comm-ring.maximal-imp-prime-ideal* applied to *poly-irreducible-maximal*).

theorem *poly-irreducible-prime*:
assumes *irr*: *poly-irreducible* *p*
and *a*: *a* ∈ *poly-carrier* **and** *b*: *b* ∈ *poly-carrier*
and *dvd*: *poly-dvd* *p* (*a* ⊗_{*P*} *b*)
shows *poly-dvd* *p* *a* ∨ *poly-dvd* *p* *b*
proof —
have *p*: *p* ∈ *poly-carrier* **using** *irr* **by** (rule *poly-irreducibleD-carrier*)
interpret *I*: *ideal-in-comm-ring poly-pideal p poly-carrier* (⊕_{*P*}) (⊗_{*P*}) **0**_{*P*} **1**_{*P*}
by (rule *poly-pideal-in-comm-ring*[*OF p*])
have *I.prime-ideal* **using** *poly-irreducible-maximal*[*OF irr*] **by** (rule *I.maximal-imp-prime-ideal*)
— Primality: *a* ⊗ *b* ∈ (*p*) gives *a* ∈ (*p*) or *b* ∈ (*p*).
have *a* ∈ *poly-pideal* *p* ∨ *b* ∈ *poly-pideal* *p*
using *I.prime-ideal-def* ⟨*I.prime-ideal*⟩ *a b dvd poly-dvd-iff-mem-pideal* **by** *blast*
then show ?thesis **by** (*simp add: poly-dvd-iff-mem-pideal*)
qed

An irreducible polynomial does not divide **1**_{*P*} (else it would be a unit).

lemma *poly-irreducible-not-dvd-one*:
assumes *irr*: *poly-irreducible* *p* **shows** ¬ *poly-dvd* *p* **1**_{*P*}
proof
assume *poly-dvd* *p* **1**_{*P*}
then obtain *q* **where** *q*: *q* ∈ *poly-carrier* **and** **1**_{*P*} = *q* ⊗_{*P*} *p*
unfolding *poly-dvd-def* **by** *blast*
then have *poly-unit* *p*
using *poly-unitI irr poly-irreducibleD-carrier poly-mult-comm* **by** *presburger*
with *poly-irreducibleD-nonunit*[*OF irr*] **show** *False* ..
qed

Euclid's lemma for products. If an irreducible *p* divides a product of a list of polynomials, it divides one of them. (Base case: it cannot divide the empty product **1**_{*P*}; step: split off the head with *poly-irreducible-prime*.)

lemma *poly-irreducible-prime-list*:
assumes *irr*: *poly-irreducible* *p*
shows [∀ *f* ∈ *set fs*. *f* ∈ *poly-carrier*; *poly-dvd* *p* (*poly-prod fs*)]
⇒ ∃ *f* ∈ *set fs*. *poly-dvd* *p* *f*
proof (*induct fs*)
case *Nil*
then have *poly-dvd* *p* **1**_{*P*} **by** *simp*
with *poly-irreducible-not-dvd-one*[*OF irr*] **show** ?case ..
next
case (*Cons g gs*)
have *gc*: *g* ∈ *poly-carrier* **using** *Cons.prem*s(1) **by** *simp*
have *gsc*: ∀ *f* ∈ *set gs*. *f* ∈ *poly-carrier* **using** *Cons.prem*s(1) **by** *simp*
have *gsprod*: *poly-prod* *gs* ∈ *poly-carrier*
by (rule *poly-prod-closed*) (use *gsc* **in** *blast*)
have *poly-dvd* *p* (*g* ⊗_{*P*} *poly-prod* *gs*) **using** *Cons.prem*s(2) **by** *simp*
then have *poly-dvd* *p* *g* ∨ *poly-dvd* *p* (*poly-prod* *gs*)
using *poly-irreducible-prime*[*OF irr gc gsprod*] **by** *simp*
then show ?case

using *Cons.hyps gsc* by force
qed

50.3 Uniqueness of the irreducible factorisation

A nonzero divisor has degree at most that of its multiple.

lemma *poly-dvd-degree-le*:
 assumes *dvd*: *poly-dvd* *d x* and *dc*: $d \in \text{poly-carrier}$ and *xnz*: $x \neq \mathbf{0}_P$
 shows $\text{degree } d \leq \text{degree } x$
proof –
 obtain *q* where $q: q \in \text{poly-carrier}$ and *xeq*: $x = q \otimes_P d$
 using *dvd unfolding poly-dvd-def* by blast
 have $\text{degree } x = \text{degree } q + \text{degree } d$ using *xeq degree-mult[OF q dc]*
 using *dc poly-mult-comm poly-mult-zero-right q xnz* by force
 then show ?thesis by simp
 qed

Two monic irreducibles that divide one another are equal: *p* dividing the irreducible *q* makes *p* a unit or an associate of *q*; being a non-unit monic, it equals the monic *q*.

lemma *monic-irreducible-dvd-eq*:
 assumes *ip*: *poly-irreducible* *p* and *iq*: *poly-irreducible* *q*
 and *mp*: *poly-monic* *p* and *mq*: *poly-monic* *q* and *dvd*: *poly-dvd* *p q*
 shows $p = q$
proof –
 have *p*: $p \in \text{poly-carrier}$ using *ip* by (rule *poly-irreducibleD-carrier*)
 have *qc*: $q \in \text{poly-carrier}$ using *iq* by (rule *poly-irreducibleD-carrier*)
 obtain *c* where $c: c \in \text{poly-carrier}$ and *qcp*: $q = c \otimes_P p$ and *uc*: *poly-unit* *c*
 using *dvd unfolding poly-dvd-def*
 by (metis *ip iq poly-irreducible-def*)
 — *q* irreducible and $q = c \otimes_P p$: one of *c*, *p* is a unit.
 — A unit has degree 0, so *c* is a nonzero constant and *p*, *q* have equal degree; being monic, they have equal leading terms too.
 have *cconst*: $c = \text{poly-const } (c \ 0) \wedge c \ 0 \neq \mathbf{0}$ and *dc0*: $\text{degree } c = 0$
 using *poly-unit-imp-const[OF uc]* by auto
 have *c0R*: $c \ 0 \in R$ using *c* by (rule *poly-carrier-coeff-closed*)
 have *pnz*: $p \neq \mathbf{0}_P$ using *ip* by (rule *poly-irreducibleD-nonzero*)
 — Leading coefficient of $q = \text{poly-const } (c \ 0) \otimes_P p$ is $c \ 0 \cdot \text{lead-coeff } p$.
 have $\text{lead-coeff } q = c \ 0 \cdot \text{lead-coeff } p$
 using *qcp cconst lead-coeff-poly-const-mult[OF c0R - p pnz]* by simp
 moreover have $\text{lead-coeff } q = \mathbf{1}$ $\text{lead-coeff } p = \mathbf{1}$
 using *mp mq* by (simp-all add: *poly-monic-def*)
 ultimately have $c \ 0 = \mathbf{1}$ using *c0R* by simp
 — Hence $c = \mathbf{1}_P$ and $q = p$.
 then show ?thesis using *qcp p*
 using *cconst poly-mult-one-left* by auto
 qed

Pulling a factor out of a product: if *g* occurs in *gs*, the product factors as *g* times

the product of the rest.

```

lemma poly-prod-remove1:
  assumes  $g \in \text{set } gs$  and  $\forall f \in \text{set } gs. f \in \text{poly-carrier}$ 
  shows  $\text{poly-prod } gs = g \otimes_P \text{poly-prod } (\text{remove1 } g \text{ } gs)$ 
  using assms
proof (induct gs)
  case Nil then show ?case by simp
next
  case (Cons h hs)
  show ?case
  proof (cases g = h)
    case True then show ?thesis by simp
  next
  case False
    then have  $gs: g \in \text{set } hs$  and  $hc: h \in \text{poly-carrier}$  and  $hsc: \forall f \in \text{set } hs. f \in$ 
    poly-carrier
    using Cons.prems by auto
    have  $gc: g \in \text{poly-carrier}$  using  $gs \text{ } hsc$  by blast
    have  $hsprod: \text{poly-prod } (\text{remove1 } g \text{ } hs) \in \text{poly-carrier}$ 
    by (meson hsc notin-set-remove1 poly-prod-closed)
    have  $\text{poly-prod } (h \# hs) = h \otimes_P \text{poly-prod } hs$  by simp
    also have  $\dots = g \otimes_P (h \otimes_P \text{poly-prod } (\text{remove1 } g \text{ } hs))$ 
    using  $hc \text{ } gc \text{ } hsprod \text{ } Cons$ 
    by (metis gs hsc poly-mult-assoc poly-mult-comm)
    finally show ?thesis
    using False by simp
  qed
qed

```

Uniqueness of the monic irreducible factorisation. Two lists of monic irreducibles with equal products are permutations of one another. By induction on the first list: the head f divides the other product, hence equals some monic irreducible g in it; cancel f and recurse on the remaining factors.

```

theorem factorization-unique:
   $\llbracket \forall f \in \text{set } fs. \text{poly-irreducible } f \wedge \text{poly-monic } f;$ 
   $\forall g \in \text{set } gs. \text{poly-irreducible } g \wedge \text{poly-monic } g;$ 
   $\text{poly-prod } fs = \text{poly-prod } gs \rrbracket \implies \text{mset } fs = \text{mset } gs$ 
proof (induct fs arbitrary: gs)
  case Nil
  — Empty product is  $\mathbf{1}_P$ ; a nonempty  $gs$  would have an irreducible dividing  $\mathbf{1}_P$ .
  have  $gs = []$ 
  proof (rule ccontr)
    assume  $gs \neq []$ 
    then obtain  $g \text{ } gs'$  where  $gs: gs = g \# gs'$  by (cases gs) auto
    have  $ig: \text{poly-irreducible } g$  using Nil.prems(2)  $gs$  by simp
    have  $gsc: \forall f \in \text{set } gs. f \in \text{poly-carrier}$ 
    using Nil.prems(2) by (auto simp: poly-irreducibleD-carrier)
    have  $\text{poly-prod } gs = \mathbf{1}_P$  using Nil.prems(3) by simp
    then have  $g \otimes_P \text{poly-prod } gs' = \mathbf{1}_P$  using  $gs$  by simp

```

— So g divides 1_P — impossible for an irreducible.

moreover have $\text{poly-prod } gs' \in \text{poly-carrier}$
using $\text{gsc } gs$ **by** $(\text{auto intro!}: \text{poly-prod-closed})$
ultimately have $\text{poly-dvd } g \ 1_P$
unfolding poly-dvd-def
using $\text{poly-irreducibleD-carrier}[OF \ ig]$ **by** $(\text{metis poly-mult-comm})$
with $\text{poly-irreducible-not-dvd-one}[OF \ ig]$ **show** $\text{False} \dots$
qed
then show $?case$ **using** $\text{Nil.premis}(3)$ **by** simp
next
case $(\text{Cons } f \ fs')$
have $\text{iff}: \text{poly-irreducible } f$ **and** $\text{mf}: \text{poly-monic } f$ **using** $\text{Cons.premis}(1)$ **by** auto
have $\text{fc}: f \in \text{poly-carrier}$ **using** iff **by** $(\text{rule poly-irreducibleD-carrier})$
have $\text{fs'c}: \forall h \in \text{set } fs'. \text{poly-irreducible } h \wedge \text{poly-monic } h$ **using** $\text{Cons.premis}(1)$ **by** simp
have $\text{gsc}: \forall g \in \text{set } gs. \text{poly-irreducible } g \wedge \text{poly-monic } g$ **using** $\text{Cons.premis}(2)$.
have $\text{gscarr}: \forall g \in \text{set } gs. g \in \text{poly-carrier}$
using gsc **by** $(\text{auto simp: poly-irreducibleD-carrier})$
have $\text{fs'carr}: \forall h \in \text{set } fs'. h \in \text{poly-carrier}$
using fs'c **by** $(\text{auto simp: poly-irreducibleD-carrier})$
have $\text{fs'prod}: \text{poly-prod } fs' \in \text{poly-carrier}$
by $(\text{rule poly-prod-closed})$ $(\text{use } \text{fs'carr} \text{ in blast})$
— f divides $\text{poly-prod } gs = \text{poly-prod } (f \ \# \ fs')$, so divides some g in gs ; both monic irreducibles, hence $f = g$.
have $\text{poly-dvd } f \ (\text{poly-prod } gs)$
unfolding poly-dvd-def **using** $\text{Cons.premis}(3)$ fs'prod **by** $(\text{metis poly-prod-Cons poly-mult-comm } \text{fc})$
then obtain g **where** $g: g \in \text{set } gs$ **and** $\text{poly-dvd } f \ g$
using $\text{poly-irreducible-prime-list}[OF \ \text{iff } \text{gscarr}]$ **by** blast
have $\text{ig}: \text{poly-irreducible } g$ **and** $\text{mg}: \text{poly-monic } g$ **using** $\text{gsc } g$ **by** auto
have $\text{feq}: f = g$ **using** $\text{monic-irreducible-dvd-eq}[OF \ \text{iff } \text{ig } \text{mf } \text{mg } \langle \text{poly-dvd } f \ g \rangle]$.
— Cancel f : $\text{poly-prod } fs' = \text{poly-prod } (\text{remove1 } g \ gs)$.
have $\text{gc}: g \in \text{poly-carrier}$ **using** ig **by** $(\text{rule poly-irreducibleD-carrier})$
have $\text{rprod}: \text{poly-prod } (\text{remove1 } g \ gs) \in \text{poly-carrier}$
by $(\text{meson } \text{gscarr} \text{ notin-set-remove1 } \text{poly-prod-closed})$
have $\text{gnz}: g \neq 0_P$ **using** ig **by** $(\text{rule poly-irreducibleD-nonzero})$
have $g \otimes_P \text{poly-prod } fs' = g \otimes_P \text{poly-prod } (\text{remove1 } g \ gs)$
using $\text{Cons.premis}(3)$ $\text{feq } g \ \text{gscarr } \text{poly-prod-remove1}$ **by** auto
then have $\text{poly-prod } fs' = \text{poly-prod } (\text{remove1 } g \ gs)$
using $\text{poly-mult-left-cancel}[OF \ \text{gc } \text{gnz } \text{fs'prod } \text{rprod}]$ **by** simp
— Recurse on the remaining factors.
moreover have $\forall g' \in \text{set } (\text{remove1 } g \ gs). \text{poly-irreducible } g' \wedge \text{poly-monic } g'$
using gsc **by** $(\text{meson notin-set-remove1})$
ultimately show $?case$
using $\text{Cons.hyps}[OF \ \text{fs'c}]$
by $(\text{simp add: } \text{feq } g)$
qed
end

end

51 The prime fields $GF(p) = \mathbb{Z}/p\mathbb{Z}$

```
theory GF-p
  imports Ring-Theory Poly-Ring Poly-Ideal HOL-Computational-Algebra.Primes
begin
```

51.1 The residue rings $\mathbb{Z}/n\mathbb{Z}$

Ring-Theory suppresses HOL's arithmetic notation; restore it locally for the modular arithmetic below.

```
notation plus (infixl <+> 65)
notation minus (infixl <-> 65)
unbundle uminus-syntax
```

For any $1 < n$, the integers modulo n form a commutative ring: carrier $\{0..<n\}$ of *nat*, arithmetic modulo n . This is the abstract residue ring $\mathbb{Z}/n\mathbb{Z}$; when n is prime it is in fact a field, developed in *GF-p*. The bound $1 < n$ ensures both 0 and 1 lie in $\{0..<n\}$ and are distinct in *nat*; the trivial ring case $n = 1$ is not modelled by this carrier.

```
definition gfp-add :: nat ⇒ nat ⇒ nat ⇒ nat
  where gfp-add n x y = (x + y) mod n
```

```
definition gfp-mult :: nat ⇒ nat ⇒ nat ⇒ nat
  where gfp-mult n x y = (x * y) mod n
```

```
context
  fixes n :: nat
  assumes n-gt-1: n > 1
begin
```

```
lemma n-gt-0: n > 0 using n-gt-1 by simp
```

Closure: results of modular arithmetic land in $\{0..<n\}$.

```
lemma gfp-add-closed [simp]: gfp-add n x y ∈ {0..<n}
  using n-gt-0 by (simp add: gfp-add-def)
```

```
lemma gfp-mult-closed [simp]: gfp-mult n x y ∈ {0..<n}
  using n-gt-0 by (simp add: gfp-mult-def)
```

```
lemma zero-mem: (0::nat) ∈ {0..<n} and one-mem: (1::nat) ∈ {0..<n}
  using n-gt-1 by auto
```

The additive part is an abelian group: associative and commutative modular addition, with $n - x$ (reduced) as the inverse of x .

```
lemma gfp-add-group: Group {0..<n} (gfp-add n) 0
```



```

proof (rule GroupI)
  show  $\bigwedge x y z. \llbracket x \in \{0..<n\}; y \in \{0..<n\}; z \in \{0..<n\} \rrbracket$ 
     $\implies \text{gfp-add } n (\text{gfp-add } n x y) z = \text{gfp-add } n x (\text{gfp-add } n y z)$ 
    by (simp add: gfp-add-def mod-add-left-eq mod-add-right-eq add.assoc)
next
  fix  $x$  assume  $x: x \in \{0..<n\}$ 
  then have key:  $(x + (n - x) \bmod n) \bmod n = 0$ 
    by (simp-all add: mod-add-right-eq)
  then show  $\exists y \in \{0..<n\}. \text{gfp-add } n x y = 0 \wedge \text{gfp-add } n y x = 0$ 
    by (metis add.commute add.right-neutral gfp-add-closed gfp-add-def)
qed (use zero-mem n-gt-0 in ⟨simp-all add: add.commute gfp-add-def⟩)

lemma gfp-add-commutative-monoid: commutative-monoid  $\{0..<n\}$  (gfp-add n) 0
unfolding commutative-monoid-def
  by (metis Group-def add.commute commutative-monoid-axioms.intro gfp-add-def
    gfp-add-group)

lemma gfp-additive: Abelian-Group  $\{0..<n\}$  (gfp-add n) 0
  by (rule Abelian-Group.intro [OF gfp-add-group gfp-add-commutative-monoid])

```

The multiplicative part is a commutative monoid with unit 1.

```

lemma gfp-multiplicative: commutative-monoid  $\{0..<n\}$  (gfp-mult n) 1
proof
  show  $\bigwedge x y z. \llbracket x \in \{0..<n\}; y \in \{0..<n\}; z \in \{0..<n\} \rrbracket$ 
     $\implies \text{gfp-mult } n (\text{gfp-mult } n x y) z = \text{gfp-mult } n x (\text{gfp-mult } n y z)$ 
    by (simp add: gfp-mult-def mod-mult-left-eq mod-mult-right-eq mult.assoc)
qed (use one-mem n-gt-1 in ⟨simp-all add: mult.commute gfp-mult-def⟩)

lemma zfact-ring: Ring  $\{0..<n\}$  (gfp-add n) (gfp-mult n) 0 1
proof (intro Ring.intro gfp-additive)
  show Monoid  $\{0..<n\}$  (gfp-mult n) 1
    by (rule commutative-monoid.axioms(1) [OF gfp-multiplicative])
  show Ring-axioms  $\{0..<n\}$  (gfp-add n) (gfp-mult n)
    by unfold-locales
      (simp-all add: gfp-add-def gfp-mult-def mod-mult-left-eq mod-mult-right-eq
        add-mult-distrib add-mult-distrib2 mod-add-left-eq mod-add-right-eq)
qed

```

```

lemma zfact-commutative-ring: commutative-ring  $\{0..<n\}$  (gfp-add n) (gfp-mult n)
  0 1
  by (rule commutative-ring.intro [OF zfact-ring gfp-multiplicative])

```

The residue ring $\mathbb{Z}/n\mathbb{Z}$ has exactly n elements.

```

lemma card-zfact: card  $\{0..<n\} = n$  by simp

```

end

Hence, for every $1 < n$, the locale *commutative-ring* is non-vacuous with a witness of cardinality n .

51.2 The residue ring as a field

For a prime p , the residue ring $ZFact$ becomes a field: the only extra step over the general $0 < n$ case is that every nonzero residue has a multiplicative inverse, which follows from Bézout's identity since p is prime.

Carrier $\{0..<p\}$ of nat , arithmetic modulo p , arithmetic operations $gfp-add$ and $gfp-mult$ shared with the ring-level development in $ZFact$.

```

context
  fixes  $p :: nat$ 
  assumes  $prime-p$ :  $prime\ p$ 
begin

lemma  $p-gt-1$ :  $p > 1$  using  $prime-p$  by ( $simp\ add:\ prime-nat-iff$ )
lemma  $p-gt-0$ :  $p > 0$  using  $p-gt-1$  by  $simp$ 

Inverses. For  $a \in \{1..<p\}$ ,  $a$  is coprime to the prime  $p$ , so Bézout gives integers
with  $u \cdot a + v \cdot p = 1$ ; reducing  $u$  modulo  $p$  yields a multiplicative inverse of  $a$ .

lemma  $gfp-inverse-exists$ :
  assumes  $a$ :  $a \in \{0..<p\}$  and  $anz$ :  $a \neq 0$ 
  shows  $\exists b \in \{0..<p\}.$   $gfp-mult\ p\ a\ b = 1$ 
proof -
  have  $ap$ :  $a < p$  and  $a1$ :  $a \geq 1$  using  $a\ anz$  by  $auto$ 
  —  $p$  prime and  $a < p$ ,  $a \neq 0$ , so  $p$  does not divide  $a$ .
  have  $ndvd$ :  $\neg p\ dvd\ a$  using  $ap\ anz$  by ( $simp\ add:\ nat-dvd-not-less$ )
  then have  $gcd\ a\ p = 1$ 
  by ( $metis\ gcd-unique\ prime-nat-iff\ prime-p$ )
  then have  $cop$ :  $coprime\ (int\ a)\ (int\ p)$  by ( $simp\ add:\ coprime-iff-gcd-eq-1$ )
  — Bézout:  $u \cdot a + v \cdot p = 1$  for some integers  $u, v$ .
  obtain  $u\ v :: int$  where  $uv$ :  $u * int\ a + v * int\ p = 1$ 
  using  $cop\ bezout-int$  by ( $metis\ coprime-imp-gcd-eq-1$ )
  define  $b$  where  $b = nat\ (u\ mod\ int\ p)$ 
  have  $brange$ :  $b \in \{0..<p\}$  using  $p-gt-0$  by ( $simp\ add:\ b-def$ ) ( $simp\ add:\ nat-less-iff$ )
  — Then  $a \cdot b \equiv a \cdot u \equiv 1 \pmod{p}$ .
  have  $int\ (gfp-mult\ p\ a\ b) = (int\ a * u)\ mod\ int\ p$ 
  by ( $simp\ add:\ b-def\ gfp-mult-def\ mod-mult-right-eq\ of-nat-mod\ p-gt-0$ )
  also have  $\dots = (1 - v * int\ p + v * int\ p)\ mod\ int\ p$ 
  by ( $metis\ diff-add-cancel\ mod-mult-self1\ mult.commute\ uv$ )
  also have  $\dots = 1$  using  $p-gt-1$  by  $simp$ 
  finally show  $?thesis$  using  $brange$ 
  using  $of-nat-eq-1-iff$  by  $blast$ 
qed

lemma  $gfp-field$ :  $Field\ \{0..<p\}\ (gfp-add\ p)\ (gfp-mult\ p)\ 0\ 1$ 
proof -
  interpret  $R$ :  $commutative-ring\ \{0..<p\}\ gfp-add\ p\ gfp-mult\ p\ 0\ 1$ 
  using  $zfact-commutative-ring[OF\ p-gt-1]$  .
  show  $?thesis$ 
proof
  show  $(1::nat) \neq 0$  using  $p-gt-1$  by  $simp$ 

```

```

    show  $\bigwedge a. a \in \{0..<p\} \implies a \neq 0 \implies R.\text{multiplicative.invertible } a$ 
    by (metis R.multiplicative.invertibleI gfp-inverse-exists gfp-mult-def mult.commute)
  qed
qed

```

The prime field has exactly p elements.

```

lemma card-gfp:  $\text{card } \{0..<p\} = p$  by simp

```

```

end

```

Hence prime fields of every prime order exist — in particular the *Field* locale is non-vacuous for every prime cardinality.

51.3 The two-element field as a non-vacuity witness

A concrete field with two elements, witnessing that the *Field* and *integral-domain* locales (and hence the abstract '*a ring*' type as a field) are non-vacuous. Carrier $\{0, 1\}$ of *nat*.

```

definition gf2-add ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$  where gf2-add  $x\ y \equiv (x + y) \bmod 2$ 

```

```

definition gf2-mult ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$  where gf2-mult  $x\ y \equiv (x * y) \bmod 2$ 

```

Enumeration helper: membership in the two-element carrier.

```

lemma gf2-cases:  $x \in \{0, 1::\text{nat}\} \implies x = 0 \vee x = 1$  by auto

```

```

lemma gf2-add-simps [simp]:
  gf2-add 0 0 = 0 gf2-add 0 1 = 1 gf2-add 1 0 = 1 gf2-add 1 1 = 0
by (simp-all add: gf2-add-def)

```

```

lemma gf2-mult-simps [simp]:
  gf2-mult 0 0 = 0 gf2-mult 0 1 = 0 gf2-mult 1 0 = 0 gf2-mult 1 1 = 1
by (simp-all add: gf2-mult-def)

```

```

lemma gf2-add-closed [simp]:  $\llbracket x \in \{0, 1::\text{nat}\}; y \in \{0, 1\} \rrbracket \implies \text{gf2-add } x\ y \in \{0, 1\}$ 
by (auto simp: gf2-add-def)

```

```

lemma gf2-mult-closed [simp]:  $\llbracket x \in \{0, 1::\text{nat}\}; y \in \{0, 1\} \rrbracket \implies \text{gf2-mult } x\ y \in \{0, 1\}$ 
by (auto simp: gf2-mult-def)

```

The additive part is a group (each element is its own inverse), via *GroupI*.

```

lemma gf2-add-group: Group  $\{0, 1::\text{nat}\}$  gf2-add 0

```

```

proof (rule GroupI)

```

```

  fix  $u :: \text{nat}$  assume  $u \in \{0, 1\}$ 

```

```

  then have gf2-add  $u\ u = 0$  by (auto simp: gf2-add-def)

```

```

  with  $u$  show  $\exists v \in \{0, 1::\text{nat}\}. \text{gf2-add } u\ v = 0 \wedge \text{gf2-add } v\ u = 0$  by blast

```

```

qed (auto simp: gf2-add-def)

```

lemma *gf2-add-commutative-monoid: commutative-monoid* $\{0, 1::\text{nat}\}$ *gf2-add 0*
by *unfold-locales (auto simp: gf2-add-def)*

The additive part is therefore an abelian group.

lemma *gf2-additive: Abelian-Group* $\{0, 1::\text{nat}\}$ *gf2-add 0*
by (*rule Abelian-Group.intro [OF gf2-add-group gf2-add-commutative-monoid]*)

The multiplicative part is a commutative monoid.

lemma *gf2-multiplicative: commutative-monoid* $\{0, 1::\text{nat}\}$ *gf2-mult 1*
by *unfold-locales (auto simp: gf2-mult-def)*

lemma *gf2-ring: Ring* $\{0, 1::\text{nat}\}$ *gf2-add gf2-mult 0 1*
proof (*rule Ring.intro*)
show *Abelian-Group* $\{0, 1::\text{nat}\}$ *gf2-add 0* **by** (*rule gf2-additive*)
show *Monoid* $\{0, 1::\text{nat}\}$ *gf2-mult 1* **by** (*rule commutative-monoid.axioms(1) [OF gf2-multiplicative]*)
show *Ring-axioms* $\{0, 1::\text{nat}\}$ *gf2-add gf2-mult* **by** *unfold-locales (auto simp: gf2-add-def gf2-mult-def)*
qed

lemma *gf2-commutative-ring: commutative-ring* $\{0, 1::\text{nat}\}$ *gf2-add gf2-mult 0 1*
by (*rule commutative-ring.intro [OF gf2-ring gf2-multiplicative]*)

The two-element field. This is the single witness the downstream constructions consume (via *interpretation ... by (rule gf2-field)* in *GF4* and *GF8*); it also witnesses non-vacuity of the *Field*, *integral-domain* and *commutative-ring* locales, the last two by the locale hierarchy (a *Field* is in particular an *integral-domain*).

lemma *gf2-field: Field* $\{0, 1::\text{nat}\}$ *gf2-add gf2-mult 0 1*
proof –
interpret *R: commutative-ring* $\{0, 1::\text{nat}\}$ *gf2-add gf2-mult 0 1* **by** (*rule gf2-commutative-ring*)
show *?thesis*
proof
show $(1::\text{nat}) \neq 0$ **by** *simp*
next
fix *a :: nat* **assume** *a: a ∈ {0, 1}* *a ≠ 0*
then have *a1: a = 1* **by** *auto*
then have *gf2-mult a a = 1* **by** (*simp add: gf2-mult-def*)
then show *R.multiplicative.invertible a*
using *a1* **by** (*intro R.multiplicative.invertibleI [where v = a]*) *auto*
qed
qed

51.4 The two-element field with polynomial machinery

interpretation *GF2: Field* $\{0, 1::\text{nat}\}$ *gf2-add gf2-mult 0 1*
by (*rule gf2-field*)

lemma *GF2-poly-const-carrier [simp, intro]: GF2.poly-const 1 ∈ GF2.poly-carrier*
by (*rule GF2.poly-const-closed*) *simp*

lemma *GF2-monom-carrier* [*simp*, *intro*]: *GF2.monom 1 n* $\in$ *GF2.poly-carrier*
by (*rule GF2.monom-closed*) *simp*

51.5 The 3-element prime field with polynomial machinery

A single, canonical interpretation of the *Field* locale on $GF(3)$.

interpretation *GF3*: *Field* $\{0..<3::nat\}$ *gfp-add 3 gfp-mult 3 0 1*
using *gfp-field[of 3]* **by** *simp*

lemma *GF3-poly-const-carrier* [*simp*, *intro*]: *GF3.poly-const 1* $\in$ *GF3.poly-carrier*
by (*rule GF3.poly-const-closed*) *simp*

lemma *GF3-monom-carrier* [*simp*, *intro*]: *GF3.monom 1 n* $\in$ *GF3.poly-carrier*
by (*rule GF3.monom-closed*) *simp*

An end-to-end witness for the carrier-set field machinery, over a non-binary prime field: the polynomial $X^2 + 1$ is irreducible over $GF(3)$ (it has no root there, as -1 is not a square modulo 3), so by Kronecker's theorem the quotient $GF(3)[X] / (X^2 + 1)$ is a field, and by *Field.card-poly-quotient* it has $3^2 = 9$ elements — the field $GF(9)$.

The witness polynomial $\Psi = X^2 + 1$ over $GF(3)$.

definition *Psi* :: *nat* $\Rightarrow$ *nat*
where *Psi* $\equiv$ *GF3.poly-add (GF3.monom 1 2) (GF3.poly-const 1)*

lemma *Psi-coeff*: *Psi i* = (if $i = 0 \vee i = 2$ then 1 else 0)
by (*simp add: Psi-def GF3.poly-add-def GF3.monom-def GF3.poly-const-def gfp-add-def*)

lemma *Psi-closed*: *Psi* $\in$ *GF3.poly-carrier*
unfolding *Psi-def* **by** (*intro GF3.poly-add-closed GF3-monom-carrier GF3-poly-const-carrier*)

Psi has degree exactly 2.

lemma *degree-Psi*: *GF3.degree Psi* = 2

proof (*rule antisym*)
show *GF3.degree Psi* $\leq$ 2
by (*rule GF3.degree-leI[OF Psi-closed]*) (*simp add: Psi-coeff*)
have *Psi 2* $\neq$ 0 **by** (*simp add: Psi-coeff*)
moreover have *finite* $\{i. Psi i \neq 0\}$ **using** *Psi-closed* **by** (*rule GF3.poly-carrier-finite*)
ultimately show 2 $\leq$ *GF3.degree Psi*
by (*simp add: GF3.degree-def*) (*metis (mono-tags, lifting) Max-ge empty-iff mem-Collect-eq*)
qed

Psi has no root in $\{0..<3\}$: $\alpha^2 + 1$ is 1, 2, 2 at $\alpha = 0, 1, 2$ respectively — never 0.

lemma *Psi-no-root*: $\alpha \in \{0..<3::nat\} \implies GF3.eval \alpha Psi \neq 0$

proof —

```

assume  $a: \alpha \in \{0..<3::nat\}$ 
— Evaluate term by term.
have  $GF3.eval\ \alpha\ Psi = gfp-add\ 3\ (GF3.eval\ \alpha\ (GF3.monom\ 1\ 2))\ (GF3.eval\ \alpha\ (GF3.poly-const\ 1))$ 
unfolding  $Psi-def$  by (rule  $GF3.eval-add[OF\ GF3-monom-carrier\ GF3-poly-const-carrier\ a]$ )
also have  $GF3.eval\ \alpha\ (GF3.monom\ 1\ 2) = gfp-mult\ 3\ 1\ (GF3.rpow\ \alpha\ 2)$ 
using  $GF3.eval-monom\ a$  by blast
also have  $GF3.eval\ \alpha\ (GF3.poly-const\ 1) = 1$ 
using  $GF3.eval-const\ a$  by blast
finally have  $ev: GF3.eval\ \alpha\ Psi = gfp-add\ 3\ (gfp-mult\ 3\ 1\ (GF3.rpow\ \alpha\ 2))\ 1$  .
— Compute the square:  $\alpha^2 = \alpha \cdot \alpha$ .
have  $r2: GF3.rpow\ \alpha\ 2 = gfp-mult\ 3\ \alpha\ (gfp-mult\ 3\ \alpha\ 1)$ 
unfolding  $GF3.rpow-def$  by (simp add: eval-nat-numeral)
have  $val: GF3.eval\ \alpha\ Psi = Suc\ (\alpha * \alpha\ mod\ 3)\ mod\ 3$ 
using  $ev\ r2$  by (simp add: gfp-add-def gfp-mult-def mod-mult-right-eq mod-mult-left-eq)
— Check the three residues  $\alpha = 0, 1, 2$  explicitly.
have  $\alpha = 0 \vee \alpha = 1 \vee \alpha = 2$  using  $a$  by auto
then show  $GF3.eval\ \alpha\ Psi \neq 0$  using  $val$  by auto
qed

```

Therefore Psi is irreducible over $GF(3)$.

```

theorem  $Psi-irreducible: GF3.poly-irreducible\ Psi$ 
using  $GF3.degree23-no-root-irreducible[OF\ Psi-closed]\ degree-Psi\ Psi-no-root$  by blast

```

51.6 The four-element field $GF(4)$ via Kronecker's construction

The polynomial $X^2 + X + 1$ is irreducible over the two-element field $GF(2)$ (it has no root there), so by Kronecker's theorem (*Field.poly-quotient-field*) the quotient $GF(2)[X] / (X^2 + X + 1)$ is a field — the four-element field $GF(4)$.

The witness polynomial $\Phi = X^2 + X + 1$ over $GF(2)$.

```

definition  $Phi :: nat \Rightarrow nat$ 
where  $Phi \equiv GF2.poly-add\ (GF2.poly-add\ (GF2.monom\ 1\ 2)\ (GF2.monom\ 1\ 1))\ (GF2.poly-const\ 1)$ 

```

Its coefficients: 1 at degrees 0, 1, 2, and 0 above.

```

lemma  $Phi-coeff: Phi\ i = (if\ i \leq 2\ then\ 1\ else\ 0)$ 
by (simp add:  $Phi-def\ GF2.poly-add-def\ GF2.monom-def\ GF2.poly-const-def\ gf2-add-def$ )

```

```

lemma  $Phi-closed: Phi \in GF2.poly-carrier$ 
unfolding  $Phi-def$  by (intro  $GF2.poly-add-closed\ GF2-monom-carrier\ GF2-poly-const-carrier$ )

```

Phi has degree exactly 2.

```

lemma  $degree-Phi: GF2.degree\ Phi = 2$ 
proof (rule antisym)

```

```

show GF2.degree Phi ≤ 2
  using GF2.degree-le-iff Phi-closed Phi-coeff by auto
have finite {i. Phi i ≠ 0} using Phi-closed by (rule GF2.poly-carrier-finite)
then show 2 ≤ GF2.degree Phi
  using GF2.coeff-gt-degree Phi-closed Phi-coeff linorder-not-le by fastforce
qed

```

Φ has no root in $\{0, 1\}$: both candidate values evaluate to 1, since $X^2 + X + 1$ at 0 is 1 and at 1 is $1 + 1 + 1 = 1$ modulo 2.

lemma *Phi-no-root*:

```

assumes a: α ∈ {0, 1::nat}
shows GF2.eval α Phi ≠ 0
proof —
have sum2: GF2.poly-add (GF2.monom 1 2) (GF2.monom 1 1) ∈ GF2.poly-carrier
  by (intro GF2.poly-add-closed GF2-monom-carrier)
  — Evaluate term by term, using additivity and GF2.eval-monom.
have GF2.eval α Phi
  = gf2-add (GF2.eval α (GF2.poly-add (GF2.monom 1 2) (GF2.monom 1 1)))
  (GF2.eval α (GF2.poly-const 1))
  unfolding Phi-def using GF2.eval-add[OF sum2 GF2-poly-const-carrier assms] .
also have ... = gf2-add (gf2-add (gf2-mult 1 (GF2.rpow α 2)) (gf2-mult 1 (GF2.rpow
α 1))) 1
  using GF2.eval-add[OF GF2-monom-carrier GF2-monom-carrier assms]
  GF2.eval-const[OF assms] GF2.eval-monom[OF - assms] by simp
finally have GF2.eval α Phi
  = gf2-add (gf2-add (gf2-mult 1 (GF2.rpow α 2)) (gf2-mult 1 (GF2.rpow α
1))) 1 .
  — Compute the powers: α2 = α·α, α1 = α.
then show GF2.eval α Phi ≠ 0
  using gf2-cases[OF a] gf2-add-simps(4) gf2-mult-simps(4)
  unfolding GF2.rpow-def by (fastforce simp add: eval-nat-numeral)
qed

```

Therefore Φ is irreducible over $GF(2)$.

theorem *Phi-irreducible*: $GF2.poly-irreducible \Phi$

```

using GF2.degree23-no-root-irreducible[OF Phi-closed] degree-Phi Phi-no-root by
blast

```

Kronecker. The principal ideal (Φ) is maximal in $GF(2)[X]$, so the quotient $GF(2)[X]/(\Phi)$ is a field — a four-element field, $GF(4)$.

theorem *GF4-is-field*:

```

ideal-in-comm-ring (GF2.poly-pideal Phi) GF2.poly-carrier GF2.poly-add GF2.poly-mult
GF2.poly-zero GF2.poly-one
  ∧ Ideal.maximal-ideal (GF2.poly-pideal Phi) GF2.poly-carrier GF2.poly-add GF2.poly-mult
GF2.poly-zero GF2.poly-one
using GF2.poly-quotient-field[OF Phi-irreducible] .

```

51.7 The eight-element field $GF(8)$ via Kronecker's construction

A cubic witness for the carrier-set field machinery: the polynomial $X^3 + X + 1$ is irreducible over $GF(2)$ (degree 3 with no root), so by Kronecker's theorem the quotient $GF(2)[X] / (X^3 + X + 1)$ is a field, and by *Field.card-poly-quotient* it has $2^3 = 8$ elements — the field $GF(8)$.

The witness polynomial $\Theta = X^3 + X + 1$ over $GF(2)$.

definition *Theta* :: nat $\Rightarrow$ nat

where *Theta* = *GF2.poly-add* (*GF2.poly-add* (*GF2.monom* 1 3) (*GF2.monom* 1 1)) (*GF2.poly-const* 1)

lemma *Theta-coeff*: *Theta* *i* = (if *i* = 0 $\vee$ *i* = 1 $\vee$ *i* = 3 then 1 else 0)

by (*simp add*: *Theta-def* *GF2.poly-add-def* *GF2.monom-def* *GF2.poly-const-def* *gf2-add-def*)

lemma *Theta-closed*: *Theta* $\in$ *GF2.poly-carrier*

unfolding *Theta-def* **by** (*intro* *GF2.poly-add-closed* *GF2-monom-carrier* *GF2-poly-const-carrier*)

Theta has degree exactly 3.

lemma *degree-Theta*: *GF2.degree* *Theta* = 3

proof (*rule antisym*)

show *GF2.degree* *Theta* $\leq$ 3

by (*rule* *GF2.degree-leI*[*OF* *Theta-closed*]) (*simp add*: *Theta-coeff*)

have *Theta* 3 $\neq$ 0 **by** (*simp add*: *Theta-coeff*)

moreover have *finite* {*i*. *Theta* *i* $\neq$ 0} **using** *Theta-closed* **by** (*rule* *GF2.poly-carrier-finite*)

ultimately show 3 $\leq$ *GF2.degree* *Theta*

by (*simp add*: *GF2.degree-def*) (*metis* (*mono-tags*, *lifting*) *Max-ge empty-iff mem-Collect-eq*)

qed

Theta has no root in {0, 1}: $\alpha^3 + \alpha + 1$ is 1 at both 0 and 1 (modulo 2).

lemma *Theta-no-root*: $\alpha \in \{0, 1::\text{nat}\} \implies GF2.eval \alpha \text{ Theta} \neq 0$

proof —

assume *a*: $\alpha \in \{0, 1::\text{nat}\}$

have *sum2*: *GF2.poly-add* (*GF2.monom* 1 3) (*GF2.monom* 1 1) $\in$ *GF2.poly-carrier*

by (*intro* *GF2.poly-add-closed* *GF2-monom-carrier*)

— Evaluate term by term.

have *GF2.eval* α *Theta*

= *gf2-add* (*GF2.eval* α (*GF2.poly-add* (*GF2.monom* 1 3) (*GF2.monom* 1 1)))

(*GF2.eval* α (*GF2.poly-const* 1))

unfolding *Theta-def*

using *GF2.eval-add*[*OF* *sum2* *GF2-poly-const-carrier* *a*].

also have *GF2.eval* α (*GF2.poly-add* (*GF2.monom* 1 3) (*GF2.monom* 1 1))

= *gf2-add* (*GF2.eval* α (*GF2.monom* 1 3)) (*GF2.eval* α (*GF2.monom* 1

1))

using *GF2.eval-add* *a* **by** *blast*

also have *GF2.eval* α (*GF2.monom* 1 3) = *gf2-mult* 1 (*GF2.rpow* α 3)

using *GF2.eval-monom* *a* **by** *blast*

also have *GF2.eval* α (*GF2.monom* 1 1) = *gf2-mult* 1 (*GF2.rpow* α 1)


```

    using GF2.eval-monom a by blast
  also have GF2.eval  $\alpha$  (GF2.poly-const 1) = 1
    using GF2.eval-const a by blast
  finally have ev: GF2.eval  $\alpha$  Theta
    = gf2-add (gf2-add (gf2-mult 1 (GF2.rpow  $\alpha$  3)) (gf2-mult 1 (GF2.rpow  $\alpha$ 
1))) 1 .
  then have GF2.eval  $\alpha$  Theta = gf2-add (gf2-add (gf2-mult 1 (gf2-mult  $\alpha$  (gf2-mult
 $\alpha$  (gf2-mult  $\alpha$  1))))
    (gf2-mult 1 (gf2-mult  $\alpha$  1))) 1
    unfolding GF2.rpow-def by (simp add: eval-nat-numeral)
  then show GF2.eval  $\alpha$  Theta  $\neq$  0
    using gf2-cases[OF a] by (auto simp: gf2-add-def gf2-mult-def)
qed

```

Therefore *Theta* is irreducible over $GF(2)$.

```

theorem Theta-irreducible: GF2.poly-irreducible Theta
  using GF2.degree23-no-root-irreducible[OF Theta-closed] degree-Theta Theta-no-root
  by blast

```

Kronecker. The quotient $GF(2)[X]/(\Theta)$ is a field with $2^3 = 8$ elements.

```

theorem GF8-is-field:
  ideal-in-comm-ring (GF2.poly-pideal Theta) GF2.poly-carrier (GF2.poly-add) (GF2.poly-mult)
  GF2.poly-zero GF2.poly-one
   $\wedge$  Ideal.maximal-ideal (GF2.poly-pideal Theta) GF2.poly-carrier (GF2.poly-add)
  (GF2.poly-mult) GF2.poly-zero GF2.poly-one
  using GF2.poly-quotient-field[OF Theta-irreducible] .

```

theorem card-GF8:

```

  card (Ideal.quotient-set (GF2.poly-pideal Theta) GF2.poly-carrier (GF2.poly-add)
  GF2.poly-zero) = 8
proof -
  have pnz: Theta  $\neq$  GF2.poly-zero
    using degree-Theta by (metis GF2.degree-zero zero-neq-numeral)
  have card (Ideal.quotient-set (GF2.poly-pideal Theta) GF2.poly-carrier (GF2.poly-add)
  GF2.poly-zero)
    = card {0, 1::nat}  $\wedge$  GF2.degree Theta
    by (rule GF2.card-poly-quotient[OF - Theta-closed pnz]) simp
  also have ... = 8 by (simp add: degree-Theta eval-nat-numeral)
  finally show ?thesis .
qed

```

51.8 The nine-element field $GF(9)$ via Kronecker's construction

Kronecker. The quotient $GF(3)[X]/(\Psi)$ is a field with $3^2 = 9$ elements.

theorem GF9-is-field:

```

  ideal-in-comm-ring (GF3.poly-pideal Psi) GF3.poly-carrier (GF3.poly-add) (GF3.poly-mult)
  GF3.poly-zero GF3.poly-one

```

```

     $\wedge$  Ideal.maximal-ideal (GF3.poly-pideal Psi) GF3.poly-carrier (GF3.poly-add) (GF3.poly-mult)
    GF3.poly-zero GF3.poly-one
    using GF3.poly-quotient-field[OF Psi-irreducible] .

```

theorem card-GF9:

```

    card (Ideal.quotient-set (GF3.poly-pideal Psi) GF3.poly-carrier (GF3.poly-add) GF3.poly-zero)
    = 9

```

proof –

```

    have pnz: Psi  $\neq$  GF3.poly-zero
    using degree-Psi by (metis GF3.degree-zero zero-neq-numeral)
    have card (Ideal.quotient-set (GF3.poly-pideal Psi) GF3.poly-carrier (GF3.poly-add)
    GF3.poly-zero)
    = card {0.. $\leq$ 3::nat}  $\wedge$  GF3.degree Psi
    by (rule GF3.card-poly-quotient[OF - Psi-closed pnz]) simp
    also have ... = 9 by (simp add: degree-Psi)
    finally show ?thesis .

```

qed

end

52 Divisibility in commutative rings: units, associates, irreducibles, primes

theory Divisibility-Theory

imports Ideal-Theory

begin

Elementary multiplicative divisibility theory for a commutative ring, in the locale idiom of *Ring-Theory*. We work directly with ring elements rather than lifting the multiplicative monoid, which keeps the statements close to textbook divisibility.

context commutative-ring

begin

52.1 Divisibility and units

divides a b: *a* divides *b*, i.e. $b = a \cdot c$ for some ring element *c*.

definition divides :: 'a $\Rightarrow$ 'a $\Rightarrow$ bool

where divides a b $\equiv (\exists c \in R. b = a \cdot c)$

lemma dividesI [intro]: $\llbracket c \in R; b = a \cdot c \rrbracket \Longrightarrow \text{divides } a \ b$

unfolding divides-def by blast

lemma dividesE [elim]:

assumes divides a b obtains c where $c \in R$ $b = a \cdot c$

using assms unfolding divides-def by blast

lemma *divides-refl* [simp]: $a \in R \implies \text{divides } a \ a$
using *Monoid.unit-closed multiplicative.Monoid-axioms* **by** *fastforce*

lemma *divides-trans* [trans]:
assumes *divides a b divides b c* **and** *a: a ∈ R*
shows *divides a c*
using *a assms divides-def* **by** *auto*

lemma *divides-zero* [simp]: $a \in R \implies \text{divides } a \ 0$
by (rule *dividesI[of 0]*) *simp-all*

lemma *one-divides* [simp]: $a \in R \implies \text{divides } 1 \ a$
by (simp add: *divides-def*)

lemma *divides-mult-right*:
assumes $b \in R$ **shows** *divides a (a · b)*
using *assms* **by** *blast*

A unit is an invertible element of the multiplicative monoid.

abbreviation *is-unit* :: $'a \Rightarrow \text{bool}$
where *is-unit u* $\equiv$ *multiplicative.invertible u*

lemma *is-unitI* [intro]: $\llbracket u \cdot v = 1; v \cdot u = 1; v \in R \rrbracket \implies \text{is-unit } u$
by (rule *multiplicative.invertibleI*)

lemma *unit-divides-one*:
assumes *is-unit u* **shows** *divides u 1*
using *assms* **by** *auto*

lemma *one-divides-unit-iff*:
assumes $u: u \in R$ **shows** *divides u 1* $\longleftrightarrow$ *is-unit u*
using *is-unitI multiplicative.commutative u* **by** *blast*

lemma *unit-divides-all*:
assumes *is-unit u* $u \in R$ $a \in R$ **shows** *divides u a*
using *assms divides-trans one-divides one-divides-unit-iff* **by** *meson*

52.2 Associated elements

Two elements are associated if each divides the other.

definition *associated* :: $'a \Rightarrow \text{bool}$
where *associated a b* $\longleftrightarrow$ *divides a b* $\wedge$ *divides b a*

lemma *associatedI* [intro]: $\llbracket \text{divides } a \ b; \text{divides } b \ a \rrbracket \implies \text{associated } a \ b$
unfolding *associated-def* **by** *blast*

lemma *associatedD1*: *associated a b* $\implies \text{divides } a \ b$
unfolding *associated-def* **by** *blast*

lemma *associatedD2*: $\text{associated } a \ b \implies \text{divides } b \ a$
unfolding *associated-def* **by** *blast*

lemma *associated-refl* [*simp*]: $a \in R \implies \text{associated } a \ a$
by (*simp add: associated-def*)

lemma *associated-sym*: $\text{associated } a \ b \implies \text{associated } b \ a$
unfolding *associated-def* **by** *blast*

lemma *associated-trans* [*trans*]:
assumes *associated* $a \ b$ *associated* $b \ c$ $a \in R \ b \in R \ c \in R$
shows *associated* $a \ c$
using *assms* **unfolding** *associated-def* **by** (*meson divides-trans*)

52.3 Greatest common divisors and least common multiples

is-gcd $d \ a \ b$: d is a common divisor of a and b that every common divisor divides. Gcds need not exist in a general ring, but when they do they are unique up to association; existence is established in a principal ideal domain (Bezout) below.

definition *is-gcd* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$
where *is-gcd* $d \ a \ b \longleftrightarrow$
 $d \in R \wedge \text{divides } d \ a \wedge \text{divides } d \ b \wedge (\forall c \in R. \text{divides } c \ a \wedge \text{divides } c \ b \longrightarrow \text{divides } c \ d)$

is-lcm $m \ a \ b$: m is a common multiple of a and b dividing every common multiple.

definition *is-lcm* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$
where *is-lcm* $m \ a \ b \longleftrightarrow$
 $m \in R \wedge \text{divides } a \ m \wedge \text{divides } b \ m \wedge (\forall c \in R. \text{divides } a \ c \wedge \text{divides } b \ c \longrightarrow \text{divides } m \ c)$

lemma *is-gcdI*:
 $\llbracket d \in R; \text{divides } d \ a; \text{divides } d \ b; \bigwedge c. \llbracket c \in R; \text{divides } c \ a; \text{divides } c \ b \rrbracket \implies \text{divides } c \ d \rrbracket \implies \text{is-gcd } d \ a \ b$
unfolding *is-gcd-def* **by** *blast*

lemma *is-gcdD*:
assumes *is-gcd* $d \ a \ b$
shows $d \in R$ *divides* $d \ a$ *divides* $d \ b$
and $\bigwedge c. \llbracket c \in R; \text{divides } c \ a; \text{divides } c \ b \rrbracket \implies \text{divides } c \ d$
using *assms* **unfolding** *is-gcd-def* **by** *blast+*

lemma *is-gcd-sym*: $\text{is-gcd } d \ a \ b \implies \text{is-gcd } d \ b \ a$
unfolding *is-gcd-def* **by** *blast*

lemma *is-lcm-sym*: $\text{is-lcm } m \ a \ b \implies \text{is-lcm } m \ b \ a$
unfolding *is-lcm-def* **by** *blast*

Any two gcds of the same pair are associated.

```

lemma is-gcd-unique:
  assumes is-gcd d a b and is-gcd d' a b
  shows associated d d'
  using assms associated-def is-gcd-def by force

```

Dually, any two lcms are associated.

```

lemma is-lcm-unique:
  assumes is-lcm m a b and is-lcm m' a b
  shows associated m m'
  using assms associated-def is-lcm-def by force

```

end

52.4 Irreducible and prime elements

```

context commutative-ring
begin

```

An irreducible element is a nonzero non-unit whose only factorisations are trivial.

```

definition irreducible-elem :: 'a  $\Rightarrow$  bool
where irreducible-elem p  $\longleftrightarrow$ 
   $p \in R \wedge p \neq \mathbf{0} \wedge \neg \text{is-unit } p \wedge$ 
   $(\forall a \in R. \forall b \in R. p = a \cdot b \longrightarrow \text{is-unit } a \vee \text{is-unit } b)$ 

```

```

lemma irreducible-elemI:
  assumes  $p \in R$   $p \neq \mathbf{0}$   $\neg \text{is-unit } p$ 
  and  $\bigwedge a b. \llbracket a \in R; b \in R; p = a \cdot b \rrbracket \Longrightarrow \text{is-unit } a \vee \text{is-unit } b$ 
  shows irreducible-elem p
  using assms unfolding irreducible-elem-def by blast

```

```

lemma irreducible-elemD:
  assumes irreducible-elem p
  shows  $p \in R$   $p \neq \mathbf{0}$   $\neg \text{is-unit } p$ 
  and  $\bigwedge a b. \llbracket a \in R; b \in R; p = a \cdot b \rrbracket \Longrightarrow \text{is-unit } a \vee \text{is-unit } b$ 
  using assms unfolding irreducible-elem-def by blast+

```

A prime element is a nonzero non-unit that, when dividing a product, divides a factor.

```

definition prime-elem :: 'a  $\Rightarrow$  bool
where prime-elem p  $\longleftrightarrow$ 
   $p \in R \wedge p \neq \mathbf{0} \wedge \neg \text{is-unit } p \wedge$ 
   $(\forall a \in R. \forall b \in R. \text{divides } p (a \cdot b) \longrightarrow \text{divides } p a \vee \text{divides } p b)$ 

```

```

lemma prime-elemI:
  assumes  $p \in R$   $p \neq \mathbf{0}$   $\neg \text{is-unit } p$ 
  and  $\bigwedge a b. \llbracket a \in R; b \in R; \text{divides } p (a \cdot b) \rrbracket \Longrightarrow \text{divides } p a \vee \text{divides } p b$ 
  shows prime-elem p
  using assms unfolding prime-elem-def by blast

```

```

lemma prime-elimD:
  assumes prime-elim p
  shows  $p \in R \ p \neq \mathbf{0} \neg \text{is-unit } p$ 
    and  $\bigwedge a \ b. \llbracket a \in R; b \in R; \text{divides } p \ (a \cdot b) \rrbracket \implies \text{divides } p \ a \vee \text{divides } p \ b$ 
  using assms unfolding prime-elim-def by blast+

end

```

52.5 In an integral domain, prime elements are irreducible

```

context integral-domain
begin

```

Cancellation of a nonzero factor: the defining property that separates domains from general commutative rings. From $c \cdot a = c \cdot b$ with $c \neq \mathbf{0}$ we get $c \cdot (a - b) = \mathbf{0}$, and absence of zero divisors forces $a - b = \mathbf{0}$.

```

lemma mult-cancel-left:
  assumes c:  $c \in R$  and a:  $a \in R$  and b:  $b \in R$  and cnz:  $c \neq \mathbf{0}$  and eq:  $c \cdot a = c \cdot b$ 
  shows  $a = b$ 
proof -
  have  $c \cdot (a - b) = c \cdot a + c \cdot (-b)$  using c a b by (simp add: distributive)
  then have  $c \cdot (a - b) = \mathbf{0}$ 
    by (simp add: b c eq local.right-minus)
  then have  $c = \mathbf{0} \vee a - b = \mathbf{0}$ 
    using c no-zero-divisors by (simp add: a b)
  then have z:  $a - b = \mathbf{0}$  using cnz by blast
  then show ?thesis
    using a additive.commutative additive.inverse-equality b by fastforce
qed

```

The classical characterization of association in a domain: a and b are associated iff they differ by a unit factor, $b = u \cdot a$. (The right-to-left direction holds in any commutative ring; the forward direction is where cancellation — hence the domain hypothesis — is used.)

```

lemma associated-iff-unit-multiple:
  assumes a:  $a \in R$  and b:  $b \in R$ 
  shows  $\text{associated } a \ b \longleftrightarrow (\exists u. \text{is-unit } u \wedge u \in R \wedge b = u \cdot a)$ 
proof
  assume associated a b
  then have ab: divides a b and ba: divides b a
    by (auto dest: associatedD1 associatedD2)
  obtain s where s:  $s \in R \ b = a \cdot s$  using ab by blast
  obtain t where t:  $t \in R \ a = b \cdot t$  using ba by blast
  show  $\exists u. \text{is-unit } u \wedge u \in R \wedge b = u \cdot a$ 
proof (cases a = 0)
  case True
  then show ?thesis using s a by force

```

```

next
  case False
  have eq1:  $a \cdot 1 = a \cdot (s \cdot t)$ 
    using multiplicative.associative a s t by force
  have stR:  $s \cdot t \in R$  using s t by simp
  have one-st:  $1 = s \cdot t$ 
    by (rule mult-cancel-left[OF a multiplicative.unit-closed stR False eq1])
  then show ?thesis using s(1)
    using a is-unitI multiplicative.commutative s(2) t(1) by blast
qed
next
  assume  $\exists u. \text{is-unit } u \wedge u \in R \wedge b = u \cdot a$ 
  then obtain u where u:  $\text{is-unit } u \wedge u \in R$  and beq:  $b = u \cdot a$  and iuR: multiplicative.inverse  $u \in R$ 
    by blast
  show associated a b
  proof (rule associatedI)
    — divides a b since  $b = u \cdot a = a \cdot u$ .
  show divides a b
    using a beq multiplicative.commutative u(2) by blast
    — divides b a since  $a = u^{-1} \cdot b$ .
  show divides b a
    unfolding divides-def
    using a b beq iuR multiplicative.commutative multiplicative.invertible-left-inverse2
  u
    by metis
  qed
qed

lemma prime-imp-irreducible:
  assumes p: prime-elem p
  shows irreducible-elem p
proof (rule irreducible-elemI)
  show pR:  $p \in R$  and pnz:  $p \neq 0$  and pnu:  $\neg \text{is-unit } p$ 
    using p by (auto dest: prime-elemD)
  fix a b assume a:  $a \in R$  and b:  $b \in R$  and eq:  $p = a \cdot b$ 
  — p divides  $a \cdot b$ , so it divides one factor; say p divides a, giving  $a = p \cdot d$ .
  then have divides p a  $\vee$  divides p b using a b p by (auto simp: prime-elem-def)
  then show is-unit a  $\vee$  is-unit b
  proof
    assume divides p a
    then obtain d where d:  $d \in R \wedge a = p \cdot d$  by blast
    have  $p \cdot 1 = p \cdot (d \cdot b)$ 
      using eq d a b pR by (simp add: multiplicative.associative)
    then have db:  $d \cdot b = 1$  using mult-cancel-left[OF pR - - pnz] d b pR by simp
    then show ?thesis
      using b d(1) is-unitI multiplicative.commutative by blast
  next
    assume divides p b
    then obtain d where d:  $d \in R \wedge b = p \cdot d$  by blast

```

```

then have p · 1 = p · (d · a)
  using a eq multiplicative.commutative multiplicative.left-commutative pR by force

then have da: d · a = 1 using mult-cancel-left[OF pR - - pnz] d a pR by simp
then show ?thesis
  using a d(1) is-unitI multiplicative.commutative by blast
qed
qed

end

```

52.6 Divisibility via principal ideals

Divisibility is containment of principal ideals the other way round: a divides b iff $(b) \subseteq (a)$ iff $b \in (a)$. This is the bridge that turns the ideal-theoretic PID argument into statements about ring elements.

```

context commutative-ring
begin

```

```

lemma divides-iff-mem-principal:
  assumes a:  $a \in R$  and b:  $b \in R$ 
  shows divides a b  $\longleftrightarrow b \in \text{principal-ideal } a$ 
  using a divides-def multiplicative.commutative principal-ideal-def by auto

```

```

lemma divides-iff-principal-subset:
  assumes a:  $a \in R$  and b:  $b \in R$ 
  shows divides a b  $\longleftrightarrow \text{principal-ideal } b \subseteq \text{principal-ideal } a$ 
proof
  assume dvd: divides a b
  show principal-ideal b  $\subseteq$  principal-ideal a
  proof
    fix x assume x  $\in$  principal-ideal b
    then obtain r c where  $r \in R$   $x = r \cdot b$  and  $c \in R$   $b = a \cdot c$ 
    unfolding principal-ideal-def using dvd by blast
    then show x  $\in$  principal-ideal a using principal-ideal-memI[of r · c a]
    using a multiplicative.associative multiplicative.commutative by auto
  qed
next
  assume principal-ideal b  $\subseteq$  principal-ideal a
  moreover have  $b \in \text{principal-ideal } b$  using b by (simp add: principal-ideal-contains)
  ultimately have  $b \in \text{principal-ideal } a$  by blast
  then show divides a b using divides-iff-mem-principal[OF a b] by simp
qed

```

Associated elements generate the same principal ideal.

```

lemma associated-iff-same-principal:
  assumes a:  $a \in R$  and b:  $b \in R$ 
  shows associated a b  $\longleftrightarrow \text{principal-ideal } a = \text{principal-ideal } b$ 
  unfolding associated-def

```


using *divides-iff-principal-subset*[*OF a b*] *divides-iff-principal-subset*[*OF b a*] **by** *blast*

An ideal containing **1** is the whole ring.

lemma *ideal-contains-one-eq-whole*:

assumes *I*: *Ideal I R (+) (·) 0 1* **and** *one*: **1** $\in I$

shows $I = R$

proof –

interpret *I*: *Ideal I R (+) (·) 0 1* **by** (*rule I*)

show *?thesis* **using** *one I.additive.subset*

by (*metis I.Ideal(1) multiplicative.right-unit subsetI subset-antisym*)

qed

A principal ideal is the whole ring exactly when its generator is a unit.

lemma *principal-ideal-eq-whole-iff-unit*:

assumes *a*: $a \in R$

shows *principal-ideal a = R* $\longleftrightarrow$ *is-unit a*

proof

assume *principal-ideal a = R*

then have **1** $\in$ *principal-ideal a* **by** *simp*

then obtain *r* **where** *r*: $r \in R$ $r \cdot a = 1$ $a \cdot r = 1$

unfolding *principal-ideal-def*

using *assms multiplicative.commutative* **by** *auto*

then show *is-unit a*

using *is-unitI* **by** *blast*

next

assume *u*: *is-unit a*

have **1** $\in$ *principal-ideal a*

using *multiplicative.invertibleE principal-ideal-memI u* **by** *metis*

then show *principal-ideal a = R*

using *assms ideal-proper-iff-one-notin principal-ideal-is-ideal* **by** *blast*

qed

end

52.7 Noetherian rings

A (two-sided) noetherian ring is one in which every ascending chain of ideals eventually stabilises. This is the ascending-chain formulation of noetherianity, and is the form used by the factorisation arguments below.

locale *Noetherian-Ring* = *Ring* +

assumes *ideal-chain-stabilises*:

$\llbracket \bigwedge n. \text{Ideal } (A\ n) \ R \ (+) \ (\cdot) \ 0 \ 1; \bigwedge n. A\ n \subseteq A\ (Suc\ n) \rrbracket \implies$

$\exists N. \forall n. N \leq n \longrightarrow A\ n = A\ N$

A noetherian domain is both noetherian and an integral domain.

locale *Noetherian-Domain* = *Noetherian-Ring* + *integral-domain*

52.8 Principal ideal domains

A principal ideal domain (PID) is an integral domain in which every ideal is principal.

```
locale principal-ideal-domain = integral-domain +
  assumes principal: Ideal I R (+) (·) 0 1  $\implies \exists a \in R. I = \text{principal-ideal } a$ 
```

```
context principal-ideal-domain
begin
```

In a PID an irreducible element generates a maximal ideal. We use the maximality criterion of *Ideal*: any ideal J containing (p) is principal, $J = (q)$, so q divides p ; by irreducibility one factor is a unit, giving $J = (p)$ or $J = R$.

theorem *irreducible-imp-maximal-ideal*:

```
  assumes p: irreducible-elem p
  shows Ideal.maximal-ideal (principal-ideal p) R (+) (·) 0 1
```

proof –

```
  have pR:  $p \in R$  and pnu:  $\neg \text{is-unit } p$  using p by (auto dest: irreducible-elemD)
```

```
  interpret P: Ideal principal-ideal p R (+) (·) 0 1
```

```
    by (rule principal-ideal-is-ideal[OF pR])
```

```
  show ?thesis
```

```
    unfolding P.maximal-ideal-def
```

```
  proof (intro conjI allI impI)
```

```
    show principal-ideal  $p \neq R$  using principal-ideal-eq-whole-iff-unit[OF pR] pnu by
```

```
blast
```

```
  next
```

```
    fix J assume J: Ideal J R (+) (·) 0 1 and sub: principal-ideal  $p \subseteq J$ 
```

```
    obtain q where q:  $q \in R$   $J = \text{principal-ideal } q$  using principal[OF J] by blast
```

```
    —  $(p) \subseteq (q)$  means  $q$  divides  $p$ , so  $p = q \cdot r$ .
```

```
    have principal-ideal  $p \subseteq \text{principal-ideal } q$  using sub q by simp
```

```
    then have divides q p using divides-iff-principal-subset[OF q(1) pR] by simp
```

```
    then obtain r where r:  $r \in R$   $p = q \cdot r$  by blast
```

```
    have is-unit  $q \vee \text{is-unit } r$  using p q(1) r by (auto dest: irreducible-elemD)
```

```
    then show  $J = \text{principal-ideal } p \vee J = R$ 
```

```
  proof
```

```
    assume is-unit q
```

```
    then show ?thesis using principal-ideal-eq-whole-iff-unit q by blast
```

```
  next
```

```
    assume ur: is-unit r
```

```
    have  $p \cdot \text{multiplicative.inverse } r = q$ 
```

```
      by (simp add: multiplicative.associative q(1) r ur)
```

```
    then have divides p q
```

```
      using r(1) ur by auto
```

```
    then show ?thesis using q(2)
```

```
      using divides-iff-principal-subset pR q(1) sub by blast
```

```
  qed
```

```
qed
```

```
qed
```

Hence in a PID every irreducible element is prime (the converse of $\llbracket \text{integral-domain } ?R \text{ ?addition ?multiplication ?zero ?unit; commutative-ring.prime-elem } ?R \text{ ?multiplication ?zero ?unit ?p} \rrbracket \implies \text{commutative-ring.irreducible-elem } ?R \text{ ?multiplication ?zero ?unit ?p}$).

theorem *irreducible-imp-prime:*

assumes p : irreducible-elem p

shows prime-elem p

proof –

have pR : $p \in R$ and pnz : $p \neq 0$ and pnu : $\neg \text{is-unit } p$

using p **by** (auto dest: irreducible-elemD)

interpret P : ideal-in-comm-ring principal-ideal p R (+) ($\cdot$) 0 1

by (intro ideal-in-comm-ring.intro principal-ideal-is-ideal[OF pR] commutative-ring-axioms)

have prime: P .prime-ideal

using P .maximal-imp-prime-ideal irreducible-imp-maximal-ideal p **by** blast

show ?thesis

using P .prime-ideal-def divides-iff-mem-principal pR pnu pnz prime prime-elemI

by auto

qed

52.8.1 The ascending chain condition on principal ideals

The union of a countable ascending chain of ideals is an ideal.

lemma *Union-chain-ideal:*

assumes I : $\bigwedge n. \text{Ideal } (A \ n) \ R$ (+) ($\cdot$) 0 1

and mono: $\bigwedge n. A \ n \subseteq A \ (\text{Suc } n)$

shows Ideal $(\bigcup n. A \ n) \ R$ (+) ($\cdot$) 0 1

proof –

have mono-le: $A \ m \subseteq A \ n$ if $m \leq n$ for $m \ n$

using that **by** (induct n) (simp-all add: lift-Suc-mono-le mono)

have dir: $\bigwedge x \ y. x \in (\bigcup n. A \ n) \implies y \in (\bigcup n. A \ n) \implies \exists k. x \in A \ k \wedge y \in A \ k$

using mono mono-le **by** (metis UN-E not-less-eq-eq subset-eq)

interpret $A0$: Ideal $A \ 0 \ R$ (+) ($\cdot$) 0 1 **by** (rule I)

have sub: Subgroup $(\bigcup n. A \ n) \ R$ (+) 0

proof (rule additive.subgroupI)

show $(\bigcup n. A \ n) \subseteq R$

proof

fix x **assume** $x \in (\bigcup n. A \ n)$

then obtain n **where** $x \in A \ n$ **by** blast

interpret An : Ideal $A \ n \ R$ (+) ($\cdot$) 0 1 **by** (rule I)

show $x \in R$ **using** $\langle x \in A \ n \rangle$ An .additive.subset **by** blast

qed

next

show 0 $\in (\bigcup n. A \ n)$ **using** $A0$.additive.sub-unit-closed **by** blast

next

fix $g \ h$ **assume** $g \in (\bigcup n. A \ n)$ $h \in (\bigcup n. A \ n)$

then obtain k **where** gh : $g \in A \ k$ $h \in A \ k$ **using** dir **by** blast

interpret Ak : Ideal $A \ k \ R$ (+) ($\cdot$) 0 1 **by** (rule I)

show $g + h \in (\bigcup n. A \ n)$ **using** gh Ak .additive.sub-composition-closed **by** blast

next

```

fix g assume g ∈ (⋃ n. A n)
then obtain n where g ∈ A n by blast
interpret An: Ideal A n R (+) (·) 0 1 by (rule I)
show additive.invertible g using ⟨g ∈ A n⟩ An.additive.subset by auto
show additive.inverse g ∈ (⋃ n. A n)
  using ⟨g ∈ A n⟩ An.additive.submonoid-inverse-closed An.additive.sub.invertible
by blast
qed
interpret U: Subgroup ⋃ n. A n R (+) 0 by (rule sub)
show ?thesis
proof
  fix a x assume a: a ∈ R and x: x ∈ (⋃ n. A n)
  obtain n where x ∈ A n using x by blast
  interpret An: Ideal A n R (+) (·) 0 1 by (rule I)
  show a · x ∈ (⋃ n. A n) x · a ∈ (⋃ n. A n)
    using a ⟨x ∈ A n⟩ An.Ideal by blast+
qed
qed

```

Ascending chain condition: every ascending chain of (principal) ideals in a PID stabilises. Proof: the union is an ideal, hence principal = (c) ; the generator c lies in some $A N$, and from there on the chain cannot grow beyond $A N$.

```

theorem ascending-chain-stabilises:
  assumes I: ⋀ n. Ideal (A n) R (+) (·) 0 1
    and mono: ⋀ n. A n ⊆ A (Suc n)
  obtains N where ⋀ n. N ≤ n ⟹ A n = A N
proof -
  have mono-le: A m ⊆ A n if m ≤ n for m n
    using that
    by (induct n) (simp-all add: lift-Suc-mono-le mono)
  have Uideal: Ideal (⋃ n. A n) R (+) (·) 0 1
    using I mono by (rule Union-chain-ideal)
  obtain c where c: c ∈ R and Uc: (⋃ n. A n) = principal-ideal c
    using principal[OF Union-chain-ideal] I mono by metis
  with principal-ideal-contains obtain N where cN: c ∈ A N
    by fastforce
  have A n = A N if N ≤ n for n
  proof
    show A N ⊆ A n using mono-le that .
    show A n ⊆ A N
  proof
    fix x assume x ∈ A n
    then have x ∈ principal-ideal c using Uc by blast
    then obtain r where r: r ∈ R x = r · c unfolding principal-ideal-def by blast
    interpret AN: Ideal A N R (+) (·) 0 1
      by (rule I)
    show x ∈ A N using r cN AN.Ideal(1) by blast
  qed
qed

```

then show ?thesis using that by blast
qed

52.8.2 Bezout: existence of greatest common divisors

The ideal generated by two elements, $(a, b) = \{x \cdot a + y \cdot b\}$.

definition *bezout-ideal* :: 'a $\Rightarrow$ 'a $\Rightarrow$ 'a set
where *bezout-ideal* a b $\equiv \{x \cdot a + y \cdot b \mid x \ y. x \in R \wedge y \in R\}$

lemma *bezout-ideal-memI*:
[[$x \in R; y \in R$]] $\implies x \cdot a + y \cdot b \in \text{bezout-ideal } a \ b$
unfolding *bezout-ideal-def* by blast

lemma *bezout-ideal-is-ideal*:
assumes a: $a \in R$ and b: $b \in R$
shows *Ideal* (*bezout-ideal* a b) R (+) ($\cdot$) 0 1
proof –
have sg: *Subgroup* (*bezout-ideal* a b) R (+) 0
proof (rule *additive.subgroupI*)
show *bezout-ideal* a b $\subseteq R$
using a b *principal-ideal-domain.bezout-ideal-def* *principal-ideal-domain-axioms*
by *fastforce*
show 0 $\in$ *bezout-ideal* a b using a b
using *additive.left-unit* *additive.unit-closed* *bezout-ideal-memI* *left-zero* by *metis*
next
fix g h assume $g \in \text{bezout-ideal } a \ b$ $h \in \text{bezout-ideal } a \ b$
then obtain $x1 \ y1 \ x2 \ y2$ where
 $g: x1 \in R \ y1 \in R \ g = x1 \cdot a + y1 \cdot b$ and $h: x2 \in R \ y2 \in R \ h = x2 \cdot a + y2 \cdot b$
unfolding *bezout-ideal-def* by blast
have $g + h = (x1 + x2) \cdot a + (y1 + y2) \cdot b$
by (simp add: *a* *additive.commutative* *additive.left-commutative* b *distributive*(2))
 $g \ h$)
then show $g + h \in \text{bezout-ideal } a \ b$
using g h *bezout-ideal-memI*[of $x1+x2 \ y1+y2 \ a \ b$] by simp
next
fix g assume $g \in \text{bezout-ideal } a \ b$
then obtain x y where $xy: x \in R \ y \in R \ g = x \cdot a + y \cdot b$
unfolding *bezout-ideal-def* by blast
show *additive.invertible* g
using xy a b by simp
show *additive.inverse* g $\in \text{bezout-ideal } a \ b$
using xy a b *bezout-ideal-memI*[of $-x -y \ a \ b$]
by (simp add: *left-minus* *additive.inverse-composition-commute* *additive.commutative*)
qed
interpret S: *Subgroup* *bezout-ideal* a b R (+) 0 by (rule sg)
show ?thesis
proof
fix r z assume $r: r \in R$ and $z \in \text{bezout-ideal } a \ b$

```

then obtain  $x\ y$  where  $z: x \in R\ y \in R\ z = x \cdot a + y \cdot b$ 
  unfolding bezout-ideal-def by blast
have  $r \cdot z = (r \cdot x) \cdot a + (r \cdot y) \cdot b$ 
  using  $r\ z\ a\ b$  by (simp add: distributive multiplicative.associative)
then show  $r \cdot z \in \text{bezout-ideal}\ a\ b$ 
  using  $r\ z\ \text{bezout-ideal-memI}[of\ r \cdot x\ r \cdot y\ a\ b]$  by simp
then show  $z \cdot r \in \text{bezout-ideal}\ a\ b$ 
  using  $r\ z\ a\ b$  by (simp add: multiplicative.commutative)
qed
qed

```

a and b both lie in (a, b) .

```

lemma a-in-bezout-ideal:  $\llbracket a \in R; b \in R \rrbracket \implies a \in \text{bezout-ideal}\ a\ b$ 
  using bezout-ideal-memI[of 1 0 a b] by simp

```

```

lemma b-in-bezout-ideal:  $\llbracket a \in R; b \in R \rrbracket \implies b \in \text{bezout-ideal}\ a\ b$ 
  using bezout-ideal-memI[of 0 1 a b] by simp

```

Bezout's theorem. The gcd generates the ideal (a, b) : divisibility of a and b is membership, and any common divisor divides every element of (a, b) , in particular d .

theorem *bezout*:

```

assumes  $a: a \in R$  and  $b: b \in R$ 
obtains  $d\ x\ y$  where is-gcd  $d\ a\ b\ x \in R\ y \in R\ d = x \cdot a + y \cdot b$ 
proof -
  have  $I: \text{Ideal}(\text{bezout-ideal}\ a\ b)\ R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}$  by (rule bezout-ideal-is-ideal[OF a b])
  obtain  $d$  where  $d: d \in R$  and  $\text{gen}: \text{bezout-ideal}\ a\ b = \text{principal-ideal}\ d$ 
    using principal[OF I] by blast
  have  $d \in \text{bezout-ideal}\ a\ b$  using  $\text{gen}\ d$  by (simp add: principal-ideal-contains)
  then obtain  $x\ y$  where  $xy: x \in R\ y \in R\ d = x \cdot a + y \cdot b$ 
    unfolding bezout-ideal-def by blast
  obtain  $dda: \text{divides}\ d\ a$  and  $ddb: \text{divides}\ d\ b$ 
    using a-in-bezout-ideal b-in-bezout-ideal gen divides-iff-mem-principal[OF d]  $a\ b$ 
  by metis
  — Any common divisor  $c$  divides  $d = x \cdot a + y \cdot b$ .
  have is-gcd  $d\ a\ b$ 
  proof (rule is-gcdI[OF d dda ddb])
    fix  $c$  assume  $c: c \in R$  and  $ca: \text{divides}\ c\ a$  and  $cb: \text{divides}\ c\ b$ 
    obtain  $s\ t$  where  $s: s \in R\ a = c \cdot s$  and  $t: t \in R\ b = c \cdot t$  using  $ca\ cb$  by blast
    then have  $d = c \cdot (x \cdot s + y \cdot t)$ 
      by (simp add:  $c\ \text{distributive}(1)\ \text{multiplicative.left-commutative}\ xy$ )
    moreover have  $x \cdot s + y \cdot t \in R$  using  $xy\ s\ t$  by simp
    ultimately show  $\text{divides}\ c\ d$  by (blast intro: dividesI)
  qed
  then show ?thesis using  $xy$  that by blast
qed

```

Consequently a greatest common divisor exists for every pair.

corollary *gcd-exists*:

```

assumes  $a \in R \ b \in R$  shows  $\exists d. \text{is-gcd } d \ a \ b$ 
using bezout[OF assms] by blast

```

end

Principal ideal domains satisfy the noetherian ascending-chain condition.

sublocale *principal-ideal-domain* $\subseteq$ *Noetherian-Domain*

proof *unfold-locales*

```

fix  $A$ 
assume  $I: \bigwedge n. \text{Ideal } (A \ n) \ R \ (+) \ (\cdot) \ 0 \ 1$ 
and mono:  $\bigwedge n. A \ n \subseteq A \ (\text{Suc } n)$ 
show  $\exists N. \forall n. N \leq n \longrightarrow A \ n = A \ N$ 
using I ascending-chain-stabilises mono by meson
qed

```

52.9 Euclidean domains

A Euclidean domain is an integral domain equipped with a degree function φ into the naturals admitting division with remainder: for a, b nonzero there are q, r with $a = b \cdot q + r$ and either $r = 0$ or $\varphi \ r < \varphi \ b$.

```

locale Euclidean-Domain = integral-domain +
fixes  $\varphi :: 'a \Rightarrow \text{nat}$ 
assumes euclidean-division:
   $\llbracket a \in R; b \in R; a \neq 0; b \neq 0 \rrbracket \Longrightarrow$ 
   $\exists q \in R. \exists r \in R. a = b \cdot q + r \wedge (r = 0 \vee \varphi \ r < \varphi \ b)$ 
begin

```

Every Euclidean domain is a principal ideal domain: an ideal is generated by any nonzero element of least φ -value, since division with remainder against it lands back in the ideal and minimality forces a zero remainder.

sublocale *principal-ideal-domain*

proof *unfold-locales*

```

fix  $I$  assume  $I: \text{Ideal } I \ R \ (+) \ (\cdot) \ 0 \ 1$ 
interpret  $I: \text{Ideal } I \ R \ (+) \ (\cdot) \ 0 \ 1$  by (rule I)
show  $\exists a \in R. I = \text{principal-ideal } a$ 
proof (cases  $I = \{0\}$ )
  case True
  have principal-ideal  $0 = \{0\}$ 
  using principal-ideal-memI[of 0 0] by (force simp: principal-ideal-def)
  then show ?thesis using True by auto
next
  case False
  — Choose  $a$  in the nonzero part of  $I$  minimising  $\varphi$ .
  define Inz where  $\text{Inz} \equiv \{x \in I. x \neq 0\}$ 
  have ne:  $\text{Inz} \neq \{\}$  using False I.additive.sub-unit-closed Inz-def by blast
  define  $\varphi \text{img}$  where  $\varphi \text{img} \equiv \varphi \ ` \text{Inz}$ 
  have  $\varphi \text{img} \neq \{\}$  using ne  $\varphi \text{img-def}$  by simp

```

```

then obtain a where a: a ∈ Inz and amin:  $\bigwedge b. b \in \text{Inz} \implies \varphi a \leq \varphi b$ 
using exists-least-iff[of  $\lambda n. n \in \varphi \text{img}$ ] unfolding  $\varphi \text{img-def}$   $\text{image-def}$   $\text{mem-Collect-eq}$ 
  by (metis ne all-not-in-conv linorder-not-le)
have aI: a ∈ I using a Inz-def by blast
have aR: a ∈ R using aI I.additive.subset by blast
have anz: a ≠ 0 using a Inz-def by blast
have I = principal-ideal a
proof
  — (a) ⊆ I since a ∈ I and I absorbs products.
  show principal-ideal a ⊆ I
    using I.Ideal(1) aI principal-ideal-def by auto
next
  — I ⊆ (a) by Euclidean division against a.
  show I ⊆ principal-ideal a
  proof
    fix b assume bI: b ∈ I
    have bR: b ∈ R using bI I.additive.subset by blast
    show b ∈ principal-ideal a
    proof (cases b = 0)
      case True then show ?thesis using principal-ideal-memI[of 0 a] aR by simp
    next
      case bnz: False
      obtain q r where q: q ∈ R and rR: r ∈ R and beq: b = a · q + r
        and rem: r = 0 ∨  $\varphi r < \varphi a$ 
        using euclidean-division[OF bR aR bnz anz] by blast
      have aqI: a · q ∈ I using I.Ideal(2)[OF q aI] .
      then have req: r = b − a · q
        using I.additive.sub additive.commutative additive commute-iff-inverse beq
      rR by metis
      have − (a · q) ∈ I using aqI by simp
      then have b + (− (a · q)) ∈ I
        using bI I.additive.sub-composition-closed by blast
      then have rI: r ∈ I using req by simp
      — Minimality of  $\varphi a$  rules out  $\varphi r < \varphi a$ , so r = 0 and a divides b.
      have r = 0
        using rI rem Inz-def amin linorder-not-less by blast
      then show ?thesis using q principal-ideal-memI[of q a]
        using aR beq multiplicative.commutative by auto
    qed
  qed
qed
then show ?thesis using aR by blast
qed
qed
end

```

A field is a Euclidean domain with the constant degree function (division with remainder is exact: $a = b \cdot (b^{-1} \cdot a) + 0$).


```

sublocale Field  $\subseteq$  Euclidean-Domain R (+) ( $\cdot$ ) 0 1  $\lambda$ -. 0
proof
  fix a b assume a: a  $\in$  R and b: b  $\in$  R and bnz: b  $\neq$  0
  have a = b  $\cdot$  (multiplicative.inverse b  $\cdot$  a) + 0
    by (simp add: a b bnz field-inverse multiplicative.invertible-right-inverse2)
  then show  $\exists q \in R. \exists r \in R. a = b \cdot q + r \wedge (r = 0 \vee (0::nat) < 0)$ 
    using a b bnz field-inverse by auto
qed

```

52.10 Factorizations into irreducibles

The product of a list of ring elements.

```

context commutative-ring
begin

```

```

definition list-prod :: 'a list  $\Rightarrow$  'a
  where list-prod xs = foldr ( $\cdot$ ) xs 1

```

```

lemma list-prod-Nil [simp]: list-prod [] = 1
  by (simp add: list-prod-def)

```

```

lemma list-prod-Cons [simp]: list-prod (x # xs) = x  $\cdot$  list-prod xs
  by (simp add: list-prod-def)

```

```

lemma list-prod-closed [intro, simp]:
  set xs  $\subseteq$  R  $\implies$  list-prod xs  $\in$  R
  by (induct xs) auto

```

A factorization of a is a list of irreducible ring elements whose product is a .

```

definition factorization :: 'a list  $\Rightarrow$  'a  $\Rightarrow$  bool
  where factorization fs a  $\equiv$  ( $\forall p \in$  set fs. irreducible-elem p)  $\wedge$  list-prod fs = a

```

```

lemma factorizationI:
  [ $\bigwedge p. p \in$  set fs  $\implies$  irreducible-elem p; list-prod fs = a]  $\implies$  factorization fs a
  unfolding factorization-def by blast

```

```

lemma factorization-irreducible: [factorization fs a; p  $\in$  set fs]  $\implies$  irreducible-elem
p
  unfolding factorization-def by blast

```

```

lemma factorization-prod: factorization fs a  $\implies$  list-prod fs = a
  unfolding factorization-def by blast

```

```

lemma factorization-set-R: factorization fs a  $\implies$  set fs  $\subseteq$  R
  unfolding factorization-def using irreducible-elemD(1) by blast

```

```

lemma list-prod-append [simp]:
  assumes set xs  $\subseteq$  R set ys  $\subseteq$  R
  shows list-prod (xs @ ys) = list-prod xs  $\cdot$  list-prod ys

```

```

using assms
by (induct xs) (simp-all add: multiplicative.associative)

```

An irreducible element has the one-item factorization $[p]$.

```

lemma factorization-single: irreducible-elem p  $\implies$  factorization [p] p
by (simp add: factorization-def irreducible-elemD)

```

Concatenating factorizations multiplies the factored elements.

```

lemma factorization-append:
  assumes factorization fs a factorization gs b
  shows factorization (fs @ gs) (a · b)
  using assms factorization-def factorization-set-R by auto

```

end

52.11 Uniqueness of factorization

```

context integral-domain
begin

```

A prime element dividing a list-product divides one of the factors.

```

lemma prime-divides-list-prod:
  assumes p: prime-elem p and fs: set fs  $\subseteq$  R and dvd: divides p (list-prod fs)
  shows  $\exists f \in \text{set } fs. \text{ divides } p f$ 
  using fs dvd

```

```

proof (induct fs)
  case Nil with one-divides-unit-iff p prime-elem-def show ?case
    by force

```

```

next
  case (Cons f fs)
  then have divides p f  $\vee$  divides p (list-prod fs)
    using p prime-elem-def by auto
  then show ?case
    using Cons by force

```

qed

If an irreducible q divides an irreducible p then they are associated.

```

lemma irreducible-dvd-irreducible-assoc:
  assumes q: irreducible-elem q and p: irreducible-elem p and dvd: divides q p
  shows associated q p
proof —
  have qR: q  $\in$  R and pR: p  $\in$  R using q p by (auto dest: irreducible-elemD)
  obtain c where c: c  $\in$  R p = q · c using dvd by blast
  — p = q · c irreducible: q or c is a unit; q is not, so c is, giving q  $\sim$  p.
  have is-unit q  $\vee$  is-unit c using irreducible-elemD(4)[OF p qR c(1) c(2)] .
  then have cu: is-unit c using irreducible-elemD(3)[OF q] by blast
  have divides p q
    using associated-iff-unit-multiple c cu local.associatedD2 multiplicative.commutative
    pR qR

```

```

    by metis
  then show ?thesis using dvd
    by blast
qed

```

Cancellation for association: a common nonzero factor may be dropped.

```

lemma associated-cancel:
  assumes c:  $c \in R$  and x:  $x \in R$  and y:  $y \in R$  and cnz:  $c \neq 0$ 
    and as: associated (c · x) (c · y)
  shows associated x y
proof (rule associatedI)
  have d1: divides (c · x) (c · y) and d2: divides (c · y) (c · x)
    using as by (auto dest: associatedD1 associatedD2)
  show divides x y
proof -
  obtain d where d:  $d \in R$  c · y = (c · x) · d using d1 by blast
  then have y = x · d
    using mult-cancel-left[OF c y - cnz] x d c by auto
  then show ?thesis using d by blast
qed
show divides y x
proof -
  obtain d where d:  $d \in R$  c · x = (c · y) · d using d2 by blast
  then show ?thesis using d
    using <divides x y> c cnz divides-def local.mult-cancel-left multiplicative.associative
      multiplicative.composition-closed x by metis
qed
qed

```

Multiplying by a unit yields an associate.

```

lemma associated-unit-mult:
  assumes u: is-unit u u ∈ R and b: b ∈ R
  shows associated (u · b) b
proof (rule associatedI)
  show divides (u · b) b
    using associated-iff-unit-multiple b local.associatedD2 u by blast
  show divides b (u · b) using u b by (metis dividesI multiplicative.commutative)
qed

```

Product of a list is unchanged by removing then re-multiplying a member (commutativity).

```

lemma list-prod-remove1:
  assumes set gs ⊆ R and g ∈ set gs
  shows list-prod gs = g · list-prod (remove1 g gs)
  using assms
proof (induct gs)
  case (Cons h hs)
  have hR: h ∈ R and hsR: set hs ⊆ R using Cons.prem1 by auto
  show ?case

```

```

proof (cases g = h)
  case False
    have tR: list-prod (remove1 g hs) ∈ R
      using hsR set-remove1-subset by fastforce
    have list-prod (h # hs) = h · (g · list-prod (remove1 g hs))
      using Cons.hyps Cons.prem(2) False hsR by force
    also have ... = g · list-prod (h # remove1 g hs)
      using Cons.prem tR by (auto simp: mult-ac)
    also have h # remove1 g hs = remove1 g (h # hs) using False by simp
    finally show ?thesis .
  qed auto
qed auto

```

Uniqueness of factorization (length form): any two prime factorizations of associated products have the same number of factors. This is the well-definedness of the number of irreducible factors — the core uniqueness statement of unique factorization.

```

theorem prime-factorization-length-unique:
  assumes  $\bigwedge f. f \in \text{set } fs \implies \text{prime-elem } f$  and  $\bigwedge g. g \in \text{set } gs \implies \text{prime-elem } g$ 
  and associated (list-prod fs) (list-prod gs)
  shows length fs = length gs
  using assms
proof (induct fs arbitrary: gs)
  case Nil
    — Empty on the left: the right product is a unit, so it too must be empty.
    have associated 1 (list-prod gs) using Nil.prem(3) by simp
    then have 1: divides (list-prod gs) 1 by (auto dest: associatedD2)
    have gs = []
    proof (rule ccontr)
      assume gs ≠ []
      then obtain g gs' where gs: gs = g # gs' using list.exhaust by blast
      have gp: prime-elem g using Nil.prem(2) gs by simp
      have gR: g ∈ R and gsR: set gs' ⊆ R
        using gp gs Nil.prem(2) by (auto dest: prime-elemD(1))
      have divides g (list-prod gs) using gs gR gsR by (auto intro: dividesI)
      then show False using prime-elemD(3)[OF gp]
        using 1 divides-trans gR one-divides-unit-iff by blast
    qed
    then show ?case by simp
  next
    case (Cons f fs')
    have fp: prime-elem f using Cons.prem(1) by simp
    then have fR: f ∈ R and fnz: f ≠ 0 by (auto dest: prime-elemD)
    have fs'p:  $\bigwedge x. x \in \text{set } fs' \implies \text{prime-elem } x$  using Cons.prem(1) by simp
    have fsR: set fs' ⊆ R using fs'p prime-elemD(1) by blast
    have gsR: set gs ⊆ R using Cons.prem(2) prime-elemD(1) by blast
    — f divides the right product, so divides some g ∈ gs; they are associated.
    have divides f (list-prod (f # fs')) using fR fsR by (auto intro: dividesI)
    then have divides f (list-prod gs)

```

```

    using Cons.premis(3) fR list-prod-closed[OF gsR] by (meson associatedD1 divides-trans)
  then obtain g where g: g ∈ set gs and fg: divides f g
    using Cons.premis(1) prime-divides-list-prod[OF - gsR] by auto
  have gp: prime-elem g using Cons.premis(2) g by simp
  have gR: g ∈ R and gnz: g ≠ 0 using gp by (auto dest: prime-elemD)
  have assoc-fg: associated f g
    by (simp add: Cons.premis(1) fg gp irreducible-dvd-irreducible-assoc prime-imp-irreducible)
  define rest where rest ≡ remove1 g gs
  have restR: set rest ⊆ R
    using gsR rest-def by force
  have restp:  $\bigwedge x. x \in \text{set rest} \implies \text{prime-elem } x$ 
    unfolding rest-def using Cons.premis(2) set-remove1-subset by (meson subset-iff)
  have gs-split: list-prod gs = g · list-prod rest
    using list-prod-remove1[OF gsR g] rest-def by simp
  — g = f · u for a unit u; substitute and cancel f.
  obtain u where u: u ∈ R g = f · u
    using assoc-fg by (auto dest: associatedD1)
  then have uu: is-unit u
    using fp gp irreducible-elem-def prime-imp-irreducible by metis
  have associated (f · list-prod fs') (g · list-prod rest)
    using Cons.premis(3) gs-split by simp
  then have associated (f · list-prod fs') (f · (u · list-prod rest))
    using u by (simp add: multiplicative.associative fR u(1) restR)
  then have associated (list-prod fs') (u · list-prod rest)
    using associated-cancel[OF fR list-prod-closed[OF fsR]] fnz u(1) restR by simp
  moreover have associated (u · list-prod rest) (list-prod rest)
    using associated-unit-mult[OF uu u(1) list-prod-closed[OF restR]] .
  ultimately have associated (list-prod fs') (list-prod rest)
    using fsR restR u(1) associated-trans by (meson associated-def divides-trans list-prod-closed)
  then have eqlen: length fs' = length rest using Cons.hyps[OF fs'p restp] by simp
  have 0 < length gs using g by (cases gs) auto
  then show ?case using eqlen
    using g rest-def by (simp add: length-remove1)
qed

end

```

52.12 Factorial domains (unique factorization domains)

context *principal-ideal-domain*
begin

52.12.1 Existence of factorizations

Every nonzero non-unit of a PID admits a factorization into irreducibles. If not, the set of “bad” elements (nonzero non-units without a factorization) is nonempty; a bad element is not irreducible, so it factors into two non-units, at

least one of which is bad and is a *proper* divisor. Iterating (dependent choice) yields a strictly ascending chain of principal ideals, contradicting the ascending chain condition.

theorem *factorization-exists:*

assumes $a: a \in R$ **and** $anz: a \neq 0$ **and** $anu: \neg \text{is-unit } a$

shows $\exists fs. \text{factorization } fs \ a$

proof (rule ccontr)

define *bad* **where** $bad \ x \equiv x \in R \wedge x \neq 0 \wedge \neg \text{is-unit } x \wedge \neg (\exists fs. \text{factorization } fs \ x)$ **for** x

assume $non: \neg (\exists fs. \text{factorization } fs \ a)$

— *bad* x : nonzero non-unit with no factorization.

then have *bad-a: bad* a **using** *assms bad-def* **by** *simp*

— Every bad element has a bad proper divisor.

have *step: $\exists y. bad \ y \wedge \text{divides } y \ x \wedge \neg \text{divides } x \ y$* **if** $bx: bad \ x$ **for** x

proof —

have $xR: x \in R$ **and** $xnz: x \neq 0$ **and** $xnu: \neg \text{is-unit } x$

and $xnf: \neg (\exists fs. \text{factorization } fs \ x)$ **using** bx *bad-def* **by** *auto*

have $\neg \text{irreducible-elem } x$ **using** xnf *factorization-single* **by** *blast*

then obtain $p \ q$ **where** $pq: p \in R \ q \in R \ x = p \cdot q \wedge \neg \text{is-unit } p \wedge \neg \text{is-unit } q$

using *irreducible-elemI* $[OF \ xR \ xnz \ xnu]$ **by** *blast*

have $pnz: p \neq 0$ **and** $qnz: q \neq 0$ **using** $pq(3)$ xnz $pq(1,2)$ **by** *auto*

— If both p and q had factorizations, so would $x = p \cdot q$; hence one is bad.

have $\neg (\exists fs. \text{factorization } fs \ p) \vee \neg (\exists fs. \text{factorization } fs \ q)$

using *factorization-append* $pq(3)$ xnf **by** *blast*

then show *?thesis*

proof

assume $npf: \neg (\exists fs. \text{factorization } fs \ p)$

have *bad* p **using** $pq(1)$ pnz $pq(4)$ npf *bad-def* **by** *simp*

moreover have $\neg \text{divides } x \ p$

proof

assume *divides* $x \ p$

— $x \mid p$ and $x = p \cdot q$ force q to be a unit, contradiction.

then obtain d **where** $d: d \in R \ p = x \cdot d$ **by** *blast*

then have $x \cdot 1 = x \cdot (q \cdot d)$

using *d multiplicative.associative multiplicative.commutative* $pq(2,3)$ xR **by**

force

then have $qd: q \cdot d = 1$ **using** *mult-cancel-left* $[OF \ xR \ - \ xnz]$ $pq(2)$ d **by**

simp

have $d \cdot q = 1$ **using** qd $pq(2)$ $d(1)$ **by** (*simp add: multiplicative.commutative*)

then show *False*

using *d is-unitI* $pq(5)$ qd **by** *blast*

qed

ultimately show *?thesis* **using** pq **by** *blast*

next

assume $nqf: \neg (\exists fs. \text{factorization } fs \ q)$

have *bad* q **using** $pq(2)$ qnz $pq(5)$ nqf *bad-def* **by** *simp*

moreover have *divides* $q \ x$

using *multiplicative.commutative* pq **by** *blast*

moreover have $\neg \text{divides } x \ q$

```

proof
  assume divides  $x \ q$ 
  then obtain  $d$  where  $d \in R \ q = x \cdot d$  by blast
  then have  $x \cdot 1 = x \cdot (p \cdot d)$ 
    using multiplicative.left-commutative  $pq \ xR$  by auto
  then have  $pd: p \cdot d = 1$  using mult-cancel-left[OF  $xR - - \ xnz$ ]  $pq(1) \ d$  by
simp
  have  $d \cdot p = 1$  using  $pd \ pq(1) \ d(1)$  by (simp add: multiplicative.commutative)
  then show False
    using  $d(1) \ is-unitI \ pd \ pq(4)$  by presburger
  qed
  ultimately show ?thesis by blast
qed
qed
  — Build an infinite bad proper-divisor chain by dependent choice.
  have  $\exists f. \forall n. \text{bad } (f \ n) \wedge (\text{divides } (f \ (\text{Suc } n)) \ (f \ n) \wedge \neg \text{divides } (f \ n) \ (f \ (\text{Suc } n)))$ 
    by (rule dependent-nat-choice) (use bad-a step in blast)+
  then obtain  $f$  where  $f\text{bad}: \bigwedge n. \text{bad } (f \ n)$ 
    and  $f\text{dvd}: \bigwedge n. \text{divides } (f \ (\text{Suc } n)) \ (f \ n)$  and  $f\text{ndvd}: \bigwedge n. \neg \text{divides } (f \ n) \ (f \ (\text{Suc } n))$ 
    by blast
  have  $fR: \bigwedge n. f \ n \in R$  using  $f\text{bad} \ \text{bad-def}$  by simp
  — The principal ideals  $(f \ n)$  ascend strictly.
  define  $A$  where  $A \ n \equiv \text{principal-ideal } (f \ n)$  for  $n$ 
  have  $\bigwedge n. A \ n \subseteq A \ (\text{Suc } n)$ 
    using A-def dvd divides-iff-principal-subset[OF  $fR \ fR$ ] by simp
  then obtain  $N$  where  $\text{stab}: \bigwedge n. N \leq n \implies A \ n = A \ N$ 
    using A-def principal-ideal-is-ideal[OF  $fR$ ] ideal-chain-stabilises by force
  —  $A \ N = A \ (\text{Suc } N)$  makes  $f \ N$  and  $f \ (\text{Suc } N)$  mutually divide.
  have  $A \ (\text{Suc } N) = A \ N$  using  $\text{stab}[of \ \text{Suc } N]$  by simp
  then show False using  $f\text{ndvd}$ 
    unfolding A-def using divides-iff-mem-principal fR principal-ideal-contains by
metis
qed
end

```

A factorial domain (UFD) is an integral domain in which every nonzero non-unit factors into irreducibles, uniquely up to the number of factors (and, via $\llbracket \text{integral-domain } ?R \ ?\text{addition } ?\text{multiplication } ?\text{zero } ?\text{unit}; \bigwedge f. f \in \text{set } ?fs \implies \text{commutative-ring.prime-elem } ?R \ ?\text{multiplication } ?\text{zero } ?\text{unit } f; \bigwedge g. g \in \text{set } ?gs \implies \text{commutative-ring.prime-elem } ?R \ ?\text{multiplication } ?\text{zero } ?\text{unit } g; \text{commutative-ring.associated } ?R \ ?\text{multiplication } (\text{commutative-ring.list-prod } ?\text{multiplication } ?\text{unit } ?fs) \ (\text{commutative-ring.list-prod } ?\text{multiplication } ?\text{unit } ?gs) \rrbracket \implies \text{length } ?fs = \text{length } ?gs$, up to associates and order once the factors are prime).

locale *Factorial-Domain* = *integral-domain* +
assumes *factors-exist*: $\llbracket a \in R; a \neq 0; \neg \text{is-unit } a \rrbracket \implies \exists fs. \text{factorization } fs \ a$

Every principal ideal domain is a factorial domain: existence of factorizations is

$\llbracket \text{principal-ideal-domain } ?R \text{ ?addition ?multiplication ?zero ?unit; } ?a \in ?R; ?a \neq ?\text{zero}; \neg \text{Monoid.invertible } ?R \text{ ?multiplication ?unit ?a} \rrbracket \implies \exists \text{fs. commutative-ring.factorization } ?R \text{ ?multiplication ?zero ?unit fs } ?a$, established via the ascending chain condition.

sublocale *principal-ideal-domain* $\subseteq$ *Factorial-Domain*
by *unfold-locales (rule factorization-exists)*

52.13 Euclidean type classes as locale Euclidean domains; the integers

Isabelle's type class *euclidean-ring* (an *idom* with a *euclidean-size* and division with remainder) is exactly the type-class analogue of our locale *Euclidean-Domain*. The ring of integers $\mathbb{Z}$ is the primary instance.

Restore HOL arithmetic notation locally (*Ring-Theory* suppresses it).

notation *plus* (**infixl** $\langle + \rangle$ 65)
notation *minus* (**infixl** $\langle - \rangle$ 65)
unbundle *uminus-syntax*

52.14 The general type-class bridge

The division-with-remainder axiom of *Euclidean-Domain* is witnessed by $q = a \text{ div } b$, $r = a \text{ mod } b$: the identity $?b * (?a \text{ div } ?b) + ?a \text{ mod } ?b = ?a$ gives $a = b \cdot q + r$, and $?b \neq 0 \implies \text{euclidean-size } (?a \text{ mod } ?b) < \text{euclidean-size } ?b$ bounds the remainder's size. (The class's extra axioms — *euclidean-size* $0 = 0$, $?b \neq 0 \implies \text{euclidean-size } ?a \leq \text{euclidean-size } (?a * ?b)$ — exceed what the locale requires, so this is a one-directional bridge, which is all we need.)

lemma *euclidean-domain-TC*:

Euclidean-Domain (*UNIV* :: 'a :: euclidean-ring set) (+) (*) 0 1 euclidean-size

proof (rule *Euclidean-Domain.intro*)

show *integral-domain* (*UNIV* :: 'a set) (+) (*) 0 1 **by** (rule *idom-TC*)

next

show *Euclidean-Domain-axioms* (*UNIV* :: 'a set) (+) (*) 0 euclidean-size

proof

fix $a \ b :: 'a$ **assume** $a \in \text{UNIV}$ $b \in \text{UNIV}$ $a \neq 0$ **and** $\text{bnz}: b \neq 0$

have $a = b * (a \text{ div } b) + a \text{ mod } b$ **by** (simp add: mult-div-mod-eq)

moreover have $a \text{ mod } b = 0 \vee \text{euclidean-size } (a \text{ mod } b) < \text{euclidean-size } b$

using *mod-size-less[OF bnz]* **by** blast

ultimately show $\exists q \in \text{UNIV}. \exists r \in \text{UNIV}.$

$a = b * q + r \wedge (r = 0 \vee \text{euclidean-size } r < \text{euclidean-size } b)$

by (intro *beI[of - a div b]* *beI[of - a mod b]*) auto

qed

qed

52.15 The integers

$\mathbb{Z}$ is a Euclidean ring with Euclidean size the absolute value, so the bridge specialises to it directly. Through the interpretation $\mathbb{Z}$ inherits the whole tower: it is a principal ideal domain and a factorial (unique factorization) domain — the fundamental theorem of arithmetic.

interpretation *Int: Euclidean-Domain*

UNIV :: *int set* (+) (*) 0 1 *euclidean-size* :: *int* $\Rightarrow$ *nat*
by (*rule euclidean-domain-TC*)

Consequences, transported from the abstract tower.

lemmas *int-factorization-exists* = *Int.factorization-exists*

lemmas *int-irreducible-imp-prime* = *Int.irreducible-imp-prime*

lemmas *int-bezout* = *Int.bezout*

end

53 The prime subfield of a finite integral domain

theory *Prime-Subfield*

imports *Characteristic GF-p Divisibility-Theory*

begin

53.1 Finite integral domains

A finite integral domain is a field. Multiplication by a nonzero element is an injective self-map of the finite carrier, hence surjective; a preimage of the unit is a multiplicative inverse.

theorem (*in integral-domain*) *finite-integral-domain-imp-field*:

assumes *fin*: *finite R*

shows *Field R* (+) ($\cdot$) 0 1

proof (*rule Field.intro*[*OF commutative-ring-axioms nontrivial-ring-axioms*])

show *Field-axioms R* ($\cdot$) 0 1

proof

fix *a*

assume *aR*: $a \in R$ **and** *anz*: $a \neq 0$

have *inj*: *inj-on* (($\cdot$) *a*) *R*

by (*rule inj-onI*) (*rule mult-cancel-left*[*OF aR - - anz*])

have *closed*: (($\cdot$) *a*) ‘ *R* $\subseteq$ *R*

using *aR* **by** (*blast intro: multiplicative.composition-closed*)

have $1 \in (\cdot) a$ ‘ *R*

using *endo-inj-surj*[*OF fin closed inj*] *multiplicative.unit-closed* **by** *simp*

then obtain *b* **where** *bR*: $b \in R$ **and** *ab*: $a \cdot b = 1$ **by** *blast*

have *ba*: $b \cdot a = 1$

using *multiplicative.commutative*[*OF bR aR*] *ab* **by** *simp*

show *multiplicative.invertible a*

by (*rule multiplicative.invertibleI*[*OF ab ba bR*])

qed
qed

The characteristic map factors through the existing concrete field $GF(\text{characteristic})$. Its image is the canonical prime subfield of the finite domain. This is the bridge between the carrier-set ring hierarchy and the finite-field constructions.

locale *finite-integral-domain* = *integral-domain* +
 assumes *finite-carrier*: *finite R*
begin

definition *prime-subfield* :: 'a set
 where *prime-subfield* = *natmult* ' {0..*characteristic*}

definition *characteristic-embedding* :: *nat* $\Rightarrow$ 'a
 where *characteristic-embedding* = *restrict natmult* {0..*characteristic*}

lemma *natmult-characteristic-multiple*:
 shows *natmult* (*k* * *characteristic*) = 0
proof –
 have *natmult characteristic* = 0 **using** *characteristic-pos*[OF *finite-carrier*] **by** *simp*
 then show ?thesis **by** (*simp add: natmult-mult*)
qed

lemma *natmult-mod-characteristic*:
 shows *natmult* (*n mod characteristic*) = *natmult n*
proof –
 have *n*: *n* = *n div characteristic* * *characteristic* + *n mod characteristic*
 using *div-mult-mod-eq*[of *n characteristic*] **by** *simp*
 have *natmult n* = *natmult* (*n div characteristic* * *characteristic* + *n mod characteristic*)
 by (*rule arg-cong*[OF *n*])
 also have ... = *natmult* (*n div characteristic* * *characteristic*) + *natmult* (*n mod characteristic*)
 by (*rule natmult-add*)
 also have ... = *natmult* (*n mod characteristic*)
 using *natmult-characteristic-multiple* **by** *simp*
 finally show ?thesis **by** (*rule sym*)
qed

lemma *natmult-inj-on-characteristic*:
 shows *inj-on* *natmult* {0..*characteristic*}
proof (*rule inj-onI*)
 fix *m n*
 assume *m*: *m* $\in$ {0..*characteristic*} **and** *n*: *n* $\in$ {0..*characteristic*}
 and *eq*: *natmult m* = *natmult n*
 show *m* = *n*
 proof (*rule ccontr*)
 assume *m* $\neq$ *n*
 consider (*mn*) *m* < *n* | (*nm*) *n* < *m* **using** $\langle m \neq n \rangle$ **by** *linarith*

```

then show False
proof cases
  case mn
    have pos:  $0 < n - m$  using mn by simp
    have zero:  $\text{natmult } (n - m) = 0$ 
      using  $\text{natmult-diff-zero}[OF \text{ less-imp-le}[OF \text{ mn}] \text{ eq}]$  .
    have  $\text{characteristic} \leq n - m$  by (rule  $\text{characteristic-least}[OF \text{ pos zero}]$ )
    moreover have  $n - m < \text{characteristic}$  using n by simp
    ultimately show False by simp
  next
    case nm
    have pos:  $0 < m - n$  using nm by simp
    have zero:  $\text{natmult } (m - n) = 0$ 
      using  $\text{natmult-diff-zero}[OF \text{ less-imp-le}[OF \text{ nm}] \text{ eq}[\text{symmetric}]]$  .
    have  $\text{characteristic} \leq m - n$  by (rule  $\text{characteristic-least}[OF \text{ pos zero}]$ )
    moreover have  $m - n < \text{characteristic}$  using m by simp
    ultimately show False by simp
qed
qed
qed

lemma characteristic-embedding-add:
  assumes m:  $m \in \{0..<\text{characteristic}\}$  and n:  $n \in \{0..<\text{characteristic}\}$ 
  shows  $\text{characteristic-embedding } (\text{gfp-add } \text{characteristic } m \ n) =$ 
     $\text{characteristic-embedding } m + \text{characteristic-embedding } n$ 
proof -
  have cgt1:  $\text{characteristic} > 1$ 
    using  $\text{characteristic-prime}[OF \text{ finite-carrier}]$  by (simp add: prime-nat-iff)
  have closed:  $\text{gfp-add } \text{characteristic } m \ n \in \{0..<\text{characteristic}\}$ 
    by (rule  $\text{gfp-add-closed}[OF \text{ cgt1}]$ )
  have  $\text{characteristic-embedding } (\text{gfp-add } \text{characteristic } m \ n) =$ 
     $\text{natmult } ((m + n) \bmod \text{characteristic})$ 
    using m n closed by (simp add:  $\text{characteristic-embedding-def } \text{gfp-add-def}$ )
  also have  $\dots = \text{natmult } (m + n)$  by (rule  $\text{natmult-mod-characteristic}$ )
  also have  $\dots = \text{natmult } m + \text{natmult } n$  by (rule  $\text{natmult-add}$ )
  also have  $\dots = \text{characteristic-embedding } m + \text{characteristic-embedding } n$ 
    using m n by (simp add:  $\text{characteristic-embedding-def}$ )
  finally show ?thesis .
qed

lemma characteristic-embedding-mult:
  assumes m:  $m \in \{0..<\text{characteristic}\}$  and n:  $n \in \{0..<\text{characteristic}\}$ 
  shows  $\text{characteristic-embedding } (\text{gfp-mult } \text{characteristic } m \ n) =$ 
     $\text{characteristic-embedding } m \cdot \text{characteristic-embedding } n$ 
proof -
  have cgt1:  $\text{characteristic} > 1$ 
    using  $\text{characteristic-prime}[OF \text{ finite-carrier}]$  by (simp add: prime-nat-iff)
  have closed:  $\text{gfp-mult } \text{characteristic } m \ n \in \{0..<\text{characteristic}\}$ 
    by (rule  $\text{gfp-mult-closed}[OF \text{ cgt1}]$ )
  have  $\text{characteristic-embedding } (\text{gfp-mult } \text{characteristic } m \ n) =$ 

```

```

    natmult ((m * n) mod characteristic)
  using m n closed by (simp add: characteristic-embedding-def gfp-mult-def)
  also have ... = natmult (m * n) by (rule natmult-mod-characteristic)
  also have ... = natmult m · natmult n by (rule natmult-mult)
  also have ... = characteristic-embedding m · characteristic-embedding n
    using m n by (simp add: characteristic-embedding-def)
  finally show ?thesis .
qed

theorem characteristic-embedding-monomorphism:
  shows ring-monomorphism characteristic-embedding
    {0..E R
    by (auto simp: characteristic-embedding-def)
  show  $\bigwedge x y. \llbracket x \in \{0..
    characteristic-embedding (gfp-add characteristic x y) =
    characteristic-embedding x + characteristic-embedding y
    by (rule characteristic-embedding-add)
  show characteristic-embedding 0 = 0
  using characteristic-pos[OF finite-carrier] by (simp add: characteristic-embedding-def)
qed
show Monoid-homomorphism characteristic-embedding
  {0..E R
    by (auto simp: characteristic-embedding-def)
  show  $\bigwedge x y. \llbracket x \in \{0..
    characteristic-embedding (gfp-mult characteristic x y) =
    characteristic-embedding x · characteristic-embedding y
    by (rule characteristic-embedding-mult)
  show characteristic-embedding 1 = 1
    using characteristic-prime[OF finite-carrier]$$ 
```

```

    by (simp add: characteristic-embedding-def prime-nat-iff)
  qed
qed
show injective-map characteristic-embedding {0..<characteristic} R
proof
  show characteristic-embedding ∈ {0..<characteristic} →E R
    by (auto simp: characteristic-embedding-def)
  show inj-on characteristic-embedding {0..<characteristic}
    using natmult-inj-on-characteristic
    by (simp add: characteristic-embedding-def inj-on-def)
  qed
qed

lemma characteristic-embedding-image:
  characteristic-embedding ‘ {0..<characteristic} = prime-subfield
  by (auto simp: characteristic-embedding-def prime-subfield-def)

theorem prime-subfield-subring:
  shows Subring prime-subfield R (+) (·) 0 1
proof -
  interpret E: ring-monomorphism characteristic-embedding
    {0..<characteristic} gfp-add characteristic gfp-mult characteristic 0 1
    R addition multiplication zero unit
  by (rule characteristic-embedding-monomorphism)
  show ?thesis using E.image.Subring-axioms by (simp add: characteristic-embedding-image)
qed

theorem prime-subfield-field:
  shows Field prime-subfield (+) (·) 0 1
proof -
  interpret P: Subring prime-subfield R addition multiplication zero unit
    by (rule prime-subfield-subring)
  have ag: Abelian-Group prime-subfield (+) 0
  proof -
    interpret A: Group prime-subfield addition zero
      by (rule P.additive.sub.Group-axioms)
    show ?thesis
  proof
    fix x y
    assume x: x ∈ prime-subfield and y: y ∈ prime-subfield
    show x + y = y + x
      by (rule additive.commutative[OF
        subsetD[OF P.additive.subset x] subsetD[OF P.additive.subset y]])
  qed
  qed
  have ring: Ring prime-subfield (+) (·) 0 1
  proof (intro Ring.intro)
    show Abelian-Group prime-subfield (+) 0 by (rule ag)
    show Monoid prime-subfield (·) 1
      by (rule P.multiplicative.sub.Monoid-axioms)
  qed

```

```

show Ring-axioms prime-subfield (+) (·)
proof
  fix a b c
  assume a:  $a \in \text{prime-subfield}$  and b:  $b \in \text{prime-subfield}$ 
    and c:  $c \in \text{prime-subfield}$ 
  show  $a \cdot (b + c) = a \cdot b + a \cdot c$ 
    by (rule distributive(1)[OF subsetD[OF P.additive.subset a]
      subsetD[OF P.additive.subset b] subsetD[OF P.additive.subset c]])
  show  $(b + c) \cdot a = b \cdot a + c \cdot a$ 
    by (rule distributive(2)[OF subsetD[OF P.additive.subset a]
      subsetD[OF P.additive.subset b] subsetD[OF P.additive.subset c]])
  qed
qed
have cr: commutative-ring prime-subfield (+) (·) 0 1
proof -
  interpret S: Ring prime-subfield addition multiplication zero unit by (rule ring)
  show ?thesis
  proof
    fix x y
    assume x:  $x \in \text{prime-subfield}$  and y:  $y \in \text{prime-subfield}$ 
    show  $x \cdot y = y \cdot x$ 
      by (rule multiplicative.commutative[OF
        subsetD[OF P.multiplicative.subset x]
        subsetD[OF P.multiplicative.subset y]])
    qed
  qed
  interpret C: commutative-ring prime-subfield addition multiplication zero unit by
  (rule cr)
  have nt: nontrivial-ring prime-subfield (+) (·) 0 1
  proof
    show  $1 \neq 0$  by (rule nontrivial)
  qed
  interpret N: nontrivial-ring prime-subfield addition multiplication zero unit by (rule
  nt)
  have dom: integral-domain prime-subfield (+) (·) 0 1
  proof
    fix a b
    assume a:  $a \in \text{prime-subfield}$  and b:  $b \in \text{prime-subfield}$ 
      and zero:  $a \cdot b = 0$ 
    show  $a = 0 \vee b = 0$ 
      by (rule no-zero-divisors[OF subsetD[OF P.additive.subset a]
        subsetD[OF P.additive.subset b] zero])
    qed
  qed
  interpret D: integral-domain prime-subfield addition multiplication zero unit by
  (rule dom)
  have finite prime-subfield unfolding prime-subfield-def by simp
  then show ?thesis by (rule D.finite-integral-domain-imp-field)
qed

theorem card-prime-subfield:

```

```

shows card prime-subfield = characteristic
unfolding prime-subfield-def
using natmult-inj-on-characteristic by (simp add: card-image)

```

end

end

54 Modules over a ring

```

theory Module-Basics
imports Ring-Theory Finite-Composite
begin

```

A (left) *module* over the ring R : an abelian group of elements M under $\oplus$ with zero $\mathbf{0}_M$, together with a scalar multiplication $\odot$ by ring elements satisfying the module axioms. This is the exact generalisation of theory *Vector-Space* from a field of scalars to a ring; a vector space is a module whose scalar ring is a field. We develop the elementary arithmetic (scaling by zero, by the ring negation, additivity of finite sums) and the notions of submodule and of a linear combination.

```

locale Module = Ring +
  fixes madd :: 'b  $\Rightarrow$  'b  $\Rightarrow$  'b (infixl  $\langle \oplus \rangle$  65)
  and mzero :: 'b ( $\langle \mathbf{0}_M \rangle$ )
  and mcarrier :: 'b set ( $\langle M \rangle$ )
  and scale :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'b (infixr  $\langle \odot \rangle$  75)
  assumes madd-group: Abelian-Group  $M$  ( $\oplus$ )  $\mathbf{0}_M$ 
  and scale-closed:  $\llbracket a \in R; v \in M \rrbracket \Longrightarrow a \odot v \in M$ 
  and scale-distrib-madd:  $\llbracket a \in R; u \in M; v \in M \rrbracket \Longrightarrow a \odot (u \oplus v) = (a \odot u) \oplus (a \odot v)$ 
  and scale-distrib-add:  $\llbracket a \in R; b \in R; v \in M \rrbracket \Longrightarrow (a + b) \odot v = (a \odot v) \oplus (b \odot v)$ 
  and scale-scale:  $\llbracket a \in R; b \in R; v \in M \rrbracket \Longrightarrow (a \cdot b) \odot v = a \odot (b \odot v)$ 
  and scale-one:  $v \in M \Longrightarrow \mathbf{1} \odot v = v$ 
begin

```

The additive group of the module, available through the *madd* prefix.

```

sublocale madd: Abelian-Group  $M$  ( $\oplus$ )  $\mathbf{0}_M$ 
  by (rule madd-group)

```

54.1 Elementary closure and arithmetic

```

lemma mzero-closed [simp, intro]:  $\mathbf{0}_M \in M$  by simp

```

```

lemma madd-closed [simp, intro]:  $\llbracket u \in M; v \in M \rrbracket \Longrightarrow u \oplus v \in M$  by simp

```

Scaling the zero element gives the zero element.

```

lemma scale-zero-elem:

```

```

assumes  $a: a \in R$ 
shows  $a \odot \mathbf{0}_M = \mathbf{0}_M$ 
proof –
  have  $w: a \odot \mathbf{0}_M \in M$  using  $a$  by (simp add: scale-closed)
  then have  $eq: a \odot \mathbf{0}_M = (a \odot \mathbf{0}_M) \oplus (a \odot \mathbf{0}_M)$ 
    by (metis assms madd.left-unit mzero-closed scale-distrib-madd)
  then show ?thesis
    using madd.invertible-left-inverse2 w by fastforce
qed

```

Scaling by the ring zero gives the zero element.

```

lemma scale-zero-scalar:
  assumes  $v: v \in M$ 
  shows  $\mathbf{0} \odot v = \mathbf{0}_M$ 
proof –
  have  $w: \mathbf{0} \odot v \in M$  using  $v$  by (simp add: scale-closed)
  have  $eq: \mathbf{0} \odot v = (\mathbf{0} \odot v) \oplus (\mathbf{0} \odot v)$ 
    using scale-distrib-add v by force
  then show ?thesis
    using madd.commute-iff-inverse w by fastforce
qed

```

Scaling by the ring negation is the additive inverse of the scaling.

```

lemma scale-neg-scalar:
  assumes  $a: a \in R$  and  $v: v \in M$ 
  shows  $(- a) \odot v = \text{madd.inverse } (a \odot v)$ 
proof –
  have  $(a \odot v) \oplus ((- a) \odot v) = \mathbf{0}_M$  using  $a v$ 
    by (simp add: scale-zero-scalar flip: scale-distrib-add)
  then show ?thesis
    using  $a v$  by (simp add: scale-closed madd.inverse-equality madd.commutative)
qed

```

54.2 Submodules

A *submodule* is a subset containing the zero element and closed under addition and under scaling by arbitrary ring elements.

```

definition submodule :: 'b set  $\Rightarrow$  bool
  where submodule  $N \equiv N \subseteq M \wedge \mathbf{0}_M \in N \wedge (\forall u \in N. \forall v \in N. u \oplus v \in N) \wedge (\forall a \in R. \forall v \in N. a \odot v \in N)$ 

```

```

lemma submoduleI:
  assumes  $N \subseteq M$  and  $\mathbf{0}_M \in N$ 
  and  $\bigwedge u v. [u \in N; v \in N] \Longrightarrow u \oplus v \in N$ 
  and  $\bigwedge a v. [a \in R; v \in N] \Longrightarrow a \odot v \in N$ 
  shows submodule N
  using assms unfolding submodule-def by blast

```


lemma *submodule-subset*: $\text{submodule } N \implies N \subseteq M$
by (*simp add: submodule-def*)

lemma *submodule-zero*: $\text{submodule } N \implies \mathbf{0}_M \in N$
by (*simp add: submodule-def*)

lemma *submodule-add*: $\llbracket \text{submodule } N; u \in N; v \in N \rrbracket \implies u \oplus v \in N$
by (*simp add: submodule-def*)

lemma *submodule-scale*: $\llbracket \text{submodule } N; a \in R; v \in N \rrbracket \implies a \odot v \in N$
by (*simp add: submodule-def*)

A submodule is closed under the module negation (scale by -1).

lemma *submodule-neg*:
assumes N : *submodule* N **and** v : $v \in N$ **shows** *madd.inverse* $v \in N$
proof –
have vM : $v \in M$ **using** N v *submodule-subset* **by** *blast*
have *madd.inverse* $v = (-1) \odot v$ **using** vM **by** (*simp add: scale-neg-scalar scale-one*)
moreover have $(-1) \odot v \in N$ **using** N v **by** (*auto intro: submodule-scale*)
ultimately show *?thesis* **by** *simp*
qed

The whole module and the trivial submodule are submodules.

lemma *submodule-whole*: *submodule* M
by (*rule submoduleI*) (*auto simp: scale-closed*)

lemma *submodule-trivial*: *submodule* $\{\mathbf{0}_M\}$
by (*rule submoduleI*) (*auto simp: scale-zero-elem*)

The intersection of two submodules is a submodule.

lemma *submodule-inter*:
assumes N : *submodule* N **and** K : *submodule* K **shows** *submodule* $(N \cap K)$
proof (*rule submoduleI*)
show $N \cap K \subseteq M$ **using** *submodule-subset* [*OF* N] **by** *blast*
show $\mathbf{0}_M \in N \cap K$ **using** *submodule-zero* [*OF* N] *submodule-zero* [*OF* K] **by** *blast*
show $\bigwedge u v. \llbracket u \in N \cap K; v \in N \cap K \rrbracket \implies u \oplus v \in N \cap K$
using *submodule-add* [*OF* N] *submodule-add* [*OF* K] **by** *blast*
show $\bigwedge a v. \llbracket a \in R; v \in N \cap K \rrbracket \implies a \odot v \in N \cap K$
using *submodule-scale* [*OF* N] *submodule-scale* [*OF* K] **by** *blast*
qed

54.3 The sum of two submodules

The *sum* of two subsets of the module: all sums of an element of each. This is the module analogue of *ideal-sum* in *Chinese-Remainder*, and is the smallest submodule containing both summands when they are themselves submodules.

definition *submodule-sum* :: ' b set $\Rightarrow$ ' b set $\Rightarrow$ ' b set (infixl $\langle \oplus_S \rangle$ 65)

where $N \oplus_S K = \{u \oplus v \mid u \in N \wedge v \in K\}$

lemma *submodule-sum-memI*: $\llbracket u \in N; v \in K \rrbracket \implies u \oplus v \in N \oplus_S K$
unfolding *submodule-sum-def* **by** *blast*

lemma *submodule-sum-memE*:
assumes $x \in N \oplus_S K$
obtains $u \ v$ **where** $u \in N \ v \in K \ x = u \oplus v$
using *assms* **unfolding** *submodule-sum-def* **by** *blast*

The sum contains both summands, since each submodule contains $\mathbf{0}_M$.

lemma *submodule-sum-incl-left*:
assumes N : *submodule* N **and** K : *submodule* K **shows** $N \subseteq N \oplus_S K$
proof
fix x **assume** $x: x \in N$
then have $xM: x \in M$ **using** *submodule-subset[OF N]* **by** *blast*
have $x \oplus \mathbf{0}_M \in N \oplus_S K$ **using** x *submodule-zero[OF K]* **by** (*rule submodule-sum-memI*)
then show $x \in N \oplus_S K$ **using** xM **by** *simp*
qed

lemma *submodule-sum-incl-right*:
assumes N : *submodule* N **and** K : *submodule* K **shows** $K \subseteq N \oplus_S K$
proof
fix x **assume** $x: x \in K$
then have $xM: x \in M$ **using** *submodule-subset[OF K]* **by** *blast*
have $\mathbf{0}_M \oplus x \in N \oplus_S K$ **using** *submodule-zero[OF N]* x **by** (*rule submodule-sum-memI*)
then show $x \in N \oplus_S K$ **using** xM **by** *simp*
qed

The sum of two submodules is a submodule. Closure under addition needs commutativity of the module addition to re-pair the four summands; closure under scaling distributes.

theorem *submodule-sum-submodule*:
assumes N : *submodule* N **and** K : *submodule* K **shows** *submodule* $(N \oplus_S K)$
proof (*rule submoduleI*)
show *sub*: $N \oplus_S K \subseteq M$
proof
fix x **assume** $x \in N \oplus_S K$
then obtain $u \ v$ **where** $uv: u \in N \ v \in K \ x = u \oplus v$ **by** (*rule submodule-sum-memE*)
then show $x \in M$ **using** *submodule-subset[OF N]* *submodule-subset[OF K]* **by**
auto
qed
show $\mathbf{0}_M \in N \oplus_S K$
using *submodule-zero[OF N]* *submodule-zero[OF K]*
by (*metis madd.left-unit mzero-closed submodule-sum-memI*)
next
fix $x \ y$ **assume** $x \in N \oplus_S K$ **and** $y \in N \oplus_S K$
then obtain $u \ v \ u' \ v'$ **where**
 $a: u \in N \ v \in K \ x = u \oplus v$ **and** $b: u' \in N \ v' \in K \ y = u' \oplus v'$

by (*meson submodule-sum-memE*)
 have $uM: u \in M$ and $u'M: u' \in M$ using $a(1)$ $b(1)$ *submodule-subset[OF N]* by
auto
 have $vM: v \in M$ and $v'M: v' \in M$ using $a(2)$ $b(2)$ *submodule-subset[OF K]* by
auto
 — Re-pair $(u \oplus v) \oplus (u' \oplus v')$ as $(u \oplus u') \oplus (v \oplus v')$, using the confluent associative-commutative bundle of the additive group (bare *associative/commutative* do not reassociate reliably here).
 have $x \oplus y = (u \oplus v) \oplus (u' \oplus v')$ using $a(3)$ $b(3)$ by *simp*
 also have $\dots = (u \oplus u') \oplus (v \oplus v')$
 using uM vM $u'M$ $v'M$ by (*simp add: madd.ac*)
 finally have $x \oplus y = (u \oplus u') \oplus (v \oplus v')$.
 moreover have $u \oplus u' \in N$ using *submodule-add[OF N a(1) b(1)]* .
 moreover have $v \oplus v' \in K$ using *submodule-add[OF K a(2) b(2)]* .
 ultimately show $x \oplus y \in N \oplus_S K$ by (*simp add: submodule-sum-memI*)
 next
 fix a x assume $aR: a \in R$ and $xNK: x \in N \oplus_S K$
 — xNK named explicitly: a bare *then* would feed both assumptions to the rule.
 from xNK obtain u v where $uv: u \in N$ $v \in K$ $x = u \oplus v$ by (*rule submodule-sum-memE*)
 have $uM: u \in M$ using $uv(1)$ *submodule-subset[OF N]* by *blast*
 have $vM: v \in M$ using $uv(2)$ *submodule-subset[OF K]* by *blast*
 have $a \odot x = (a \odot u) \oplus (a \odot v)$
 using $uv(3)$ aR uM vM by (*simp add: scale-distrib-madd*)
 moreover have $a \odot u \in N$ using *submodule-scale[OF N aR uv(1)]* .
 moreover have $a \odot v \in K$ using *submodule-scale[OF K aR uv(2)]* .
 ultimately show $a \odot x \in N \oplus_S K$ by (*simp add: submodule-sum-memI*)
 qed

The sum is commutative, as a set.

lemma *submodule-sum-commute:*

assumes N : *submodule N* and K : *submodule K* shows $N \oplus_S K = K \oplus_S N$
 proof
 show $N \oplus_S K \subseteq K \oplus_S N$
 proof
 fix x assume $x \in N \oplus_S K$
 then obtain u v where $uv: u \in N$ $v \in K$ $x = u \oplus v$ by (*rule submodule-sum-memE*)
 have $uM: u \in M$ using $uv(1)$ *submodule-subset[OF N]* by *blast*
 have $vM: v \in M$ using $uv(2)$ *submodule-subset[OF K]* by *blast*
 have $x = v \oplus u$ using $uv(3)$ uM vM by (*simp add: madd.commutative*)
 then show $x \in K \oplus_S N$ using $uv(1,2)$ by (*simp add: submodule-sum-memI*)
 qed
 next
 show $K \oplus_S N \subseteq N \oplus_S K$
 proof
 fix x assume $x \in K \oplus_S N$
 then obtain u v where $uv: u \in K$ $v \in N$ $x = u \oplus v$ by (*rule submodule-sum-memE*)
 have $uM: u \in M$ using $uv(1)$ *submodule-subset[OF K]* by *blast*
 have $vM: v \in M$ using $uv(2)$ *submodule-subset[OF N]* by *blast*
 have $x = v \oplus u$ using $uv(3)$ uM vM by (*simp add: madd.commutative*)

then show $x \in N \oplus_S K$ using $uv(1,2)$ by (simp add: submodule-sum-memI)
 qed
 qed

A trivial intersection makes representations in the submodule sum unique. The existence of such a representation is already exactly *submodule-sum-memE*; this cancellation lemma records the additional content supplied by disjointness.

lemma *submodule-sum-decomposition-unique*:

assumes N : submodule N and K : submodule K
 and disjoint: $N \cap K = \{0_M\}$
 and u : $u \in N$ and v : $v \in K$
 and u' : $u' \in N$ and v' : $v' \in K$
 and eq: $u \oplus v = u' \oplus v'$
 shows $u = u' \wedge v = v'$
 proof –
 have uM : $u \in M$ and $u'M$: $u' \in M$
 using u u' submodule-subset[OF N] by auto
 have vM : $v \in M$ and $v'M$: $v' \in M$
 using v v' submodule-subset[OF K] by auto
 have $invu'M$: $\text{madd.inverse } u' \in M$ using $u'M$ by simp
 have $invvM$: $\text{madd.inverse } v \in M$ using vM by simp
 have differenceM : $u \oplus \text{madd.inverse } u' \in M$ using uM $invu'M$ by simp
 have difference-eq : $u \oplus \text{madd.inverse } u' = v' \oplus \text{madd.inverse } v$
 proof –
 have $(u \oplus \text{madd.inverse } u') \oplus v = (u \oplus v) \oplus \text{madd.inverse } u'$
 using uM $u'M$ vM by (simp add: madd.ac)
 also have $\dots = (u' \oplus v') \oplus \text{madd.inverse } u'$ using eq by simp
 also have $\dots = (v' \oplus u') \oplus \text{madd.inverse } u'$
 using madd.commutative[OF $u'M$ $v'M$] by simp
 also have $\dots = v' \oplus (u' \oplus \text{madd.inverse } u')$
 by (rule madd.associative[OF $v'M$ $u'M$ $invu'M$])
 also have $\dots = v'$ using $u'M$ $v'M$ by simp
 finally have cancellation : $(u \oplus \text{madd.inverse } u') \oplus v = v'$.
 have $((u \oplus \text{madd.inverse } u') \oplus v) \oplus \text{madd.inverse } v =$
 $(u \oplus \text{madd.inverse } u') \oplus (v \oplus \text{madd.inverse } v)$
 by (rule madd.associative[OF differenceM vM $invvM$])
 also have $\dots = u \oplus \text{madd.inverse } u'$ using differenceM vM by simp
 finally have recovered : $((u \oplus \text{madd.inverse } u') \oplus v) \oplus$
 $\text{madd.inverse } v = u \oplus \text{madd.inverse } u'$.
 have $u \oplus \text{madd.inverse } u' =$
 $((u \oplus \text{madd.inverse } u') \oplus v) \oplus \text{madd.inverse } v$
 using recovered by simp
 also have $\dots = v' \oplus \text{madd.inverse } v$ using cancellation by simp
 finally show ?thesis .
 qed
 have difference-N : $u \oplus \text{madd.inverse } u' \in N$
 using submodule-add[OF N u submodule-neg[OF N u']] .
 have difference-K : $u \oplus \text{madd.inverse } u' \in K$
 using difference-eq submodule-add[OF K v' submodule-neg[OF K v]] by simp

```

have difference-zero:  $u \oplus \text{madd.inverse } u' = \mathbf{0}_M$ 
  using difference-N difference-K disjoint by blast
have uu':  $u = u'$ 
proof -
  have madd.inverse  $u' \oplus u' = \mathbf{0}_M$ 
    using  $u'M$  by (simp add: madd.invertible-left-inverse)
  then show ?thesis
    using madd.inverse-unique  $uM$   $u'M$  difference-zero by blast
qed
have vv':  $v = v'$ 
  using eq uu'  $uM$   $vM$   $v'M$  by simp
show ?thesis using uu' vv' by blast
qed

```

54.4 Linear combinations

A linear combination of a finite set B of module elements with ring coefficients c : the module sum $\bigoplus_{v \in B} c \ v \odot v$.

definition *lincomb* :: $('b \Rightarrow 'a) \Rightarrow 'b \text{ set} \Rightarrow 'b$
where *lincomb* $c \ B = \text{madd.fincomp } (\lambda v. c \ v \odot v) \ B$

lemma *lincomb-closed*:
assumes $B: B \subseteq M$ **and** $c: \bigwedge v. v \in B \implies c \ v \in R$
shows $\text{lincomb } c \ B \in M$
unfolding *lincomb-def* **using** $B \ c$ **by** (auto intro!: madd.fincomp-closed scale-closed)

lemma *lincomb-empty* [simp]: $\text{lincomb } c \ \{\} = \mathbf{0}_M$
by (simp add: *lincomb-def*)

The span of a set of module elements: all linear combinations over its finite subsets.

definition *span* :: $'b \text{ set} \Rightarrow 'b$
where $\text{span } S = \{\text{lincomb } c \ B \mid c \ B. \text{ finite } B \wedge B \subseteq S \wedge (\forall v \in B. c \ v \in R)\}$

lemma *span-closed*:
assumes $S \subseteq M$ **and** $x \in \text{span } S$ **shows** $x \in M$
using *assms* **unfolding** *span-def* **by** (auto intro: *lincomb-closed*)

lemma *span-mono*: $S \subseteq T \implies \text{span } S \subseteq \text{span } T$
unfolding *span-def* **by** blast

end

54.5 A vector space is a module over a field

The module axioms are exactly those of theory *Vector-Space*, so any vector space is a module. (We record the relationship at the locale level; the field's

ring reduct supplies the scalar ring.) A dedicated theory could re-derive the theory *Vector-Space* development as the field specialisation of the present one.

end

55 Generators, independence, bases and free modules

```
theory Free-Module
imports Module-Basics
begin
```

Over a ring the two halves of “basis” come apart. A spanning set need not be linearly independent, a linearly independent set need not span, and — unlike the vector-space case — a linearly independent spanning set is genuinely stronger than either. So *spanning* and *lin-indep* are defined separately, and a module is *free* when it has a basis, meaning a set that is both.

These definitions are stated for arbitrary — possibly infinite — subsets. Finiteness is not part of either concept; it enters only through the finite index sets inside *lincomb* and *span*. This matters because *madd.fincomp* returns the unit on an infinite index set (*fincomp-infinite*), so a definition that applied *lincomb* directly to an infinite set would silently be talking about the zero of the module rather than about an infinite sum. Quantifying over finite subsets, as *lin-indep* does, avoids that trap; guarding the definition with *finite* would merely hide it.

Vector-Space defines its own finite-dimensional *basis* as bijectivity of the coordinate map, which is the right packaging for the cardinality count there but presumes finiteness. The bridging lemmas relating the two live in that theory, with *finite* as an explicit hypothesis where it does real work rather than baked into the concept.

The uniqueness-of-coordinates theorem *lin-indep-lincomb-unique* below is the reason linear independence is worth isolating: it is what makes a free module the right domain for defining a homomorphism by prescribing values on a basis.

```
context Module
begin
```

55.1 Coefficient functions

A *coefficient function on B* assigns a ring element to each element of *B*. We do not require extensionality: the linear combination *lincomb c B* only ever inspects *c* on *B*, so two coefficient functions agreeing on *B* give the same combination (*lincomb-cong* below).

```
definition coeffs-on :: ('b  $\Rightarrow$  'a)  $\Rightarrow$  'b set  $\Rightarrow$  bool
where coeffs-on c B  $\longleftrightarrow$  ( $\forall v \in B. c v \in R$ )
```

```
lemma coeffs-onI: ( $\bigwedge v. v \in B \implies c v \in R$ )  $\implies$  coeffs-on c B
```

by (simp add: coeffs-on-def)

lemma coeffs-onD: $\llbracket \text{coeffs-on } c \ B; v \in B \rrbracket \implies c \ v \in R$
 by (simp add: coeffs-on-def)

lemma coeffs-on-subset: $\llbracket \text{coeffs-on } c \ B; A \subseteq B \rrbracket \implies \text{coeffs-on } c \ A$
 by (auto simp: coeffs-on-def)

The zero coefficient function.

abbreviation zero-coeffs :: $'b \Rightarrow 'a$
 where zero-coeffs $\equiv (\lambda \cdot. \mathbf{0})$

lemma coeffs-on-zero [simp, intro]: coeffs-on zero-coeffs B
 by (simp add: coeffs-on-def)

A linear combination depends on the coefficients only through their values on the index set.

lemma lincomb-cong:
 assumes $B: B \subseteq M$ and eq: $\bigwedge v. v \in B \implies c \ v = d \ v$ and $d: \text{coeffs-on } d \ B$
 shows $\text{lincomb } c \ B = \text{lincomb } d \ B$
 unfolding lincomb-def
proof (rule madd.fincomp-cong')
 show $B = B$..
 show $(\lambda v. d \ v \odot v) \in B \rightarrow M$ using $B \ d$ by (auto intro: scale-closed coeffs-onD)
 show $\bigwedge v. v \in B \implies c \ v \odot v = d \ v \odot v$ using eq by simp
qed

The zero combination. Every term is $\mathbf{0} \odot v = \mathbf{0}_M$.

lemma lincomb-zero-coeffs [simp]:
 assumes $B: B \subseteq M$ shows $\text{lincomb } \text{zero-coeffs } B = \mathbf{0}_M$
 unfolding lincomb-def
 using B by (auto intro!: madd.fincomp-unit-eqI scale-zero-scalar)

Scaling distributes over a finite module sum. Proved by induction on the index set; kept as a named lemma because inline it is slow enough to hit the proof timeout.

lemma scale-fincomp:
 assumes $a: a \in R$ and $f: f \in A \rightarrow M$
 shows $a \odot \text{madd.fincomp } f \ A = \text{madd.fincomp } (\lambda v. a \odot f \ v) \ A$
 using f
proof (induct A rule: infinite-finite-induct)
 case (infinite A) then show ?case using a by (simp add: scale-zero-elem)
next
 case empty then show ?case using a by (simp add: scale-zero-elem)
next
 case (insert $v \ A$)
 then have $fv: f \ v \in M$ and $fA: f \in A \rightarrow M$ by auto
 have $sc: (\lambda u. a \odot f \ u) \in A \rightarrow M$ using $a \ fA$ by (auto intro: scale-closed)

```

have a  $\odot$  madd.fincomp f (insert v A) = a  $\odot$  (f v  $\oplus$  madd.fincomp f A)
  using insert fv fA by simp
also have ... = (a  $\odot$  f v)  $\oplus$  (a  $\odot$  madd.fincomp f A)
  using a fv fA by (auto intro!: scale-distrib-madd madd.fincomp-closed)
also have ... = (a  $\odot$  f v)  $\oplus$  madd.fincomp ( $\lambda u.$  a  $\odot$  f u) A
  using insert fA by simp
also have ... = madd.fincomp ( $\lambda u.$  a  $\odot$  f u) (insert v A)
  using insert a fv sc by (auto intro!: madd.fincomp-insert[symmetric] scale-closed)
finally show ?case .
qed

```

Enlarging the index set by elements whose coefficient is $\mathbf{0}$ does not change the combination: the extra terms are all $\mathbf{0} \odot v = \mathbf{0}_M$. This is what lets two combinations over different finite index sets be added on their union.

```

lemma lincomb-mono-zero:
  assumes B: finite B and AB:  $A \subseteq B$  and BM:  $B \subseteq M$ 
    and zero:  $\bigwedge v. v \in B - A \implies c v = \mathbf{0}$  and c: coeffs-on c B
  shows lincomb c A = lincomb c B
  unfolding lincomb-def
proof (rule madd.fincomp-mono-neutral-cong-left)
  show finite B and  $A \subseteq B$  using B AB by simp-all
  show  $\bigwedge v. v \in B - A \implies c v \odot v = \mathbf{0}_M$ 
    using zero BM by (auto simp: scale-zero-scalar)
  show  $\bigwedge v. v \in A \implies c v \odot v = c v \odot v$  by simp
  show ( $\lambda v. c v \odot v$ )  $\in B \rightarrow M$ 
    using BM c by (auto intro!: scale-closed intro: coeffs-onD)
qed

```

55.2 Membership in the span

Introduction and elimination for *span*, in the form the proofs below want.

```

lemma spanI:
  assumes finite A and  $A \subseteq S$  and coeffs-on c A and  $x = \text{lincomb } c A$ 
  shows  $x \in \text{span } S$ 
  using assms unfolding span-def coeffs-on-def by blast

```

```

lemma spanE:
  assumes  $x \in \text{span } S$ 
  obtains c A where finite A  $A \subseteq S$  coeffs-on c A  $x = \text{lincomb } c A$ 
  using assms unfolding span-def coeffs-on-def by blast

```

A combination over a singleton is just the scaling. Kept separate: unfolding *lincomb* to *madd.fincomp* inline makes the simplifier work far harder than it needs to (it timed out at 12s in an earlier draft).

```

lemma lincomb-singleton [simp]:
  assumes v:  $v \in M$  and c:  $c v \in R$ 
  shows  $\text{lincomb } c \{v\} = c v \odot v$ 
  unfolding lincomb-def using v c by (simp add: scale-closed)

```


Each element of a set lies in its span: take the singleton combination with coefficient **1**.

lemma *span-incl*:

assumes $S: S \subseteq M$ **shows** $S \subseteq \text{span } S$

proof

fix v **assume** $v: v \in S$

then have $vM: v \in M$ **using** S **by** *blast*

have $\text{lincomb } (\lambda\cdot. \mathbf{1}) \{v\} = \mathbf{1} \odot v$ **using** vM **by** *simp*

also have $\dots = v$ **using** vM **by** (*rule scale-one*)

finally have $v = \text{lincomb } (\lambda\cdot. \mathbf{1}) \{v\}$ **..**

then show $v \in \text{span } S$ **using** v **by** (*auto intro!; spanI[where $A = \{v\}$] coeffs-onI*)

qed

Adding coefficient functions pointwise adds the combinations. Both the span-closure proof and the uniqueness proof need this; factored out so each does the *madd.fincomp* manipulation once.

lemma *lincomb-add-coeffs*:

assumes $AM: A \subseteq M$ **and** $c: \text{coeffs-on } c \ A$ **and** $d: \text{coeffs-on } d \ A$

shows $\text{lincomb } (\lambda v. c \ v + d \ v) \ A = \text{lincomb } c \ A \oplus \text{lincomb } d \ A$

unfolding *lincomb-def*

proof (*rule trans*)

show $\text{madd.fincomp } (\lambda v. (c \ v + d \ v) \odot v) \ A$

$= \text{madd.fincomp } (\lambda v. (c \ v \odot v) \oplus (d \ v \odot v)) \ A$

using $AM \ c \ d$

by (*intro madd.fincomp-cong'*)

(*auto intro!; scale-distrib-add scale-closed madd-closed intro: coeffs-onD*)

show $\text{madd.fincomp } (\lambda v. (c \ v \odot v) \oplus (d \ v \odot v)) \ A$

$= \text{madd.fincomp } (\lambda v. c \ v \odot v) \ A \oplus \text{madd.fincomp } (\lambda v. d \ v \odot v) \ A$

using $AM \ c \ d$

by (*intro madd.fincomp-comp*) (*auto intro!; scale-closed intro: coeffs-onD*)

qed

The span is a submodule. Closure under addition merges two finite index sets; closure under scaling multiplies through the coefficients.

lemma *span-zero* [*simp, intro*]: $\mathbf{0}_M \in \text{span } S$

by (*rule spanI[where $A = \{\}$ and $c = \text{zero-coeffs}$] auto*)

lemma *span-madd-closed*:

assumes $S: S \subseteq M$ **and** $x: x \in \text{span } S$ **and** $y: y \in \text{span } S$

shows $x \oplus y \in \text{span } S$

proof —

from x **obtain** $c \ A$ **where** $A: \text{finite } A \ A \subseteq S \ \text{coeffs-on } c \ A \ x = \text{lincomb } c \ A$

by (*rule spanE*)

from y **obtain** $d \ B$ **where** $B: \text{finite } B \ B \subseteq S \ \text{coeffs-on } d \ B \ y = \text{lincomb } d \ B$

by (*rule spanE*)

— Extend both coefficient functions by $\mathbf{0}$ off their own index sets and add them on the union; each combination is unchanged by *lincomb-mono-zero*.

define c' **where** $c' = (\lambda v. \text{if } v \in A \text{ then } c \ v \text{ else } \mathbf{0})$

```

define d' where d' = ( $\lambda v. \text{if } v \in B \text{ then } d \ v \text{ else } \mathbf{0}$ )
have UM:  $A \cup B \subseteq M$  using A(2) B(2) S by blast
have finU: finite ( $A \cup B$ ) using A(1) B(1) by simp
have c'R: coeffs-on c' ( $A \cup B$ )
proof (rule coeffs-onI)
  fix v assume v  $\in A \cup B$ 
  show c' v  $\in R$  unfolding c'-def using coeffs-onD[OF A(3)] by simp
qed
have d'R: coeffs-on d' ( $A \cup B$ )
proof (rule coeffs-onI)
  fix v assume v  $\in A \cup B$ 
  show d' v  $\in R$  unfolding d'-def using coeffs-onD[OF B(3)] by simp
qed
— Each extended combination over the union agrees with the original over its own
index set, because the added coefficients are  $\mathbf{0}$ .
have cA: lincomb c' ( $A \cup B$ ) = lincomb c A
proof —
  have lincomb c' A = lincomb c' ( $A \cup B$ )
  proof (rule lincomb-mono-zero[OF finU Un-upper1 UM])
    show  $\bigwedge v. v \in A \cup B - A \implies c' v = \mathbf{0}$  unfolding c'-def by simp
    show coeffs-on c' ( $A \cup B$ ) by (rule c'R)
  qed
  moreover have lincomb c' A = lincomb c A
  proof (rule lincomb-cong)
    show  $A \subseteq M$  using A(2) S by blast
    show  $\bigwedge v. v \in A \implies c' v = c v$  unfolding c'-def by simp
    show coeffs-on c A by (rule A(3))
  qed
  ultimately show ?thesis by simp
qed
have dB: lincomb d' ( $A \cup B$ ) = lincomb d B
proof —
  have lincomb d' B = lincomb d' ( $A \cup B$ )
  proof (rule lincomb-mono-zero[OF finU Un-upper2 UM])
    show  $\bigwedge v. v \in A \cup B - B \implies d' v = \mathbf{0}$  unfolding d'-def by simp
    show coeffs-on d' ( $A \cup B$ ) by (rule d'R)
  qed
  moreover have lincomb d' B = lincomb d B
  proof (rule lincomb-cong)
    show  $B \subseteq M$  using B(2) S by blast
    show  $\bigwedge v. v \in B \implies d' v = d v$  unfolding d'-def by simp
    show coeffs-on d B by (rule B(3))
  qed
  ultimately show ?thesis by simp
qed
have sum: lincomb ( $\lambda v. c' v + d' v$ ) ( $A \cup B$ ) =  $x \oplus y$ 
  using lincomb-add-coeffs[OF UM c'R d'R] cA dB A(4) B(4) by simp
have sumR: coeffs-on ( $\lambda v. c' v + d' v$ ) ( $A \cup B$ )
  by (rule coeffs-onI) (use coeffs-onD[OF c'R] coeffs-onD[OF d'R] in simp)
show ?thesis

```

by (rule spanI[OF finU - sumR sum[symmetric]]) (use A(2) B(2) in blast)
 qed

lemma span-scale-closed:
 assumes $S: S \subseteq M$ and $a: a \in R$ and $x: x \in \text{span } S$
 shows $a \odot x \in \text{span } S$
proof –
 from x obtain $c \ A$ where $A: \text{finite } A \ A \subseteq S \ \text{coeffs-on } c \ A \ x = \text{lincomb } c \ A$
 by (rule spanE)
 have $AM: A \subseteq M$ using A(2) S by blast
 have $pi: (\lambda v. c \ v \odot v) \in A \rightarrow M$
 using $AM \ A(3)$ by (auto intro!: scale-closed intro: coeffs-onD)
 have $a \odot \text{lincomb } c \ A = \text{lincomb } (\lambda v. a \cdot c \ v) \ A$
 unfolding lincomb-def
proof (rule trans)
 show $a \odot \text{madd.fincomp } (\lambda v. c \ v \odot v) \ A = \text{madd.fincomp } (\lambda v. a \odot (c \ v \odot v)) \ A$
 using $a \ pi$ by (rule scale-fincomp)
 — scale-scale is oriented $(a \cdot b) \odot v = a \odot b \odot v$, so each term is rewritten
 right-to-left, one element at a time.
 show $\text{madd.fincomp } (\lambda v. a \odot (c \ v \odot v)) \ A = \text{madd.fincomp } (\lambda v. (a \cdot c \ v) \odot v) \ A$
proof (rule madd.fincomp-cong')
 show $A = A$..
 show $(\lambda v. (a \cdot c \ v) \odot v) \in A \rightarrow M$
 using $AM \ a \ \text{coeffs-onD}[OF \ A(3)]$ by (auto intro!: scale-closed)
 fix v assume $v: v \in A$
 then have $vM: v \in M$ and $cv: c \ v \in R$ using $AM \ \text{coeffs-onD}[OF \ A(3)]$ by auto
 show $a \odot (c \ v \odot v) = (a \cdot c \ v) \odot v$ by (rule scale-scale[OF a cv vM, symmetric])
 qed
 qed
 moreover have $\text{coeffs-on } (\lambda v. a \cdot c \ v) \ A$
 by (rule coeffs-onI) (use a coeffs-onD[OF A(3)] in simp)
 ultimately show ?thesis using A by (blast intro: spanI)
 qed

theorem span-submodule:
 assumes $S: S \subseteq M$ shows submodule (span S)
proof (rule submoduleI)
 show $\text{span } S \subseteq M$ using S by (auto intro: span-closed)
 show $0_M \in \text{span } S$ by simp
 show $\bigwedge u \ v. \llbracket u \in \text{span } S; v \in \text{span } S \rrbracket \implies u \oplus v \in \text{span } S$
 using S by (rule span-madd-closed)
 show $\bigwedge a \ v. \llbracket a \in R; v \in \text{span } S \rrbracket \implies a \odot v \in \text{span } S$
 using S by (rule span-scale-closed)
 qed

The span is the least submodule containing its generators. This elimination
 form complements *span-submodule*: any submodule containing S contains every
 finite linear combination of elements of S .

theorem span-minimal:

```

assumes  $N$ : submodule  $N$  and  $SN$ :  $S \subseteq N$ 
shows  $\text{span } S \subseteq N$ 
proof
  fix  $x$  assume  $x \in \text{span } S$ 
  then obtain  $c A$  where  $A$ : finite  $A$   $A \subseteq S$   $\text{coeffs-on } c A$   $x = \text{lincomb } c A$ 
    by (rule  $\text{spanE}$ )
  have  $\text{lincomb-in-}N$ :  $\text{lincomb } c A \in N$ 
    if finite  $A$   $A \subseteq N$   $\text{coeffs-on } c A$  for  $A$   $c$ 
    using that
  proof (induction  $A$  arbitrary:  $c$  rule: finite-induct)
    case empty
    then show ?case using submodule-zero[OF  $N$ ] by simp
  next
    case (insert  $v A$ )
    have  $vN$ :  $v \in N$  and  $AN$ :  $A \subseteq N$  using insert.prem by auto
    have  $NM$ :  $N \subseteq M$  by (rule submodule-subset[OF  $N$ ])
    have  $vM$ :  $v \in M$  and  $AM$ :  $A \subseteq M$  using  $vN$   $AN$   $NM$  by auto
    have  $cv$ :  $c v \in R$  by (rule coeffs-onD[OF insert.prem(2)]) simp
    have  $cA$ :  $\text{coeffs-on } c A$  by (rule coeffs-on-subset[OF insert.prem(2)]) blast
    have terms:  $(\lambda u. c u \odot u) \in A \rightarrow M$ 
      using  $AM$   $cA$  by (auto intro!: scale-closed intro: coeffs-onD)
    have  $eq$ :  $\text{lincomb } c (\text{insert } v A) = (c v \odot v) \oplus \text{lincomb } c A$ 
      unfolding  $\text{lincomb-def}$  using insert.hyps terms  $cv$   $vM$  by (simp add: scale-closed)
    have  $\text{term-}N$ :  $c v \odot v \in N$  by (rule submodule-scale[OF  $N$   $cv$   $vN$ ])
    have  $\text{rest-}N$ :  $\text{lincomb } c A \in N$  by (rule insert.IH[OF  $AN$   $cA$ ])
    have  $(c v \odot v) \oplus \text{lincomb } c A \in N$ 
      by (rule submodule-add[OF  $N$   $\text{term-}N$   $\text{rest-}N$ ])
    then show ?case using  $eq$  by simp
  qed
have  $\text{lincomb } c A \in N$ 
  by (rule  $\text{lincomb-in-}N$ [OF  $A(1)$  -  $A(3)$ ]) (use  $A(2)$   $SN$  in blast)
then show  $x \in N$  using  $A(4)$  by simp
qed

```

55.3 Generating sets, independence, bases

S generates the module when its span is everything.

definition $\text{spanning} :: 'b \text{ set} \Rightarrow \text{bool}$
where $\text{spanning } S \iff S \subseteq M \wedge \text{span } S = M$

lemma spanningI : $\llbracket S \subseteq M; \bigwedge v. v \in M \implies v \in \text{span } S \rrbracket \implies \text{spanning } S$
unfolding spanning-def **using** span-closed **by** *blast*

lemma spanningD : $\llbracket \text{spanning } S; v \in M \rrbracket \implies v \in \text{span } S$
by (simp add: spanning-def)

The whole module generates itself.

lemma spanning-whole : $\text{spanning } M$
by (rule spanningI) (auto intro: span-incl[THEN subsetD])

S is *linearly independent* when no non-trivial finite combination of *distinct* elements vanishes: whenever a finite $A \subseteq S$ has $\text{lincomb } c \ A = \mathbf{0}_M$, all the coefficients on A are $\mathbf{0}$. Over a ring this is the right notion — note it forbids torsion among the generators, which is why not every module is free.

definition *lin-indep* :: 'b set $\Rightarrow$ bool
where *lin-indep* $S \longleftrightarrow S \subseteq M \wedge$
 $(\forall c \ A. \text{finite } A \longrightarrow A \subseteq S \longrightarrow \text{coeffs-on } c \ A \longrightarrow \text{lincomb } c \ A = \mathbf{0}_M \longrightarrow (\forall v \in A. c \ v = \mathbf{0}))$

lemma *lin-indepI*:

$\llbracket S \subseteq M;$
 $\bigwedge c \ A \ v. \llbracket \text{finite } A; A \subseteq S; \text{coeffs-on } c \ A; \text{lincomb } c \ A = \mathbf{0}_M; v \in A \rrbracket \Longrightarrow c \ v =$
 $\mathbf{0} \rrbracket$
 $\Longrightarrow \text{lin-indep } S$
unfolding *lin-indep-def* **by** *blast*

lemma *lin-indepD*:

$\llbracket \text{lin-indep } S; \text{finite } A; A \subseteq S; \text{coeffs-on } c \ A; \text{lincomb } c \ A = \mathbf{0}_M; v \in A \rrbracket \Longrightarrow c \ v =$
 $\mathbf{0}$
unfolding *lin-indep-def* **by** *blast*

lemma *lin-indep-subset*: $\llbracket \text{lin-indep } S; T \subseteq S \rrbracket \Longrightarrow \text{lin-indep } T$

unfolding *lin-indep-def* **using** *subset-trans* **by** *blast*

The empty set is linearly independent.

lemma *lin-indep-empty* [*simp*, *intro*]: *lin-indep* $\{\}$

by (*rule lin-indepI*) *auto*

Independence rules out $\mathbf{0}_M$ as a member — the combination $\mathbf{1} \odot \mathbf{0}_M$ vanishes with coefficient $\mathbf{1}$ — but *only over a non-trivial ring*. The hypothesis $\mathbf{1} \neq \mathbf{0}$ is genuinely needed: *Module* extends *Ring*, which does not assume it, and over the trivial ring $\mathbf{1} = \mathbf{0}$ makes every coefficient zero, so $\{\mathbf{0}_M\}$ counts as *lin-indep*.

lemma *lin-indep-not-mzero*:

assumes $S: \text{lin-indep } S$ **and** $\text{nz}: \mathbf{1} \neq \mathbf{0}$ **shows** $\mathbf{0}_M \notin S$

proof

assume $z: \mathbf{0}_M \in S$

have $\text{lincomb } (\lambda_. \mathbf{1}) \ \{\mathbf{0}_M\} = \mathbf{1} \odot \mathbf{0}_M$ **by** *simp*

also have $\dots = \mathbf{0}_M$ **by** (*rule scale-one*) *simp*

finally have $lc: \text{lincomb } (\lambda_. \mathbf{1}) \ \{\mathbf{0}_M\} = \mathbf{0}_M$.

have $(\lambda_. \mathbf{1}) \ \mathbf{0}_M = \mathbf{0}$

by (*rule lin-indepD[OF S - - lc]*) (*use z in <auto intro!> coeffs-onI*)

then have $\mathbf{1} = \mathbf{0}$ **by** *simp*

then show *False* **using** *nz* **by** *simp*

qed

A *basis* is a linearly independent spanning set; a module is *free* when it has one.

definition *module-basis* :: 'b set $\Rightarrow$ bool

where *module-basis* $B \longleftrightarrow \text{spanning } B \wedge \text{lin-indep } B$

```

definition free :: bool
  where free  $\longleftrightarrow (\exists B. \text{module-basis } B)$ 

lemma module-basisI:  $\llbracket \text{spanning } B; \text{lin-indep } B \rrbracket \implies \text{module-basis } B$ 
  by (simp add: module-basis-def)

lemma module-basis-spanning:  $\text{module-basis } B \implies \text{spanning } B$ 
  by (simp add: module-basis-def)

lemma module-basis-lin-indep:  $\text{module-basis } B \implies \text{lin-indep } B$ 
  by (simp add: module-basis-def)

lemma freeI:  $\text{module-basis } B \implies \text{free}$ 
  unfolding free-def by blast

```

The trivial module is free, on the empty basis: its only element $\mathbf{0}_M$ is the empty linear combination, which lies in $\text{span } \{\}$.

```

lemma module-basis-empty-trivial:
  assumes triv:  $M = \{\mathbf{0}_M\}$  shows module-basis  $\{\}$ 
proof (rule module-basisI)
  show spanning  $\{\}$ 
  proof (rule spanningI)
    show  $\{\} \subseteq M$  by simp
    fix v assume v  $\in M$ 
    then have v =  $\mathbf{0}_M$  using triv by blast
    then show v  $\in \text{span } \{\}$  by simp
  qed
qed auto

```

```

lemma free-trivial:
  assumes  $M = \{\mathbf{0}_M\}$  shows free
  using module-basis-empty-trivial[OF assms] by (rule freeI)

```

55.4 Uniqueness of coordinates

The point of independence. Over a linearly independent set the coefficients of a linear combination are determined by its value. Stated for a common finite index set A : if two coefficient functions give the same combination, they agree on A .

The proof forms the difference of the two coefficient functions. Its combination is $\mathbf{0}_M$, so independence forces every difference to vanish, hence $c \ v = d \ v$ by cancellation in the ring's additive group.

```

theorem lin-indep-lincomb-unique:
  assumes S: lin-indep S and A: finite A  $A \subseteq S$ 
  and c: coeffs-on c A and d: coeffs-on d A
  and eq: lincomb c A = lincomb d A
  and v: v  $\in A$ 

```

shows $c v = d v$
proof —
have $AM: A \subseteq M$ **using** $A(2)$ S **unfolding** *lin-indep-def* **by** *blast*
— The difference function, and the observation that its combination is the difference of the combinations — so it vanishes.
define e **where** $e = (\lambda u. c u + (- d u))$
have $negd: coeffs-on (\lambda u. - d u) A$
by (*rule coeffs-onI*) (*use coeffs-onD[OF d] in simp*)
have $eR: coeffs-on e A$
unfolding *e-def*
by (*rule coeffs-onI*)
(*use coeffs-onD[OF c] coeffs-onD[OF negd] in simp*)
have $lincomb e A = lincomb c A \oplus madd.inverse (lincomb d A)$
proof —
have $lincomb e A = lincomb c A \oplus lincomb (\lambda u. - d u) A$
unfolding *e-def* **by** (*rule lincomb-add-coeffs[OF AM c negd]*)
also have $lincomb (\lambda u. - d u) A = madd.inverse (lincomb d A)$
unfolding *lincomb-def*
proof (*rule trans*)
show $madd.fincomp (\lambda u. (- d u) \odot u) A = madd.fincomp (\lambda u. madd.inverse (d u \odot u)) A$
proof (*rule madd.fincomp-cong'*)
show $A = A$..
show $(\lambda u. madd.inverse (d u \odot u)) \in A \rightarrow M$
using AM *coeffs-onD[OF d]* **by** (*auto intro!: scale-closed*)
fix u **assume** $u: u \in A$
then have $uM: u \in M$ **and** $du: d u \in R$ **using** AM *coeffs-onD[OF d]* **by** *auto*
show $(- d u) \odot u = madd.inverse (d u \odot u)$ **by** (*rule scale-neg-scalar[OF du uM]*)
qed
show $madd.fincomp (\lambda u. madd.inverse (d u \odot u)) A = madd.inverse (madd.fincomp (\lambda u. d u \odot u) A)$
using AM d **by** (*intro madd.fincomp-inverse*) (*auto intro!: scale-closed intro: coeffs-onD*)
qed
finally show *?thesis* .
qed
also have $\dots = \mathbf{0}_M$
using *eq lincomb-closed[OF AM]* $d c$
by (*simp add: madd.invertible-right-inverse coeffs-on-def*)
finally have $lincomb e A = \mathbf{0}_M$.
— Independence kills every coefficient of the difference.
then have $e v = \mathbf{0}$ **using** S A eR v **by** (*rule-tac lin-indepD*) *auto*
then have $cd: c v + (- d v) = \mathbf{0}$ **unfolding** *e-def* .
— Cancel the common summand $- d v$: both $c v$ and $d v$ added to it give $\mathbf{0}$. (Stated by explicit cancellation rather than left to *metis*, which on goals of this shape tends not to terminate.)
have $cvR: c v \in R$ **and** $dvR: d v \in R$ **using** *coeffs-onD[OF c v]* *coeffs-onD[OF d v]*
by *simp-all*
have $ndvR: - d v \in R$ **using** dvR **by** *simp*

```

have  $d\ v + (-\ d\ v) = \mathbf{0}$  using  $dvR$  by simp
with  $cd$  have  $eq2: c\ v + (-\ d\ v) = d\ v + (-\ d\ v)$  by simp
show  $c\ v = d\ v$ 
  by (rule additive.invertible-right-cancel
      [OF additive.invertible[OF ndvR] ndvR cvR dvR, THEN iffD1, OF eq2])
qed

```

Consequently, for a basis every module element has *unique* coordinates on any finite index set: existence from generation, uniqueness from independence.

corollary *module-basis-coords-unique*:

```

assumes  $B$ : module-basis  $B$  and  $A$ : finite  $A\ A \subseteq B$ 
  and  $c$ : coeffs-on  $c\ A$  and  $d$ : coeffs-on  $d\ A$ 
  and  $eq$ : lincomb  $c\ A = \text{lincomb } d\ A$ 
shows  $\forall v \in A. c\ v = d\ v$ 
using lin-indep-lincomb-unique[OF module-basis-lin-indep[OF  $B$ ]  $A\ c\ d\ eq$ ] by blast

```

55.5 Coordinates over a basis

For the universal property below we need uniqueness across *different* index sets, not just a common one: two representations of the same element, over possibly different finite subsets of an independent set, must agree once each coefficient function is extended by $\mathbf{0}$ outside its own index set. The proof pushes both representations onto the union of the two index sets — which changes neither combination, by *lincomb-mono-zero* — and then applies *lin-indep-lincomb-unique* there.

theorem *lin-indep-lincomb-unique-gen*:

```

assumes  $S$ : lin-indep  $S$ 
  and  $A$ : finite  $A\ A \subseteq S$  and  $A'$ : finite  $A'\ A' \subseteq S$ 
  and  $c$ : coeffs-on  $c\ A$  and  $d$ : coeffs-on  $d\ A'$ 
  and  $eq$ : lincomb  $c\ A = \text{lincomb } d\ A'$ 
shows (if  $v \in A$  then  $c\ v$  else  $\mathbf{0}$ ) = (if  $v \in A'$  then  $d\ v$  else  $\mathbf{0}$ )
proof -
  have  $SM: S \subseteq M$  using  $S$  unfolding lin-indep-def by blast
  have  $UM: A \cup A' \subseteq M$  using  $A(2)\ A'(2)\ SM$  by blast
  have  $finU$ : finite  $(A \cup A')$  using  $A(1)\ A'(1)$  by simp
  have  $US: A \cup A' \subseteq S$  using  $A(2)\ A'(2)$  by blast
  define  $c'$  where  $c' = (\lambda u. \text{if } u \in A \text{ then } c\ u \text{ else } \mathbf{0})$ 
  define  $d'$  where  $d' = (\lambda u. \text{if } u \in A' \text{ then } d\ u \text{ else } \mathbf{0})$ 
  have  $c'R$ : coeffs-on  $c'\ (A \cup A')$ 
    by (rule coeffs-onI) (use coeffs-onD[OF  $c$ ] in ⟨simp add:  $c'$ -def⟩)
  have  $d'R$ : coeffs-on  $d'\ (A \cup A')$ 
    by (rule coeffs-onI) (use coeffs-onD[OF  $d$ ] in ⟨simp add:  $d'$ -def⟩)
  — Each extended combination over the union agrees with the original over its own
  index set.
  have  $cU$ : lincomb  $c'\ (A \cup A') = \text{lincomb } c\ A$ 
proof -
  have lincomb  $c'\ A = \text{lincomb } c'\ (A \cup A')$ 
  proof (rule lincomb-mono-zero[OF  $finU\ Un-upper1\ UM$ ])

```



```

    show  $\bigwedge u. u \in A \cup A' - A \implies c' u = \mathbf{0}$  unfolding  $c'$ -def by simp
    show coeffs-on  $c' (A \cup A')$  by (rule  $c'R$ )
  qed
  moreover have  $\text{lincomb } c' A = \text{lincomb } c A$ 
  proof (rule lincomb-cong)
    show  $A \subseteq M$  using  $A(2)$  SM by blast
    show  $\bigwedge u. u \in A \implies c' u = c u$  unfolding  $c'$ -def by simp
    show coeffs-on  $c A$  by (rule  $c$ )
  qed
  ultimately show ?thesis by simp
  qed
  have  $dU: \text{lincomb } d' (A \cup A') = \text{lincomb } d A'$ 
  proof -
    have  $\text{lincomb } d' A' = \text{lincomb } d' (A \cup A')$ 
    proof (rule lincomb-mono-zero[OF finU Un-upper2 UM])
      show  $\bigwedge u. u \in A \cup A' - A' \implies d' u = \mathbf{0}$  unfolding  $d'$ -def by simp
      show coeffs-on  $d' (A \cup A')$  by (rule  $d'R$ )
    qed
    moreover have  $\text{lincomb } d' A' = \text{lincomb } d A'$ 
    proof (rule lincomb-cong)
      show  $A' \subseteq M$  using  $A'(2)$  SM by blast
      show  $\bigwedge u. u \in A' \implies d' u = d u$  unfolding  $d'$ -def by simp
      show coeffs-on  $d A'$  by (rule  $d$ )
    qed
    ultimately show ?thesis by simp
  qed
  from  $\text{eq } cU \ dU$  have  $\text{eq}U: \text{lincomb } c' (A \cup A') = \text{lincomb } d' (A \cup A')$  by simp
  show ?thesis
  proof (cases  $v \in A \cup A'$ )
    case True
    then have  $c' v = d' v$ 
      by (rule lin-indep-lincomb-unique[OF S finU US c'R d'R eqU])
    then show ?thesis unfolding  $c'$ -def  $d'$ -def .
  next
    case False
    then show ?thesis by simp
  qed
  qed

```

Spans generated by disjoint parts of one independent set meet only at zero. The generating sets themselves need not be finite: span membership supplies the two finite supports on which uniqueness of coordinates is applied.

theorem *span-inter-eq-trivial*:

assumes S : *lin-indep* S **and** AS : $A \subseteq S$ **and** BS : $B \subseteq S$

and *disjoint*: $A \cap B = \{\}$

shows $\text{span } A \cap \text{span } B = \{\mathbf{0}_M\}$

proof

show $\text{span } A \cap \text{span } B \subseteq \{\mathbf{0}_M\}$

proof

```

fix  $x$  assume  $x: x \in \text{span } A \cap \text{span } B$ 
have  $xA: x \in \text{span } A$  and  $xB: x \in \text{span } B$  using  $x$  by simp-all
from  $xA$  obtain  $c \ C$  where  $C: \text{finite } C \ C \subseteq A \ \text{coeffs-on } c \ C$ 
 $x = \text{lincomb } c \ C$ 
by (rule spanE)
from  $xB$  obtain  $d \ D$  where  $D: \text{finite } D \ D \subseteq B \ \text{coeffs-on } d \ D$ 
 $x = \text{lincomb } d \ D$ 
by (rule spanE)
have  $CS: C \subseteq S$  using  $C(2) \ AS$  by blast
have  $DS: D \subseteq S$  using  $D(2) \ BS$  by blast
have  $eq: \text{lincomb } c \ C = \text{lincomb } d \ D$  using  $C(4) \ D(4)$  by simp
have  $c\text{-zero}: c \ v = \mathbf{0}$  if  $v: v \in C$  for  $v$ 
proof –
  have  $v \notin D$  using  $v \ C(2) \ D(2)$  disjoint by blast
  moreover have (if  $v \in C$  then  $c \ v$  else  $\mathbf{0}$ ) =
    (if  $v \in D$  then  $d \ v$  else  $\mathbf{0}$ )
    by (rule lin-indep-lincomb-unique-gen[OF  $S \ C(1) \ CS \ D(1) \ DS \ C(3) \ D(3) \ eq$ ])
  ultimately show ?thesis using  $v$  by simp
qed
have  $CM: C \subseteq M$  using  $CS \ S$  unfolding lin-indep-def by blast
have  $x = \text{lincomb } c \ C$  by (rule C(4))
also have  $\dots = \text{lincomb } \text{zero-coeffs } C$ 
  by (rule lincomb-cong[OF  $CM \ c\text{-zero} \ \text{coeffs-on-zero}$ ])
also have  $\dots = \mathbf{0}_M$  using  $CM$  by simp
finally show  $x \in \{\mathbf{0}_M\}$  by simp
qed
show  $\{\mathbf{0}_M\} \subseteq \text{span } A \cap \text{span } B$  by simp
qed

```

Every element of a module with a basis has a representation over some finite subset of it.

lemma *module-basis-repr-exists:*

```

assumes  $B: \text{module-basis } B$  and  $x: x \in M$ 
obtains  $A \ c$  where  $\text{finite } A \ A \subseteq B \ \text{coeffs-on } c \ A \ x = \text{lincomb } c \ A$ 
proof –
  have  $x \in \text{span } B$  by (rule spanningD[OF module-basis-spanning[OF  $B$ ]  $x$ ])
  then show thesis using that by (rule spanE)
qed

```

And that representation is unique in the sense of *lin-indep-lincomb-unique-gen*: the coefficients, extended by $\mathbf{0}$, are determined by the element.

corollary *module-basis-repr-unique:*

```

assumes  $B: \text{module-basis } B$ 
and  $A: \text{finite } A \ A \subseteq B$  and  $A': \text{finite } A' \ A' \subseteq B$ 
and  $c: \text{coeffs-on } c \ A$  and  $d: \text{coeffs-on } d \ A'$ 
and  $eq: \text{lincomb } c \ A = \text{lincomb } d \ A'$ 
shows (if  $v \in A$  then  $c \ v$  else  $\mathbf{0}$ ) = (if  $v \in A'$  then  $d \ v$  else  $\mathbf{0}$ )
by (rule lin-indep-lincomb-unique-gen
  [OF module-basis-lin-indep[OF  $B$ ]  $A \ A' \ c \ d \ eq$ ])

```

end

end

56 Finite-dimensional vector spaces over a field

```
theory Vector-Space
  imports Free-Module
begin
```

The import of *FiniteProduct* (through *Module*) re-merges HOL's $+/−$ concrete syntax alongside the ring locale's, which is harmless in term positions (type inference disambiguates) but ambiguous in a locale-header instantiation argument.

```
no-notation plus (infixl ⟨+⟩ 65)
no-notation minus (infixl ⟨−⟩ 65)
unbundle no uminus-syntax
```

A vector space over the field R (with field operations $+$, $\cdot$, $\mathbf{0}$, $\mathbf{1}$): an abelian group of vectors V under $\oplus$ with zero $\mathbf{0}_V$, together with a scalar multiplication $\odot$ satisfying the module axioms. The purpose of this layer is the structural theorem that a finite field has prime-power order: such a field is a finite-dimensional vector space over its prime subfield, so its cardinality is $|F|^{dim}$.

```
locale Vector-Space = Field +
  fixes vadd :: 'b  $\Rightarrow$  'b  $\Rightarrow$  'b (infixl ⟨ $\oplus$ ⟩ 65)
  and vzero :: 'b (⟨ $\mathbf{0}_V$ ⟩)
  and vcarrier :: 'b set (⟨ $V$ ⟩)
  and scale :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'b (infixr ⟨ $\odot$ ⟩ 75)
  assumes vadd-group: Abelian-Group  $V$  ( $\oplus$ )  $\mathbf{0}_V$ 
  and scale-closed:  $\llbracket a \in R; v \in V \rrbracket \Longrightarrow a \odot v \in V$ 
  and scale-distrib-vadd:  $\llbracket a \in R; u \in V; v \in V \rrbracket \Longrightarrow a \odot (u \oplus v) = (a \odot u) \oplus (a \odot v)$ 
  and scale-distrib-add:  $\llbracket a \in R; b \in R; v \in V \rrbracket \Longrightarrow (a + b) \odot v = (a \odot v) \oplus (b \odot v)$ 
  and scale-scale:  $\llbracket a \in R; b \in R; v \in V \rrbracket \Longrightarrow (a \cdot b) \odot v = a \odot (b \odot v)$ 
  and scale-one:  $v \in V \Longrightarrow \mathbf{1} \odot v = v$ 
begin
```

The vector additive group, available through the *vadd* prefix.

```
sublocale vadd: Abelian-Group  $V$  ( $\oplus$ )  $\mathbf{0}_V$ 
  by (rule vadd-group)
```

Every vector space is a module (its scalar ring being a field is the additional constraint). This makes the module-level constructions — *lincomb*, *span*, and the submodule/quotient machinery — available inside the *Vector-Space* locale under the *vadd*-prefixed operations.

```

sublocale mod: Module R (+) (·) 0 1 (⊕) 0V V (⊙)
  by unfold-locales
    (auto simp: vadd-group scale-closed scale-distrib-vadd scale-distrib-add scale-scale
scale-one)

```

Every submodule of a vector space is itself a vector space under the restricted carrier and the same operations. This bridge lets the module-level kernel, image, and quotient constructions reuse the vector-space basis and dimension API.

```

lemma submodule-vector-space:
  assumes N: mod.submodule N
  shows Vector-Space R (+) (·) 0 1 (⊕) 0V N (⊙)
proof (intro Vector-Space.intro Vector-Space-axioms.intro)
  show Field R (+) (·) 0 1 by unfold-locales
  have N-subgroup: Subgroup N V (⊕) 0V
  proof (intro vadd.subgroupI mod.submodule-subset mod.submodule-zero mod.submodule-add
mod.submodule-neg)
    fix v assume v: v ∈ N
    then show vadd.invertible v
      using N mod.submodule-def by auto
  qed (use N in auto)
  interpret Nadd: Subgroup N V (⊕) 0V by (rule N-subgroup)
  have comm: u ⊕ v = v ⊕ u if u ∈ N v ∈ N for u v
    using N that mod.submodule-subset vadd.commutative by blast
  show Abelian-Group N (⊕) 0V
    by (simp add: Abelian-Group-def Nadd.sub.Group-axioms Nadd.sub.Monoid-axioms
comm
commutative-monoid.intro commutative-monoid-axioms.intro)
  show ∧ a v. [a ∈ R; v ∈ N] ⇒ a ⊙ v ∈ N
    using N by (rule mod.submodule-scale)
  show ∧ a u v. [a ∈ R; u ∈ N; v ∈ N] ⇒
    a ⊙ (u ⊕ v) = (a ⊙ u) ⊕ (a ⊙ v)
    using N mod.submodule-subset by (blast intro: scale-distrib-vadd)
  show ∧ v. v ∈ N ⇒ 1 ⊙ v = v
    using N mod.submodule-subset by (blast intro: scale-one)
  fix a b v
  assume a ∈ R b ∈ R v ∈ N
  then show (a + b) ⊙ v = (a ⊙ v) ⊕ (b ⊙ v) (a · b) ⊙ v = a ⊙ (b ⊙ v)
    using N mod.submodule-subset by (blast intro: scale-distrib-add scale-scale)+
qed

```

Basic closure and unit facts.

```

lemma vzero-closed [simp, intro]: 0V ∈ V by simp

```

```

lemma vadd-closed [simp, intro]: [u ∈ V; v ∈ V] ⇒ u ⊕ v ∈ V
  by simp

```

```

lemma scale-zero-vec:
  assumes a ∈ R
  shows a ⊙ 0V = 0V

```

proof –

```

have  $w: a \odot \mathbf{0}_V \in V$  using assms by (simp add: scale-closed)
then have  $(a \odot \mathbf{0}_V) \oplus \mathbf{0}_V = (a \odot \mathbf{0}_V) \oplus (a \odot \mathbf{0}_V)$ 
  by (metis assms scale-distrib-vadd vadd.right-unit vzero-closed)
then show ?thesis
  using vadd.invertible-left-cancel[OF vadd.invertible[OF w] w vzero-closed w] by
simp
qed

```

Linear combinations and spans come from the module layer. The *mod* sublocale above already supplies *mod.lincomb* and *mod.span* on V , with definitions identical to the ones this theory used to repeat verbatim. We introduce local abbreviations so existing proofs continue to read *lincomb* and *span* unqualified, and re-export the two facts those proofs cite by name.

abbreviation *lincomb* :: $('b \Rightarrow 'a) \Rightarrow 'b \text{ set} \Rightarrow 'b$
where *lincomb* $\equiv \text{mod.lincomb}$

abbreviation *span* :: $'b \text{ set} \Rightarrow 'b \text{ set}$
where *span* $\equiv \text{mod.span}$

lemmas *lincomb-def* = *mod.lincomb-def*
lemmas *span-def* = *mod.span-def*

lemma *lincomb-closed*:
assumes $B: B \subseteq V$ **and** $c: \bigwedge v. v \in B \implies c \ v \in R$
shows *lincomb* $c \ B \in V$
using $B \ c$ **by** (*rule mod.lincomb-closed*)

Spanning and linear independence: the finite-dimensional forms. The module layer defines *mod.spanning* and *mod.lin-indep* for arbitrary subsets (see *Free-Module*). This theory is about the finite-dimensional case — its basis and dimension results count coordinate vectors — so we keep *finite* variants phrased with a total coefficient function $c \in B \rightarrow_E R$, which is what the coordinate map needs, and relate them to the general notions by the bridging lemmas below.

definition *spanning* :: $'b \text{ set} \Rightarrow \text{bool}$
where *spanning* $B \equiv \text{finite } B \wedge B \subseteq V \wedge (\forall v \in V. \exists c \in B \rightarrow_E R. v = \text{lincomb } c \ B)$

definition *lin-indep* :: $'b \text{ set} \Rightarrow \text{bool}$
where *lin-indep* $B \equiv \text{finite } B \wedge B \subseteq V \wedge (\forall c \in B \rightarrow_E R. \text{lincomb } c \ B = \mathbf{0}_V \implies (\forall v \in B. c \ v = \mathbf{0}))$

A set B is a *basis* of V when it is a finite subset whose coordinate map — sending each coefficient vector $c \in B \rightarrow_E R$ to the linear combination *lincomb* $c \ B$ — is a bijection onto V . This packages spanning (surjectivity) and linear independence (injectivity) in exactly the form the cardinality count needs.

definition *basis* :: $'b \text{ set} \Rightarrow \text{bool}$
where *basis* $B \equiv \text{finite } B \wedge B \subseteq V \wedge \text{bij-betw } (\lambda c. \text{lincomb } c \ B) (B \rightarrow_E R) \ V$

Bridge: the finite forms agree with the module-level notions. On a finite set the two spanning conditions coincide: *mod.spanning* says every vector lies in the span, and on a finite B a span membership is realised by a coefficient function on B itself (extended by $\mathbf{0}$ where the witnessing subset omits an element).

lemma *spanning-iff-mod-spanning:*
assumes *finB*: finite B **and** BV : $B \subseteq V$
shows *spanning* $B \longleftrightarrow$ *mod.spanning* B
proof
assume *spanning* B
show *mod.spanning* B
proof (rule *mod.spanningI*[OF BV])
fix v **assume** $v \in V$
with $\langle$ *spanning* $B\rangle$ *mod.Module-axioms* **show** $v \in$ *mod.span* B
using *mod.spanI* *Module.spanI* *finB* *mod.coeffs-on-def*
by (*fastforce simp: PiE-iff spanning-def*)
qed
next
assume *ms*: *mod.spanning* B
have $\forall v \in V. \exists c \in B \rightarrow_E R. v = \text{lincomb } c \ B$
proof (rule *ballI*)
fix v **assume** $v \in V$
then obtain $c \ A$ **where** A : finite $A \ A \subseteq B$ *mod.coeffs-on* $c \ A \ v = \text{lincomb } c \ A$
using *mod.spanE* *mod.spanning-def ms* **by** *metis*
— Extend c by $\mathbf{0}$ off A and make it extensional on B .
define c' **where** $c' \equiv (\lambda u. \text{if } u \in A \text{ then } c \ u \text{ else } \mathbf{0})$
have $c'R$: *mod.coeffs-on* $c' \ B$
using $A(3)$ c' -def *mod.coeffs-on-def* **by** *force*
have *lincomb* $c' \ A = \text{lincomb } c' \ B$
using $\langle A \subseteq B \rangle$ BV $c'R$ c' -def *finB* *mod.lincomb-mono-zero* **by** *force*
moreover have *lincomb* $c' \ A = \text{lincomb } c \ A$
using $\langle A \subseteq B \rangle$ BV *mod.coeffs-onD*[OF $A(3)$]
by (*intro mod.lincomb-cong*) (*auto simp: c'-def intro!: mod.coeffs-onI*)
ultimately have vc' : $v = \text{lincomb } c' \ B$ **using** $A(4)$ **by** *simp*
— Make it extensional on B ; that changes no value on B .
have *lincomb* (*restrict* $c' \ B$) $B = \text{lincomb } c' \ B$
by (rule *mod.lincomb-cong*[OF BV]) (*use c'R in simp-all*)
with vc' **have** $v = \text{lincomb } (\text{restrict } c' \ B) \ B$ **by** *simp*
moreover have *restrict* $c' \ B \in B \rightarrow_E R$
using *mod.coeffs-onD*[OF $c'R$] **by** *auto*
ultimately show $\exists c \in B \rightarrow_E R. v = \text{lincomb } c \ B$ **by** *blast*
qed
with *finB* BV **show** *spanning* B **unfolding** *spanning-def* **by** *blast*
qed

Likewise for independence. The finite form quantifies over total coefficient functions on B ; the module form over coefficient functions on finite subsets. On a finite B these agree: restrict to the subset in one direction, extend by $\mathbf{0}$ in the other.

lemma *lin-indep-iff-mod-lin-indep:*

```

assumes finB: finite B and BV:  $B \subseteq V$ 
shows lin-indep B  $\longleftrightarrow$  mod.lin-indep B
proof
  assume li: lin-indep B
  show mod.lin-indep B
  proof (rule mod.lin-indepI[OF BV])
    fix c A v
    assume finite A  $A \subseteq B$  and cA: mod.coeffs-on c A
    and zero: lincomb c A =  $\mathbf{0}_V$  and vA:  $v \in A$ 
    define c' where c' = restrict ( $\lambda u.$  if  $u \in A$  then  $c\ u$  else  $\mathbf{0}$ ) B
    have c'R: mod.coeffs-on c' B
    by (rule mod.coeffs-onI) (use mod.coeffs-onD[OF cA] in  $\langle \text{simp add: } c'\text{-def} \rangle$ )
    have c'PiE:  $c' \in B \rightarrow_E R$  using mod.coeffs-onD[OF c'R] by (auto simp: c'-def)
    have lincomb c' A = lincomb c A
    using  $\langle A \subseteq B \rangle$  BV mod.coeffs-onD[OF cA]
    by (intro mod.lincomb-cong) (auto simp: c'-def intro!: mod.coeffs-onI)
    moreover have lincomb c' A = lincomb c' B
    using  $\langle A \subseteq B \rangle$  BV c'R c'-def finB mod.lincomb-mono-zero by force
    ultimately have lincomb c' B =  $\mathbf{0}_V$  using zero by simp
    then have  $c' v = \mathbf{0}$  using c'PiE li using  $\langle A \subseteq B \rangle$  vA by (auto simp: lin-indep-def)
    then show  $c v = \mathbf{0}$  using  $\langle A \subseteq B \rangle$  c'-def vA by auto
  qed
next
  assume ms: mod.lin-indep B
  — rule twice, not intro: intro is greedy and would also strip the inner bounded
  quantifier, leaving a goal the show below cannot match.
  have  $\forall c \in B \rightarrow_E R. \text{lincomb } c\ B = \mathbf{0}_V \longrightarrow (\forall v \in B. c\ v = \mathbf{0})$ 
    using finB mod.coeffs-on-def mod.lin-indepD ms by (meson PiE-mem subset-refl)
  with finB BV show lin-indep B unfolding lin-indep-def by blast
qed

```

A basis spans: surjectivity of the coordinate map.

lemma *basis-spanning*:

```

assumes basis B shows spanning B
unfolding spanning-def
using assms basis-def by (metis (no-types, lifting) bij-betw-def imageE)

```

The dimension counting theorem. Over a finite field, a space with a basis of size n has exactly $(\text{card } R)^n$ elements: its coordinate map is a bijection onto the $(\text{card } R)^n$ coefficient vectors $B \rightarrow_E R$.

theorem *card-eq-card-base-pow-dim*:

```

assumes basis B
shows  $\text{card } V = \text{card } R \wedge \text{card } B$ 
by (metis assms basis-def bij-betw-same-card card-funcsetE)

```

Over a finite field, any two bases have the same size — so the dimension is well defined. (Both give $\text{card } V = (\text{card } R)^{\text{card } B_i}$, and $2 \leq \text{card } R$ is injective as a base of exponentiation.)

theorem *basis-card-unique-finite*:

```

assumes finR: finite R and B1: basis B1 and B2: basis B2
shows card B1 = card B2
proof –
  have card R ^ card B1 = card V using card-eq-card-base-pow-dim[OF B1] by simp
  also have ... = card R ^ card B2 using card-eq-card-base-pow-dim[OF B2] by simp
  finally have eq: card R ^ card B1 = card R ^ card B2 .
  — The field has at least two elements ( $1 \neq 0$ ), so exponentiation is injective.
  have card R ≥ 2 using nontrivial card-mono[OF finR, of {1,0}] by auto
  then show ?thesis using eq by simp
qed

```

The dimension of the space: the size of a basis (well defined over a finite field).

```

definition dimension :: nat
  where dimension = card (SOME B. basis B)

```

```

lemma dimension-eq:
  assumes finR: finite R and B: basis B shows dimension = card B
  by (metis B basis-card-unique-finite dimension-def finR someI)

```

end

A vector subspace packages a module-level submodule of a vector space and exposes the induced vector-space structure under the prefix *sub*.

```

locale vector-subspace = Vector-Space +
  fixes W :: 'b set
  assumes W-submodule: mod.submodule W
begin

  sublocale sub: Vector-Space R (+) (·) 0 1 (⊕) 0V W (⊙)
    by (rule submodule-vector-space[OF W-submodule])

```

end

Following the *field-TC* pattern of *Ring-Theory*, we bridge the locale-based *Vector-Space* to Isabelle’s type-class arithmetic. A type-class field '*a*' plays the scalar role, and a type-class abelian additive group '*b*' plays the vector role; a scale operation *scale* satisfying the four vector-space axioms then witnesses *Vector-Space* on the whole types as carriers.

```

lemma vector-space-TC:
  fixes scale :: 'a::field ⇒ 'b::ab-group-add ⇒ 'b
  assumes scale-add-right:  $\bigwedge a\ x\ y. \text{scale } a\ (\text{plus } x\ y) = \text{plus } (\text{scale } a\ x)\ (\text{scale } a\ y)$ 
    and scale-add-left:  $\bigwedge a\ b\ x. \text{scale } (\text{plus } a\ b)\ x = \text{plus } (\text{scale } a\ x)\ (\text{scale } b\ x)$ 
    and scale-scale:  $\bigwedge a\ b\ x. \text{scale } a\ (\text{scale } b\ x) = \text{scale } (a * b)\ x$ 
    and scale-one:  $\bigwedge x. \text{scale } 1\ x = x$ 
  shows Vector-Space (UNIV :: 'a set) plus (*) 0 1 plus 0 (UNIV :: 'b set) scale
proof –
  interpret F: Field UNIV :: 'a set plus (*) 0 1 by (rule field-TC.Field-axioms)
  show ?thesis
  proof (intro Vector-Space.intro Vector-Space-axioms.intro)

```



```

    show Field (UNIV :: 'a set) plus (*) 0 1 by (rule field-TC.Field-axioms)
    show Abelian-Group (UNIV :: 'b set) plus 0 by (rule add-abelian-TC)
  qed (simp-all add: scale-add-right scale-add-left scale-scale scale-one)
end

```

Concrete specialisations to Isabelle's built-in *real-vector* and *complex-vector* classes can be added by any consumer.

end

57 The Steinitz exchange lemma and dimension over an arbitrary field

```

theory Steinitz
  imports Vector-Space
begin

```

Restore HOL's arithmetic notation, which *Vector-Space* (via its ancestor *Ring-Theory*) suppresses to disambiguate locale-parameter binding. Steinitz does not bind $+$ or $-$ as locale parameters, so it is safe (and needed for e.g. nat subtraction) to restore them.

```

notation plus (infixl <+> 65)
notation minus (infixl <-> 65)
unbundle uminus-syntax

```

Basis-cardinality uniqueness over a *finite* field was obtained in *Vector-Space* by a counting argument. Here we develop the general (any field, in particular an infinite base such as $\mathbb{Q}$) theory via the *Steinitz exchange lemma*: from a spanning set and a linearly independent set, independent vectors can be exchanged into the spanning set one at a time, whence $|independent| \leq |spanning|$ and any two bases have equal size. It is the linchpin for a general field-extension degree $[K:F]$ and for transitivity of algebraicity.

```

context Vector-Space
begin

```

57.1 Span basics

```

lemma span-closed:
  assumes  $S \subseteq V$  and  $x \in \text{span } S$ 
  shows  $x \in V$ 
  using assms lincomb-closed span-def by auto

```

```

lemma span-mono:
  assumes  $S \subseteq T$  shows  $\text{span } S \subseteq \text{span } T$ 
  using assms unfolding span-def by blast

```

Every vector of S lies in its span.

```

lemma span-incl:

```

assumes $S \subseteq V$ $v \in S$
shows $v \in \text{span } S$
using *assms mod.span-incl* **by** (*meson in-mono*)

lemma *span-subset-V*:
assumes $S \subseteq V$ **shows** $\text{span } S \subseteq V$
using *assms span-closed* **by** *blast*

57.2 Extension of a linear combination by zero coefficients

Scaling by the field zero gives the zero vector: from $(\mathbf{0} + \mathbf{0}) \odot v = \mathbf{0} \odot v \oplus \mathbf{0} \odot v$ and cancellation.

lemmas *scale-zero-scalar* = *mod.scale-zero-scalar*

Extending a linear combination's support by vectors with zero coefficient leaves it unchanged: the extra terms $\mathbf{0} \odot v = \mathbf{0}_V$ vanish in the vector sum.

lemma *lincomb-extend-zero*:
assumes $B: B \subseteq V$ **and** $C: \text{finite } C$ $C \subseteq V$ **and** $c: \bigwedge v. v \in B \implies c \ v \in R$
and *disj*: $B \cap C = \{\}$ **and** *czero*: $\bigwedge v. v \in C \implies c \ v = \mathbf{0}$
and *finB*: *finite B*
shows $\text{lincomb } c \ (B \cup C) = \text{lincomb } c \ B$
proof –
have *Rc*: $(\lambda v. c \ v \odot v) \in B \rightarrow V$ **using** *B c* **by** (*auto intro: scale-closed*)
have *RcC*: $(\lambda v. c \ v \odot v) \in C \rightarrow V$ **using** *C czero* **by** (*auto simp: scale-zero-scalar*)
have $\text{lincomb } c \ (B \cup C) = \text{vadd.fincomp } (\lambda v. c \ v \odot v) \ (B \cup C)$ **by** (*simp add: lincomb-def*)
also have $\dots = \text{vadd.fincomp } (\lambda v. c \ v \odot v) \ B \oplus \text{vadd.fincomp } (\lambda v. c \ v \odot v) \ C$
using *finB C(1) disj Rc RcC* **by** (*simp add: vadd.fincomp-Un-disjoint*)
also have $\text{vadd.fincomp } (\lambda v. c \ v \odot v) \ C = \mathbf{0}_V$
using *C czero* **by** (*auto intro!: vadd.fincomp-unit-eqI simp: scale-zero-scalar*)
also have $\text{vadd.fincomp } (\lambda v. c \ v \odot v) \ B \oplus \mathbf{0}_V = \text{vadd.fincomp } (\lambda v. c \ v \odot v) \ B$
using *Rc finB* **by** (*simp add: vadd.right-unit vadd.fincomp-closed*)
finally show *?thesis* **by** (*simp add: lincomb-def*)
qed

57.3 Span is a subspace

Over a common finite support B , linear combinations add coefficientwise: $\text{lincomb } c \ B \oplus \text{lincomb } d \ B = \text{lincomb } (\lambda v. c \ v + d \ v) \ B$ (by *vadd.fincomp-comp* and *scale-distrib-add*).

lemma *lincomb-add*:
assumes $B: B \subseteq V$ **and** *finB*: *finite B*
and $c: \bigwedge v. v \in B \implies c \ v \in R$ **and** $d: \bigwedge v. v \in B \implies d \ v \in R$
shows $\text{lincomb } c \ B \oplus \text{lincomb } d \ B = \text{lincomb } (\lambda v. c \ v + d \ v) \ B$
proof –
have *Rc*: $(\lambda v. c \ v \odot v) \in B \rightarrow V$ **using** *B c* **by** (*auto intro: scale-closed*)
have *Rd*: $(\lambda v. d \ v \odot v) \in B \rightarrow V$ **using** *B d* **by** (*auto intro: scale-closed*)
have $\text{lincomb } c \ B \oplus \text{lincomb } d \ B$

```

      = vadd.fincomp (λv. (c v ⊙ v) ⊕ (d v ⊙ v)) B
    using Rc Rd by (simp add: lincomb-def vadd.fincomp-comp)
  also have ... = vadd.fincomp (λv. (c v + d v) ⊙ v) B
    by (rule vadd.fincomp-cong') (use assms in (auto simp: scale-distrib-add scale-closed))
  finally show ?thesis by (simp add: lincomb-def)
qed

```

A scalar pulls through a vector finite sum: $a \odot \text{vadd.fincomp } g \ A = \text{vadd.fincomp } (\lambda v. a \odot g \ v) \ A$ (by *scale-distrib-vadd*, induction on A).

lemmas *scale-fincomp* = *mod.scale-fincomp*

Scaling a linear combination scales its coefficients: $a \odot \text{lincomb } c \ B = \text{lincomb } (\lambda v. a \cdot c \ v) \ B$.

lemma *lincomb-scale*:

assumes $B: B \subseteq V$ **and** *finB*: *finite B* **and** $a: a \in R$

and $c: \bigwedge v. v \in B \implies c \ v \in R$

shows $a \odot \text{lincomb } c \ B = \text{lincomb } (\lambda v. a \cdot c \ v) \ B$

proof –

have $Rc: (\lambda v. c \ v \odot v) \in B \rightarrow V$ **using** $B \ c$ **by** (*auto intro: scale-closed*)

have $a \odot \text{lincomb } c \ B = \text{vadd.fincomp } (\lambda v. a \odot (c \ v \odot v)) \ B$

by (*simp add: Rc a finB lincomb-def scale-fincomp*)

also have ... = $\text{vadd.fincomp } (\lambda v. (a \cdot c \ v) \odot v) \ B$

using *assms* **by** (*auto simp: scale-scale intro!: vadd.fincomp-cong' scale-closed*)

finally show ?thesis **by** (*simp add: lincomb-def*)

qed

57.4 Span closure, idempotence and transitivity

Two coefficient functions agreeing on the support give the same linear combination.

lemma *lincomb-cong*:

assumes $B \subseteq V$ **and** $\bigwedge v. v \in B \implies d \ v \in R$ **and** $\bigwedge v. v \in B \implies c \ v = d \ v$

shows $\text{lincomb } c \ B = \text{lincomb } d \ B$

unfolding *lincomb-def*

using *assms*

by (*force intro!: vadd.fincomp-cong'[OF refl] scale-closed*)

Enlarging the support of a linear combination by vectors whose coefficient is **0** leaves its value unchanged — the general (non-disjoint) form of *lincomb-extend-zero*, obtained by taking the extra part to be the vectors of D outside B .

lemma *lincomb-zero-extend-eq*:

assumes $B: B \subseteq V$ **and** $D: \text{finite } D \ D \subseteq V$ **and** $BD: B \subseteq D$

and $c: \bigwedge v. v \in B \implies c \ v \in R$ **and** $czero: \bigwedge v. [v \in D; v \notin B] \implies c \ v = \mathbf{0}$

shows $\text{lincomb } c \ D = \text{lincomb } c \ B$

proof –

define E **where** $E \equiv \{v \in D. v \notin B\}$

obtain *finB*: *finite B* **and** *finE*: *finite E* **and** $E \subseteq V$

using *finite-subset D BD* **unfolding** *E-def* **by** *auto*

```

have disj:  $B \cap E = \{\}$  unfolding  $E\text{-def}$  by  $\text{blast}$ 
have Ezero:  $\bigwedge v. v \in E \implies c\ v = \mathbf{0}$  using  $\text{czero}$  unfolding  $E\text{-def}$  by  $\text{blast}$ 
have lincomb  $c\ (B \cup E) = \text{lincomb}\ c\ B$ 
  by  $(\text{rule}\ \text{lincomb-extend-zero}[OF\ B\ \text{finE}\ \langle E \subseteq V \rangle\ c\ \text{disj}\ Ezero\ \text{finB}])$ 
moreover have  $B \cup E = D$  using  $BD$  unfolding  $E\text{-def}$  by  $\text{blast}$ 
ultimately show  $?thesis$  by  $\text{simp}$ 
qed

```

The zero vector is the empty linear combination, hence lies in every span.

```

lemma  $\text{span-zero}$ :  $\mathbf{0}_V \in \text{span}\ S$ 
by  $\text{simp}$ 

```

A span is closed under scalar multiplication.

```

lemmas  $\text{span-scale} = \text{mod.span-scale-closed}$ 

```

A span is closed under vector addition.

```

lemmas  $\text{span-vadd} = \text{mod.span-madd-closed}$ 

```

Consequently a linear combination of vectors *drawn from a span* stays in the span: induction on the support, using span-zero , span-scale , span-vadd .

```

lemma  $\text{span-lincomb-closed}$ :
  assumes  $S$ :  $S \subseteq V$  and  $\text{finB}$ : finite  $B$  and  $BS$ :  $B \subseteq \text{span}\ S$ 
  and  $cR$ :  $\bigwedge v. v \in B \implies c\ v \in R$ 
  shows  $\text{lincomb}\ c\ B \in \text{span}\ S$ 
  using  $\text{finB}\ BS\ cR$ 
proof  $(\text{induct}\ B\ \text{rule:}\ \text{finite-induct})$ 
  case  $\text{empty}$ 
  have  $\text{lincomb}\ c\ \{\} = \mathbf{0}_V$  by  $(\text{simp}\ \text{add:}\ \text{lincomb-def})$ 
  then show  $?case$  by  $(\text{simp}\ \text{add:}\ \text{span-zero})$ 
next
  case  $(\text{insert}\ x\ F)$ 
  then have  $sF$ :  $\text{lincomb}\ c\ F \in \text{span}\ S$ 
  by  $\text{blast}$ 
  have  $\text{split}$ :  $\text{lincomb}\ c\ (\text{insert}\ x\ F) = (c\ x \odot x) \oplus \text{lincomb}\ c\ F$ 
  using  $\text{insert}\ S$  by  $(\text{simp}\ \text{add:}\ \text{lincomb-def}\ \text{scale-closed}\ \text{span-closed}\ \text{subset-eq})$ 
  have  $sx$ :  $c\ x \odot x \in \text{span}\ S$ 
  using  $S\ \text{insert.prem1}\ 2\ \text{span-scale}$  by  $\text{auto}$ 
  show  $?case$  using  $\text{split}\ sx\ sF\ S$  by  $(\text{auto}\ \text{intro:}\ \text{span-vadd})$ 
qed

```

Span is idempotent, so it is a genuine closure operator.

```

lemma  $\text{span-span}$ :
  assumes  $S$ :  $S \subseteq V$  shows  $\text{span}\ (\text{span}\ S) = \text{span}\ S$ 
  by  $(\text{simp}\ \text{add:}\ \text{assms}\ \text{mod.span-incl}\ \text{mod.span-minimal}\ \text{mod.span-submodule}\ \text{span-subset-}V\ \text{subset-antisym})$ 

```

Transitivity: if S lies inside the span of T , its span does too.

```

lemma  $\text{span-trans}$ :

```

```

assumes  $ST: S \subseteq \text{span } T$  and  $T: T \subseteq V$ 
shows  $\text{span } S \subseteq \text{span } T$ 
by (metis  $ST$   $T$  span-mono span-span)

```

The membership form of transitivity, matching the shape used in the exchange lemma: absorbing a spanned vector into the generating set does not enlarge the span.

```

lemma span-trans-mem:
  assumes  $S \subseteq V$  and  $x \in \text{span } S$  and  $y \in \text{span } (\text{insert } x \ S)$ 
  shows  $y \in \text{span } S$ 
  by (metis (no-types, opaque-lifting) assms insert-subset span-incl span-trans subset-eq)

```

57.5 Vector subtraction inside a span

Negating a scalar coefficient negates the scaled vector: $(- \ a) \odot v$ is the vector-group inverse of $a \odot v$. From $(a - a) \odot v = \mathbf{0}_V$.

```

lemmas scale-neg-scalar = mod.scale-neg-scalar

```

A span is closed under vector negation.

```

lemma span-neg:
  assumes  $S \subseteq V$  and  $x \in \text{span } S$ 
  shows vadd.inverse  $x \in \text{span } S$ 
  using mod.span-submodule mod.submodule-neg assms by presburger

```

A span is closed under vector subtraction.

```

lemma span-diff:
  assumes  $S: S \subseteq V$  and  $x: x \in \text{span } S$  and  $y: y \in \text{span } S$ 
  shows  $x \oplus \text{vadd.inverse } y \in \text{span } S$ 
  by (simp add:  $S$  span-neg span-vadd  $x$   $y$ )

```

57.6 The exchange step

Exchange step. If a lies in the span of $\text{insert } b \ S$ but not in the span of S , then b lies in the span of $\text{insert } a \ S$ — so b may be swapped out for a .

```

lemma in-span-insert:
  assumes  $S: S \subseteq V$  and  $b: b \in V$ 
  and  $a: a \in \text{span } (\text{insert } b \ S)$  and  $na: a \notin \text{span } S$ 
  shows  $b \in \text{span } (\text{insert } a \ S)$ 
proof —
  have  $aV: a \in V$  using  $a \ S \ b$  span-closed by (metis insert-subset span-subset-V subsetD)
  — Split a linear combination over  $\text{insert } b \ S$  into its  $b$ -part and the rest.
  from  $a$  obtain  $c \ B$  where a-eq:  $a = \text{lincomb } c \ B$  and finB: finite  $B$ 
  and  $BbS: B \subseteq \text{insert } b \ S$  and cR:  $\bigwedge v. v \in B \implies c \ v \in R$  unfolding span-def
by blast
  define  $B0$  where  $B0 = B \setminus \{b\}$ 

```

```

define k where k = (if b ∈ B then c b else 0)
have kR: k ∈ R using cR by (auto simp: k-def additive.unit-closed)
have finB0: finite B0 using finB by (simp add: B0-def)
have B0 ⊆ S B0 ⊆ V using BbS S by (auto simp: B0-def)
have bnB0: b ∉ B0 by (simp add: B0-def)
have cR0:  $\bigwedge v. v \in B0 \implies c v \in R$  using cR by (auto simp: B0-def)
define r where r ≡ lincomb c B0 — @the part of a inside span S
have rspan: r ∈ span S using finB0 ⟨B0 ⊆ S⟩ cR0 unfolding r-def span-def by
blast
have rV: r ∈ V using rspan S span-closed by blast
have a-split: a = (k ⊙ b) ⊕ r
proof (cases b ∈ B)
case True
then have lincomb c B = lincomb c (insert b B0)
by (simp add: B0-def insert-absorb)
also have ... = (c b ⊙ b) ⊕ lincomb c B0
unfolding lincomb-def using finB0 bnB0 cR0 ⟨B0 ⊆ V⟩ b cR True
by (subst vadd.fincomp-insert) (auto intro: scale-closed)
finally show ?thesis using True by (simp add: a-eq k-def r-def)
next
case False
then show ?thesis using rV na rspan by (force simp: B0-def r-def a-eq)
qed
— k is nonzero, otherwise a = r would lie in span S.
have kb: k ⊙ b ∈ V using kR b by (rule scale-closed)
have knz: k ≠ 0
using a-split b na rV rspan scale-zero-scalar by force
— Recover b: from a = k ⊙ b ⊕ r we get k ⊙ b = a ⊕ (− r).
have kinv: multiplicative.inverse k ∈ R using kR knz field-inverse by simp
have kk: multiplicative.inverse k · k = 1
using kR knz field-inverse by (simp add: multiplicative.invertible-left-inverse)
have (k ⊙ b) = a ⊕ vadd.inverse r
by (simp add: a-split kb rV vadd.associative)
then have b-eq: b = multiplicative.inverse k ⊙ (a ⊕ vadd.inverse r)
by (metis b kR kinv kk scale-one scale-scale)
— Both a and − r lie in span (insert a S), so does their sum, then its scaling.
have aS: a ∈ span (insert a S) insert a S ⊆ V
using aV S by (auto intro: span-incl)
have rS: r ∈ span (insert a S) using rspan span-mono[of S insert a S] by blast
have a ⊕ vadd.inverse r ∈ span (insert a S) using aS rS by (auto intro: span-diff)
then show ?thesis using aS kinv b-eq by (auto intro: span-scale)
qed

```

Delete form. If a is spanned by S but not by $S - \{b\}$, then b can be exchanged into the generating set in place of a .

lemma *in-span-delete*:

```

assumes S: S ⊆ V and b: b ∈ V
and a: a ∈ span S and na: a ∉ span (S \ {b})
shows b ∈ span (insert a (S \ {b}))

```

proof –
have $a \in \text{span } (\text{insert } b \ (S \setminus \{b\}))$ **using** $a \text{ span-mono}[of \ S \ \text{insert } b \ (S \setminus \{b\})]$ **by**
 blast
with assms **show** $?thesis$
by $(\text{meson } \text{Diff-subset in-span-insert subset-trans})$
qed

57.7 Independence as irredundancy

The geometric reading of linear independence: no member of an independent set lies in the span of the others. This is the bridge from the coordinate definition *lin-indep* to the span-theoretic exchange arguments. If some $w \in B$ were in $\text{span } (B - \{w\})$, its witnessing combination, moved to one side with coefficient -1 on w , would be a nontrivial dependence.

lemma *lin-indep-not-in-span*:

assumes $\text{ind}: \text{lin-indep } B$ **and** $w: w \in B$
shows $w \notin \text{span } (B \setminus \{w\})$

proof

assume $w \in \text{span } (B \setminus \{w\})$

then obtain $d \ C$ **where** $w\text{-eq}: w = \text{lincomb } d \ C$ **and** $\text{finC}: \text{finite } C$

and $\text{CB}: C \subseteq B \setminus \{w\}$ **and** $\text{dR}: \bigwedge v. v \in C \implies d \ v \in R$ **unfolding** *span-def* **by**

blast

have $\text{finB}: \text{finite } B$ **and** $B \subseteq V$ **using** ind **by** $(\text{auto simp: lin-indep-def})$

have $wV: w \in V$ **using** $w \in B \subseteq V$ **by** blast

have $\text{wnC}: w \notin C$ **using** CB **by** blast

define c **where** $c \equiv (\lambda v. \text{if } v = w \text{ then } -1 \text{ else if } v \in C \text{ then } d \ v \text{ else } 0)$

have $\text{cR}: \bigwedge v. v \in B \implies c \ v \in R$ **using** dR **by** $(\text{auto simp: c-def additive.unit-closed})$

have $\text{cPiE}: \text{restrict } c \ B \in B \rightarrow_E R$ **using** cR **by** $(\text{auto simp: c-def})$

— The combination over B splits into the w -term and the C -term (all other coefficients are zero), and equals 0_V .

have $\text{CsubB}: C \subseteq B$ **using** CB **by** blast

have $\text{insCB}: \text{insert } w \ C \subseteq B$ **using** $w \ CsubB$ **by** blast

have $\text{lincomb } (\text{restrict } c \ B) \ B = \text{lincomb } (\text{restrict } c \ B) \ (\text{insert } w \ C)$

proof $(\text{intro lincomb-zero-extend-eq finB } \langle B \subseteq V \rangle \text{ insCB})$

show $\text{insert } w \ C \subseteq V$ **using** $\text{insCB } \langle B \subseteq V \rangle$ **by** blast

show $\bigwedge v. v \in \text{insert } w \ C \implies \text{restrict } c \ B \ v \in R$ **using** $\text{insCB } \text{cR}$ **by** auto

qed $(\text{auto simp add: c-def})$

also have $\dots = \text{lincomb } c \ (\text{insert } w \ C)$

by $(\text{rule lincomb-cong}) \ (\text{use insCB } \langle B \subseteq V \rangle \text{ cR in auto})$

also have $\dots = (c \ w \odot w) \oplus \text{lincomb } c \ C$

unfolding *lincomb-def* **using** $\text{finC } \text{wnC } wV \text{ cR}[OF \ w] \text{ cR } \text{CsubB } \langle B \subseteq V \rangle$

by $(\text{subst vadd.fincomp-insert}) \ (\text{auto intro!: scale-closed})$

also have $\text{lincomb } c \ C = \text{lincomb } d \ C$

by $(\text{rule lincomb-cong}) \ (\text{use CsubB } \langle B \subseteq V \rangle \text{ dR } \text{wnC in } \langle \text{auto simp: c-def} \rangle)$

also have $c \ w \odot w = \text{vadd.inverse } w$ **using** wV **by** $(\text{simp add: c-def scale-neg-scalar scale-one})$

also have $\text{vadd.inverse } w \oplus \text{lincomb } d \ C = \text{vadd.inverse } w \oplus w$ **using** $w\text{-eq}$ **by**
 simp

also have $\dots = 0_V$ **using** wV **by** simp

finally have $\text{zero: } \text{lincomb } (\text{restrict } c \ B) \ B = \mathbf{0}_V$.
 — Independence forces every coefficient to vanish — but the w -coefficient is $-1 \neq 0$.
from $\text{ind } c\text{PiE } \text{zero}$ **have** $\text{restrict } c \ B \ w = \mathbf{0}$ **using** w
 using lin-indep-def **by** blast
then have $(-1) = \mathbf{0}$ **using** w **by** $(\text{simp add: } c\text{-def})$
then show False
 using $\text{additive.commute-iff-inverse nontrivial}$ **by** fastforce
qed

Conversely, over a field, a finite set none of whose members lies in the span of the others is linearly independent. Given a dependence $\text{lincomb } c \ B = \mathbf{0}_V$ with some $c \ w \neq \mathbf{0}$, solving for w exhibits w in the span of the rest.

lemma $\text{not-in-span-lin-indep}$:
 assumes $\text{finB: finite } B$ **and** $B \subseteq V$
 and $\text{irred: } \bigwedge w. w \in B \implies w \notin \text{span } (B \setminus \{w\})$
 shows $\text{lin-indep } B$
 unfolding lin-indep-def
proof $(\text{intro strip conjI assms})$
 show $c \ v = \mathbf{0}$ **if** $c\text{PiE: } c \in B \rightarrow_E R$ **and** $\text{zero: } \text{lincomb } c \ B = \mathbf{0}_V$ **and** $v: v \in B$ **for**
 $c \ v$
 proof (rule ccontr)
 assume $\text{cwnz: } c \ v \neq \mathbf{0}$
 define r **where** $r \equiv \text{lincomb } c \ (B \setminus \{v\})$
 have $\text{rspan: } r \in \text{span } (B \setminus \{v\})$ **using** $\text{finB } c\text{PiE}$ **unfolding** $r\text{-def span-def}$ **by**
 blast
 have $rV: r \in V$ **using** $\text{rspan } \langle B \subseteq V \rangle \text{ span-closed}$ **by** blast
 have $wV: v \in V$ **using** $v \ \langle B \subseteq V \rangle$ **by** blast
 have $\text{cwR: } c \ v \in R$ **using** $c\text{PiE } v$ **by** blast
 — Split off the v -term: $\mathbf{0}_V = c \ v \odot v \oplus \text{lincomb } c \ (B - \{v\})$.
 have $\text{cww: } c \ v \odot v \in V$ **using** $\text{cwR } wV$ **by** $(\text{rule scale-closed})$
 have $B = \text{insert } v \ (B \setminus \{v\})$ **using** v **by** blast
 then have $\text{lincomb } c \ B = \text{lincomb } c \ (\text{insert } v \ (B \setminus \{v\}))$ **by** simp
 also have $\dots = (c \ v \odot v) \oplus \text{lincomb } c \ (B \setminus \{v\})$
 unfolding lincomb-def **using** $\text{finB } wV \text{ cwR } c\text{PiE } \langle B \subseteq V \rangle$
 by $(\text{subst vadd.fincomp-insert})$ $(\text{auto intro!: scale-closed})$
 finally have $c \ v \odot v = \text{vadd.inverse } r$
 using $\text{cww } rV \text{ r-def vadd.commutative vadd.inverse-equality zero}$ **by** metis
 have $\text{cwinv: multiplicative.inverse } (c \ v) \in R$ **using** $\text{cwR } \text{cwnz field-inverse}$ **by** simp
 have $\text{cwcw: multiplicative.inverse } (c \ v) \cdot c \ v = \mathbf{1}$
 using $\text{cwR } \text{cwnz field-inverse}$ **by** $(\text{simp add: multiplicative.invertible-left-inverse})$
 have $v = \text{multiplicative.inverse } (c \ v) \odot (c \ v \odot v)$
 using $\text{cwinv } \text{cwR } wV$ **by** $(\text{simp add: scale-scale[symmetric] cwcw scale-one})$
 also have $\dots = \text{multiplicative.inverse } (c \ v) \odot \text{vadd.inverse } r$
 using $\langle c \ v \odot v = \text{vadd.inverse } r \rangle$ **by** simp
 finally have $v = \text{multiplicative.inverse } (c \ v) \odot \text{vadd.inverse } r$.
 moreover have $\text{multiplicative.inverse } (c \ v) \odot \text{vadd.inverse } r \in \text{span } (B \setminus \{v\})$
 using $\langle B \subseteq V \rangle \text{ cwinv span-neg[OF - rspan]}$ **by** $(\text{auto intro: span-scale})$
 ultimately have $v \in \text{span } (B \setminus \{v\})$ **by** simp


```

    with irred[OF v] show False by simp
  qed
qed

```

Adjoining a vector outside the span preserves linear independence.

```

lemma lin-indep-insert:
  assumes ind: lin-indep A and x:  $x \in V$  and outside:  $x \notin \text{span } A$ 
  shows lin-indep (insert x A)
proof (rule not-in-span-lin-indep)
  show finite (insert x A) using ind by (simp add: lin-indep-def)
  show insert x A  $\subseteq V$  using ind x by (auto simp: lin-indep-def)
  fix w assume w:  $w \in \text{insert } x A$ 
  have AV:  $A \subseteq V$  using ind by (simp add: lin-indep-def)
  have xA:  $x \notin A$ 
    using AV outside span-incl by blast
  show  $w \notin \text{span } (\text{insert } x A \setminus \{w\})$ 
proof
  assume wsp:  $w \in \text{span } (\text{insert } x A \setminus \{w\})$ 
  have wA:  $w \in A$  using w
    using insert-iff outside wsp by fastforce
  have w-outside:  $w \notin \text{span } (A \setminus \{w\})$ 
    using ind wA by (rule lin-indep-not-in-span)
  have w-span:  $w \in \text{span } (\text{insert } x (A \setminus \{w\}))$ 
    using wsp w-outside by (metis insert-Diff-iff)
  have diffV:  $A \setminus \{w\} \subseteq V$  using AV by auto
  then show False
    using outside in-span-insert wA w-outside w-span x by (metis insert-Diff)
qed
qed

```

57.8 Bases span and are independent

A spanning set spans the whole space: every vector of V lies in its span.

```

lemma spanning-span-all:
  assumes spanning B v  $\in V$ 
  shows  $v \in \text{span } B$ 
  using assms mod.spanningD spanning-def spanning-iff-mod-spanning by meson

```

A basis is linearly independent: the coordinate map is injective, and the zero vector is *lincomb mod.zero-coeffs* B , so any coordinate vector mapping to $\mathbf{0}_V$ must be the zero one.

```

lemma basis-lin-indep:
  assumes B: basis B
  shows lin-indep B
proof -
  have finB: finite B and B  $\subseteq V$  using B by (auto simp: basis-def)
  have inj: inj-on ( $\lambda c. \text{lincomb } c B$ ) (B  $\rightarrow_E R$ ) using B by (simp add: basis-def
    bij-betw-def)

```

```

have c v = 0 if cPiE: c ∈ B →E R and zero: lincomb c B = 0V and v ∈ B for c v
proof -
  define z where z ≡ restrict (λ-. 0) B
  have zPiE: z ∈ B →E R using additive.unit-closed by (auto simp: z-def)
  have lincomb z B = lincomb (λ-. 0) B by (rule lincomb-cong) (use ⟨B ⊆ V⟩ in
    ⟨auto simp: z-def⟩)
  also have ... = 0V by (simp add: ⟨B ⊆ V⟩)
  finally have lincomb z B = 0V .
  with zero inj cPiE zPiE have c = z
    by (metis inj-on-eq-iff)
  then show c v = 0
    using ⟨v ∈ B⟩ z-def by force
qed
with finB ⟨B ⊆ V⟩ show ?thesis unfolding lin-indep-def by blast
qed

```

A finite vector-space basis is also a basis in the underlying free-module sense. This bridge lets constructions defined by the universal property of free modules consume the coordinate-map basis used by the dimension theory.

```

lemma basis-module-basis:
  assumes B: basis B
  shows mod.module-basis B
proof (rule mod.module-basisI)
  have finB: finite B and BV: B ⊆ V using B by (auto simp: basis-def)
  show mod.spanning B
    using basis-spanning[OF B] spanning-iff-mod-spanning[OF finB BV] by simp
  show mod.lin-indep B
    using basis-lin-indep[OF B] lin-indep-iff-mod-lin-indep[OF finB BV] by simp
qed

```

Spanning forces equality with a containing independent set. If $T \subseteq S$, S is independent and $S \subseteq \text{span } T$, then $S = T$: any $w \in S - T$ would lie in $\text{span } T \subseteq \text{span } (S - \{w\})$, contradicting independence.

```

lemma spanning-subset-independent:
  assumes TS: T ⊆ S and iS: lin-indep S and SsT: S ⊆ span T
  shows S = T
proof (rule antisym[OF - TS])
  show S ⊆ T
  proof (rule ccontr)
    assume ¬ S ⊆ T
    then obtain w where w: w ∈ S w ∉ T T ⊆ S \ {w} using TS by blast
    then have w ∈ span (S \ {w})
      using SsT span-mono by blast
    with lin-indep-not-in-span[OF iS w(1)] show False by simp
  qed
qed

```

57.9 The Steinitz exchange lemma

An independent set contained in the span of a set T is contained in the span of an equinumerous “exchanged” set drawn from $S \cup T$: independent vectors are traded into T one at a time. Induction on $\text{card } (T - S)$, following Steinitz.

lemma *exchange-lemma*:

assumes *finite* T **and** *lin-indep* S **and** $S \subseteq \text{span } T$ **and** $T \subseteq V$

shows $\exists T'. \text{card } T' = \text{card } T \wedge \text{finite } T' \wedge S \subseteq T' \wedge T' \subseteq S \cup T \wedge S \subseteq \text{span } T'$

using *assms*

proof (*induct card* $(T \setminus S)$ *arbitrary: S T rule: less-induct*)

case *less*

note $ft = \langle \text{finite } T \rangle$ **and** $iS = \langle \text{lin-indep } S \rangle$ **and** $spST = \langle S \subseteq \text{span } T \rangle$ **and** $TV = \langle T \subseteq V \rangle$

have $SV: S \subseteq V$ **using** iS **by** (*simp add: lin-indep-def*)

have $finS: \text{finite } S$ **using** iS **by** (*simp add: lin-indep-def*)

let $?P = \lambda T'. \text{card } T' = \text{card } T \wedge \text{finite } T' \wedge S \subseteq T' \wedge T' \subseteq S \cup T \wedge S \subseteq \text{span } T'$

show $?case$

proof (*cases* $S \subseteq T \vee T \subseteq S$)

case *True*

then show $?thesis$

using ft iS $spST$ *spanning-subset-independent* **by** *auto*

next

case *False*

then obtain b **where** $\neg S \subseteq T$ **and** $b: b \in T \wedge b \notin S \wedge T \setminus \{b\} \setminus S \subseteq T \setminus S$ **by**

blast

then have $cardlt: \text{card } (T \setminus \{b\} \setminus S) < \text{card } (T \setminus S)$

using ft **by** (*auto intro: psubset-card-mono*)

from b ft **have** $ct0: \text{card } T \neq 0$ **by** *auto*

show $?thesis$

proof (*cases* $S \subseteq \text{span } (T \setminus \{b\})$)

case *True*

from ft **have** $ftb: \text{finite } (T \setminus \{b\})$ **by** *auto*

have $TbV: T \setminus \{b\} \subseteq V$ **using** TV **by** *blast*

from $less(1)[OF \text{ cardlt } ftb \ iS \ \text{True } TbV]$

obtain U **where** $U: \text{card } U = \text{card } (T \setminus \{b\}) \wedge S \subseteq U \wedge U \subseteq S \cup (T \setminus \{b\}) \wedge S \subseteq$

$\text{span } U$

and $fu: \text{finite } U$ **by** *blast*

have $bu: b \notin U$ **using** b U **by** *blast*

then have $th2: \text{card } (\text{insert } b \ U) = \text{card } T$ **using** *card-insert-disjoint* $[OF \ fu \ bu]$

ct0

using $U(1)$ $b(1)$ ft **by** (*metis card.remove*)

have $S \subseteq \text{span } (\text{insert } b \ U)$

using $U(4)$ *span-mono* **by** *blast*

then show $?thesis$ **using** U b $th2$ fu **by** *blast*

next

case *False*

then obtain a **where** $a: a \in S \wedge a \notin \text{span } (T \setminus \{b\})$ **and** $aV: a \in V$ **and** $ab: a \neq b$

```

    using SV b(2) by blast
  have at:  $a \notin T$ 
    using a ab TV by (auto intro: span-incl)
  have ft': finite (insert a ( $T \setminus \{b\}$ )) using ft by auto
  have insV:  $\text{insert } a (T \setminus \{b\}) \subseteq V$  using aV TV by blast
  —  $b$  may be exchanged for  $a$ , so  $S \subseteq \text{span } (\text{insert } a (T - \{b\}))$ .
  have sp':  $S \subseteq \text{span } (\text{insert } a (T \setminus \{b\}))$ 
proof
  fix x assume xs:  $x \in S$ 
  have TbV:  $T \setminus \{b\} \subseteq V$  using TV by blast
  have aspan:  $a \in \text{span } (\text{insert } b (T \setminus \{b\}))$ 
    using a(1) spST b by (metis insert-Diff subsetD)
  have bV:  $b \in V$  using b TV by blast
  have bs:  $b \in \text{span } (\text{insert } a (T \setminus \{b\}))$ 
    by (rule in-span-insert[OF TbV bV aspan a(2)])
  have xspanT:  $x \in \text{span } T$  using xs spST by blast
  have  $T \subseteq \text{insert } a (\text{insert } b (T \setminus \{b\}))$  using b by auto
  with xspanT have x:  $x \in \text{span } (\text{insert } a (\text{insert } b (T \setminus \{b\})))$ 
    using span-mono by blast
  — Absorb  $b$  (which is spanned) into the generating set.
  show  $x \in \text{span } (\text{insert } a (T \setminus \{b\}))$ 
    using span-trans-mem[OF insV bs] x by (simp add: insert-commute)
qed
have mlt:  $\text{card } (\text{insert } a (T \setminus \{b\}) \setminus S) < \text{card } (T \setminus S)$ 
  using cardlt ft a b by (auto intro: psubset-card-mono)
from less(1)[OF mlt ft' iS sp' insV] a b ft at ct0 ab show ?thesis by auto
qed
qed
qed

```

57.10 The cardinality bound and uniqueness of dimension

Steinitz bound. A linearly independent set is no larger than any set whose span contains it — in particular, than any spanning set.

theorem *independent-le-span:*

assumes iS : *lin-indep* S **and** fT : *finite* T **and** TV : $T \subseteq V$ **and** $spST$: $S \subseteq \text{span } T$
shows $\text{card } S \leq \text{card } T$
by (metis exchange-lemma[OF fT iS spST TV] card-mono)

Uniqueness of dimension over an arbitrary field. Any two bases of V have the same cardinality: each is independent and spans, so each bounds the other by the Steinitz bound. This generalises *basis-card-unique-finite*, dropping the finiteness of the base field.

theorem *basis-card-unique:*

assumes $B1$: *basis* B_1 **and** $B2$: *basis* B_2
shows $\text{card } B_1 = \text{card } B_2$

proof —

have $sp1$: *spanning* B_1 **and** $sp2$: *spanning* B_2 **using** $B1 B2$ **by** (auto intro: basis-spanning)

```

have i1: lin-indep B1 and i2: lin-indep B2 using B1 B2 by (auto intro: ba-
sis-lin-indep)
have finite B1 finite B2 and B1V: B1 ⊆ V and B2V: B2 ⊆ V
using B1 B2 by (auto simp: basis-def)
— Every basis vector lies in the span of the other basis (which spans V).
moreover have B1 ⊆ span B2 B2 ⊆ span B1
using sp2 B1V sp1 B2V by (auto intro: spanning-span-all)
ultimately show ?thesis
by (simp add: antisym i1 i2 independent-le-span)
qed

```

Consequently the *dimension* is the size of *any* basis, over an arbitrary field — the finite-field hypothesis of *dimension-eq* is no longer needed.

theorem *dimension-eq-any-field*:

```

assumes B: basis B shows dimension = card B
by (metis assms basis-card-unique dimension-def someI)

```

57.11 Existence of a basis

basis is defined as bijectivity of the coordinate map, and *basis-spanning* and *basis-lin-indep* take it apart. The converse — that spanning together with independence *gives* a basis — is what one needs in order to *build* a basis, and it is the missing introduction rule: surjectivity of the coordinate map is spanning, and injectivity is independence, applied to the difference of two coordinate vectors.

lemma *basisI*:

```

assumes sp: spanning B and ind: lin-indep B
shows basis B
unfolding basis-def
proof (intro conjI)
show finB: finite B and BV: B ⊆ V using sp by (auto simp: spanning-def)
show bij-betw (λc. lincomb c B) (B →E R) V
unfolding bij-betw-def
proof
— Injectivity. If two coordinate vectors give the same vector, their difference is a
dependence, so independence makes it zero coefficientwise.
show inj-on (λc. lincomb c B) (B →E R)
proof (rule inj-onI)
fix c d assume c: c ∈ B →E R and d: d ∈ B →E R and eq: lincomb c B =
lincomb d B
have cR: ⋀v. v ∈ B ⇒ c v ∈ R and dR: ⋀v. v ∈ B ⇒ d v ∈ R using c d
by auto
— The coefficientwise difference. Adding it to d recovers c, so by lincomb-add its
combination is 0V.
define e where e ≡ restrict (λv. c v + additive.inverse (d v)) B
have eR: ⋀v. v ∈ B ⇒ e v ∈ R using cR dR by (auto simp: e-def)
have ePiE: e ∈ B →E R using eR by (simp add: e-def)
have sum-back: ⋀v. v ∈ B ⇒ e v + d v = c v
using cR dR by (simp add: e-def additive.associative additive.inverse-composition-commute)

```

```

have lincomb e B  $\oplus$  lincomb d B = lincomb ( $\lambda v. e\ v + d\ v$ ) B
using BV finB eR dR by (rule lincomb-add)
also have ... = lincomb c B by (rule lincomb-cong) (use BV sum-back cR in
auto)
finally have lincomb e B  $\oplus$  lincomb d B = lincomb d B using eq by simp
then have lincomb e B =  $\mathbf{0}_V$ 
using BV finB eR dR
by (metis lincomb-closed vadd.invertible vadd.invertible-right-cancel vadd.left-unit
vzero-closed)
then have  $\bigwedge v. v \in B \implies e\ v = \mathbf{0}$  using ind ePiE by (simp add: lin-indep-def)
then have  $\bigwedge v. v \in B \implies c\ v = d\ v$ 
using cR dR sum-back by (metis additive.left-unit)
then show c = d using c d by (blast intro: PiE-ext)
qed
— Surjectivity is spanning, in the coordinate form the definition uses.
show ( $\lambda c. \text{lincomb } c\ B$ ) ' (B  $\rightarrow_E$  R) = V
proof
show ( $\lambda c. \text{lincomb } c\ B$ ) ' (B  $\rightarrow_E$  R)  $\subseteq V$ 
using finB BV by (auto intro: lincomb-closed)
show V  $\subseteq$  ( $\lambda c. \text{lincomb } c\ B$ ) ' (B  $\rightarrow_E$  R)
using sp by (auto simp: spanning-def)
qed
qed
qed

```

The zero vector space has the empty basis. This small introduction rule is useful when dimension arguments reduce a kernel or intersection to the trivial subspace.

```

lemma basis-empty-trivial:
assumes triv: V =  $\{\mathbf{0}_V\}$ 
shows basis {}
proof (rule basisI)
show spanning {}
unfolding spanning-def
using mod.lincomb-empty triv by auto
show lin-indep {} by (simp add: lin-indep-def)
qed

```

Basis extension. Every finite independent set extends to a basis inside its union with any given basis. At each step, a basis vector outside the current span is adjoined; the finite difference from the given basis is the termination measure.

```

theorem basis-extension:
assumes ind: lin-indep A and B: basis B
shows  $\exists C. A \subseteq C \wedge C \subseteq A \cup B \wedge \text{basis } C$ 
using assms
proof (induction card (B \ A) arbitrary: A rule: less-induct)
case less
note ind = less.prem(1) and B = less.prem(2)
have finA: finite A and AV: A  $\subseteq V$  using ind by (auto simp: lin-indep-def)

```

```

show ?case
proof (cases spanning A)
  case True
    then show ?thesis
      using basisI ind by blast
  next
    case False
      have  $\neg B \subseteq \text{span } A$ 
      proof
        assume Bspan:  $B \subseteq \text{span } A$ 
        have  $\bigwedge v. v \in V \implies v \in \text{span } A$ 
          using AV B Bspan basis-spanning span-trans spanning-span-all by blast
        with mod.spanningI[OF AV] have spanning A
          unfolding spanning-iff-mod-spanning[OF finA AV] .
        then show False using False by contradiction
      qed
    then obtain b where b:  $b \in B$   $b \notin \text{span } A$  by blast
    have BV:  $B \subseteq V$  and finB: finite B using B by (auto simp: basis-def)
    have bV:  $b \in V$  using b BV by blast
    have bA:  $b \notin A$ 
      using AV b(2) span-incl by blast
    have  $0 < \text{card } (B \setminus A)$ 
      using finB b bA by (simp add: card-gt-0-iff) blast
    then have measure:  $\text{card } (B \setminus \text{insert } b A) < \text{card } (B \setminus A)$ 
      by (simp add: b(1) bA)
    obtain C where C:  $\text{insert } b A \subseteq C$   $C \subseteq \text{insert } b A \cup B$  basis C
      using B b(2) bV ind less.hyps lin-indep-insert measure by meson
    have  $A \subseteq C$  using C(1) by blast
    moreover have  $C \subseteq A \cup B$  using C b(1) by blast
    ultimately show ?thesis using C(3) by blast
  qed
qed

```

Every finitely spanned space has a basis, extracted from any finite spanning set. Discard redundant vectors one at a time: if some $w \in B$ lies in the span of the others then $B - \{w\}$ still spans (by *span-trans-mem*), and it is strictly smaller, so the process terminates. What it terminates at is a set no member of which is spanned by the rest, which by *not-in-span-lin-indep* is exactly independence. This is the result that turns “finite-dimensional” into “has a basis”, and hence the one that lets the dimension theory above be applied to a space presented by generators.

theorem *basis-exists-from-spanning*:

```

assumes spanning B
shows  $\exists B' \subseteq B.$  basis B'
using assms
proof (induction card B arbitrary: B rule: less-induct)
  case less
    have finB: finite B and BV:  $B \subseteq V$  using less.prem1 by (auto simp: spanning-def)
    show ?case

```

proof (cases $\exists w \in B. w \in \text{span } (B \setminus \{w\})$)
case *True*
— A redundant vector: drop it. The smaller set still spans, since anything in the span of B is already in the span of $B - \{w\}$.
then obtain w **where** $w: w \in B$ **and** $wsp: w \in \text{span } (B \setminus \{w\})$ **by** *blast*
have $\text{sub}: B \setminus \{w\} \subseteq V$ **using** BV **by** *blast*
have $\text{spanning } (B \setminus \{w\})$
unfolding *spanning-def*
proof (intro *conjI ballI finB sub*)
show $\text{finite } (B \setminus \{w\})$ **using** finB **by** *blast*
fix v **assume** $v: v \in V$
have $v \in \text{span } (B \setminus \{w\})$ **using** sub wsp
using *less.prem span-trans-mem spanning-span-all v w by (metis insert-Diff)*
then show $\exists c \in (B \setminus \{w\}) \rightarrow_E R. v = \text{lincomb } c (B \setminus \{w\})$
proof —
obtain $c C$ **where** $\text{veq}: v = \text{lincomb } c C$ **and** $\text{finC}: \text{finite } C$ **and** $CB: C \subseteq B \setminus \{w\}$
and $cR: \bigwedge u. u \in C \implies c u \in R$ **using** $\langle v \in \text{span } (B \setminus \{w\}) \rangle$ **unfolding** *span-def* **by** *blast*
define d **where** $d \equiv \text{restrict } (\lambda u. \text{if } u \in C \text{ then } c u \text{ else } \mathbf{0}) (B \setminus \{w\})$
have $dR: \bigwedge u. u \in B \setminus \{w\} \implies d u \in R$ **using** cR **by** (auto *simp: d-def*)
have $dPiE: d \in (B \setminus \{w\}) \rightarrow_E R$ **using** dR **by** (simp *add: d-def*)
— Note the orientation: the lemma reads $\text{lincomb } c D = \text{lincomb } c B$ with $B \subseteq D$, so the *small* set C is its B and the big one its D .
have $\text{lincomb } d (B \setminus \{w\}) = \text{lincomb } d C$
proof (intro $CB \text{ sub lincomb-zero-extend-eq}$)
show $\bigwedge u. \llbracket u \in B \setminus \{w\}; u \notin C \rrbracket \implies d u = \mathbf{0}$ **by** (simp *add: d-def*)
qed (use $\text{finB } CB \text{ sub } dR$ **in** *auto*)
also have $\dots = \text{lincomb } c C$ **by** (rule *lincomb-cong*) (use $CB \text{ sub } cR$ **in** $\langle \text{auto simp: d-def} \rangle$)
finally have $\text{lincomb } d (B \setminus \{w\}) = v$ **using** veq **by** *simp*
then show *?thesis* **using** $dPiE$ **by** *blast*
qed
qed
moreover have $\text{card } (B \setminus \{w\}) < \text{card } B$
by (meson $\text{finB } w \text{ card-Diff1-less-iff}$)
ultimately show *?thesis* **using** *less.hyps* **by** *blast*
next
case *False*
then show *?thesis*
using $BV \text{ basisI finB less.prem not-in-span-lin-indep}$ **by** *auto*
qed
qed

A finite vector space has cardinality $(\text{card } R)^{\text{dimension}}$.

theorem *card-eq-card-base-pow-dimension:*

assumes $\text{finV}: \text{finite } V$

shows $\text{card } V = \text{card } R \wedge \text{dimension}$

proof —

have $\text{spanV}: \text{spanning } V$


```

    using finV mod.spanning-whole spanning-iff-mod-spanning by blast
  obtain B where basis: basis B
    using basis-exists-from-spanning[OF spanV] by blast
  then show ?thesis
    using card-eq-card-base-pow-dim dimension-eq-any-field by presburger
qed

```

A uniform finite bound on the cardinalities of independent sets already gives a basis. Choose an independent set of maximum cardinality; if a vector lay outside its span, adjoining that vector would produce a strictly larger independent set.

```

theorem basis-exists-of-independent-card-bound:
  assumes bound:  $\bigwedge A. \text{lin-indep } A \implies \text{card } A \leq n$ 
  shows  $\exists A. \text{basis } A$ 
proof -
  define sizes where sizes  $\equiv \{\text{card } A \mid A. \text{lin-indep } A\}$ 
  obtain sizes-nonempty: sizes  $\neq \{\}$  and sizes-bounded: sizes  $\subseteq \{..n\}$ 
    using assms unfolding sizes-def lin-indep-def by auto
  have finite-sizes: finite sizes
    using sizes-bounded by (rule finite-subset) simp
  have Max sizes  $\in$  sizes by (rule Max-in[OF finite-sizes sizes-nonempty])
  then obtain A where indA: lin-indep A and maxA: Max sizes = card A
    unfolding sizes-def by blast
  have finA: finite A and AV:  $A \subseteq V$  using indA by (auto simp: lin-indep-def)
  have mod.spanning A
  proof (rule mod.spanningI[OF AV])
    fix x assume xV:  $x \in V$ 
    show  $x \in \text{span } A$ 
  proof (rule ccontr)
    assume outside:  $x \notin \text{span } A$ 
    have ind-insert: lin-indep (insert x A)
      by (rule lin-indep-insert[OF indA xV outside])
    have xA:  $x \notin A$ 
      using AV outside span-incl by force
    have card (insert x A)  $\leq$  Max sizes
      using Max-ge finite-sizes ind-insert sizes-def by blast
    then show False
      using finA maxA xA by force
  qed
  qed
  then show ?thesis
    using basisI finA indA mod.spanning-def spanning-iff-mod-spanning by meson
qed

```

Hence the dimension of a finitely spanned space is bounded by any spanning set.

```

corollary dimension-le-spanning:
  assumes spanning B shows dimension  $\leq$  card B
proof -
  obtain B' where B'B:  $B' \subseteq B$  and B': basis B'

```

```

    using assms by (blast dest: basis-exists-from-spanning)
  then show ?thesis using dimension-eq-any-field
    using assms card-mono spanning-def by auto
qed

```

end

```

context vector-subspace
begin

```

Linear combinations and spans do not depend on whether their vectors are viewed in the ambient space or in the subspace; only the carrier side-condition changes.

```

lemma sub-fincomp-eq:
  assumes finA: finite A and f: f ∈ A → W
  shows sub.vadd.fincomp f A = vadd.fincomp f A
  using finA f
proof (induction A rule: finite-induct)
  case empty
  then show ?case by simp
next
  case (insert x A)
  obtain fxV: f x ∈ V and fAV: f ∈ A → V
  using Module.submodule-subset W-submodule insert.premis mod.Module-axioms by
  fastforce
  with insert show ?case
  by auto
qed

```

```

lemma sub-lincomb-eq:
  assumes finA: finite A and AW: A ⊆ W
  and c: ⋀ v. v ∈ A ⇒ c v ∈ R
  shows sub.lincomb c A = lincomb c A
  unfolding sub.lincomb-def lincomb-def
  using AW c by (auto intro!: sub-fincomp-eq[OF finA] sub.scale-closed)

```

```

lemma sub-span-eq:
  assumes AW: A ⊆ W
  shows sub.span A = span A
proof
  show sub.span A ⊆ span A
  proof
    fix x assume x ∈ sub.span A
    then obtain c C where C: finite C C ⊆ A sub.mod.coeffs-on c C
      x = sub.lincomb c C and CW: C ⊆ W using assms sub.mod.spanE by (metis
      subset-trans)
    have x-eq: x = lincomb c C using C sub-lincomb-eq[OF C(1) CW] sub.mod.coeffs-onD
    by simp
    show x ∈ span A

```

```

    using C mod.spanI x-eq by force
  qed
show span A  $\subseteq$  sub.span A
proof
  fix x assume x  $\in$  span A
  then obtain c C where C: finite C C  $\subseteq$  A mod.coeffs-on c C x = lincomb c C
    and CW: C  $\subseteq$  W using assms mod.spanE by (metis subset-trans)
  have x-eq: x = sub.lincomb c C using C sub.lincomb-eq[OF C(1) CW] mod.coeffs-onD
by simp
  show x  $\in$  sub.span A
    using C sub.mod.spanI x-eq by force
  qed
qed

lemma sub-lin-indep-iff:
  assumes AW: A  $\subseteq$  W
  shows sub.lin-indep A  $\longleftrightarrow$  lin-indep A
proof
  assume ind: sub.lin-indep A
  show lin-indep A
    unfolding lin-indep-def
  proof (intro conjI ballI impI)
    show finA: finite A
      using ind by (simp add: sub.lin-indep-def)
    show A  $\subseteq$  V using AW W-submodule mod.submodule-subset by blast
    fix c v
    assume c: c  $\in$  A  $\rightarrow_E$  R and zero: lincomb c A =  $\mathbf{0}_V$  and v: v  $\in$  A
    have coeff:  $\bigwedge u. u \in A \implies c u \in R$  using c by auto
    have sub.lincomb c A =  $\mathbf{0}_V$  using zero sub.lincomb-eq[OF finA AW coeff] by simp
    then show c v =  $\mathbf{0}$  using ind c v by (simp add: sub.lin-indep-def)
  qed
next
  assume ind: lin-indep A
  show sub.lin-indep A
    unfolding sub.lin-indep-def
  proof (intro conjI ballI impI AW)
    show finA: finite A
      using ind by (simp add: lin-indep-def)
    fix c v
    assume c: c  $\in$  A  $\rightarrow_E$  R and zero: sub.lincomb c A =  $\mathbf{0}_V$  and v: v  $\in$  A
    have coeff:  $\bigwedge u. u \in A \implies c u \in R$  using c by auto
    have lincomb c A =  $\mathbf{0}_V$  using zero sub.lincomb-eq[OF finA AW coeff] by simp
    then show c v =  $\mathbf{0}$  using ind c v by (simp add: lin-indep-def)
  qed
qed

```

Subspaces of finite-dimensional spaces have bases. The possible cardinalities of independent subsets of W are bounded by the size of an ambient basis. Choose an independent set of maximum cardinality; adjoining any vector outside its span would contradict maximality, so it spans the subspace.

```

theorem subspace-basis-exists:
  assumes B: basis B
  shows  $\exists A. \text{sub.basis } A$ 
proof –
  define sizes where sizes  $\equiv \{\text{card } A \mid A. \text{sub.lin-indep } A\}$ 
  have finB: finite B and BV:  $B \subseteq V$  using B by (auto simp: basis-def)
  have sizes-nonempty: sizes  $\neq \{\}$ 
    unfolding sizes-def using sub.lin-indep-def by force
  have sizes-bounded: sizes  $\subseteq \{.. \text{card } B\}$ 
proof
  fix n assume  $n \in \text{sizes}$ 
  then obtain A where A: sub.lin-indep A  $n = \text{card } A$  and AW:  $A \subseteq W$ 
    using sizes-def sub.lin-indep-def by auto
  then have indA: lin-indep A using sub.lin-indep-iff[OF AW] A by simp
  have  $A \subseteq \text{span } B$  using basis-spanning[OF B]
    using indA lin-indep-def spanning-span-all by blast
  then show  $n \in \{.. \text{card } B\}$  using A
    by (simp add: BV finB indA independent-le-span)
qed
have finite-sizes: finite sizes
  using sizes-bounded by (rule finite-subset) simp
have Max sizes  $\in \text{sizes}$  by (rule Max-in[OF finite-sizes sizes-nonempty])
then obtain A where indA: sub.lin-indep A and maxA: Max sizes  $= \text{card } A$ 
  and finA: finite A and AW:  $A \subseteq W$ 
  unfolding sizes-def using sub.lin-indep-def by auto
have sub.mod.spanning A
proof (rule sub.mod.spanningI[OF AW])
  fix x assume xW:  $x \in W$ 
  show  $x \in \text{sub.span } A$ 
  proof (rule ccontr)
    assume outside:  $x \notin \text{sub.span } A$ 
    have ind-insert: sub.lin-indep (insert x A)
      by (rule sub.lin-indep-insert[OF indA xW outside])
    have insert-size:  $\text{card } (\text{insert } x \ A) = \text{Suc } (\text{card } A)$ 
      by (meson AW finA outside sub.span-incl card-insert-disjoint)
    then show False
      using insert-size maxA Max-ge finite-sizes ind-insert sizes-def by fastforce
  qed
qed
then have sub.basis A using indA
  by (simp add: AW finA sub.basisI sub.spanning-iff-mod-spanning)
then show ?thesis by blast
qed

```

A subspace of a finite-dimensional space cannot have larger dimension than the ambient space. This is the set-based counterpart of the standard monotonicity theorem for dimension: a subspace basis is also ambiently independent, so Steinitz bounds it by any ambient basis.

theorem *subspace-dimension-le*:

```

assumes  $B$ : basis  $B$ 
shows  $\text{sub.dimension} \leq \text{dimension}$ 
proof –
  obtain  $A$  where  $A$ : sub.basis  $A$  and  $AW$ :  $A \subseteq W$  and  $\text{ind}A$ : lin-indep  $A$ 
    using subspace-basis-exists[ $OF$   $B$ ] sub.basis-def sub.basis-lin-indep sub-lin-indep-iff
by metis
  have  $\text{fin}B$ : finite  $B$  and  $BV$ :  $B \subseteq V$  using  $B$  by (auto simp: basis-def)
  have  $AV$ :  $A \subseteq V$  using  $AW$  W-submodule mod.submodule-subset by blast
  have  $A \subseteq \text{span } B$ 
    using basis-spanning[ $OF$   $B$ ]  $AV$  by (auto intro: spanning-span-all)
  then show ?thesis
    using sub.dimension-eq-any-field[ $OF$   $A$ ] dimension-eq-any-field[ $OF$   $B$ ]
    by (simp add: BV finB indA independent-le-span)
qed

```

In finite dimension, a subspace has full dimension exactly when it is the whole ambient space. For the non-trivial direction, extend a subspace basis to an ambient basis; equality of dimensions leaves no additional basis vectors.

```

theorem subspace-eq-ambient-iff-dimension-eq:
  assumes  $B$ : basis  $B$ 
  shows  $W = V \longleftrightarrow \text{sub.dimension} = \text{dimension}$ 
proof
  assume  $WV$ :  $W = V$ 
  have  $\text{sub}B$ : sub.basis  $B$  using  $B$   $WV$  by simp
  show  $\text{sub.dimension} = \text{dimension}$ 
    using sub.dimension-eq-any-field[ $OF$   $\text{sub}B$ ] dimension-eq-any-field[ $OF$   $B$ ] by simp
next
  assume  $\text{dim}$ :  $\text{sub.dimension} = \text{dimension}$ 
  obtain  $A$  where  $A$ : sub.basis  $A$ 
    using subspace-basis-exists[ $OF$   $B$ ] by blast
  have  $AW$ :  $A \subseteq W$  using  $A$  by (simp add: sub.basis-def)
  have  $\text{ind}A$ : lin-indep  $A$ 
    using sub.basis-lin-indep[ $OF$   $A$ ] sub-lin-indep-iff[ $OF$   $AW$ ] by simp
  obtain  $C$  where  $AC$ :  $A \subseteq C$  and  $C$ : basis  $C$  and  $\text{fin}C$ : finite  $C$ 
    using basis-extension[ $OF$   $\text{ind}A$   $B$ ] basis-def by blast
  then have  $A=C$ 
    using  $A$  card-subset-eq dim dimension-eq-any-field sub.dimension-eq-any-field by
auto
  then have  $V \subseteq W$ 
    using  $C$  basis-spanning spanning-span-all AW W-submodule mod.span-minimal by
blast
  then show  $W = V$ 
    using  $W$ -submodule mod.submodule-subset by blast
qed

```

The only zero-dimensional subspace is the trivial one. The ambient basis assumption supplies a basis of the subspace; dimension zero then forces that basis to be empty.

```

theorem subspace-dimension-eq-zero-iff:

```

```

assumes B: basis B
shows sub.dimension = 0  $\longleftrightarrow$   $W = \{0_V\}$ 
proof
  assume dim: sub.dimension = 0
  obtain A where A: sub.basis A finite A
    using subspace-basis-exists[OF B] sub.basis-def by blast
  then have  $A = \{\}$  using sub.dimension-eq-any-field dim by fastforce
  then have empty-spanning: sub.spanning  $\{\}$ 
    using A sub.basis-spanning by blast
  then show  $W = \{0_V\}$ 
    unfolding sub.spanning-def by auto
next
  assume W:  $W = \{0_V\}$ 
  then show sub.dimension = 0
    using sub.dimension-eq-any-field sub.basis-empty-trivial by force
qed

end

end

```

58 A finite subgroup of the multiplicative group of a field is cyclic

```

theory Field-Mult-Cyclic
imports Poly-Ring Group-Operations
begin

```

In particular the multiplicative group of a finite field (e.g. $GF(p^n)$) is cyclic. The argument splits into a group-theoretic part and a field-theoretic part. If a finite group G satisfies the *root bound* that for every $0 < n$ at most n elements x satisfy $x^n = 1$, then G is cyclic. The field part supplies the root bound: the elements x of the multiplicative group with $x^n = 1$ are roots of the polynomial $X^n - 1$, and a field has at most n such roots.

58.1 The order of a power in a finite group

```

context Group
begin

```

In a finite group, the order of *local.power a k* is *element-order a div gcd (element-order a) k*. This is the key computation behind counting the elements of each order in a cyclic group.

```

lemma element-order-power:
  assumes fin: finite G and aG:  $a \in G$ 
  shows element-order (power a k) = element-order a div gcd (element-order a) k
proof –

```

```

define e where e = element-order a
define d where d = gcd e k
have e-pos: e > 0 unfolding e-def using element-order-pos[OF fin aG] .
have ak: power a k ∈ G using aG by simp
have pow-iff: power (power a k) m = 1 ⟷ (e div d) dvd m for m
proof -
  have step: power (power a k) m = 1 ⟷ e dvd (k * m)
    using aG e-def element-order-dvd fin power-mult power-element-order power-unit
    by (metis dvd-def)
  — Now reduce the divisibility using d.
  have d-pos: d > 0 unfolding d-def using e-pos by simp
  have d-dvd-e: d dvd e and d-dvd-k: d dvd k unfolding d-def by auto
  have coprime (e div d) (k div d)
    unfolding d-def using e-pos by (simp add: div-gcd-coprime)
  with step coprime-dvd-mult-right-iff d-dvd-e d-dvd-k show ?thesis
    by fastforce
qed
then show ?thesis
  using ak d-def e-def element-order-dvd fin power-element-order
  by (metis gcd-nat.asym gcd-nat.refl)
qed
end

```

58.2 Counting elements of each order in a cyclic group

```

context Group
begin

```

The elements of order exactly e in the cyclic group generated by a (where a has order e) are exactly its generators: the powers $\text{local.power } a \ k$ with k below e and coprime to it.

lemma *card-order-e-in-cyclic*:

```

assumes fin: finite G and aG: a ∈ G and ord-a: element-order a = e
shows card {x ∈ cyclic-subgroup a. element-order x = e} = card {k. k < e ∧ gcd e k = 1}

```

proof —

```

have e-pos: e > 0
  using ord-a element-order-pos[OF fin aG] by simp
have set-eq: {x ∈ cyclic-subgroup a. element-order x = e}
  = power a ‘ {k. k < e ∧ gcd e k = 1} (is ?L=?R)

```

proof

```

show ?L ⊆ ?R

```

proof

```

fix x assume xc: x ∈ ?L
then obtain k where xk: x = power a k and kl: k < e
  unfolding cyclic-subgroup-def ord-a by auto
have element-order (power a k) = e
  using element-order-power[OF fin aG] ord-a xc xk by blast

```

```

    then have gcd e k = 1
      by (simp add: aG div-eq-dividend-iff e-pos element-order-power fin ord-a)
    then show  $x \in ?R$  using  $xk\ kl$  by auto
  qed
qed (auto simp: cyclic-subgroup-def aG element-order-power fin ord-a)
— local.power a is injective on  $\{..<e\}$ .
have inj-on (power a)  $\{..<e\}$  using power-inj-on-order[OF fin aG] ord-a by simp
then have inj: inj-on (power a)  $\{k. k < e \wedge \gcd e k = 1\}$ 
  by (rule inj-on-subset) (auto simp: lessThan-def)
show card  $\{x \in \text{cyclic-subgroup } a. \text{element-order } x = e\} = \text{card } \{k. k < e \wedge \gcd e$ 
 $k = 1\}$ 
  using set-eq inj card-image by auto
qed

end

```

58.3 Counting the Residues Coprime to a Modulus

The cyclicity criterion for a finite group needs two arithmetic facts about the number of residues below e coprime to e : that it is positive, and that summing it over the divisors of d recovers d .

lemma *coprime-count-pos*:

```

  fixes e :: nat
  assumes 0 < e shows 0 < card  $\{k. k < e \wedge \gcd e k = 1\}$ 
    using Suc-lessI [OF assms] by (fastforce simp: card-gt-0-iff)

```

The residues below d whose greatest common divisor with d is exactly f are the multiples $j * f$ of f by residues below $d \text{ div } f$ coprime to it.

lemma *card-gcd-class*:

```

  fixes d f :: nat
  assumes d: 0 < d and f: f dvd d
  shows card  $\{j. j < d \text{ div } f \wedge \gcd (d \text{ div } f) j = 1\} = \text{card } \{k. k < d \wedge \gcd k d = f\}$ 
proof (rule bij-betw-same-card)
  have fpos: 0 < f using d f by (auto intro: dvd-pos-nat)
  have deg: d div f * f = d using f by simp
  show bij-betw  $(\lambda j. j * f) \{j. j < d \text{ div } f \wedge \gcd (d \text{ div } f) j = 1\} \{k. k < d \wedge \gcd k d = f\}$ 
    unfolding bij-betw-def
  proof
    show inj-on  $(\lambda j. j * f) \{j. j < d \text{ div } f \wedge \gcd (d \text{ div } f) j = 1\}$ 
      using fpos by (simp add: inj-on-def)
    next
      show  $(\lambda j. j * f) \{j. j < d \text{ div } f \wedge \gcd (d \text{ div } f) j = 1\} = \{k. k < d \wedge \gcd k d = f\}$ 
        proof (intro set-eqI iffI)
          fix k assume k  $\in (\lambda j. j * f) \{j. j < d \text{ div } f \wedge \gcd (d \text{ div } f) j = 1\}$ 
          then obtain j where j:  $j < d \text{ div } f$  and gcd:  $\gcd (d \text{ div } f) j = 1$  and k:  $k = j * f$  by auto
          from j(1) fpos have j * f < d div f * f by (rule mult-less-mono1)

```



```

then have  $k < d$  using  $k \text{ deq}$  by simp
moreover have  $\gcd k d = f$ 
proof -
  have  $\gcd k d = \gcd (j * f) (d \text{ div } f * f)$  using  $k \text{ deq}$  by simp
  also have  $\dots = \gcd (d \text{ div } f) j * f$  by (simp add: gcd-mult-right)
  also have  $\dots = f$  using  $j(2)$  by simp
  finally show ?thesis .
qed
ultimately show  $k \in \{k. k < d \wedge \gcd k d = f\}$  by simp
next
fix  $k$  assume  $k \in \{k. k < d \wedge \gcd k d = f\}$ 
then have  $k: k < d$  and  $g: \gcd k d = f$  and  $keq: k \text{ div } f * f = k$  by auto
have  $\gcd (d \text{ div } f) (k \text{ div } f) * f = f$  using  $g$ 
  using deq keq by (metis gcd.commute gcd-mult-distrib-nat mult.commute)
moreover have  $k \text{ div } f < d \text{ div } f$  using  $k \text{ fpos } f$  by (simp add: div-less-iff-less-mult)
ultimately show  $k \in (\lambda j. j * f) \text{ ' } \{j. j < d \text{ div } f \wedge \gcd (d \text{ div } f) j = 1\}$ 
  using keq fpos by (force simp: image-iff)
qed
qed
qed

```

Summing the coprime counts over the divisors of d recovers d : the residues below d are partitioned by their greatest common divisor with d .

```

theorem coprime-count-divisor-sum:
  fixes  $d :: \text{nat}$ 
  assumes  $d: 0 < d$ 
  shows  $(\sum e \mid e \text{ dvd } d. \text{card } \{k. k < e \wedge \gcd e k = 1\}) = d$ 
proof -
  have  $\text{part: } \{k. k < d\} = (\bigcup f \in \{f. f \text{ dvd } d\}. \{k. k < d \wedge \gcd k d = f\})$ 
    by auto
  have  $d = \text{card } \{k. k < d\}$  by simp
  also have  $\dots = (\sum f \mid f \text{ dvd } d. \text{card } \{k. k < d \wedge \gcd k d = f\})$ 
    unfolding part using  $d$  by (intro card-UN-disjoint) auto
  also have  $\dots = (\sum f \mid f \text{ dvd } d. \text{card } \{j. j < d \text{ div } f \wedge \gcd (d \text{ div } f) j = 1\})$ 
    using card-gcd-class [OF d] by simp
  also have  $\dots = (\sum e \mid e \text{ dvd } d. \text{card } \{k. k < e \wedge \gcd e k = 1\})$ 
  proof (rule sum.reindex-bij-witness [of -  $\lambda e. d \text{ div } e \lambda f. d \text{ div } f$ ])
    qed (use  $d \text{ div-div-eq-right}$  in auto)
  finally show ?thesis by simp
qed

```

58.4 The abstract group criterion for being cyclic

context *Group*
begin

The set of elements of order exactly e in a finite group.

Every element of a finite group has order dividing $\text{card } G$, so the group is partitioned by order over the divisors of $\text{card } G$.

```

lemma card-G-sum-orders:
  assumes fin: finite G
  shows  $\text{card } G = (\sum e \mid e \text{ dvd } \text{card } G. \text{card } \{x \in G. \text{element-order } x = e\})$ 
proof –
  have part:  $G = (\bigcup e \in \{e. e \text{ dvd } \text{card } G\}. \{x \in G. \text{element-order } x = e\})$ 
    using element-order-dvd-card[OF fin] by auto
  have  $\text{card } G > 0$ 
    using fin card-gt-0-iff unit-closed by blast
  then have fin-div: finite  $\{e. e \text{ dvd } \text{card } G\}$  by (rule finite-divisors-nat)
  have  $\text{card } G = \text{card } (\bigcup e \in \{e. e \text{ dvd } \text{card } G\}. \{x \in G. \text{element-order } x = e\})$ 
    using part by simp
  also have  $\dots = (\sum e \mid e \text{ dvd } \text{card } G. \text{card } \{x \in G. \text{element-order } x = e\})$ 
    using fin by (intro card-UN-disjoint fin-div) auto
  finally show ?thesis .
qed

```

The root bound: for each positive n , at most n elements satisfy $x^n = 1$. Under this hypothesis, at most $\text{card } \{k. k < e \wedge \text{gcd } e \ k = 1\}$ elements have order exactly e .

```

lemma card-order-e-le-coprime-count:
  assumes fin: finite G
    and bound:  $\bigwedge n. n > 0 \implies \text{card } \{x \in G. \text{power } x \ n = 1\} \leq n$ 
    and e-pos:  $e > 0$ 
  shows  $\text{card } \{x \in G. \text{element-order } x = e\} \leq \text{card } \{k. k < e \wedge \text{gcd } e \ k = 1\}$ 
proof (cases  $\{x \in G. \text{element-order } x = e\} = \{\}$ )
  case True show ?thesis
    by (simp add: True)
next
  case False
  then obtain a where aG:  $a \in G$  and ord-a:  $\text{element-order } a = e$  by auto
  — The  $e$  powers of  $a$  all solve  $x^e = 1$ ; the bound forces them to be all solutions.
  have cyc-sub: cyclic-subgroup  $a \subseteq \{x \in G. \text{power } x \ e = 1\}$ 
  proof
    fix x assume  $x \in \text{cyclic-subgroup } a$ 
    then obtain k where xk:  $x = \text{power } a \ k$  and  $k < e$ 
      unfolding cyclic-subgroup-def ord-a by auto
    have  $\text{power } x \ e = 1$ 
      using aG fin power-mult ord-a power-element-order power-unit xk by (metis
mult.commute)
    with xk aG show  $x \in \{x \in G. \text{power } x \ e = 1\}$  by simp
  qed
  have  $\text{card } (\text{cyclic-subgroup } a) = e$ 
    using card-cyclic-subgroup[OF fin aG] ord-a by simp
  with cyc-sub bound[OF e-pos] fin have  $*$ :  $\text{cyclic-subgroup } a = \{x \in G. \text{power } x \ e = 1\}$ 
    by (simp add: card-seteq)
  have  $\{x \in G. \text{element-order } x = e\} = \{x \in \text{cyclic-subgroup } a. \text{element-order } x = e\}$ 
    by (auto simp: * fin power-element-order)
  then show ?thesis

```

using *card-order-e-in-cyclic*[*OF fin aG ord-a*] by *simp*
qed

The abstract criterion: a finite group satisfying the root bound is cyclic.

theorem *cyclic-of-root-bound*:

assumes *fin*: *finite G* and *ne*: $G \neq \{\}$
and *bound*: $\bigwedge n. n > 0 \implies \text{card } \{x \in G. \text{power } x \ n = 1\} \leq n$
shows $\exists g \in G. \text{cyclic-subgroup } g = G$

proof –

define *d* where $d = \text{card } G$

have *d-pos*: $d > 0$ unfolding *d-def* using *fin ne* by (*simp add: card-gt-0-iff*)

— Every order divides *d*; summing over divisors gives *d*.

have *sum-orders*: $(\sum e \mid e \text{ dvd } d. \text{card } \{x \in G. \text{element-order } x = e\}) = d$

unfolding *d-def* using *card-G-sum-orders*[*OF fin*] by *simp*

have *sum-counts*: $(\sum e \mid e \text{ dvd } d. \text{card } \{k. k < e \wedge \text{gcd } e \ k = 1\}) = d$

using *coprime-count-divisor-sum* [*OF d-pos*] by *simp*

have *fin-div*: *finite* $\{e. e \text{ dvd } d\}$

using *d-pos finite-divisors-nat* by (*metis gr-implies-not0*)

have *le*: $\text{card } \{x \in G. \text{element-order } x = e\} \leq \text{card } \{k. k < e \wedge \text{gcd } e \ k = 1\}$

if $e \text{ dvd } d$ for *e*

using *bound card-0-eq card-order-e-le-coprime-count d-def fin that* by *fastforce*

have *eq-all*: $\text{card } \{x \in G. \text{element-order } x = e0\} = \text{card } \{k. k < e0 \wedge \text{gcd } e0 \ k = 1\}$

if $e0 \text{ dvd } d$ for *e0*

proof (*rule ccontr*)

assume *neq*: $\text{card } \{x \in G. \text{element-order } x = e0\} \neq \text{card } \{k. k < e0 \wedge \text{gcd } e0 \ k = 1\}$
= 1}

with *le that* have *strict*: $\text{card } \{x \in G. \text{element-order } x = e0\}$

$< \text{card } \{k. k < e0 \wedge \text{gcd } e0 \ k = 1\}$ by *force*

have $(\sum e \mid e \text{ dvd } d. \text{card } \{x \in G. \text{element-order } x = e\})$

$< (\sum e \mid e \text{ dvd } d. \text{card } \{k. k < e \wedge \text{gcd } e \ k = 1\})$

by (*intro sum-strict-mono-ex1 fin-div*) (*use that strict le in auto*)

with *sum-orders sum-counts* show *False* by *simp*

qed

— In particular there is an element of order *d*, which generates *G*.

have $\text{card } \{x \in G. \text{element-order } x = d\} = \text{card } \{k. k < d \wedge \text{gcd } d \ k = 1\}$

using *eq-all* by *simp*

moreover have $\text{card } \{k. k < d \wedge \text{gcd } d \ k = 1\} > 0$ using *d-pos* by (*rule coprime-count-pos*)

ultimately have $\{x \in G. \text{element-order } x = d\} \neq \{\}$ by (*metis card.empty less-irrefl*)

then obtain *g* where *gG*: $g \in G$ and *gord*: $\text{element-order } g = d$ by *auto*

have *cyclic-subgroup* $g = G$

by (*simp add: card-cyclic-subgroup card-subset-eq cyclic-subgroup-subset d-def fin gG gord*)

with *gG* show *?thesis* by *blast*

qed

end

58.5 The multiplicative group of a field and the root bound

context *Field*
begin

The nonzero elements of the field.

definition *Fstar* :: 'a set
where *Fstar* = *R* - {0}

lemma *Fstar-iff* [*simp*]: $x \in Fstar \longleftrightarrow x \in R \wedge x \neq 0$
unfolding *Fstar-def* **by** *auto*

lemma *one-in-Fstar*: $1 \in Fstar$
using *nontrivial* **by** *simp*

The nonzero elements form a group under multiplication.

theorem *Group-Fstar*: *Group Fstar* ($\cdot$) **1**
proof (*intro GroupI one-in-Fstar*)
fix *a b* **assume** $a \in Fstar$ $b \in Fstar$
then show $a \cdot b \in Fstar$
using *local.no-zero-divisors* **by** *auto*
next
fix *u* **assume** $u \in Fstar$
then show $\exists v \in Fstar. u \cdot v = 1 \wedge v \cdot u = 1$
using *field-inverse nontrivial* **by** *fastforce*
qed (*auto simp: multiplicative.associative*)
end

58.6 The polynomial root bound, via the polynomial ring

The number of n th roots of unity in a field is at most n . Rather than reproving a bound on polynomial roots by hand, we reuse *Field.card-roots-le-degree* from *Poly-Ring*: the n th roots of unity are exactly the roots of the polynomial $X^n - 1$, which is nonzero of degree n .

context *Field*
begin

The ring power *rpow* of *Poly-Ring* agrees with the multiplicative group power.

lemma *rpow-eq-power*: $rpow\ x\ n = multiplicative.power\ x\ n$
by (*induction n*) *simp-all*

The polynomial $X^n - 1$, as a coefficient sequence: -1 at 0, 1 at n , and 0 elsewhere (valid for $0 < n$).

definition *xn-minus-1* :: $nat \Rightarrow 'a$
where $xn-minus-1\ n \equiv (\lambda i. \text{if } i = 0 \text{ then } -1 \text{ else if } i = n \text{ then } 1 \text{ else } 0)$

lemma *xn-minus-1-closed*: $xn-minus-1\ n \in poly-carrier$

proof (rule poly-carrierI)
 have $\{i. \text{xn-minus-1 } n \ i \neq 0\} \subseteq \{0, n\}$ **by** (auto simp: xn-minus-1-def)
 then show finite $\{i. \text{xn-minus-1 } n \ i \neq 0\}$ **by** (rule finite-subset) simp
 show $\bigwedge i. \text{xn-minus-1 } n \ i \in R$ **by** (simp add: xn-minus-1-def)
qed

lemma xn-minus-1-coeff: $\text{xn-minus-1 } n \ i \in R$
by (simp add: xn-minus-1-def)

lemma degree-xn-minus-1:
 assumes $n: n > 0$ **shows** $\text{degree } (\text{xn-minus-1 } n) = n$
proof (rule le-antisym)
 show $\text{degree } (\text{xn-minus-1 } n) \leq n$
 using degree-le-iff xn-minus-1-closed xn-minus-1-def **by** auto
 then show $n \leq \text{degree } (\text{xn-minus-1 } n)$
 using coeff-gt-degree le-eq-less-or-eq nontrivial xn-minus-1-closed xn-minus-1-def
by fastforce
qed

lemma xn-minus-1-nonzero:
 assumes $n: n > 0$ **shows** $\text{xn-minus-1 } n \neq 0_P$
by (metis degree-xn-minus-1 degree-zero less-zeroE n)

Its value at x is $\text{multiplicative.power } x \ n - 1$.

lemma eval-xn-minus-1:
 assumes $n: n > 0$ **and** $x: x \in R$
 shows $\text{eval } x \ (\text{xn-minus-1 } n) = \text{multiplicative.power } x \ n + (-1)$
proof –
 have $\text{eval } x \ (\text{xn-minus-1 } n) = \text{additive.fincomp } (\lambda i. \text{xn-minus-1 } n \ i \cdot \text{rpow } x \ i) \ \{..n\}$
 using degree-xn-minus-1 local.eval-def n **by** force
 also have $\dots = \text{additive.fincomp } (\lambda i. \text{xn-minus-1 } n \ i \cdot \text{rpow } x \ i) \ \{0, n\}$
 using x
by (intro additive.fincomp-mono-neutral-cong-right) (auto simp: xn-minus-1-def
 xn-minus-1-coeff)
 also have $\dots = (-1) + \text{multiplicative.power } x \ n$
 using n x **by** (simp add: xn-minus-1-coeff xn-minus-1-def rpow-eq-power)
 also have $\dots = \text{multiplicative.power } x \ n + (-1)$
 using x **by** (simp add: additive.commutative)
 finally show ?thesis .
qed

The roots of unity are exactly the roots of $X^n - 1$.

lemma card-nth-roots-of-unity:
 assumes $n: n > 0$
 shows finite $\{x \in R. \text{multiplicative.power } x \ n = 1\}$
 $\wedge \text{card } \{x \in R. \text{multiplicative.power } x \ n = 1\} \leq n$
proof –
 have $\text{multiplicative.power } x \ n = 1$
 if $x \in R$ **and** $\text{multiplicative.power } x \ n - 1 = 0$ **for** x

```

proof –
  have multiplicative.power  $x\ n = (\text{multiplicative.power } x\ n + (-\ 1)) + 1$ 
  using additive.invertible-left-inverse[of 1]  $\langle x \in R \rangle$  by (simp add: additive.associative)
  also have  $\dots = 1$  using that by (simp add: additive.associative)
  finally show multiplicative.power  $x\ n = 1$  .
qed
then have  $\{x \in R. \text{multiplicative.power } x\ n = 1\} = \{x \in R. \text{eval } x\ (xn\text{-minus-1 } n)$ 
 $= 0\}$ 
  by (auto simp: eval-xn-minus-1 n)
then show ?thesis
  using degree-xn-minus-1 [OF  $n$ ] xn-minus-1-closed xn-minus-1-nonzero [OF  $n$ ] card-roots-le-degree

  by force
qed

end

```

58.7 The main theorem

The *Fstar*-power (multiplicative power in the group of nonzero elements) coincides with the field's multiplicative power, so the root bound transfers to the group.

```

context Field
begin

```

```

lemma Fstar-power-eq:
  Monoid.power  $(\cdot)\ 1\ x\ n = \text{multiplicative.power } x\ n$ 
  by (simp only: Monoid.power-def)

```

Any finite subgroup of the multiplicative group of a field is cyclic.

```

theorem finite-subgroup-Fstar-cyclic:
  assumes sub: Subgroup  $H\ Fstar\ (\cdot)\ 1$  and fin: finite  $H$ 
  shows  $\exists g \in H. \text{Group.cyclic-subgroup } (\cdot)\ 1\ g = H$ 
proof –
  interpret  $G$ : Group  $Fstar\ (\cdot)\ 1$  by (rule Group-Fstar)
  interpret  $H$ : Subgroup  $H\ Fstar\ (\cdot)\ 1$  by (rule sub)
  have  $HR$ :  $H \subseteq R$  using  $H.\text{subset}$  by (auto simp: Fstar-def)
  — The root bound restricted to  $H$ : at most  $n$  elements of  $H$  solve  $x^n = 1$ .
  have bound:  $\text{card } \{x \in H. \text{Monoid.power } (\cdot)\ 1\ x\ n = 1\} \leq n$  if  $n > 0$  for  $n$ 
  proof –
    have sub-set:  $\{x \in H. \text{Monoid.power } (\cdot)\ 1\ x\ n = 1\} \subseteq \{x \in R. \text{multiplicative.power}$ 
 $x\ n = 1\}$ 
    using  $HR$  by blast
    then show ?thesis
    by (metis (no-types, lifting) card-mono card-nth-roots-of-unity [of  $n$ ] order-trans
that)
  qed
  have ne:  $H \neq \{\}$  using  $H.\text{sub.unit-closed}$  by auto
  show ?thesis

```

```

    using H.sub.cyclic-of-root-bound[OF fin ne] bound by blast
qed

```

In particular, if the whole multiplicative group $Fstar$ is finite (e.g. a finite field), it is cyclic.

```

corollary finite-field-mult-cyclic:
  assumes fin: finite Fstar
  shows  $\exists g \in Fstar. \text{Group.cyclic-subgroup } (\cdot) \text{ } 1 \text{ } g = Fstar$ 
  by (simp add: Group.group-self-subgroup Group-Fstar fin finite-subgroup-Fstar-cyclic)

end

end

```

59 A field extension as a vector space over a subfield

```

theory Extension-Vector-Space
  imports Steinitz Ring-Theory Subfield
begin

```

We feed the type-class field operations into the abstract *Vector-Space*. For subfields $K \subseteq L$ of a type-class field, L is a vector space over K : the vectors are the elements of L , addition and the zero are the field's, and scalar multiplication is field multiplication (restricted to scalars from K). This is the linchpin that carries the Steinitz results — *Vector-Space.span*, *Vector-Space.basis*, *Vector-Space.dimension* and *Vector-Space.basis-card-unique* — over to field extensions, giving a general $[L:K]$ and its tower law.

```

notation plus (infixl  $\langle + \rangle$  65)
notation minus (infixl  $\langle - \rangle$  65)
unbundle uminus-syntax

```

59.1 A subfield is a locale-based field

The carrier K of a *Subfield* satisfies the locale-based *Field* axioms under the ambient type-class operations. Built bottom-up exactly as in *Ring-Theory*, but with carrier K in place of the whole type.

```

context Subfield
begin

lemma sf-add-group: Group K (+) 0
  using zero-closed add-closed uminus-closed
  by (intro GroupI; force simp: algebra-simps)

lemma sf-add-abelian: Abelian-Group K (+) 0
proof (rule Abelian-Group.intro[OF sf-add-group])

```

```

show commutative-monoid  $K$  (+) 0
  by unfold-locales (auto intro: add-closed simp: ac-simps)
qed

lemma sf-mult-cmonoid: commutative-monoid  $K$  (*) 1
  by unfold-locales (auto intro: mult-closed simp: ac-simps)

lemma sf-ring: Ring  $K$  (+) (*) 0 1
  by (simp add: Ring-axioms.intro Ring-def commutative-monoid.axioms(1) distrib-left
    mult commute sf-add-abelian sf-mult-cmonoid)

lemma sf-commutative-ring: commutative-ring  $K$  (+) (*) 0 1
  by (rule commutative-ring.intro[OF sf-ring sf-mult-cmonoid])

lemma sf-field: Field  $K$  (+) (*) 0 1
proof –
  interpret  $R$ : commutative-ring  $K$  (+) (*) 0 1 by (rule sf-commutative-ring)
  show Field  $K$  (+) (*) 0 1
  proof qed (use  $R$ .multiplicative.invertible-def inverse-closed in fastforce)+
qed

end

```

59.2 The extension as a vector space

Two nested subfields $K \subseteq L$. The larger field L is the space of vectors, the smaller field K supplies the scalars.

```

locale subfield-tower = base: Subfield  $K$  + ext: Subfield  $L$ 
  for  $K$   $L$  :: 'a :: field set +
  assumes base-subset:  $K \subseteq L$ 
begin

```

L is a K -vector space under the ambient field operations. (The locale name is qualified: *HOL.Vector-Spaces* is imported transitively and also defines a *vector-space* locale.)

```

sublocale vs: Vector-Space  $K$  (+) (*) 0 1 (+) 0  $L$  (*)
proof (intro Vector-Space.intro Vector-Space.Vector-Space-axioms.intro)
  show Field  $K$  (+) (*) 0 1 by (rule base.sf-field)
  show Abelian-Group  $L$  (+) 0 by (rule ext.sf-add-abelian)
next
  fix  $a$   $v$  assume  $a \in K$   $v \in L$ 
  then show  $a * v \in L$  using base-subset by (auto intro: ext.mult-closed)
qed (auto simp: algebra-simps)

```

end

```

context subfield-tower
begin

```


Over the extension, the abstract additive *commutative-monoid.fincomp* of the vector group is just the ordinary finite sum *sum*, since *vadd* = (+) and the summands lie in *L*.

```

lemma vadd-fincomp-eq-sum:
  assumes finite B and  $f \in B \rightarrow L$ 
  shows  $vs.vadd.fincomp\ f\ B = sum\ f\ B$ 
  using assms by (induct B rule: finite-induct) auto

```

Hence a *Vector-Space.lincomb* over a finite index set of vectors is the field sum of the scaled vectors.

```

lemma vs-lincomb-eq-sum:
  assumes finite B and  $B \subseteq L$  and  $\bigwedge v. v \in B \implies c\ v \in K$ 
  shows  $vs.lincomb\ c\ B = (\sum_{v \in B}. c\ v * v)$ 
proof –
  have  $(\lambda v. c\ v * v) \in B \rightarrow L$ 
  using assms base-subset by (auto intro: ext.mult-closed)
  then show ?thesis
  unfolding vs.lincomb-def by (rule vadd-fincomp-eq-sum[OF assms(1)])
qed

```

A finite K-scaled sum of vectors in L lies in the span of those vectors.

```

lemma sum-scale-in-span:
  assumes finite A and  $\bigwedge x. x \in A \implies g\ x \in L$  and  $\bigwedge x. x \in A \implies f\ x \in K$ 
  shows  $(\sum_{x \in A}. f\ x * g\ x) \in vs.span\ (g\ ` A)$ 
  using assms
proof (induct A rule: finite-induct)
  case empty
  show ?case by (simp add: vs.span-zero)
next
  case (insert x A)
  have gFL:  $g\ ` A \subseteq L$  using insert.prems by auto
  — The new term is a scalar multiple of g x in the span of the enlarged set.
  have term-span:  $f\ x * g\ x \in vs.span\ (g\ ` insert\ x\ A)$ 
  by (simp add: image-subset-iff insert.prems vs.span-incl vs.span-scale)
  — The rest lies in the span of the smaller set, hence of the enlarged one.
  have rest-span:  $(\sum_{x \in A}. f\ x * g\ x) \in vs.span\ (g\ ` insert\ x\ A)$ 
  using insert vs.span-mono[of g ` A g ` insert x A] by auto
  then show ?case
  using gFL insert term-span vs.span-vadd by auto
qed

```

Specialise the finite vector-space cardinality bridge to a finite field extension. The carrier of the larger field is the vector-space carrier, while *vs.dimension* is the extension degree in the existing tower interpretation.

```

corollary finite-subfield-tower-cardinality:
  assumes finL: finite L
  shows  $card\ L = card\ K \wedge vs.dimension$ 
  by (rule vs.card-eq-card-base-pow-dimension[OF finL])

```

Basis extension is part of the field-extension API as well: a finite independent set of vectors can be completed inside its union with an existing basis.

corollary *subfield-tower-basis-extension:*
assumes *ind*: *vs.lin-indep A and B: vs.basis B*
shows $\exists C. A \subseteq C \wedge C \subseteq A \cup B \wedge \text{vs.basis } C$
using *vs.basis-extension*[*OF ind B*] .

A finite extension has a basis in the same interpretation used for its dimension. This re-exports the basis-extraction theorem at the field-extension boundary, so clients do not have to reconstruct the finite spanning argument through the module implementation.

corollary *finite-subfield-tower-basis-exists:*
assumes *finL*: *finite L*
shows $\exists B. \text{vs.basis } B$
proof –
have *spanL*: *vs.spanning L*
by (*rule iffD2*[*OF vs.spanning-iff-mod-spanning*[*OF finL subset-refl*]])
(*rule vs.mod.spanning-whole*)
then show *?thesis*
using *vs.basis-exists-from-spanning*[*OF spanL*] **by** *blast*
qed

More generally, a finite spanning set for the larger field can be reduced to a basis. This is the reusable bridge for extensions presented by generators rather than by a finite carrier.

corollary *subfield-tower-basis-exists-of-finite-spanning:*
assumes *finB*: *finite B and BL: B ⊆ L and span: L ⊆ vs.span B*
shows $\exists C. C \subseteq B \wedge \text{vs.basis } C$
proof –
have *mod-span*: *vs.mod.spanning B*
by (*rule vs.mod.spanningI*[*OF BL*]) (*use span in blast*)
have *spanB*: *vs.spanning B*
by (*rule iffD2*[*OF vs.spanning-iff-mod-spanning*[*OF finB BL*] *mod-span*])
obtain *C* **where** $C \subseteq B$ **and** *vs.basis C*
using *vs.basis-exists-from-spanning*[*OF spanB*] **by** *blast*
then show *?thesis* **by** *blast*
qed

The dimension of a nontrivial finite field extension is positive. Stating this once avoids repeated cardinality arguments when a client needs a genuine extension degree.

corollary *finite-subfield-tower-dimension-pos:*
assumes *finL*: *finite L*
shows *vs.dimension > 0*
proof –
have *cardL*: *card L = card K ^ vs.dimension*
by (*rule finite-subfield-tower-cardinality*[*OF finL*])
have *pairL*: $\{0, 1\} \subseteq L$ **by** *auto*

```

have cardL-ge2:  $2 \leq \text{card } L$ 
  using card-mono[OF finL pairL] by simp
show ?thesis
proof (rule ccontr)
  assume  $\neg \text{vs.dimension} > 0$ 
  then have  $\text{vs.dimension} = 0$  by simp
  then have  $\text{card } L = 1$  using cardL by simp
  then show False using cardL-ge2 by simp
qed
qed

end

```

Every subfield sits in the trivial tower over itself, and every pair of nested subfields forms a tower; the sublocale then equips it with the full Steinitz vocabulary.

```

lemma subfield-tower-refl:
  assumes Subfield K shows subfield-tower K K
  by (rule subfield-tower.intro[OF assms assms]) (unfold-locales, rule subset-refl)

end

```

60 Frobenius on finite subfields

```

theory Finite-Field-Frobenius
  imports Prime-Subfield Steinitz Field-Mult-Cyclic Extension-Vector-Space
begin

```

60.1 Cardinality of finite fields

```

context Field
begin

```

A finite field is a finite-dimensional vector space over its canonical prime subfield. Taking the whole carrier as an initial finite spanning set supplies a basis; the coordinate bijection then counts the field.

```

theorem finite-field-cardinality:
  assumes fin: finite R
  shows  $\exists n > 0. \text{card } R = \text{characteristic} \wedge n$ 
proof -
  interpret F: finite-integral-domain R addition multiplication zero unit
    by unfold-locales (rule fin)
  interpret P: Field F.prime-subfield addition multiplication zero unit
    by (rule F.prime-subfield-field)
  interpret S: Subring F.prime-subfield R addition multiplication zero unit
    by (rule F.prime-subfield-subring)
  have sub:  $F.\text{prime-subfield} \subseteq R$  by (rule S.additive.subset)
  interpret V: Vector-Space F.prime-subfield addition multiplication zero unit

```

```

      addition zero R multiplication
proof (intro Vector-Space.intro Vector-Space-axioms.intro)
  show Field F.prime-subfield (+) (·) 0 1 by (rule P.Field-axioms)
  show Abelian-Group R (+) 0 by (rule additive.Abelian-Group-axioms)
next
  fix a v
  assume a: a ∈ F.prime-subfield and v: v ∈ R
  have aR: a ∈ R by (rule subsetD[OF sub a])
  show a · v ∈ R by (rule multiplicative.composition-closed[OF aR v])
next
  fix a u v
  assume a: a ∈ F.prime-subfield and u: u ∈ R and v: v ∈ R
  have aR: a ∈ R by (rule subsetD[OF sub a])
  show a · (u + v) = a · u + a · v
    by (rule distributive(1)[OF aR u v])
next
  fix a b v
  assume a: a ∈ F.prime-subfield and b: b ∈ F.prime-subfield and v: v ∈ R
  show (a + b) · v = a · v + b · v
    by (rule distributive(2)[OF v subsetD[OF sub a] subsetD[OF sub b]])
next
  fix a b v
  assume a: a ∈ F.prime-subfield and b: b ∈ F.prime-subfield and v: v ∈ R
  show a · b · v = a · (b · v)
    by (rule multiplicative.associative[OF subsetD[OF sub a] subsetD[OF sub b] v])
next
  fix v assume v: v ∈ R
  show 1 · v = v by (rule multiplicative.left-unit[OF v])
qed
have count: card R = card F.prime-subfield ^ V.dimension
  by (rule V.card-eq-card-base-pow-dimension[OF fin])
have count-char: card R = characteristic ^ V.dimension
  using count F.card-prime-subfield by simp
have pair-sub: {0, 1} ⊆ R by auto
have pair-card: card {0, 1} = 2 using nontrivial by simp
have card-ge2: 2 ≤ card R
  using card-mono[OF fin pair-sub] pair-card by simp
have dimension-pos: V.dimension > 0
proof (rule ccontr)
  assume ¬ V.dimension > 0
  then have V.dimension = 0 by simp
  then have card R = 1 using count-char by simp
  then show False using card-ge2 by simp
qed
show ?thesis using count-char dimension-pos by blast
qed

corollary finite-field-cardinality-prime-power:
assumes fin: finite R
shows ∃ p n. prime p ∧ n > 0 ∧ card R = p ^ n

```

```

proof –
  obtain  $n$  where  $n: n > 0$   $\text{card } R = \text{characteristic} \wedge n$ 
    using finite-field-cardinality[OF fin] by blast
  have prime characteristic by (rule characteristic-prime[OF fin])
  then show ?thesis using  $n$  by blast
qed

```

corollary *characteristic-dvd-card-finite-field*:

```

assumes fin: finite R
shows characteristic dvd card R

```

```

proof –
  obtain  $n$  where  $n: n > 0$   $\text{card } R = \text{characteristic} \wedge n$ 
    using finite-field-cardinality[OF fin] by blast
  have characteristic dvd characteristic  $\wedge n$  using  $n(1)$  by (cases  $n$ ) simp-all
  then show ?thesis using  $n(2)$  by simp
qed

```

end

A finite subfield is represented here as a carrier set inside an ambient type-class field. The existing finite-field cardinality theorem therefore speaks about the characteristic of the carrier-set locale. The first lemmas identify those two presentations; the remaining results expose Frobenius without introducing a second field representation.

```

definition frobenius-power ::  $\text{nat} \Rightarrow 'a :: \text{monoid-mult} \Rightarrow 'a$ 
  where frobenius-power  $q\ x = x \wedge q$ 

```

```

context Subfield
begin

```

```

interpretation carrier: Field K (+) (*) 0 1
  by (rule sf-field)

```

```

lemma carrier-natmult-eq-of-nat:
  carrier.natmult n = of-nat n
  by (induction  $n$ ) simp-all

```

lemma *carrier-characteristic-eq-CHAR*:

```

assumes fin: finite K
shows carrier.characteristic = CHAR('a)

```

```

proof –
  have small-nz: False
    if  $0 < n$  and  $n < \text{carrier.characteristic}$  and  $\text{of-nat } n = (0 :: 'a)$  for  $n$ 
    using that carrier.characteristic-least carrier-natmult-eq-of-nat
      linorder-not-less by auto
  then show ?thesis
    using carrier.characteristic-pos[OF fin] carrier-natmult-eq-of-nat by (metis CHAR-eq-posI)
qed

```

theorem *finite-subfield-cardinality-char-power*:
assumes *fin*: *finite K*
shows $\exists n > 0. \text{card } K = \text{CHAR}('a) \wedge n$
using *carrier.finite-field-cardinality*[*OF fin*]
by (*simp add: carrier-characteristic-eq-CHAR*[*OF fin*])

lemma *finite-subfield-CHAR-prime*:
assumes *fin*: *finite K*
shows *prime CHAR*('a)
using *carrier.characteristic-prime*[*OF fin*]
by (*simp add: carrier-characteristic-eq-CHAR*[*OF fin*])

end

lemma *frobenius-power-add-char-power*:
fixes *x y* :: '*a* :: *field*
assumes *char*: *prime CHAR*('a) **and** *q*: $q = \text{CHAR}('a) \wedge n$
shows *frobenius-power q* (*x* + *y*) = *frobenius-power q x* + *frobenius-power q y*
by (*simp add: char freshmans-dream' frobenius-power-def q*)

lemma *frobenius-power-mult*:
fixes *x y* :: '*a* :: *comm-monoid-mult*
shows *frobenius-power q* (*x* * *y*) = *frobenius-power q x* * *frobenius-power q y*
by (*simp add: frobenius-power-def power-mult-distrib*)

lemma *frobenius-power-one* [*simp*]:
frobenius-power q (*1* :: '*a* :: *monoid-mult*) = *1*
by (*simp add: frobenius-power-def*)

lemma *frobenius-power-zero* [*simp*]:
assumes *q* > *0*
shows *frobenius-power q* (*0* :: '*a* :: *semiring-1*) = *0*
using *assms* **by** (*simp add: frobenius-power-def zero-power*)

lemma *frobenius-power-uminus-char-power*:
fixes *x* :: '*a* :: *field*
assumes *char*: *prime CHAR*('a) **and** *q*: $q = \text{CHAR}('a) \wedge n$
shows *frobenius-power q* ($-x$) = $-(\text{frobenius-power } q \ x)$
using *char q* **by** (*metis add-cancel-left-left eq-neg-iff-add-eq-0 frobenius-power-add-char-power*)

lemma *frobenius-power-inj*:
fixes *x y* :: '*a* :: *field*
assumes *char*: *prime CHAR*('a) **and** *q*: $q = \text{CHAR}('a) \wedge n$
and *eq*: *frobenius-power q x* = *frobenius-power q y*
shows *x* = *y*
proof –
have *frobenius-power q* (*x* – *y*) = *frobenius-power q x* – *frobenius-power q y*
using *char q* **by** (*metis frobenius-power-add-char-power frobenius-power-uminus-char-power uminus-add-conv-diff*)
with *eq* **show** ?thesis

```

    by (simp add: frobenius-power-def)
qed

context Subfield
begin

interpretation carrier: Field K (+) (*) 0 1
  by (rule sf-field)

lemma frobenius-power-closed:
   $x \in K \implies \text{frobenius-power } q \ x \in K$ 
  unfolding frobenius-power-def by (rule power-closed)

lemma finite-frobenius-bij-betw:
  assumes fin: finite K and char: prime CHAR('a) and q:  $q = \text{CHAR}('a) ^ n$ 
  shows bij-betw (frobenius-power q) K K
proof -
  have inj: inj-on (frobenius-power q) K
    using frobenius-power-inj[OF char q] by (auto simp: inj-on-def)
  have image: frobenius-power q ' K = K
    using fin frobenius-power-closed inj by (meson endo-inj-surj image-subset-iff)
  show ?thesis
    by (simp add: bij-betw-def inj image)
qed

lemma finite-field-frobenius-identity:
  assumes fin: finite K and xK:  $x \in K$ 
  shows frobenius-power (card K) x = x
proof -
  have gt0: card K > 0
    using fin zero-closed card-gt-0-iff by blast
  show ?thesis
  proof (cases x = 0)
    case True
    with gt0 show ?thesis by simp
  next
    case False
    have carrier-power: carrier.multiplicative.power x (card carrier.Fstar) = 1
      using False Group.power-order-eq-1 carrier.Group-Fstar xK by fastforce
    have ambient-power: carrier.multiplicative.power x n =  $x ^ n$  for n
      by (induction n) simp-all
    with gt0 obtain  $x ^ (\text{card } K - 1) = 1$  card K = Suc (card K - 1)
      using carrier-power carrier.Fstar-def by force
    then show ?thesis
      by (metis frobenius-power-def mult.comm-neutral power-Suc)
  qed
qed
end

```

end

61 The lattice of ideals

```
theory Ideal-Lattice
  imports Ideal-Theory Order-Theory
begin
```

The ideals of a commutative ring R , ordered by inclusion, form a complete lattice: the infimum of a family of ideals is their intersection (with R , so the empty family has infimum R), and the supremum is the ideal generated by their union. Together with the subgroup lattice this exercises the abstract order theory on the two archetypal substructure lattices. As a by-product we obtain the ideal *generated by a set*, the analogue of the subgroup *generate*.

```
context commutative-ring
begin
```

61.1 Intersections of ideals

The intersection of a nonempty family of ideals is an ideal.

```
lemma ideal-Inter:
  assumes ne:  $\mathcal{A} \neq \{\}$  and id:  $\bigwedge I. I \in \mathcal{A} \implies \text{Ideal } I \text{ } R \text{ } (+) \text{ } (\cdot) \text{ } \mathbf{0} \text{ } \mathbf{1}$ 
  shows Ideal  $(\bigcap \mathcal{A}) \text{ } R \text{ } (+) \text{ } (\cdot) \text{ } \mathbf{0} \text{ } \mathbf{1}$ 
proof -
  obtain I0 where I0:  $I0 \in \mathcal{A}$  using ne by blast
  interpret I0: Ideal I0 R (+) ( $\cdot$ )  $\mathbf{0} \text{ } \mathbf{1}$  by (rule id[OF I0])
  have sg: Subgroup  $(\bigcap \mathcal{A}) \text{ } R \text{ } (+) \text{ } \mathbf{0}$ 
  proof (rule additive.subgroupI)
    show  $\bigcap \mathcal{A} \subseteq R$  using I0 I0.additive.subset by blast
  next
    show  $\mathbf{0} \in \bigcap \mathcal{A}$ 
  proof
    fix I assume I:  $I \in \mathcal{A}$ 
    then interpret I: Ideal I R (+) ( $\cdot$ )  $\mathbf{0} \text{ } \mathbf{1}$  by (rule id)
    show  $\mathbf{0} \in I$  by (rule I.additive.sub-unit-closed)
  qed
  next
    fix g h assume g:  $g \in \bigcap \mathcal{A}$  and h:  $h \in \bigcap \mathcal{A}$ 
    show  $g + h \in \bigcap \mathcal{A}$ 
  proof
    fix I assume I:  $I \in \mathcal{A}$ 
    then interpret I: Ideal I R (+) ( $\cdot$ )  $\mathbf{0} \text{ } \mathbf{1}$  by (rule id)
    show  $g + h \in I$  using g h I by (blast intro: I.additive.sub-composition-closed)
  qed
  next
    fix g assume g:  $g \in \bigcap \mathcal{A}$ 
    then have  $g \in R$  using I0 I0.additive.subset by blast
```



```

    then show additive.invertible g by simp
next
fix g assume g: g ∈ ⋂ A
show additive.inverse g ∈ ⋂ A
proof
  fix I assume I: I ∈ A
  then interpret I: Ideal I R (+) (·) 0 1 by (rule id)
  have g ∈ I using g I by blast
  then show additive.inverse g ∈ I
    using I.additive.submonoid-inverse-closed I.additive.sub.invertible by blast
qed
qed
interpret S: Subgroup ⋂ A R (+) 0 by (rule sg)
show ?thesis
proof unfold-locales
  fix a b assume a: a ∈ R and b: b ∈ ⋂ A
  show a · b ∈ ⋂ A
  proof
    fix I assume I: I ∈ A
    then interpret I: Ideal I R (+) (·) 0 1 by (rule id)
    show a · b ∈ I using a b I by (blast intro: I.Ideal(1))
  qed
  show b · a ∈ ⋂ A
  proof
    fix I assume I: I ∈ A
    then interpret I: Ideal I R (+) (·) 0 1 by (rule id)
    show b · a ∈ I using a b I by (blast intro: I.Ideal(2))
  qed
qed
qed
qed

```

The whole ring is an ideal of itself (the top of the ideal lattice).

```

lemma whole-ideal: Ideal R R (+) (·) 0 1
proof -
  have Subgroup R R (+) 0
  by (rule additive.subgroupI) auto
  then interpret S: Subgroup R R (+) 0 .
  show ?thesis by unfold-locales auto
qed

```

61.2 The ideal generated by a set

The ideal generated by a subset S of the ring: the intersection of all ideals containing $S \cap R$. (Intersecting with R in the guard mirrors the subgroup *generate*, so elements outside the carrier are ignored.)

```

definition ideal-generated :: 'a set ⇒ 'a set
where ideal-generated S = ⋂ {I. Ideal I R (+) (·) 0 1 ∧ S ∩ R ⊆ I}

```

```

lemma ideal-generated-is-ideal: Ideal (ideal-generated S) R (+) (·) 0 1

```

proof –
have $R \in \{I. \text{Ideal } I \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \wedge S \cap R \subseteq I\}$
using *whole-ideal* **by** *auto*
then have $ne: \{I. \text{Ideal } I \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \wedge S \cap R \subseteq I\} \neq \{\}$ **by** *blast*
show *?thesis*
unfolding *ideal-generated-def* **by** *(rule ideal-Inter[OF ne]) auto*
qed

lemma *ideal-generated-incl*: $S \cap R \subseteq \text{ideal-generated } S$
unfolding *ideal-generated-def* **by** *blast*

lemma *ideal-generated-minimal*:
assumes $\text{Ideal } J \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \ S \cap R \subseteq J$
shows $\text{ideal-generated } S \subseteq J$
unfolding *ideal-generated-def* **using** *assms* **by** *(blast intro: Inter-lower)*

lemma *ideal-generated-subset-carrier*: $\text{ideal-generated } S \subseteq R$
proof –
interpret *gen*: $\text{Ideal ideal-generated } S \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}$ **by** *(rule ideal-generated-is-ideal)*
show *?thesis* **by** *(rule gen.additive.subset)*
qed

61.3 The product of two ideals

The product $I \odot J$ of two ideals is the ideal generated by all products $i \cdot j$ with $i \in I$ and $j \in J$. Together with the ideal sum and the intersection this completes the semiring of ideals; the product is contained in the intersection (hence, unlike sum and intersection, is in general *not* a lattice operation).

definition *ideal-prod* :: $'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$ (**infixl** $\langle \odot \rangle$ 70)
where $I \odot J = \text{ideal-generated } \{i \cdot j \mid i \in I \wedge j \in J\}$

lemma *ideal-prod-is-ideal*: $\text{Ideal } (I \odot J) \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}$
unfolding *ideal-prod-def* **by** *(rule ideal-generated-is-ideal)*

lemma *ideal-prod-subset-carrier*: $I \odot J \subseteq R$
unfolding *ideal-prod-def* **by** *(rule ideal-generated-subset-carrier)*

Generators lie in the product.

lemma *ideal-prod-memI*:
assumes $I: \text{Ideal } I \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}$ **and** $J: \text{Ideal } J \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}$
and $i: i \in I$ **and** $j: j \in J$
shows $i \cdot j \in I \odot J$
proof –
interpret $I: \text{Ideal } I \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}$ **by** *(rule I)*
interpret $J: \text{Ideal } J \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}$ **by** *(rule J)*
have $i \cdot j \in R$ **using** $i \ j \ I.\text{additive.subset } J.\text{additive.subset}$ **by** *blast*
moreover have $i \cdot j \in \{i \cdot j \mid i \in I \wedge j \in J\}$ **using** $i \ j$ **by** *blast*
ultimately have $i \cdot j \in \{i \cdot j \mid i \in I \wedge j \in J\} \cap R$ **by** *blast*
also have $\dots \subseteq \text{ideal-generated } \{i \cdot j \mid i \in I \wedge j \in J\}$ **by** *(rule ideal-generated-incl)*

finally show ?thesis unfolding ideal-prod-def .
qed

The product is contained in both factors, hence in their intersection.

lemma ideal-prod-subset:

assumes I : Ideal I R $(+)$ $(\cdot)$ **0** **1** and J : Ideal J R $(+)$ $(\cdot)$ **0** **1**
shows $I \odot J \subseteq I \cap J$

proof –

interpret I : Ideal I R $(+)$ $(\cdot)$ **0** **1** by (rule I)

interpret J : Ideal J R $(+)$ $(\cdot)$ **0** **1** by (rule J)

have gen: $\{i \cdot j \mid i \cdot j. i \in I \wedge j \in J\} \cap R \subseteq I \cap J$

proof

fix x assume $x \in \{i \cdot j \mid i \cdot j. i \in I \wedge j \in J\} \cap R$

then obtain $i \cdot j$ where $x = i \cdot j$ and $i: i \in I$ and $j: j \in J$ by blast

have $i \cdot j \in I$ using $i \cdot j$ J .additive.subset by (blast intro: I .Ideal(2))

moreover have $i \cdot j \in J$ using $i \cdot j$ I .additive.subset by (blast intro: J .Ideal(1))

ultimately show $x \in I \cap J$ using x by simp

qed

have Ideal $(\bigcap \{I, J\})$ R $(+)$ $(\cdot)$ **0** **1**

by (rule ideal-Inter) (use I J in auto)

then have IJ : Ideal $(I \cap J)$ R $(+)$ $(\cdot)$ **0** **1** by simp

show ?thesis

unfolding ideal-prod-def by (rule ideal-generated-minimal[OF IJ]) (use gen in simp)

qed

Commutativity: the generating sets coincide because the ring is commutative.

lemma ideal-prod-commute:

assumes I : Ideal I R $(+)$ $(\cdot)$ **0** **1** and J : Ideal J R $(+)$ $(\cdot)$ **0** **1**

shows $I \odot J = J \odot I$

proof –

interpret I : Ideal I R $(+)$ $(\cdot)$ **0** **1** by (rule I)

interpret J : Ideal J R $(+)$ $(\cdot)$ **0** **1** by (rule J)

have $\{i \cdot j \mid i \cdot j. i \in I \wedge j \in J\} = \{j \cdot i \mid i \cdot j. i \in I \wedge j \in J\}$

using I .additive.subset J .additive.subset by (blast intro: multiplicative.commutative)

also have $\dots = \{j \cdot i \mid j \cdot i. j \in J \wedge i \in I\}$ by blast

finally show ?thesis unfolding ideal-prod-def by simp

qed

61.4 The complete lattice of ideals

definition ideals :: 'a set set

where ideals = $\{I. \text{Ideal } I \text{ } R \text{ } (+) \text{ } (\cdot) \text{ } \mathbf{0} \text{ } \mathbf{1}\}$

lemma ideals-iff [simp]: $I \in \text{ideals} \longleftrightarrow \text{Ideal } I \text{ } R \text{ } (+) \text{ } (\cdot) \text{ } \mathbf{0} \text{ } \mathbf{1}$

by (simp add: ideals-def)

interpretation ideal-order: Partial-Order ideals $(\subseteq)$

by unfold-locales auto

```

interpretation ideal-lattice: Complete-Lattice ideals ( $\subseteq$ )
proof (unfold-locales)
  fix  $\mathcal{A} :: 'a \text{ set set}$  assume  $\mathcal{A}: \mathcal{A} \subseteq \text{ideals}$ 
  — Supremum: the ideal generated by the union.
  let  $?s = \text{ideal-generated } (\bigcup \mathcal{A})$ 
  have  $s\text{-id}: \text{Ideal } ?s \ R \ (+) \ (\cdot) \ 0 \ 1$  by (rule ideal-generated-is-ideal)
  have  $\text{ideal-order.is-sup } ?s \ \mathcal{A}$ 
  proof (rule ideal-order.is-supI)
    show  $\text{ideal-order.upper-bound } ?s \ \mathcal{A}$ 
    proof (rule ideal-order.upper-boundI)
      show  $?s \in \text{ideals}$  using  $s\text{-id}$  by simp
      fix  $I$  assume  $I: I \in \mathcal{A}$ 
      then interpret  $I: \text{Ideal } I \ R \ (+) \ (\cdot) \ 0 \ 1$  using  $\mathcal{A}$  by auto
      have  $I \subseteq \bigcup \mathcal{A} \cap R$  using  $I \text{.additive.subset}$  by blast
      then show  $I \subseteq ?s$  using  $\text{ideal-generated-incl}$  by blast
    qed
  next
  fix  $x$  assume  $\text{ideal-order.upper-bound } x \ \mathcal{A}$ 
  then have  $x\text{id}: \text{Ideal } x \ R \ (+) \ (\cdot) \ 0 \ 1$  and  $\text{dom}: \bigwedge I. I \in \mathcal{A} \implies I \subseteq x$ 
    by (auto dest: ideal-order.upper-boundD simp: ideal-order.upper-bound-def)
  have  $\bigcup \mathcal{A} \cap R \subseteq x$ 
  proof
    fix  $y$  assume  $y \in \bigcup \mathcal{A} \cap R$ 
    then obtain  $I$  where  $I \in \mathcal{A} \ y \in I$  by blast
    then show  $y \in x$  using  $\text{dom}$  by blast
  qed
  then show  $?s \subseteq x$  using  $x\text{id}$  by (intro ideal-generated-minimal) auto
  qed
  then show  $\exists s. \text{ideal-order.is-sup } s \ \mathcal{A} ..$ 
next
  fix  $\mathcal{A} :: 'a \text{ set set}$  assume  $\mathcal{A}: \mathcal{A} \subseteq \text{ideals}$ 
  — Infimum: the intersection of  $\mathcal{A}$  together with  $R$ .
  let  $?i = \bigcap (\text{insert } R \ \mathcal{A})$ 
  have  $\text{all-id}: \bigwedge I. I \in \text{insert } R \ \mathcal{A} \implies \text{Ideal } I \ R \ (+) \ (\cdot) \ 0 \ 1$ 
    using  $\mathcal{A}$  whole-ideal by auto
  have  $i\text{-id}: \text{Ideal } ?i \ R \ (+) \ (\cdot) \ 0 \ 1$  by (rule ideal-Inter) (use all-id in auto)
  have  $\text{ideal-order.is-inf } ?i \ \mathcal{A}$ 
  proof (rule ideal-order.is-infI)
    show  $\text{ideal-order.lower-bound } ?i \ \mathcal{A}$ 
    proof (rule ideal-order.lower-boundI)
      show  $?i \in \text{ideals}$  using  $i\text{-id}$  by simp
      fix  $I$  assume  $I \in \mathcal{A}$  then show  $?i \subseteq I$  by blast
    qed
  next
  fix  $x$  assume  $\text{ideal-order.lower-bound } x \ \mathcal{A}$ 
  then have  $x\text{id}: \text{Ideal } x \ R \ (+) \ (\cdot) \ 0 \ 1$  and  $\text{dom}: \bigwedge I. I \in \mathcal{A} \implies x \subseteq I$ 
    by (auto dest: ideal-order.lower-boundD simp: ideal-order.lower-bound-def)
  interpret  $x: \text{Ideal } x \ R \ (+) \ (\cdot) \ 0 \ 1$  by (rule xid)
  show  $x \subseteq ?i$  using  $\text{dom } x \text{.additive.subset}$  by blast

```

```

qed
then show  $\exists i. \text{ideal-order.is-inf } i \ \mathcal{A} \ ..$ 
qed

end

end

```

62 Extension and contraction of ideals along a ring homomorphism

```

theory Ideal-Extension
  imports Ideal-Lattice
begin

```

Along a homomorphism η of commutative rings $R \rightarrow R'$ there are two natural operations on ideals:

- *contraction* $J \mapsto \eta^{-1} J \cap R$, the preimage of an ideal of the target, which is always an ideal of the source;
- *extension* $I \mapsto \langle \eta^{-1} I \rangle$, the ideal of the target *generated by* the image of an ideal of the source. Generation is needed: the direct image $\eta^{-1} I$ need not be an ideal of R' — it is closed under addition and under multiplication by elements of $\eta^{-1} R$, but not in general by arbitrary elements of R' . (For the inclusion of the integers in the rationals, the image of the even integers is not an ideal of the rationals.)

These form an *adjoint pair* between the ideal lattices, ordered by inclusion: $\langle \eta^{-1} I \rangle \subseteq J \iff I \subseteq \eta^{-1} J \cap R$. So this is a genuine instance of the *Galois-Connection* locale of *Order-Theory*, and all of its consequences — the unit and counit inequalities, monotonicity of both adjoints, the semi-inverse identities, and idempotence of the induced closure and kernel operators — come for free.

The two rings are commutative because *ideal-generated* and *ideals* are defined in the *commutative-ring* context. As with the module homomorphism locale, the target ring's operations are written with primes and the source's unprimed; the two rings are *separate* here (unlike the shared scalar ring of a module homomorphism), so no parameter is shared.

```

locale comm-ring-homomorphism = ring-homomorphism +
  source: commutative-ring R (+) (·) 0 1 +
  target: commutative-ring R' (+') (·') 0' 1'
begin

```

62.1 Contraction: the preimage of an ideal

The contraction of a subset of the target: its preimage, cut down to the source carrier.

definition *contract* :: 'b set $\Rightarrow$ 'a set
where *contract* $J = \eta - ' J \cap R$

lemma *contract-subset*: *contract* $J \subseteq R$
by (*auto simp: contract-def*)

lemma *contract-memI*: $\llbracket a \in R; \eta a \in J \rrbracket \Longrightarrow a \in \text{contract } J$
by (*simp add: contract-def*)

lemma *contract-memD*: $a \in \text{contract } J \Longrightarrow a \in R \wedge \eta a \in J$
by (*simp add: contract-def*)

lemma *contract-mono*: $J \subseteq J' \Longrightarrow \text{contract } J \subseteq \text{contract } J'$
by (*auto simp: contract-def*)

The preimage of an ideal is an ideal: it contains the zero because η preserves it, is closed under addition and negation because η is additive, and absorbs multiplication because η is multiplicative and J absorbs multiplication in the target.

theorem *contract-ideal*:

assumes J : *Ideal* $J R' (+') (\cdot') \mathbf{0}' \mathbf{1}'$
shows *Ideal* (*contract* J) $R (+) (\cdot) \mathbf{0} \mathbf{1}$

proof –

interpret J : *Ideal* $J R' (+') (\cdot') \mathbf{0}' \mathbf{1}'$ **by** (*rule J*)

— η carries the additive inverse to the additive inverse. This is *Monoid-homomorphism.invertible-commutes-with-inverse* for the additive homomorphism; its *invertible* hypothesis is discharged automatically because the additive monoid of a ring is a group.

have *hom-neg*: $\eta (\text{source.additive.inverse } a) = \text{target.additive.inverse } (\eta a)$ **if** a : $a \in R$ **for** a

using a **by** (*simp add: additive.invertible-commutes-with-inverse*)

— The preimage is an additive subgroup ...

have *Subgroup* (*contract* J) $R (+) \mathbf{0}$

proof (*rule source.additive.subgroupI*)

show *contract* $J \subseteq R$ **by** (*rule contract-subset*)

show $\mathbf{0} \in \text{contract } J$

by (*rule contract-memI*) (*simp-all add: additive.commutes-with-unit J.additive.sub-unit-closed*)

show $\bigwedge g h. \llbracket g \in \text{contract } J; h \in \text{contract } J \rrbracket \Longrightarrow g + h \in \text{contract } J$

proof –

fix $g h$ **assume** $g \in \text{contract } J$ **and** $h \in \text{contract } J$

then have gR : $g \in R$ **and** hR : $h \in R$ **and** hg : $\eta g \in J$ **and** hh : $\eta h \in J$

by (*auto dest: contract-memD*)

have $\eta (g + h) = \eta g + ' \eta h$

using $gR hR$ **by** (*simp add: additive.commutes-with-composition*)

then show $g + h \in \text{contract } J$

using $gR hR hg hh$ **by** (*auto intro!: contract-memI J.additive.sub-composition-closed*)

```

qed
show  $\bigwedge g. g \in \text{contract } J \implies \text{source.additive.invertible } g$ 
  using contract-subset by auto
show  $\bigwedge g. g \in \text{contract } J \implies \text{source.additive.inverse } g \in \text{contract } J$ 
proof -
  fix g assume g  $\in$  contract J
  then have gR:  $g \in R$  and hg:  $\eta \ g \in J$  by (auto dest: contract-memD)
  have  $\eta (\text{source.additive.inverse } g) = \text{target.additive.inverse } (\eta \ g)$ 
    using gR by (rule hom-neg)
  moreover have  $\text{target.additive.inverse } (\eta \ g) \in J$ 
  using hg J.additive.sub.invertible by (simp add: J.additive.submonoid-inverse-closed)
  ultimately show  $\text{source.additive.inverse } g \in \text{contract } J$ 
    using gR by (auto intro!: contract-memI)
qed
qed
then interpret S: Subgroup contract J R (+) 0 .
— ... and it absorbs multiplication from either side.
show ?thesis
proof unfold-locales
  fix a b assume a:  $a \in R$  and b:  $b \in \text{contract } J$ 
  then have bR:  $b \in R$  and hb:  $\eta \ b \in J$  by (auto dest: contract-memD)
  have  $\eta (a \cdot b) = \eta \ a \cdot' \eta \ b$ 
    using a bR by (simp add: multiplicative.commutes-with-composition)
  also have ...  $\in J$  using a hb by (blast intro: J.Ideal(1))
  finally show  $a \cdot b \in \text{contract } J$  using a bR by (blast intro: contract-memI)
next
  fix a b assume a:  $a \in R$  and b:  $b \in \text{contract } J$ 
  then have bR:  $b \in R$  and hb:  $\eta \ b \in J$  by (auto dest: contract-memD)
  have  $\eta (b \cdot a) = \eta \ b \cdot' \eta \ a$ 
    using a bR by (simp add: multiplicative.commutes-with-composition)
  also have ...  $\in J$  using a hb by (blast intro: J.Ideal(2))
  finally show  $b \cdot a \in \text{contract } J$  using a bR by (blast intro: contract-memI)
qed
qed

```

The contraction of the target's zero ideal is the kernel; contracting the whole target gives the whole source.

```

lemma contract-whole:  $\text{contract } R' = R$ 
  using map-closed by (auto simp: contract-def)

```

62.2 Extension: the ideal generated by the image

The extension of a subset of the source: the ideal of the target generated by its image.

```

definition extend :: 'a set  $\Rightarrow$  'b set
  where extend I = target.ideal-generated ( $\eta \ ` I$ )

```

```

lemma extend-ideal:  $\text{Ideal } (\text{extend } I) \ R' \ (+) \ (\cdot) \ 0' \ 1'$ 
  unfolding extend-def by (rule target.ideal-generated-is-ideal)

```

lemma *extend-subset*: $\text{extend } I \subseteq R'$
unfolding *extend-def* **by** (*rule target.ideal-generated-subset-carrier*)

The image lands inside the extension. Note the intersection with R' in *target.ideal-generated-incl* is harmless: the image of a subset of R already lies in R' .

lemma *extend-incl*:
assumes $I: I \subseteq R$ **shows** $\eta \text{ ' } I \subseteq \text{extend } I$
proof –
have $\eta \text{ ' } I \subseteq R'$ **using** *I map-closed* **by** *blast*
then have $\eta \text{ ' } I \cap R' = \eta \text{ ' } I$ **by** *blast*
then show *?thesis*
unfolding *extend-def* **using** *target.ideal-generated-incl*[*of* $\eta \text{ ' } I$] **by** *simp*
qed

Extension is minimal: any target ideal containing the image contains the extension.

lemma *extend-minimal*:
assumes $J: \text{Ideal } J R' (+') (\cdot) \mathbf{0}' \mathbf{1}'$ **and** $\text{sub}: \eta \text{ ' } I \subseteq J$
shows $\text{extend } I \subseteq J$
unfolding *extend-def*
proof (*rule target.ideal-generated-minimal*[*OF* J])
show $\eta \text{ ' } I \cap R' \subseteq J$ **using** *sub* **by** *blast*
qed

lemma *extend-mono*:
assumes $II': I \subseteq I'$ **and** $I': I' \subseteq R$ **shows** $\text{extend } I \subseteq \text{extend } I'$
proof (*rule extend-minimal*[*OF* *extend-ideal*])
show $\eta \text{ ' } I \subseteq \text{extend } I'$ **using** II' *extend-incl*[*OF* I'] **by** *blast*
qed

62.3 The adjunction

Extension is left adjoint to contraction. Both directions are immediate from minimality of the generated ideal and the definition of preimage — which is exactly why extension had to be defined by generation: without it the left-to-right direction would fail.

theorem *extend-contract-galois*:
assumes $I: I \subseteq R$ **and** $J: \text{Ideal } J R' (+') (\cdot) \mathbf{0}' \mathbf{1}'$
shows $\text{extend } I \subseteq J \longleftrightarrow I \subseteq \text{contract } J$
proof
assume $\text{extend } I \subseteq J$
with *extend-incl*[*OF* I] **have** $\eta \text{ ' } I \subseteq J$ **by** *blast*
then show $I \subseteq \text{contract } J$ **using** I **by** (*auto intro: contract-memI*)
next
assume $I \subseteq \text{contract } J$
then have $\eta \text{ ' } I \subseteq J$ **by** (*auto dest: contract-memD*)

then show $extend\ I \subseteq J$ **by** (rule *extend-minimal*[OF J])
qed

The Galois connection. On the ideal lattices of the two rings, ordered by inclusion, extension and contraction form an adjoint pair. Interpreting *Galois-Connection* makes its whole development available: *unit* ($I \subseteq contract\ (extend\ I)$), *counit* ($extend\ (contract\ J) \subseteq J$), isotonicity of both maps, the semi-inverse laws $extend\ (contract\ (extend\ I)) = extend\ I$ and $contract\ (extend\ (contract\ J)) = contract\ J$, and idempotence of the closure $contract \circ extend$ and kernel $extend \circ contract$.

theorem *ideal-galois-connection*:

Galois-Connection $source.ideals\ (\subseteq)\ target.ideals\ (\subseteq)\ extend\ contract$

proof (rule *Galois-Connection.intro*)

— Inclusion is a partial order on any set of sets; *Ideal-Lattice*’s own interpretation is inside its *commutative-ring* context and so is not citable by name from here.

show *Partial-Order* $source.ideals\ (\subseteq)$ **by** *unfold-locales auto*

show *Partial-Order* $target.ideals\ (\subseteq)$ **by** *unfold-locales auto*

show *Galois-Connection-axioms* $source.ideals\ (\subseteq)\ target.ideals\ (\subseteq)\ extend\ contract$

proof

show $extend \in source.ideals \rightarrow target.ideals$

using *extend-ideal* **by** (*auto simp: target.ideals-iff*)

show $contract \in target.ideals \rightarrow source.ideals$

using *contract-ideal* **by** (*auto simp: source.ideals-iff target.ideals-iff*)

fix $I\ J$ **assume** $I \in source.ideals$ **and** $J \in target.ideals$

then have $I: Ideal\ I\ R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}$ **and** $J: Ideal\ J\ R'\ (+')\ (\cdot')\ \mathbf{0}'\ \mathbf{1}'$

by (*simp-all add: source.ideals-iff target.ideals-iff*)

interpret $I: Ideal\ I\ R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}$ **by** (rule I)

have $IR: I \subseteq R$ **by** (rule $I.additive.subset$)

show $extend\ I \subseteq J \longleftrightarrow I \subseteq contract\ J$ **by** (rule *extend-contract-galois*[OF $IR\ J$])

qed

qed

The consequences of the adjunction, specialised to ideals. These are obtained by interpreting *ideal-galois-connection* locally rather than by a **sublocale** declaration: a *sublocale galois-connection* inside this locale sends the interpretation machinery into a loop, because *comm-ring-homomorphism* would then be a source of *galois-connection* facts about its own ideal lattices while still being elaborated. The theorem form has no such problem, and consumers can **interpret** it wherever they need the full development.

context

begin

interpretation *GC*: *Galois-Connection*

$source.ideals\ (\subseteq)\ target.ideals\ (\subseteq)\ extend\ contract$

by (rule *ideal-galois-connection*)

Unit: an ideal is contained in the contraction of its extension.

corollary *contract-extend-incl*:

Ideal $I \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \implies I \subseteq \text{contract} \ (\text{extend} \ I)$
using *GC.unit* **by** *simp*

Counit: the extension of a contraction is contained in the original ideal. The inclusion is in general strict — for $\mathbb{Z} \subseteq \mathbb{Q}$ every nonzero ideal of $\mathbb{Q}$ contracts to a nonzero ideal of $\mathbb{Z}$ whose extension is all of $\mathbb{Q}$.

corollary *extend-contract-subset*:

Ideal $J \ R' \ (+') \ (\cdot') \ \mathbf{0}' \ \mathbf{1}' \implies \text{extend} \ (\text{contract} \ J) \subseteq J$
using *GC.counit* **by** *simp*

Both adjoints are isotone on ideals.

corollary *extend-mono-ideal*:

$\llbracket \text{Ideal } I \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}; \text{Ideal } I' \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}; I \subseteq I' \rrbracket \implies \text{extend} \ I \subseteq \text{extend} \ I'$
using *isotone-le[OF GC.lower-isotone]* **by** *simp*

corollary *contract-mono-ideal*:

$\llbracket \text{Ideal } J \ R' \ (+') \ (\cdot') \ \mathbf{0}' \ \mathbf{1}'; \text{Ideal } J' \ R' \ (+') \ (\cdot') \ \mathbf{0}' \ \mathbf{1}'; J \subseteq J' \rrbracket$
 $\implies \text{contract} \ J \subseteq \text{contract} \ J'$
using *isotone-le[OF GC.upper-isotone]* **by** *simp*

The *semi-inverse* laws: extension and contraction are mutually inverse up to one round trip.

corollary *extend-contract-extend*:

Ideal $I \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \implies \text{extend} \ (\text{contract} \ (\text{extend} \ I)) = \text{extend} \ I$
using *GC.lower-semi-inverse* **by** *simp*

corollary *contract-extend-contract*:

Ideal $J \ R' \ (+') \ (\cdot') \ \mathbf{0}' \ \mathbf{1}' \implies \text{contract} \ (\text{extend} \ (\text{contract} \ J)) = \text{contract} \ J$
using *GC.upper-semi-inverse* **by** *simp*

Hence $\text{contract} \circ \text{extend}$ is a closure operator on the ideals of the source, and $\text{extend} \circ \text{contract}$ a kernel operator on those of the target.

corollary *closure-idempotent*:

Ideal $I \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \implies \text{contract} \ (\text{extend} \ (\text{contract} \ (\text{extend} \ I))) = \text{contract} \ (\text{extend} \ I)$
using *GC.closure-idempotent* **by** *simp*

corollary *kernel-idempotent*:

Ideal $J \ R' \ (+') \ (\cdot') \ \mathbf{0}' \ \mathbf{1}' \implies \text{extend} \ (\text{contract} \ (\text{extend} \ (\text{contract} \ J))) = \text{extend} \ (\text{contract} \ J)$
using *GC.kernel-idempotent* **by** *simp*

end

end

end

63 The direct product of an indexed family of rings

```

theory Ring-Family-Product
  imports Ring-Theory
begin

```

The external direct product of a family of rings $R\ i\ (i \in I)$: the carrier is the extensional dependent function space $Pi_E\ I\ R$, with pointwise addition, multiplication, zero and one. This mirrors the indexed group product but for rings; its commutative and field-of-components refinements follow. (A product of fields is *not* a field in general — it has zero divisors as soon as I has more than one element — so only the commutative-ring refinement is offered.)

63.1 The product locale

```

locale ring-family =
  fixes  $I :: 'i\ set$ 
    and  $R :: 'i \Rightarrow 'a\ set$ 
    and  $add :: 'i \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ 
    and  $mult :: 'i \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ 
    and  $zero :: 'i \Rightarrow 'a$ 
    and  $one :: 'i \Rightarrow 'a$ 
  assumes  $ring: \bigwedge i. i \in I \Longrightarrow Ring\ (R\ i)\ (add\ i)\ (mult\ i)\ (zero\ i)\ (one\ i)$ 
begin

lemma comp-ring:  $i \in I \Longrightarrow Ring\ (R\ i)\ (add\ i)\ (mult\ i)\ (zero\ i)\ (one\ i)$ 
  by (rule ring)

```

63.2 Carrier and operations

```

definition  $Pcarrier :: ('i \Rightarrow 'a)\ set$ 
  where  $Pcarrier = (\Pi_E\ i \in I. R\ i)$ 

definition  $Padd :: ('i \Rightarrow 'a) \Rightarrow ('i \Rightarrow 'a) \Rightarrow ('i \Rightarrow 'a)\ (\text{infixl } \langle \oplus_\Pi \rangle\ 65)$ 
  where  $x \oplus_\Pi y = (\lambda i \in I. add\ i\ (x\ i)\ (y\ i))$ 

definition  $Pmult :: ('i \Rightarrow 'a) \Rightarrow ('i \Rightarrow 'a) \Rightarrow ('i \Rightarrow 'a)\ (\text{infixl } \langle \odot_\Pi \rangle\ 70)$ 
  where  $x \odot_\Pi y = (\lambda i \in I. mult\ i\ (x\ i)\ (y\ i))$ 

definition  $Pzero :: 'i \Rightarrow 'a\ (\langle \mathbf{0}_\Pi \rangle)$ 
  where  $\mathbf{0}_\Pi = (\lambda i \in I. zero\ i)$ 

definition  $Pone :: 'i \Rightarrow 'a\ (\langle \mathbf{1}_\Pi \rangle)$ 
  where  $\mathbf{1}_\Pi = (\lambda i \in I. one\ i)$ 

lemma Pcarrier-memI:
   $\llbracket \bigwedge i. i \in I \Longrightarrow x\ i \in R\ i; x \in \text{extensional } I \rrbracket \Longrightarrow x \in Pcarrier$ 
  by (auto simp: Pcarrier-def PiE-iff)

```

lemma *Pcarrier-component*: $\llbracket x \in Pcarrier; i \in I \rrbracket \implies x\ i \in R\ i$
by (*auto simp: Pcarrier-def PiE-iff*)

lemma *Pcarrier-extensional*: $x \in Pcarrier \implies x \in extensional\ I$
by (*auto simp: Pcarrier-def PiE-iff*)

lemma *Pcarrier-eqI*:
 $\llbracket x \in Pcarrier; y \in Pcarrier; \bigwedge i. i \in I \implies x\ i = y\ i \rrbracket \implies x = y$
by (*metis Pcarrier-extensional extensionalityI*)

lemma *Padd-apply*: $i \in I \implies (x \oplus_{\Pi} y)\ i = add\ i\ (x\ i)\ (y\ i)$ **by** (*simp add: Padd-def*)
lemma *Pmult-apply*: $i \in I \implies (x \odot_{\Pi} y)\ i = mult\ i\ (x\ i)\ (y\ i)$ **by** (*simp add: Pmult-def*)

lemma *Pzero-apply*: $i \in I \implies \mathbf{0}_{\Pi}\ i = zero\ i$ **by** (*simp add: Pzero-def*)
lemma *Pone-apply*: $i \in I \implies \mathbf{1}_{\Pi}\ i = one\ i$ **by** (*simp add: Pone-def*)

lemma *Padd-closed* [*intro, simp*]:
assumes $x \in Pcarrier\ y \in Pcarrier$ **shows** $x \oplus_{\Pi} y \in Pcarrier$
proof (*rule Pcarrier-memI*)
fix i **assume** $i: i \in I$
interpret Ri : *Ring* $R\ i\ add\ i\ mult\ i\ zero\ i\ one\ i$ **using** i **by** (*rule ring*)
show $(x \oplus_{\Pi} y)\ i \in R\ i$ **using** *assms* i **by** (*simp add: Padd-apply Pcarrier-component*)
qed (*auto simp: Padd-def*)

lemma *Pmult-closed* [*intro, simp*]:
assumes $x \in Pcarrier\ y \in Pcarrier$ **shows** $x \odot_{\Pi} y \in Pcarrier$
proof (*rule Pcarrier-memI*)
fix i **assume** $i: i \in I$
interpret Ri : *Ring* $R\ i\ add\ i\ mult\ i\ zero\ i\ one\ i$ **using** i **by** (*rule ring*)
show $(x \odot_{\Pi} y)\ i \in R\ i$ **using** *assms* i **by** (*simp add: Pmult-apply Pcarrier-component*)
qed (*auto simp: Pmult-def*)

lemma *Pzero-closed* [*intro, simp*]: $\mathbf{0}_{\Pi} \in Pcarrier$
proof (*rule Pcarrier-memI*)
fix i **assume** $i: i \in I$
interpret Ri : *Ring* $R\ i\ add\ i\ mult\ i\ zero\ i\ one\ i$ **using** i **by** (*rule ring*)
show $\mathbf{0}_{\Pi}\ i \in R\ i$ **using** i **by** (*simp add: Pzero-apply*)
qed (*auto simp: Pzero-def*)

lemma *Pone-closed* [*intro, simp*]: $\mathbf{1}_{\Pi} \in Pcarrier$
proof (*rule Pcarrier-memI*)
fix i **assume** $i: i \in I$
interpret Ri : *Ring* $R\ i\ add\ i\ mult\ i\ zero\ i\ one\ i$ **using** i **by** (*rule ring*)
show $\mathbf{1}_{\Pi}\ i \in R\ i$ **using** i **by** (*simp add: Pone-apply*)
qed (*auto simp: Pone-def*)

63.3 The additive abelian group

lemma *additive-group*: *Group* $Pcarrier\ (\oplus_{\Pi})\ \mathbf{0}_{\Pi}$

```

proof (intro GroupI Pzero-closed Padd-closed)
  fix x y z assume xyz: x ∈ Pcarrier y ∈ Pcarrier z ∈ Pcarrier
  show (x ⊕Π y) ⊕Π z = x ⊕Π (y ⊕Π z)
  proof (rule Pcarrier-eqI)
    fix i assume i: i ∈ I
    interpret Ri: Ring R i add i mult i zero i one i using i by (rule ring)
    show ((x ⊕Π y) ⊕Π z) i = (x ⊕Π (y ⊕Π z)) i
    using xyz i by (simp add: Padd-apply Pcarrier-component Ri.additive.associative)
  qed (use xyz in auto)
next
  fix x assume x: x ∈ Pcarrier
  show 0Π ⊕Π x = x
  proof (rule Pcarrier-eqI)
    fix i assume i: i ∈ I
    interpret Ri: Ring R i add i mult i zero i one i using i by (rule ring)
    show (0Π ⊕Π x) i = x i
    by (simp add: Padd-apply Pcarrier-component Pzero-def i x)
  qed (use x in auto)
  show x ⊕Π 0Π = x
  proof (rule Pcarrier-eqI)
    fix i assume i: i ∈ I
    interpret Ri: Ring R i add i mult i zero i one i using i by (rule ring)
    show (x ⊕Π 0Π) i = x i
    using x i by (simp add: Padd-apply Pzero-apply Pcarrier-component)
  qed (use x in auto)
next
  fix x assume x: x ∈ Pcarrier
  note [[unify-search-bound=10]] — for the two following meson calls
  have ∃ v. v i ∈ R i ∧ add i (x i) (v i) = zero i ∧ add i (v i) (x i) = zero i if i: i ∈ I
for i
  proof —
    interpret Ri: Ring R i add i mult i zero i one i using i by (rule ring)
    show ?thesis
    by (meson Pcarrier-component Ri.additive.invertible Ri.additive.invertibleE i x)
  qed
  then obtain v where v: ∧ i. i ∈ I ⇒ v i ∈ R i
    ∧ add i (x i) (v i) = zero i ∧ add i (v i) (x i) = zero i by meson
  define y where y = (λ i ∈ I. v i)
  have x ⊕Π y = 0Π ∧ y ⊕Π x = 0Π
    using Padd-def Pzero-def ⟨y ≡ restrict v I⟩ v by auto
  then show ∃ y ∈ Pcarrier. x ⊕Π y = 0Π ∧ y ⊕Π x = 0Π
    using Pcarrier-def v y-def by auto
  qed

lemma additive-abelian-group: Abelian-Group Pcarrier (⊕Π) 0Π
proof (intro Abelian-Group.intro additive-group)
  show commutative-monoid Pcarrier (⊕Π) 0Π
  proof (rule commutative-monoid.intro)
    show Monoid Pcarrier (⊕Π) 0Π
    proof

```

```

fix x y z assume xyz: x ∈ Pcarrier y ∈ Pcarrier z ∈ Pcarrier
show (x ⊕Π y) ⊕Π z = x ⊕Π (y ⊕Π z)
  using Group-def Monoid.associative additive-group xyz by fastforce
next
fix x assume x: x ∈ Pcarrier
show 0Π ⊕Π x = x
  by (meson Group-def Monoid.left-unit additive-group x)
show x ⊕Π 0Π = x
  by (meson Group-def Monoid.right-unit additive-group x)
qed auto
show commutative-monoid-axioms Pcarrier (⊕Π)
proof
fix x y assume xy: x ∈ Pcarrier y ∈ Pcarrier
show x ⊕Π y = y ⊕Π x
proof (rule Pcarrier-eqI)
fix i assume i: i ∈ I
interpret Ri: Ring R i add i mult i zero i one i using i by (rule ring)
show (x ⊕Π y) i = (y ⊕Π x) i
  using xy i by (simp add: Padd-apply Pcarrier-component Ri.additive.commutative)
qed (use xy in auto)
qed
qed
qed

```

63.4 The multiplicative monoid

```

lemma multiplicative-monoid: Monoid Pcarrier (⊙Π) 1Π
proof (unfold-locales)
  show ∧x y. x ∈ Pcarrier ⇒ y ∈ Pcarrier ⇒ x ⊙Π y ∈ Pcarrier by (rule
Pmult-closed)
  show 1Π ∈ Pcarrier by (rule Pone-closed)
next
fix x y z assume xyz: x ∈ Pcarrier y ∈ Pcarrier z ∈ Pcarrier
show (x ⊙Π y) ⊙Π z = x ⊙Π (y ⊙Π z)
proof (rule Pcarrier-eqI)
fix i assume i: i ∈ I
interpret Ri: Ring R i add i mult i zero i one i using i by (rule ring)
show ((x ⊙Π y) ⊙Π z) i = (x ⊙Π (y ⊙Π z)) i
  using xyz i by (simp add: Pmult-apply Pcarrier-component Ri.multiplicative.associative)
qed (use xyz in auto)
next
fix x assume x: x ∈ Pcarrier
show 1Π ⊙Π x = x
proof (rule Pcarrier-eqI)
fix i assume i: i ∈ I
interpret Ri: Ring R i add i mult i zero i one i using i by (rule ring)
show (1Π ⊙Π x) i = x i
  using x i by (simp add: Pmult-apply Pone-apply Pcarrier-component)
qed (use x in auto)
next

```

```

fix  $x$  assume  $x: x \in Pcarrier$ 
show  $x \odot_{\Pi} 1_{\Pi} = x$ 
proof (rule Pcarrier-eqI)
  fix  $i$  assume  $i: i \in I$ 
  interpret  $Ri$ : Ring  $R$   $i$  add  $i$  mult  $i$  zero  $i$  one  $i$  using  $i$  by (rule ring)
  show  $(x \odot_{\Pi} 1_{\Pi}) i = x i$ 
    using  $x i$  by (simp add: Pmult-apply Pone-apply Pcarrier-component)
  qed (use  $x$  in auto)
qed

```

63.5 The product is a ring

```

theorem Ring-Pcarrier: Ring  $Pcarrier$   $(\oplus_{\Pi})$   $(\odot_{\Pi})$   $0_{\Pi}$   $1_{\Pi}$ 
proof (rule Ring.intro)
  show Abelian-Group  $Pcarrier$   $(\oplus_{\Pi})$   $0_{\Pi}$  by (rule additive-abelian-group)
  show Monoid  $Pcarrier$   $(\odot_{\Pi})$   $1_{\Pi}$  by (rule multiplicative-monoid)
  show Ring-axioms  $Pcarrier$   $(\oplus_{\Pi})$   $(\odot_{\Pi})$ 
proof
  fix  $x y z$  assume  $xyz: x \in Pcarrier y \in Pcarrier z \in Pcarrier$ 
  show  $x \odot_{\Pi} (y \oplus_{\Pi} z) = x \odot_{\Pi} y \oplus_{\Pi} (x \odot_{\Pi} z)$ 
proof (rule Pcarrier-eqI)
  fix  $i$  assume  $i: i \in I$ 
  interpret  $Ri$ : Ring  $R$   $i$  add  $i$  mult  $i$  zero  $i$  one  $i$  using  $i$  by (rule ring)
  show  $(x \odot_{\Pi} (y \oplus_{\Pi} z)) i = (x \odot_{\Pi} y \oplus_{\Pi} (x \odot_{\Pi} z)) i$ 
    using  $xyz i$  by (simp add: Pmult-apply Padd-apply Pcarrier-component
Ri.distributive)
  qed (use  $xyz$  in auto)
  show  $(y \oplus_{\Pi} z) \odot_{\Pi} x = y \odot_{\Pi} x \oplus_{\Pi} (z \odot_{\Pi} x)$ 
proof (rule Pcarrier-eqI)
  fix  $i$  assume  $i: i \in I$ 
  interpret  $Ri$ : Ring  $R$   $i$  add  $i$  mult  $i$  zero  $i$  one  $i$  using  $i$  by (rule ring)
  show  $((y \oplus_{\Pi} z) \odot_{\Pi} x) i = (y \odot_{\Pi} x \oplus_{\Pi} (z \odot_{\Pi} x)) i$ 
    using  $xyz i$  by (simp add: Pmult-apply Padd-apply Pcarrier-component
Ri.distributive)
  qed (use  $xyz$  in auto)
qed
qed

```

```

sublocale product: Ring  $Pcarrier$   $(\oplus_{\Pi})$   $(\odot_{\Pi})$  Pzero Pone
by (rule Ring-Pcarrier)

```

63.6 Projections are ring homomorphisms

```

definition  $Pproj :: 'i \Rightarrow ('i \Rightarrow 'a) \Rightarrow 'a$ 
  where  $Pproj i = restrict (\lambda x. x i) Pcarrier$ 

lemma Pproj-apply:  $x \in Pcarrier \implies Pproj i x = x i$ 
  by (simp add: Pproj-def)

lemma Pproj-hom:

```

```

assumes  $i: i \in I$ 
shows ring-homomorphism ( $Pproj\ i$ )  $Pcarrier\ (\oplus_\Pi)\ (\odot_\Pi)\ \mathbf{0}_\Pi\ \mathbf{1}_\Pi$ 
      ( $R\ i$ ) ( $add\ i$ ) ( $mult\ i$ ) ( $zero\ i$ ) ( $one\ i$ )
proof –
  interpret  $Ri$ : Ring  $R\ i\ add\ i\ mult\ i\ zero\ i\ one\ i$  using  $i$  by (rule ring)
  have graph:  $Pproj\ i \in Pcarrier \rightarrow_E R\ i$ 
    by (simp add: Pcarrier-component Pproj-def i)
  show ?thesis
  proof
    qed (use i in <auto simp: graph Pproj-apply Pzero-apply Pone-apply Padd-apply Pmult-apply>)
  qed
end

```

63.7 The product of a family of commutative rings is commutative

```

locale commutative-ring-family = ring-family +
  assumes comm:  $\bigwedge i. i \in I \implies commutative-ring\ (R\ i)\ (add\ i)\ (mult\ i)\ (zero\ i)\ (one\ i)$ 
begin

theorem commutative-ring-Pcarrier: commutative-ring  $Pcarrier\ (\oplus_\Pi)\ (\odot_\Pi)\ \mathbf{0}_\Pi\ \mathbf{1}_\Pi$ 
proof (rule commutative-ring.intro)
  show Ring  $Pcarrier\ (\oplus_\Pi)\ (\odot_\Pi)\ \mathbf{0}_\Pi\ \mathbf{1}_\Pi$  by (rule Ring-Pcarrier)
  — The multiplicative monoid is commutative.
  show commutative-monoid  $Pcarrier\ (\odot_\Pi)\ \mathbf{1}_\Pi$ 
  proof (rule commutative-monoid.intro)
    show Monoid  $Pcarrier\ (\odot_\Pi)\ \mathbf{1}_\Pi$  by (rule multiplicative-monoid)
    show commutative-monoid-axioms  $Pcarrier\ (\odot_\Pi)$ 
    proof
      fix  $x\ y$  assume  $xy$ :  $x \in Pcarrier\ y \in Pcarrier$ 
      show  $x \odot_\Pi y = y \odot_\Pi x$ 
      proof (rule Pcarrier-eqI)
        fix  $i$  assume  $i: i \in I$ 
        interpret  $Ri$ : commutative-ring  $R\ i\ add\ i\ mult\ i\ zero\ i\ one\ i$  using  $i$  by (rule comm)
        show  $(x \odot_\Pi y)\ i = (y \odot_\Pi x)\ i$ 
        using  $xy\ i$  by (simp add: Pmult-apply Pcarrier-component Ri.multiplicative.commutative)
        qed (use xy in auto)
      qed
    qed
  qed
end
end

```


64 The Chinese Remainder Theorem for commutative rings

```

theory Chinese-Remainder-Rings
imports Ideal-Theory Ring-Family-Product
begin

```

For two comaximal ideals I, J of a commutative ring R (i.e. $I + J = R$), the quotient $R / (I \cap J)$ is isomorphic to the product ring $(R / I) \times (R / J)$. The proof follows the classical route: the canonical map $a \mapsto (I + a, J + a)$ into the product of the two quotient rings is a ring homomorphism, surjective by comaximality, with kernel $I \cap J$; the fundamental theorem of ring homomorphisms then yields the isomorphism.

64.1 Associative-commutative rewriting in an abelian group

The *Abelian-Group* locale exposes only the raw associativity, commutativity and inverse axioms (each with a carrier side condition), so plain *simp* cannot ac-normalise sums. We prove a few confluent cancellation lemmas and a couple of rearrangement lemmas that the Chinese-Remainder arithmetic below relies on.

```

context Abelian-Group
begin

```

Left-nested inverse cancellations (safe as *simp* rules: not permutative).

```

lemma add-assoc-inverse-left [simp]:
   $\llbracket a \in G; b \in G \rrbracket \implies \text{inverse } a \cdot (a \cdot b) = b$ 
by (simp add: associative [symmetric])

```

```

lemma add-assoc-inverse-left' [simp]:
   $\llbracket a \in G; b \in G \rrbracket \implies a \cdot (\text{inverse } a \cdot b) = b$ 
by (simp add: associative [symmetric])

```

Rearrangement: $q \cdot p \cdot \text{local.inverse } (s \cdot q) = p \cdot \text{local.inverse } s$. This is the shape needed to simplify $(x \cdot j + y \cdot i) - (x \cdot i + x \cdot j)$ in the surjectivity argument.

```

lemma add-diff-cancel-mid:
  assumes  $q: q \in G$  and  $p: p \in G$  and  $s: s \in G$ 
  shows  $(q \cdot p) \cdot \text{inverse } (s \cdot q) = p \cdot \text{inverse } s$ 
proof –
  have ninv:  $\text{inverse } (s \cdot q) = \text{inverse } q \cdot \text{inverse } s$ 
    using  $q\ s$  by (simp add: inverse-composition-commute commutative)
  have  $(q \cdot p) \cdot \text{inverse } (s \cdot q) = (p \cdot q) \cdot (\text{inverse } q \cdot \text{inverse } s)$ 
    using  $q\ p$  by (simp add: ninv commutative)
  also have  $\dots = p \cdot (q \cdot (\text{inverse } q \cdot \text{inverse } s))$ 
    using  $q\ p\ s$  by (simp add: associative)
  also have  $\dots = p \cdot \text{inverse } s$  using  $q\ s$  by simp
  finally show ?thesis .

```

qed

end

64.2 The binary direct product of two rings

The external direct product of two rings, carried on pairs, with pointwise operations. This specialises the indexed direct product of *Ring-Family-Product* to two factors, which keeps the carrier type $'a \times 'b$ rather than a dependent function space — convenient for the two quotient rings occurring in the Chinese Remainder Theorem. The notation follows *ring-homomorphism*: primed operations for the second factor.

locale *ring-pair* =

A: Ring R $(+)$ $(\cdot)$ $\mathbf{0}$ $\mathbf{1}$ + *B: Ring* R' $(+')$ $(\cdot')$ $\mathbf{0}'$ $\mathbf{1}'$
for R **and** *addition* (infixl $\langle + \rangle$ 65) **and** *multiplication* (infixl $\langle \cdot \rangle$ 70) **and** *zero* ($\langle \mathbf{0} \rangle$) **and** *unit* ($\langle \mathbf{1} \rangle$)
and R' **and** *addition'* (infixl $\langle +' \rangle$ 65) **and** *multiplication'* (infixl $\langle \cdot' \rangle$ 70) **and** *zero'* ($\langle \mathbf{0}' \rangle$) **and** *unit'* ($\langle \mathbf{1}' \rangle$)
begin

definition *Bcarrier* :: $'a \times 'b$ *set*

where $Bcarrier = R \times R'$

definition *Badd* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow 'a \times 'b$ (infixl $\langle \oplus_{\times} \rangle$ 65)

where $x \oplus_{\times} y = (fst\ x + fst\ y, snd\ x +' snd\ y)$

definition *Bmult* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow 'a \times 'b$ (infixl $\langle \otimes_{\times} \rangle$ 70)

where $x \otimes_{\times} y = (fst\ x \cdot fst\ y, snd\ x \cdot' snd\ y)$

definition *Bzero* :: $'a \times 'b$ ($\langle \mathbf{0}_{\times} \rangle$)

where $\mathbf{0}_{\times} = (\mathbf{0}, \mathbf{0}')$

definition *Bone* :: $'a \times 'b$ ($\langle \mathbf{1}_{\times} \rangle$)

where $\mathbf{1}_{\times} = (\mathbf{1}, \mathbf{1}')$

lemma *Bcarrier-iff* [simp]: $(a, b) \in Bcarrier \longleftrightarrow a \in R \wedge b \in R'$

by (simp add: Bcarrier-def)

lemma *Badd-apply* [simp]: $(a_1, b_1) \oplus_{\times} (a_2, b_2) = (a_1 + a_2, b_1 +' b_2)$

by (simp add: Badd-def)

lemma *Bmult-apply* [simp]: $(a_1, b_1) \otimes_{\times} (a_2, b_2) = (a_1 \cdot a_2, b_1 \cdot' b_2)$

by (simp add: Bmult-def)

theorem *Ring-Bcarrier*: *Ring* $Bcarrier$ $(\oplus_{\times})$ $(\otimes_{\times})$ $\mathbf{0}_{\times}$ $\mathbf{1}_{\times}$

proof (rule *Ring.intro*)

show *Abelian-Group* $Bcarrier$ $(\oplus_{\times})$ $\mathbf{0}_{\times}$

proof (rule *Abelian-Group.intro*)

show *Group* $Bcarrier$ $(\oplus_{\times})$ $\mathbf{0}_{\times}$

```

proof (rule GroupI)
  show  $\bigwedge x y. x \in Bcarrier \implies y \in Bcarrier \implies x \oplus_{\times} y \in Bcarrier$ 
    by (auto simp: Badd-def Bcarrier-def)
  show  $0_{\times} \in Bcarrier$  by (simp add: Bzero-def Bcarrier-def)
next
  fix  $x y z$  assume  $x \in Bcarrier \ y \in Bcarrier \ z \in Bcarrier$ 
  then show  $(x \oplus_{\times} y) \oplus_{\times} z = x \oplus_{\times} (y \oplus_{\times} z)$ 
    by (auto simp: Badd-def Bcarrier-def A.additive.associative B.additive.associative)
next
  fix  $x$  assume  $x \in Bcarrier$  then show  $0_{\times} \oplus_{\times} x = x$ 
    by (cases  $x$ ) (simp add: Bzero-def Bcarrier-def)
next
  fix  $x$  assume  $x \in Bcarrier$  then show  $x \oplus_{\times} 0_{\times} = x$ 
    by (cases  $x$ ) (simp add: Bzero-def Bcarrier-def)
next
  fix  $x$  assume  $x: x \in Bcarrier$ 
  then have  $x1: fst\ x \in R$  and  $x2: snd\ x \in R'$  by (auto simp: Bcarrier-def)
  let  $?v = (A.additive.inverse\ (fst\ x), B.additive.inverse\ (snd\ x))$ 
  have  $?v \in Bcarrier$  using  $x1\ x2$  by (simp add: Bcarrier-def)
  moreover have  $x \oplus_{\times} ?v = 0_{\times} \wedge ?v \oplus_{\times} x = 0_{\times}$ 
    using  $x1\ x2$  by (cases  $x$ ) (simp add: Badd-def Bzero-def)
  ultimately show  $\exists y \in Bcarrier. x \oplus_{\times} y = 0_{\times} \wedge y \oplus_{\times} x = 0_{\times}$  by blast
qed
show commutative-monoid Bcarrier  $(\oplus_{\times})\ 0_{\times}$ 
proof (rule commutative-monoid.intro)
  show Monoid Bcarrier  $(\oplus_{\times})\ 0_{\times}$ 
  proof (unfold-locale)
    show  $\bigwedge x y. x \in Bcarrier \implies y \in Bcarrier \implies x \oplus_{\times} y \in Bcarrier$ 
      by (auto simp: Badd-def Bcarrier-def)
    show  $0_{\times} \in Bcarrier$  by (simp add: Bzero-def Bcarrier-def)
  next
    fix  $x y z$  assume  $x \in Bcarrier \ y \in Bcarrier \ z \in Bcarrier$ 
    then show  $(x \oplus_{\times} y) \oplus_{\times} z = x \oplus_{\times} (y \oplus_{\times} z)$ 
      by (auto simp: Badd-def Bcarrier-def A.additive.associative B.additive.associative)
  next
    fix  $x$  assume  $x \in Bcarrier$  then show  $0_{\times} \oplus_{\times} x = x$ 
      by (cases  $x$ ) (simp add: Bzero-def Bcarrier-def)
  next
    fix  $x$  assume  $x \in Bcarrier$  then show  $x \oplus_{\times} 0_{\times} = x$ 
      by (cases  $x$ ) (simp add: Bzero-def Bcarrier-def)
  qed
show commutative-monoid-axioms Bcarrier  $(\oplus_{\times})$ 
proof (unfold-locale)
  fix  $x y$  assume  $x \in Bcarrier \ y \in Bcarrier$ 
  then show  $x \oplus_{\times} y = y \oplus_{\times} x$ 
    by (auto simp: Badd-def Bcarrier-def A.additive.commutative B.additive.commutative)
qed
qed
show Monoid Bcarrier  $(\otimes_{\times})\ 1_{\times}$ 

```

```

proof (unfold-locales)
  show  $\bigwedge x y. x \in Bcarrier \implies y \in Bcarrier \implies x \otimes_{\times} y \in Bcarrier$ 
    by (auto simp: Bmult-def Bcarrier-def)
  show  $1_{\times} \in Bcarrier$  by (simp add: Bone-def Bcarrier-def)
next
  fix  $x y z$  assume  $x \in Bcarrier y \in Bcarrier z \in Bcarrier$ 
  then show  $(x \otimes_{\times} y) \otimes_{\times} z = x \otimes_{\times} (y \otimes_{\times} z)$ 
    by (auto simp: Bmult-def Bcarrier-def A.multiplicative.associative B.multiplicative.associative)
next
  fix  $x$  assume  $x \in Bcarrier$  then show  $1_{\times} \otimes_{\times} x = x$ 
    by (cases x) (simp add: Bone-def Bcarrier-def)
next
  fix  $x$  assume  $x \in Bcarrier$  then show  $x \otimes_{\times} 1_{\times} = x$ 
    by (cases x) (simp add: Bone-def Bcarrier-def)
qed
show Ring-axioms Bcarrier  $(\oplus_{\times}) (\otimes_{\times})$ 
proof
  fix  $x y z$  assume  $x \in Bcarrier y \in Bcarrier z \in Bcarrier$ 
  then show  $x \otimes_{\times} (y \oplus_{\times} z) = x \otimes_{\times} y \oplus_{\times} (x \otimes_{\times} z)$ 
    by (auto simp: Bmult-def Badd-def Bcarrier-def A.distributive B.distributive)
next
  fix  $x y z$  assume  $x \in Bcarrier y \in Bcarrier z \in Bcarrier$ 
  then show  $(y \oplus_{\times} z) \otimes_{\times} x = y \otimes_{\times} x \oplus_{\times} (z \otimes_{\times} x)$ 
    by (auto simp: Bmult-def Badd-def Bcarrier-def A.distributive B.distributive)
qed
qed

sublocale product: Ring Bcarrier  $(\oplus_{\times}) (\otimes_{\times}) Bzero Bone$ 
  by (rule Ring-Bcarrier)

end

```

64.3 Sum of two ideals

The sum $I + J$ of two ideals: all sums $i + j$ with $i \in I, j \in J$. We work in a commutative ring throughout (the setting of the Chinese Remainder Theorem).

context ideal-in-comm-ring
begin

definition ideal-sum :: 'a set $\Rightarrow$ 'a set
where ideal-sum $J = \{i + j \mid i j. i \in I \wedge j \in J\}$

lemma ideal-sum-memI: $\llbracket i \in I; j \in J \rrbracket \implies i + j \in \text{ideal-sum } J$
unfolding ideal-sum-def **by** blast

lemma ideal-sum-memE:
assumes $x \in \text{ideal-sum } J$
obtains $i j$ **where** $i \in I j \in J x = i + j$
using assms **unfolding** ideal-sum-def **by** blast

The sum contains both summands (using that ideals contain $\mathbf{0}$).

```

lemma I-subset-ideal-sum:
  assumes Ideal  $J$   $R$   $(+)$   $(\cdot)$   $\mathbf{0}$   $\mathbf{1}$  shows  $I \subseteq \text{ideal-sum } J$ 
proof
  interpret  $J$ : Ideal  $J$   $R$   $(+)$   $(\cdot)$   $\mathbf{0}$   $\mathbf{1}$  by (rule assms)
  fix  $x$  assume  $x: x \in I$ 
  then have  $xR: x \in R$  using additive.subset by blast
  have  $x + \mathbf{0} \in \text{ideal-sum } J$  using  $x$  J.additive.sub-unit-closed by (rule ideal-sum-memI)
  then show  $x \in \text{ideal-sum } J$  using  $xR$  by simp
qed

```

```

lemma J-subset-ideal-sum:
  assumes Ideal  $J$   $R$   $(+)$   $(\cdot)$   $\mathbf{0}$   $\mathbf{1}$  shows  $J \subseteq \text{ideal-sum } J$ 
proof
  interpret  $J$ : Ideal  $J$   $R$   $(+)$   $(\cdot)$   $\mathbf{0}$   $\mathbf{1}$  by (rule assms)
  fix  $x$  assume  $x: x \in J$ 
  then have  $xR: x \in R$  using J.additive.subset by blast
  have  $\mathbf{0} + x \in \text{ideal-sum } J$  using additive.sub-unit-closed  $x$  by (rule ideal-sum-memI)
  then show  $x \in \text{ideal-sum } J$  using  $xR$  by simp
qed

```

The intersection of the ambient ideal I with another ideal is again an ideal.

```

lemma inter-ideal:
  assumes Ideal  $J$   $R$   $(+)$   $(\cdot)$   $\mathbf{0}$   $\mathbf{1}$ 
  shows Ideal  $(I \cap J)$   $R$   $(+)$   $(\cdot)$   $\mathbf{0}$   $\mathbf{1}$ 
proof –
  interpret  $J$ : Ideal  $J$   $R$   $(+)$   $(\cdot)$   $\mathbf{0}$   $\mathbf{1}$  by (rule assms)
  have  $sg$ : Subgroup  $(I \cap J)$   $R$   $(+)$   $\mathbf{0}$ 
  proof (rule additive.subgroupI)
    show  $I \cap J \subseteq R$  using additive.subset by blast
    show  $\mathbf{0} \in I \cap J$  using additive.sub-unit-closed J.additive.sub-unit-closed by simp
  next
    fix  $g$   $h$  assume  $g \in I \cap J$   $h \in I \cap J$ 
    then show  $g + h \in I \cap J$ 
      using additive.sub-composition-closed J.additive.sub-composition-closed by auto
  next
    fix  $g$  assume  $g \in I \cap J$ 
    then show additive.invertible  $g$  using additive.subset by auto
  next
    fix  $g$  assume  $g: g \in I \cap J$ 
    then have  $gI$ :  $g \in I$  and  $gJ$ :  $g \in J$  by auto
    have additive.inverse  $g \in I$  using  $gI$  by simp
    moreover have additive.inverse  $g \in J$ 
      using  $gJ$  J.additive.submonoid-inverse-closed J.additive.sub.invertible by auto
    ultimately show additive.inverse  $g \in I \cap J$  by simp
  qed
  interpret  $IJ$ : Subgroup  $I \cap J$   $R$   $(+)$   $\mathbf{0}$  by (rule sg)
  show ?thesis
proof unfold-locales

```

```

    fix a b assume a: a ∈ R and b: b ∈ I ∩ J
    show a · b ∈ I ∩ J using a b Ideal J.Ideal by auto
    show b · a ∈ I ∩ J using a b Ideal J.Ideal by auto
  qed
qed

```

Comaximality is preserved under intersection: if I is comaximal with J and with K (meaning $\mathbf{1} \in \text{ideal-sum } J$ and $\mathbf{1} \in \text{ideal-sum } K$), then I is comaximal with $J \cap K$. The key step multiplies the two witnesses $i_1 + j = \mathbf{1}$, $i_2 + k = \mathbf{1}$ and regroups.

lemma *comaximal-inter*:

```

assumes J: Ideal J R (+) (·) 0 1 and K: Ideal K R (+) (·) 0 1
  and comIJ: ∃ i ∈ I. ∃ j ∈ J. i + j = 1
  and comIK: ∃ i ∈ I. ∃ k ∈ K. i + k = 1
shows ∃ i ∈ I. ∃ m ∈ J ∩ K. i + m = 1

```

proof —

```

interpret J: Ideal J R (+) (·) 0 1 by (rule J)
interpret K: Ideal K R (+) (·) 0 1 by (rule K)
obtain i1 j where i1: i1 ∈ I and j: j ∈ J and ij: i1 + j = 1
  using comIJ by blast
obtain i2 k where i2: i2 ∈ I and k: k ∈ K and ik: i2 + k = 1
  using comIK by blast
have i1R: i1 ∈ R using i1 additive.subset by blast
have i2R: i2 ∈ R using i2 additive.subset by blast
have jR: j ∈ R using j J.additive.subset by blast
have kR: k ∈ R using k K.additive.subset by blast
— Expand  $(i_1 + j) \cdot (i_2 + k) = \mathbf{1}$  by distributivity.
define m where m = i1 · i2 + i1 · k + j · i2
have mR: m ∈ R unfolding m-def using i1R i2R jR kR by simp
have expand: (i1 + j) · (i2 + k) = m + j · k
proof —
  have d1: i1 · (i2 + k) = i1 · i2 + i1 · k
    using i1R i2R kR by (simp add: distributive)
  have d2: j · (i2 + k) = j · i2 + j · k
    using jR i2R kR by (simp add: distributive)
  have (i1 + j) · (i2 + k) = i1 · (i2 + k) + j · (i2 + k)
    using distributive(2)[OF - i1R jR] i2R kR by simp
  also have ... = (i1 · i2 + i1 · k) + (j · i2 + j · k)
    by (simp add: d1 d2)
  also have ... = (i1 · i2 + i1 · k + j · i2) + j · k
    using i1R i2R jR kR by (simp add: additive.associative)
  finally show ?thesis unfolding m-def .

```

qed

```

have eq: m + j · k = 1
  using expand ij ik by simp
— m lies in I; j · k ∈ J ∩ K.

```

have *m-in-I*: $m \in I$

proof —

```

  have i1 · i2 ∈ I using i2R i1 by (simp add: Ideal)

```

```

    moreover have  $i_1 \cdot k \in I$  using  $kR\ i1$  by (simp add: Ideal)
    moreover have  $j \cdot i_2 \in I$  using  $jR\ i2$  by (simp add: Ideal)
    ultimately show ?thesis unfolding  $m\text{-def}$  by (simp add: additive.sub-composition-closed)
  qed
  have  $jk\text{-in-}JK: j \cdot k \in J \cap K$ 
  proof
    show  $j \cdot k \in J$  using  $kR\ j$  by (simp add: J.Ideal)
    show  $j \cdot k \in K$  using  $jR\ k$  by (simp add: K.Ideal)
  qed
  show ?thesis using  $m\text{-in-}I\ jk\text{-in-}JK$  eq by blast
qed

```

The reduction step underlying Chinese-Remainder surjectivity, phrased for a single ideal (where subtraction notation is unambiguous): if $i + j = \mathbf{1}$ with $i \in I$, then the element $x \cdot j + y \cdot i$ has the same coset as x modulo I (because $x \cdot j \equiv x$ and $y \cdot i \equiv \mathbf{0}$ modulo I).

lemma *coset-comaximal*:

```

  assumes  $x: x \in R$  and  $y: y \in R$  and  $i: i \in I$  and  $jR: j \in R$  and  $ij: i + j = \mathbf{1}$ 
  shows coset  $(x \cdot j + y \cdot i) = \text{coset } x$ 
  proof -
    have  $iR: i \in R$  using  $i$  additive.subset by blast
    have  $sumR: x \cdot j + y \cdot i \in R$  using  $x\ y\ iR\ jR$  by simp
    —  $x$  decomposes as  $x \cdot i + x \cdot j$  via  $i + j = \mathbf{1}$ .
    have  $xdecomp: x = x \cdot i + x \cdot j$ 
    proof -
      have  $x = x \cdot (i + j)$  using  $ij\ x$  by simp
      also have  $\dots = x \cdot i + x \cdot j$  using  $x\ iR\ jR$  by (simp add: distributive)
      finally show ?thesis .
    qed
    — An abelian-group rearrangement:  $q + p - (s + q) = p - s$  for  $q, p, s$  in  $R$ .
    — The difference  $x \cdot j + y \cdot i - x$  equals  $(y - x) \cdot i \in I$ .
    have  $(x \cdot j + y \cdot i) - x = (x \cdot j + y \cdot i) - (x \cdot i + x \cdot j)$  using  $xdecomp$  by simp
    also have  $\dots = y \cdot i - x \cdot i$ 
      using additive.add-diff-cancel-mid[ $of\ x \cdot j\ y \cdot i\ x \cdot i$ ]  $x\ y\ iR\ jR$  by simp
    also have  $\dots = (y - x) \cdot i$  using  $x\ y\ iR$  by (simp add: distributive left-minus)
    finally have  $eq: (x \cdot j + y \cdot i) - x = (y - x) \cdot i$  .
    have  $(y - x) \cdot i \in I$  using  $x\ y\ i$  by (simp add: Ideal)
    then have  $(x \cdot j + y \cdot i) - x \in I$  using  $eq$  by simp
    then show ?thesis using coset-eq-iff[OF  $sumR\ x$ ] by simp
  qed
end

```

64.4 Two comaximal ideals

The Chinese Remainder Theorem setting: a commutative ring R with two ideals I, J that are *comaximal* ($I + J = R$). We package these in a locale, giving each ideal its own quotient-ring interface ($I.$ and $J.$).

locale *comaximal-ideals* =

```

  I: quotient-ring I R (+) (·) 0 1 + J: quotient-ring J R (+) (·) 0 1
  for I and J and R and addition (infixl <+> 65) and multiplication (infixl <·> 70)
  and zero (<0>) and unit (<1>) +
  assumes comm:  $\llbracket a \in R; b \in R \rrbracket \implies a \cdot b = b \cdot a$ 
  and comaximal:  $\exists i \in I. \exists j \in J. i + j = 1$ 
begin

```

Both ideals live in a commutative ring; expose the commutative-ring structure and the *ideal-in-comm-ring* facts for each.

```

sublocale I: ideal-in-comm-ring I R (+) (·) 0 1
  by unfold-locale (rule comm)

```

```

sublocale J: ideal-in-comm-ring J R (+) (·) 0 1
  by unfold-locale (rule comm)

```

The product ring $(R/I) \times (R/J)$ of the two quotient rings.

```

sublocale prod: ring-pair
  R / I.Ring-Congruence I.additive.quotient-composition I.multiplicative.quotient-composition
  I.additive.Class 0 I.additive.Class 1
  R / J.Ring-Congruence J.additive.quotient-composition J.multiplicative.quotient-composition
  J.additive.Class 0 J.additive.Class 1
  by unfold-locale

```

The canonical projection $a \mapsto (I + a, J + a)$ into the product of the two quotients.

```

definition canon :: 'a  $\Rightarrow$  'a set  $\times$  'a set
  where canon = restrict ( $\lambda a. (I.\text{coset } a, J.\text{coset } a)$ ) R

```

```

lemma canon-apply [simp]:  $a \in R \implies \text{canon } a = (I.\text{coset } a, J.\text{coset } a)$ 
  by (simp add: canon-def)

```

```

lemma coset-I-in-carrier:  $a \in R \implies I.\text{coset } a \in R / I.\text{Ring-Congruence}$ 
  using I.coset-in-quotient-set by (simp add: I.quotient-set-def)

```

```

lemma coset-J-in-carrier:  $a \in R \implies J.\text{coset } a \in R / J.\text{Ring-Congruence}$ 
  using J.coset-in-quotient-set by (simp add: J.quotient-set-def)

```

The canonical projection is a ring homomorphism into the product ring.

```

theorem canon-hom:
  ring-homomorphism canon R (+) (·) 0 1 prod.Bcarrier (prod.Badd) (prod.Bmult)
  prod.Bzero prod.Bone
proof (unfold-locale)
  show canon  $\in R \rightarrow_E \text{prod.Bcarrier}$ 
proof (rule PiE-I)
  fix a assume a  $\in R$ 
  then show canon a  $\in \text{prod.Bcarrier}$ 
  by (simp add: prod.Bcarrier-def coset-I-in-carrier coset-J-in-carrier)
next

```



```

    fix a assume a ∉ R then show canon a = undefined by (simp add: canon-def)
  qed
next
  fix a b assume ab: a ∈ R b ∈ R
  have I.coset (a + b) = I.additive.quotient-composition (I.coset a) (I.coset b)
    using ab by (simp add: I.coset-def I.additive.Class-commutes-with-composition)
  moreover have J.coset (a + b) = J.additive.quotient-composition (J.coset a)
    (J.coset b)
    using ab by (simp add: J.coset-def J.additive.Class-commutes-with-composition)
  ultimately show canon (a + b) = prod.Badd (canon a) (canon b)
    using ab by (simp add: prod.Badd-def)
next
  fix a b assume ab: a ∈ R b ∈ R
  have I.coset (a · b) = I.multiplicative.quotient-composition (I.coset a) (I.coset b)
    using ab by (simp add: I.coset-def I.quot-mult-Class)
  moreover have J.coset (a · b) = J.multiplicative.quotient-composition (J.coset a)
    (J.coset b)
    using ab by (simp add: J.coset-def J.quot-mult-Class)
  ultimately show canon (a · b) = prod.Bmult (canon a) (canon b)
    using ab by (simp add: prod.Bmult-def)
next
  show canon 0 = prod.Bzero
    by (simp add: prod.Bzero-def I.coset-def J.coset-def)
next
  show canon 1 = prod.Bone
    by (simp add: prod.Bone-def I.coset-def J.coset-def)
qed

```

interpretation *canon: ring-homomorphism canon R (+) (·) 0 1*
prod.Bcarrier prod.Badd prod.Bmult prod.Bzero prod.Bone
by (rule canon-hom)

The kernel of the canonical projection is $I \cap J$: an element maps to zero exactly when both its cosets vanish, i.e. it lies in both ideals.

theorem *canon-kernel: canon.additive.Ker = I ∩ J*
proof (rule set-eqI)
 fix a
 have a ∈ canon.additive.Ker $\longleftrightarrow$ a ∈ R ∧ canon a = prod.Bzero
 unfolding canon.additive.Ker-def by auto
 also have ... $\longleftrightarrow$ a ∈ R ∧ I.coset a = I.coset 0 ∧ J.coset a = J.coset 0
 proof (cases a ∈ R)
 case True
 then show ?thesis by (simp add: prod.Bzero-def I.coset-def J.coset-def)
 qed simp
 also have ... $\longleftrightarrow$ a ∈ R ∧ a ∈ I ∧ a ∈ J
 by (simp add: I.coset-def J.coset-def I.Class-eq-zero-iff J.Class-eq-zero-iff cong:
 conj-cong)
 also have ... $\longleftrightarrow$ a ∈ I ∩ J
 using I.additive.subset by blast

finally show $a \in \text{canon.additive.Ker} \longleftrightarrow a \in I \cap J$.
qed

Comaximality makes the canonical projection surjective: given target cosets of x and y , the element $x \cdot j + y \cdot i$ (where $i + j = \mathbf{1}$, $i \in I$, $j \in J$) reduces to $x \bmod I$ and to $y \bmod J$.

theorem *canon-surjective*: $\text{canon} \text{ ' } R = \text{prod.Bcarrier}$

proof (*rule set-eqI*)

fix z

show $z \in \text{canon} \text{ ' } R \longleftrightarrow z \in \text{prod.Bcarrier}$

proof

assume $z \in \text{canon} \text{ ' } R$

then show $z \in \text{prod.Bcarrier}$ **using** *canon.graph by blast*

next

assume $z \in \text{prod.Bcarrier}$

then obtain $X \ Y$ **where** zXY : $z = (X, Y)$ **and** X : $X \in R / I.\text{Ring-Congruence}$
and Y : $Y \in R / J.\text{Ring-Congruence}$ **by** (*auto simp: prod.Bcarrier-def*)

obtain x **where** x : $x \in R$ **and** Xx : $X = I.\text{coset } x$

using $X \ I.\text{quotient-set-repr}$ **by** (*metis I.quotient-set-def*)

obtain y **where** y : $y \in R$ **and** Yy : $Y = J.\text{coset } y$

using $Y \ J.\text{quotient-set-repr}$ **by** (*metis J.quotient-set-def*)

— The comaximality witness $i + j = \mathbf{1}$.

obtain $i \ j$ **where** i : $i \in I$ **and** j : $j \in J$ **and** ij : $i + j = \mathbf{1}$

using *comaximal by blast*

have iR : $i \in R$ **using** $i \ I.\text{additive.subset}$ **by** *blast*

have jR : $j \in R$ **using** $j \ J.\text{additive.subset}$ **by** *blast*

define a **where** $a = x \cdot j + y \cdot i$

have aR : $a \in R$ **unfolding** $a\text{-def}$ **using** $x \ y \ iR \ jR$ **by** *simp*

— Mod I : reduces to x (since $i \in I$, $i + j = \mathbf{1}$).

have IX : $I.\text{coset } a = I.\text{coset } x$

unfolding $a\text{-def}$ **by** (*rule I.coset-comaximal[OF x y i jR ij]*)

— Mod J : symmetrically reduces to y , using $j + i = \mathbf{1}$.

have ji : $j + i = \mathbf{1}$ **using** $ij \ iR \ jR$ **by** (*simp add: I.additive.commutative*)

have $J.\text{coset } (y \cdot i + x \cdot j) = J.\text{coset } y$

by (*rule J.coset-comaximal[OF y x j iR ji]*)

moreover have $y \cdot i + x \cdot j = a$

unfolding $a\text{-def}$ **using** $x \ y \ iR \ jR$ **by** (*simp add: I.additive.commutative*)

ultimately have JY : $J.\text{coset } a = J.\text{coset } y$ **by** *simp*

have $\text{canon } a = (X, Y)$ **using** $aR \ IX \ JY \ Xx \ Yy$ **by** *simp*

then show $z \in \text{canon} \text{ ' } R$ **using** $aR \ zXY$ **by** *auto*

qed

qed

64.5 The Chinese Remainder Theorem

Feeding the canonical projection to the fundamental theorem of ring homomorphisms yields $R / \text{Ker canon}$ isomorphic to the image, which surjectivity identifies with the whole product ring, and the kernel is $I \cap J$.

interpretation *canon: ring-homomorphism-fundamental canon* $R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}$

prod.Bcarrier prod.Badd prod.Bmult prod.Bzero prod.Bone
by *unfold-locales*

The induced map $\eta|$ is an isomorphism from R / Ker onto the image; surjectivity identifies the image with the whole product ring. With *canon-kernel* this is $R / (I \cap J) \cong (R / I) \times (R / J)$.

theorem *chinese-remainder*:

$(R / (\text{subgroup-of-additive-group-of-ring.Ring-Congruence } (I \cap J) R (+) \mathbf{0}),$
canon.kernel.additive.quotient-composition, canon.kernel.multiplicative.quotient-composition,
canon.kernel.additive.Class 0, canon.kernel.additive.Class 1)
 $\cong_R (\text{prod.Bcarrier, prod.Badd, prod.Bmult, prod.Bzero, prod.Bone})$

proof —

have *ker*: *canon.additive.Ker* = $I \cap J$ **by** (*rule canon-kernel*)

— The induced map $\eta|$ is a ring isomorphism from R / Ker onto the whole product ring: it is a ring homomorphism (indeed the monomorphism supplied by *ring-homomorphism-fundamental*), and it is bijective — injective as that monomorphism, surjective because *canon* is.

have *ring-isomorphism canon.additive.induced*

$(R / (\text{subgroup-of-additive-group-of-ring.Ring-Congruence } (I \cap J) R (+) \mathbf{0}))$
canon.kernel.additive.quotient-composition canon.kernel.multiplicative.quotient-composition
(canon.kernel.additive.Class 0) (canon.kernel.additive.Class 1)
prod.Bcarrier prod.Badd prod.Bmult prod.Bzero prod.Bone

unfolding *ker[symmetric]*

proof (*rule ring-isomorphism.intro*)

show *ring-homomorphism canon.additive.induced*

$(R / (\text{subgroup-of-additive-group-of-ring.Ring-Congruence } \text{canon.additive.Ker } R (+) \mathbf{0}))$

canon.kernel.additive.quotient-composition canon.kernel.multiplicative.quotient-composition
(canon.kernel.additive.Class 0) (canon.kernel.additive.Class 1)
prod.Bcarrier prod.Badd prod.Bmult prod.Bzero prod.Bone

by (*rule canon.induced.ring-homomorphism-axioms*)

show *bijective-map canon.additive.induced*

$(R / (\text{subgroup-of-additive-group-of-ring.Ring-Congruence } \text{canon.additive.Ker } R (+) \mathbf{0}))$

prod.Bcarrier

proof

show *bij-betw canon.additive.induced*

$(R / (\text{subgroup-of-additive-group-of-ring.Ring-Congruence } \text{canon.additive.Ker } R (+) \mathbf{0}))$

prod.Bcarrier

using *canon.additive.induced-image canon.additive.induced-inj-on canon-surjective*
by (*simp add: bij-betw-def*)

qed

qed

then show *?thesis*

by (*auto simp: isomorphic-as-rings-def*)

qed

end

Transfer of the ring structure along operations that agree on the carrier. If R is

a ring under $(+)$, $(\cdot)$ and the alternative operations $(\oplus)$, $(\otimes)$ agree with them on all pairs of carrier elements, then R is also a ring under $(\oplus)$, $(\otimes)$.

lemma *Ring-cong-on-carrier*:

```

fixes  $R$  and addition (infixl  $\langle + \rangle$  65) and multiplication (infixl  $\langle \cdot \rangle$  70)
and addition' (infixl  $\langle \oplus \rangle$  65) and multiplication' (infixl  $\langle \otimes \rangle$  70) and zero ( $\langle 0 \rangle$ )
and unit ( $\langle 1 \rangle$ )
assumes ring: Ring  $R$   $(+)$   $(\cdot)$  0 1
and add-eq:  $\bigwedge a\ b. \llbracket a \in R; b \in R \rrbracket \implies a \oplus b = a + b$ 
and mult-eq:  $\bigwedge a\ b. \llbracket a \in R; b \in R \rrbracket \implies a \otimes b = a \cdot b$ 
shows Ring  $R$   $(\oplus)$   $(\otimes)$  0 1
proof –
interpret  $R$ : Ring  $R$   $(+)$   $(\cdot)$  0 1 by (rule ring)
show ?thesis
proof (rule Ring.intro)
show Abelian-Group  $R$   $(\oplus)$  0
proof (rule Abelian-Group.intro)
show Group  $R$   $(\oplus)$  0
proof (rule GroupI)
show  $\bigwedge a\ b. \llbracket a \in R; b \in R \rrbracket \implies a \oplus b \in R$  by (simp add: add-eq)
show 0  $\in R$  by simp
show  $\bigwedge a\ b\ c. \llbracket a \in R; b \in R; c \in R \rrbracket \implies (a \oplus b) \oplus c = a \oplus (b \oplus c)$ 
by (simp add: add-eq R.additive.associative)
show  $\bigwedge a. a \in R \implies \mathbf{0} \oplus a = a$  by (simp add: add-eq)
show  $\bigwedge a. a \in R \implies a \oplus \mathbf{0} = a$  by (simp add: add-eq)
fix  $u$  assume  $u: u \in R$ 
have  $u \oplus R.additive.inverse\ u = \mathbf{0} \wedge R.additive.inverse\ u \oplus u = \mathbf{0}$ 
using  $u$  by (simp add: add-eq)
then show  $\exists v \in R. u \oplus v = \mathbf{0} \wedge v \oplus u = \mathbf{0}$  using  $u$  by blast
qed
show commutative-monoid  $R$   $(\oplus)$  0
proof (rule commutative-monoid.intro)
show Monoid  $R$   $(\oplus)$  0
by unfold-locales (auto simp: add-eq R.additive.associative)
show commutative-monoid-axioms  $R$   $(\oplus)$ 
by unfold-locales (simp add: add-eq R.additive.commutative)
qed
show Monoid  $R$   $(\otimes)$  1
by unfold-locales (auto simp: mult-eq R.multiplicative.associative)
show Ring-axioms  $R$   $(\oplus)$   $(\otimes)$ 
proof
fix  $a\ b\ c$  assume  $a \in R\ b \in R\ c \in R$ 
then show  $a \otimes (b \oplus c) = a \otimes b \oplus (a \otimes c)$ 
by (simp add: add-eq mult-eq R.distributive)
next
fix  $a\ b\ c$  assume  $a \in R\ b \in R\ c \in R$ 
then show  $(b \oplus c) \otimes a = b \otimes a \oplus (c \otimes a)$ 
by (simp add: add-eq mult-eq R.distributive)
qed

```

qed
qed

65 The n-ary Chinese Remainder Theorem

Generalise the binary CRT to a finite family of pairwise-comaximal ideals: $R / (\bigcap_{i \in S} J i) \cong_R \prod_{i \in S} R / J i$. The proof proceeds by induction on the finite index set, reducing (n+1) ideals to the binary theorem via the lemma that comaximality is preserved under intersection.

65.1 Pairwise-comaximal ideals

```

locale pairwise-comaximal-ideals =
  fixes  $S :: 'i \text{ set}$ 
    and  $J :: 'i \Rightarrow 'a \text{ set}$ 
    and  $R :: 'a \text{ set}$ 
    and addition (infixl  $\langle + \rangle$  65)
    and multiplication (infixl  $\langle \cdot \rangle$  70)
    and zero ( $\langle 0 \rangle$ )
    and unit ( $\langle 1 \rangle$ )
  assumes fn: finite  $S$ 
    and nonempty:  $S \neq \{\}$ 
    and comm:  $\llbracket a \in R; b \in R \rrbracket \Longrightarrow a \cdot b = b \cdot a$ 
    and ideals:  $\bigwedge i. i \in S \Longrightarrow \text{quotient-ring } (J i) R (+) (\cdot) \mathbf{0} \mathbf{1}$ 
    and pairwise:  $\bigwedge i j. \llbracket i \in S; j \in S; i \neq j \rrbracket \Longrightarrow \exists a \in J i. \exists b \in J j. a + b = \mathbf{1}$ 
begin

  sublocale Ring  $R (+) (\cdot) \mathbf{0} \mathbf{1}$ 
  proof –
    obtain  $i$  where  $i \in S$  using nonempty by blast
    then interpret  $qi$ : quotient-ring  $J i R (+) (\cdot) \mathbf{0} \mathbf{1}$  by (rule ideals)
    show Ring  $R (+) (\cdot) \mathbf{0} \mathbf{1} ..$ 
  qed

  sublocale commutative-ring  $R (+) (\cdot) \mathbf{0} \mathbf{1}$ 
  by unfold-locales (rule comm)

  Each  $J i$  is an ideal in the commutative ring.

  lemma Ji-ideal:  $i \in S \Longrightarrow \text{Ideal } (J i) R (+) (\cdot) \mathbf{0} \mathbf{1}$ 
  proof –
    assume  $i \in S$ 
    then interpret  $qi$ : quotient-ring  $J i R (+) (\cdot) \mathbf{0} \mathbf{1}$  by (rule ideals)
    show Ideal  $(J i) R (+) (\cdot) \mathbf{0} \mathbf{1} ..$ 
  qed

  lemma Ji-ideal-comm:  $i \in S \Longrightarrow \text{ideal-in-comm-ring } (J i) R (+) (\cdot) \mathbf{0} \mathbf{1}$ 
  by (intro ideal-in-comm-ring.intro Ji-ideal commutative-ring-axioms)

```

The intersection $\bigcap (J \text{ ' } T)$ over a subset $T \subseteq S$ is again an ideal.

lemma *Inter-ideal*:

assumes $TS: T \subseteq S$ **and** $The: T \neq \{\}$

shows $Ideal (\bigcap (J \text{ ' } T)) R (+) (\cdot) \mathbf{0} \mathbf{1}$

proof –

have $finT: finite\ T$ **using** $TS\ fin\ finite\ subset$ **by** *blast*

show $?thesis$ **using** $finT\ The\ TS$

proof (*induction rule: finite-ne-induct*)

case (*singleton* x)

then show $?case$ **using** $Ji\ ideal$ **by** *simp*

next

case (*insert* $x\ F$)

then have $xS: x \in S$ **and** $FS: F \subseteq S$ **by** *auto*

interpret $Jx: ideal\ in\ comm\ ring\ J\ x\ R\ (+)\ (\cdot) \mathbf{0} \mathbf{1}$ **by** (*rule* $Ji\ ideal\ comm[OF\ xS]$)

have $Ideal (\bigcap (J \text{ ' } F)) R (+) (\cdot) \mathbf{0} \mathbf{1}$ **using** *insert* **by** *auto*

then have $Ideal (J\ x \cap \bigcap (J \text{ ' } F)) R (+) (\cdot) \mathbf{0} \mathbf{1}$

by (*rule* $Jx.inter\ ideal$)

then show $?case$ **using** *insert* **by** *simp*

qed

qed

Key induction step: if $J\ i$ is comaximal with every $J\ j$ for $j \in T$, then $J\ i$ is comaximal with $\bigcap (J \text{ ' } T)$.

lemma *comaximal-with-Inter*:

assumes $iS: i \in S$ **and** $TS: T \subseteq S$ **and** $The: T \neq \{\}$ **and** $iT: i \notin T$

and $pw: \bigwedge j. j \in T \implies \exists a \in J\ i. \exists b \in J\ j. a + b = \mathbf{1}$

shows $\exists a \in J\ i. \exists b \in \bigcap (J \text{ ' } T). a + b = \mathbf{1}$

proof –

have $finT: finite\ T$ **using** $TS\ fin\ finite\ subset$ **by** *blast*

show $?thesis$ **using** $finT\ The\ TS\ iT\ pw$

proof (*induction rule: finite-ne-induct*)

case (*singleton* x)

then show $?case$ **by** *simp*

next

case (*insert* $x\ F$)

then have $xS: x \in S$ **and** $FS: F \subseteq S$ **by** *auto*

have $Fne: F \neq \{\}$ **using** *insert* **by** *simp*

have $iF: i \notin F$ **using** *insert.prem*s **by** *simp*

have $comF: \exists a \in J\ i. \exists b \in \bigcap (J \text{ ' } F). a + b = \mathbf{1}$

using *insert* **by** *auto*

have $comx: \exists a \in J\ i. \exists b \in J\ x. a + b = \mathbf{1}$ **using** *insert.prem*s **by** *simp*

interpret $Ji: ideal\ in\ comm\ ring\ J\ i\ R\ (+)\ (\cdot) \mathbf{0} \mathbf{1}$ **by** (*rule* $Ji\ ideal\ comm[OF\ iS]$)

have $Jx\ ideal: Ideal (J\ x) R (+) (\cdot) \mathbf{0} \mathbf{1}$ **by** (*rule* $Ji\ ideal[OF\ xS]$)

have $IF\ ideal: Ideal (\bigcap (J \text{ ' } F)) R (+) (\cdot) \mathbf{0} \mathbf{1}$

by (*rule* $Inter\ ideal[OF\ FS\ Fne]$)

have $\exists a \in J\ i. \exists b \in J\ x \cap \bigcap (J \text{ ' } F). a + b = \mathbf{1}$

by (*rule* $Ji.comaximal\ inter[OF\ Jx\ ideal\ IF\ ideal\ comx\ comF]$)

then show $?case$ **using** *insert* **by** *simp*

qed
qed

65.2 The canonical projection into the indexed product

The quotient rings form a ring family. The coset *Ji-Class i a* is the equivalence class of *a* modulo the ring congruence induced by *J i*. The quotient carrier, operations, and constants are defined in terms of this class map.

definition *Ji-Cong* :: '*i* ⇒ ('*a* × '*a*) set **where**
Ji-Cong i = {(*a*, *b*). *a* ∈ *R* ∧ *b* ∈ *R* ∧ *a* − *b* ∈ *J i*}

definition *Ji-Class* :: '*i* ⇒ '*a* ⇒ '*a* set **where**
Ji-Class i a = {*b* ∈ *R*. (*a*, *b*) ∈ *Ji-Cong i*}

definition *quot-carrier* :: '*i* ⇒ '*a* set set **where**
quot-carrier i = *Ji-Class i* ' *R*

definition *quot-add* :: '*i* ⇒ '*a* set ⇒ '*a* set ⇒ '*a* set **where**
quot-add i A B = (THE *C*. ∃ *a* ∈ *A*. ∃ *b* ∈ *B*. *C* = *Ji-Class i* (*a* + *b*))

definition *quot-mult* :: '*i* ⇒ '*a* set ⇒ '*a* set ⇒ '*a* set **where**
quot-mult i A B = (THE *C*. ∃ *a* ∈ *A*. ∃ *b* ∈ *B*. *C* = *Ji-Class i* (*a* · *b*))

definition *quot-zero* :: '*i* ⇒ '*a* set **where**
quot-zero i = *Ji-Class i* **0**

definition *quot-one* :: '*i* ⇒ '*a* set **where**
quot-one i = *Ji-Class i* **1**

lemma *Ji-Cong-eq*:
assumes *i* ∈ *S*
shows *Ji-Cong i* = *subgroup-of-additive-group-of-ring.Ring-Congruence (J i) R addition 0*
proof −
interpret *qi*: *quotient-ring J i R (+) (·) 0 1* **by** (rule *ideals[OF assms]*)
show ?thesis
unfolding *Ji-Cong-def qi.Ring-Congruence-def* **by** auto
qed

Key bridge: *Ji-Class* agrees with the quotient-ring coset. The proof is by unfolding definitions within a local interpretation.

lemma *Ji-Class-unfold*:
assumes *a* ∈ *R*
shows *Ji-Class i a* = {*b* ∈ *R*. *a* − *b* ∈ *J i*}
unfolding *Ji-Class-def Ji-Cong-def* **using** *assms* **by** auto

lemma *Ji-Cong-sym*:
assumes *i*: *i* ∈ *S* **and** (*a*, *b*) ∈ *Ji-Cong i*
shows (*b*, *a*) ∈ *Ji-Cong i*

proof –
interpret *qi*: *quotient-ring J i R (+) (·) 0 1* **by** (*rule ideals[OF i]*)
from *assms*(2) **have** *aR*: $a \in R$ **and** *bR*: $b \in R$ **and** *d*: $a - b \in J i$
unfolding *Ji-Cong-def* **by** *auto*
have $-(a - b) \in J i$ **using** *d qi.additive.submonoid-inverse-closed aR bR* **by** *simp*
then have $b - a \in J i$
using *aR bR* **by** (*simp add: additive.inverse-composition-commute additive.commutative*)
then show *?thesis* **using** *aR bR* **unfolding** *Ji-Cong-def* **by** *auto*
qed

lemma *Ji-Class-mem-sym*:
assumes *i*: $i \in S$ **and** *a* $\in R$ *b* $\in Ji\text{-Class } i a$
shows *a* $\in Ji\text{-Class } i b$
proof –
from *assms*(3) **have** *bR*: $b \in R$ **and** *ab*: $(a, b) \in Ji\text{-Cong } i$
unfolding *Ji-Class-def* **by** *auto*
have $(b, a) \in Ji\text{-Cong } i$ **by** (*rule Ji-Cong-sym[OF i ab]*)
then show *?thesis* **using** *assms*(2) **unfolding** *Ji-Class-def* **by** *auto*
qed

lemma *Ji-Class-coset-bridge*:
assumes *i*: $i \in S$ **and** *aR*: $a \in R$
shows $Ji\text{-Class } i a = \{b \in R. (a, b) \in Ji\text{-Cong } i\}$
unfolding *Ji-Class-def* **by** *auto*

Within an interpretation *qi*: *quotient-ring (J i) R (+) (·) 0 1*, the coset *qi.coset a* equals *Ji-Class i a*. We package this as a fact to use in proofs.

lemma *Ji-Class-qi-coset*:
assumes *i*: $i \in S$ **and** *aR*: $a \in R$
shows $Ji\text{-Class } i a = \{b \in R. a - b \in J i\}$
by (*rule Ji-Class-unfold[OF aR]*)

The coset map is well-defined modulo the congruence.

lemma *Ji-Class-cong*:
assumes *i*: $i \in S$ **and** *a* $\in R$ *b* $\in R$ $(a, b) \in Ji\text{-Cong } i$
shows $Ji\text{-Class } i a = Ji\text{-Class } i b$
proof –
interpret *qi*: *quotient-ring J i R (+) (·) 0 1* **by** (*rule ideals[OF i]*)
have *ca*: $Ji\text{-Class } i a = qi\text{-coset } a$ **and** *cb*: $Ji\text{-Class } i b = qi\text{-coset } b$
using $\langle a \in R \rangle \langle b \in R \rangle qi\text{-additive.symmetric}$
by (*auto simp: Ji-Class-def Ji-Cong-eq i qi.coset-def*)
have $qi\text{-coset } a = qi\text{-coset } b$
using *assms* **by** (*simp add: qi.coset-def qi.additive.Class-equivalence qi.additive-congruence Ji-Cong-eq[OF i]*)
then show *?thesis* **using** *ca cb* **by** *simp*
qed

lemma *Ji-Class-repr*:
assumes *i*: $i \in S$ **and** *A* $\in quot\text{-carrier } i$


```

obtains a where  $a \in R$   $A = \text{Ji-Class } i \ a$ 
using assms unfolding quot-carrier-def by auto

lemma quot-Ring:
  assumes i:  $i \in S$ 
  shows Ring (quot-carrier i) (quot-add i) (quot-mult i) (quot-zero i) (quot-one i)
proof –
  interpret qi: quotient-ring J i R (+) (·) 0 1 by (rule ideals[OF i])
  have class-eq:  $\bigwedge a. a \in R \implies \text{Ji-Class } i \ a = \text{qi.coset } a$ 
    using qi.additive.symmetric
    by (auto simp: Ji-Class-def Ji-Cong-eq i qi.coset-def)
  have carrier: quot-carrier i = qi.quotient-set
    unfolding quot-carrier-def qi.quotient-set-def qi.additive.Partition-def qi.coset-def
    using class-eq qi.coset-def by (auto simp: image-def)
  have qzero: quot-zero i = qi.coset 0 by (simp add: quot-zero-def class-eq)
  have qone: quot-one i = qi.coset 1 by (simp add: quot-one-def class-eq)
  — The quotient operations agree with the locale's. Each element of the carrier is a
  coset, and the operations THE-select the unique class of any representative pair.
  have qadd: quot-add i A B = qi.additive.quotient-composition A B
    if AB:  $A \in \text{quot-carrier } i \ B \in \text{quot-carrier } i$  for A B
  proof –
    obtain a b where aR:  $a \in R \ A = \text{Ji-Class } i \ a$  and bR:  $b \in R \ B = \text{Ji-Class } i \ b$ 
      using AB unfolding quot-carrier-def by auto
    have Aca:  $A = \text{qi.coset } a$  using aR class-eq by simp
    have Bcb:  $B = \text{qi.coset } b$  using bR class-eq by simp
    have rhs: qi.additive.quotient-composition A B = qi.coset ( $a + b$ )
      using aR(1) bR(1) Aca Bcb by (simp add: qi.coset-def qi.additive.Class-commutes-with-composition)
    have aA:  $a \in A$  using aR(1,2) unfolding Ji-Class-def Ji-Cong-def by auto
    moreover have bB:  $b \in B$  using bR(1,2) unfolding Ji-Class-def Ji-Cong-def
  by auto
  moreover have  $\bigwedge a' \ b'. a' \in A \implies b' \in B \implies \text{Ji-Class } i \ (a' + b') = \text{Ji-Class } i \ (a + b)$ 
  proof –
    fix a' b' assume  $a' \in A \ b' \in B$ 
    then have a'R:  $a' \in R$  and b'R:  $b' \in R$ 
      using AB unfolding quot-carrier-def Ji-Class-def by auto
    have qi.coset  $a' = \text{qi.coset } a$ 
      using  $\langle a' \in A \rangle \ Aca \ a'R \ aR(1) \ \text{qi.coset-def } \text{qi.additive.Class-equivalence}$ 
      qi.additive-congruence
      by auto
    moreover have qi.coset  $b' = \text{qi.coset } b$ 
      using  $\langle b' \in B \rangle \ Bcb \ b'R \ bR(1) \ \text{qi.coset-def } \text{qi.additive.Class-equivalence}$ 
      qi.additive-congruence
      by auto
    ultimately have qi.coset ( $a' + b'$ ) = qi.coset ( $a + b$ )
  proof –
    assume eq-a: qi.coset  $a' = \text{qi.coset } a$  and eq-b: qi.coset  $b' = \text{qi.coset } b$ 
    have qi.coset ( $a' + b'$ ) = qi.additive.quotient-composition (qi.coset  $a'$ ) (qi.coset  $b'$ )
      using a'R b'R by (simp add: qi.coset-def qi.additive.Class-commutes-with-composition)

```

```

    also have ... = qi.additive.quotient-composition (qi.coset a) (qi.coset b)
      by (simp add: eq-a eq-b)
    also have ... = qi.coset (a + b)
    using aR(1) bR(1) by (simp add: qi.coset-def qi.additive.Class-commutes-with-composition)
    finally show qi.coset (a' + b') = qi.coset (a + b) .
  qed
  then show Ji-Class i (a' + b') = Ji-Class i (a + b)
    using class-eq a'R b'R aR(1) bR(1) by simp
  qed
  ultimately have quot-add i A B = Ji-Class i (a + b)
    unfolding quot-add-def by (intro the-equality; blast)
  also have ... = qi.coset (a + b) using class-eq aR(1) bR(1) by simp
  finally show ?thesis using rhs by simp
  qed
  have qmult: quot-mult i A B = qi.multiplicative.quotient-composition A B
    if AB: A ∈ quot-carrier i B ∈ quot-carrier i for A B
  proof -
    interpret qi-comm: ideal-in-comm-ring J i R (+) (·) 0 1 by (rule Ji-ideal-comm[OF
i])
    obtain a b where aR: a ∈ R A = Ji-Class i a and bR: b ∈ R B = Ji-Class i b
      using AB unfolding quot-carrier-def by auto
    have Aca: A = qi.coset a using aR class-eq by simp
    have Bcb: B = qi.coset b using bR class-eq by simp
    have rhs: qi.multiplicative.quotient-composition A B = qi.coset (a · b)
      using aR(1) bR(1) Aca Bcb by (simp add: qi.coset-def qi-comm.quot-mult-Class)
    have a ∈ A using aR(1,2) unfolding Ji-Class-def Ji-Cong-def by auto
    moreover have b ∈ B using bR(1,2) unfolding Ji-Class-def Ji-Cong-def by
auto
    moreover have  $\bigwedge a' b'. a' \in A \implies b' \in B \implies \text{Ji-Class } i (a' \cdot b') = \text{Ji-Class } i (a \cdot b)$ 
      proof -
        fix a' b' assume a'R: a' ∈ R and b'R: b' ∈ R
        then have a'R: a' ∈ R and b'R: b' ∈ R
          using AB unfolding quot-carrier-def Ji-Class-def by auto
        have (a, a') ∈ Ji-Cong i using ⟨a' ∈ A⟩ aR(2) unfolding Ji-Class-def by auto
        then have ca: (a', a) ∈ qi.Ring-Congruence using Ji-Cong-sym[OF i] Ji-Cong-eq[OF
i] by simp
        have (b, b') ∈ Ji-Cong i using ⟨b' ∈ B⟩ bR(2) unfolding Ji-Class-def by auto
        then have cb: (b', b) ∈ qi.Ring-Congruence using Ji-Cong-sym[OF i] Ji-Cong-eq[OF
i] by simp
        have (a' · b', a · b) ∈ qi.Ring-Congruence
          using ca cb by (rule qi.multiplicative-congruence)
        then have (a' · b', a · b) ∈ Ji-Cong i using Ji-Cong-eq[OF i] by simp
        then show Ji-Class i (a' · b') = Ji-Class i (a · b)
          using Ji-Class-cong[OF i] a'R b'R aR(1) bR(1) by simp
      qed
    ultimately have quot-mult i A B = Ji-Class i (a · b)
      unfolding quot-mult-def by (intro the-equality; blast)
    also have ... = qi.coset (a · b) using class-eq aR(1) bR(1) by simp
    finally show ?thesis using rhs by simp
  end

```

```

qed
— Transfer the Ring structure from qi.quotient-set to our definitional carrier.
have Ring (quot-carrier i) qi.additive.quotient-composition qi.multiplicative.quotient-composition
      (quot-zero i) (quot-one i)
  unfolding carrier qzero qone qi.quotient-set-def qi.coset-def by unfold-locales
then show ?thesis
  using qadd qmult by (rule Ring-cong-on-carrier) auto
qed

```

interpretation *prod*: ring-family *S* quot-carrier quot-add quot-mult quot-zero quot-one
 by (rule ring-family.intro) (rule quot-Ring)

The coset map for each component — directly uses *Ji-Class*.

definition *ncoset* :: '*i* ⇒ '*a* ⇒ '*a* set **where**
ncoset *i* *a* = *Ji-Class* *i* *a*

lemma *ncoset-in-carrier*:
assumes *i* ∈ *S* *a* ∈ *R*
shows *ncoset* *i* *a* ∈ quot-carrier *i*
unfolding *ncoset-def* quot-carrier-def **using** *assms*(2) **by** *auto*

The canonical map $a \mapsto (\lambda i \in S. \text{Ji-Class } i \ a)$ into the indexed product.

definition *ncanon* :: '*a* ⇒ ('*i* ⇒ '*a* set) **where**
ncanon = restrict ($\lambda a. \lambda i \in S. \text{ncoset } i \ a$) *R*

lemma *ncanon-apply*:
assumes *a* ∈ *R* *i* ∈ *S*
shows *ncanon* *a* *i* = *ncoset* *i* *a*
using *assms* **by** (simp add: *ncanon-def*)

lemma *ncanon-closed*:
assumes *a* ∈ *R*
shows *ncanon* *a* ∈ prod.Pcarrier
proof (rule prod.Pcarrier-memI)
 fix *i* **assume** *i*: *i* ∈ *S*
 show *ncanon* *a* *i* ∈ quot-carrier *i*
 using *assms* *i* **by** (simp add: *ncanon-apply* *ncoset-in-carrier*)
next
 show *ncanon* *a* ∈ extensional *S* **using** *assms* **by** (simp add: *ncanon-def*)
qed

lemma *ncanon-outside*: $a \notin R \implies \text{ncanon } a = \text{undefined}$
by (simp add: *ncanon-def*)

The canonical map is a ring homomorphism.

theorem *ncanon-hom*:
 ring-homomorphism *ncanon* *R* (+) (·) 0 1
 prod.Pcarrier prod.Padd prod.Pmult prod.Pzero prod.Pone
proof unfold-locales

```

show ncanon  $\in R \rightarrow_E \text{prod.Pcarrier}$ 
proof (rule PiE-I)
  fix a assume a  $\in R$  then show ncanon a  $\in \text{prod.Pcarrier}$  by (rule ncanon-closed)
next
  fix a assume a  $\notin R$  then show ncanon a = undefined by (rule ncanon-outside)
qed
next
fix a b assume ab: a  $\in R$  b  $\in R$ 
show ncanon (a + b) = prod.Padd (ncanon a) (ncanon b)
proof (rule prod.Pcarrier-eqI)
  show ncanon (a + b)  $\in \text{prod.Pcarrier}$  using ab by (simp add: ncanon-closed)
  show prod.Padd (ncanon a) (ncanon b)  $\in \text{prod.Pcarrier}$ 
  using ab by (simp add: ncanon-closed)
next
fix i assume i: i  $\in S$ 
interpret qi: quotient-ring J i R (+) ( $\cdot$ ) 0 1 by (rule ideals[OF i])
have lhs: ncanon (a + b) i = Ji-Class i (a + b)
  using ab i by (simp add: ncanon-apply ncaset-def)
have rhs: prod.Padd (ncanon a) (ncanon b) i =
  quot-add i (Ji-Class i a) (Ji-Class i b)
  using ab i by (simp add: ncanon-apply prod.Padd-apply ncaset-def)
also have ... = Ji-Class i (a + b)
  unfolding quot-add-def
proof (rule the-equality)
  show  $\exists a' \in \text{Ji-Class } i \ a. \exists b' \in \text{Ji-Class } i \ b. \text{Ji-Class } i \ (a + b) = \text{Ji-Class } i \ (a' + b')$ 
  proof (intro bexI)
    show a  $\in \text{Ji-Class } i \ a$  using ab(1) unfolding Ji-Class-def Ji-Cong-def by
auto
    show b  $\in \text{Ji-Class } i \ b$  using ab(2) unfolding Ji-Class-def Ji-Cong-def by auto
  qed simp
next
fix C assume  $\exists a' \in \text{Ji-Class } i \ a. \exists b' \in \text{Ji-Class } i \ b. C = \text{Ji-Class } i \ (a' + b')$ 
then obtain a' b' where a'A: a'  $\in \text{Ji-Class } i \ a$  and b'B: b'  $\in \text{Ji-Class } i \ b$ 
  and Ceq: C = Ji-Class i (a' + b') by auto
have a'R: a'  $\in R$  using a'A unfolding Ji-Class-def by auto
have b'R: b'  $\in R$  using b'B unfolding Ji-Class-def by auto
have ca: (a', a)  $\in \text{qi.Ring-Congruence}$ 
proof -
  have d: a - a'  $\in J \ i$  using a'A ab(1) unfolding Ji-Class-def Ji-Cong-def by
auto
  have dR: a - a'  $\in R$  using ab(1) a'R by simp
  have a' - a = - (a - a')
    using ab(1) a'R by (simp add: additive.inverse-composition-commute additive.commutative)
  also have - (a - a')  $\in J \ i$  using d dR qi.additive.submonoid-inverse-closed
by simp
  finally have a' - a  $\in J \ i$  .
  then show ?thesis using a'R ab(1) unfolding qi.Ring-Congruence-def by
auto

```

```

qed
have cb: (b', b) ∈ qi.Ring-Congruence
proof -
  have d: b - b' ∈ J i using b'B ab(2) unfolding Ji-Class-def Ji-Cong-def by
auto
  have dR: b - b' ∈ R using ab(2) b'R by simp
  have b' - b = - (b - b')
    using ab(2) b'R by (simp add: additive.inverse-composition-commute addi-
tive.commutative)
  also have - (b - b') ∈ J i using d dR qi.additive.submonoid-inverse-closed
by simp
  finally have b' - b ∈ J i .
  then show ?thesis using b'R ab(2) unfolding qi.Ring-Congruence-def by
auto
qed
have (a' + b', a + b) ∈ qi.Ring-Congruence
  using qi.additive.cong[simplified qi.additive-congruence, OF ca cb] .
then have (a' + b', a + b) ∈ Ji-Cong i using Ji-Cong-eq[OF i] by simp
then have Ji-Class i (a' + b') = Ji-Class i (a + b)
  using a'R b'R ab(1) ab(2) by (intro Ji-Class-cong[OF i]) auto
then show C = Ji-Class i (a + b) using Ceq by simp
qed
finally show ncanon (a + b) i = prod.Padd (ncanon a) (ncanon b) i
  using lhs by simp
qed
next
fix a b assume ab: a ∈ R b ∈ R
show ncanon (a · b) = prod.Pmult (ncanon a) (ncanon b)
proof (rule prod.Pcarrier-eqI)
  show ncanon (a · b) ∈ prod.Pcarrier using ab by (simp add: ncanon-closed)
  show prod.Pmult (ncanon a) (ncanon b) ∈ prod.Pcarrier
    using ab by (simp add: ncanon-closed)
next
fix i assume i: i ∈ S
interpret qi: quotient-ring J i R (+) (·) 0 1 by (rule ideals[OF i])
have lhs: ncanon (a · b) i = Ji-Class i (a · b)
  using ab i by (simp add: ncanon-apply ncaset-def)
have rhs: prod.Pmult (ncanon a) (ncanon b) i =
  quot-mult i (Ji-Class i a) (Ji-Class i b)
  using ab i by (simp add: ncanon-apply prod.Pmult-apply ncaset-def)
also have ... = Ji-Class i (a · b)
  unfolding quot-mult-def
proof (rule the-equality)
  show ∃ a' ∈ Ji-Class i a. ∃ b' ∈ Ji-Class i b. Ji-Class i (a · b) = Ji-Class i (a' · b')
  proof (intro bexI)
    show a ∈ Ji-Class i a using ab(1) unfolding Ji-Class-def Ji-Cong-def by
auto
    show b ∈ Ji-Class i b using ab(2) unfolding Ji-Class-def Ji-Cong-def by auto
  qed simp
next

```

```

fix C assume  $\exists a' \in \text{Ji-Class } i \ a. \exists b' \in \text{Ji-Class } i \ b. C = \text{Ji-Class } i \ (a' \cdot b')$ 
then obtain  $a' b'$  where  $a' A: a' \in \text{Ji-Class } i \ a$  and  $b' B: b' \in \text{Ji-Class } i \ b$ 
  and  $Ceq: C = \text{Ji-Class } i \ (a' \cdot b')$  by auto
have  $a'R: a' \in R$  using  $a'A$  unfolding Ji-Class-def by auto
have  $b'R: b' \in R$  using  $b'B$  unfolding Ji-Class-def by auto
have  $ca: (a', a) \in \text{qi.Ring-Congruence}$ 
proof -
  have  $a - a' \in J \ i$  using  $a'A \ ab(1)$  unfolding Ji-Class-def Ji-Cong-def by
auto
  then have  $-(a - a') \in J \ i$ 
    using  $\text{qi.additive.submonoid-inverse-closed } ab(1) \ a'R$  by simp
  then have  $a' - a \in J \ i$ 
    using  $ab(1) \ a'R$  by (simp add:  $\text{additive.inverse-composition-commute additive.commutative}$ )
  then show ?thesis using  $a'R \ ab(1)$  unfolding  $\text{qi.Ring-Congruence-def}$  by
auto
qed
have  $cb: (b', b) \in \text{qi.Ring-Congruence}$ 
proof -
  have  $b - b' \in J \ i$  using  $b'B \ ab(2)$  unfolding Ji-Class-def Ji-Cong-def by
auto
  then have  $-(b - b') \in J \ i$ 
    using  $\text{qi.additive.submonoid-inverse-closed } ab(2) \ b'R$  by simp
  then have  $b' - b \in J \ i$ 
    using  $ab(2) \ b'R$  by (simp add:  $\text{additive.inverse-composition-commute additive.commutative}$ )
  then show ?thesis using  $b'R \ ab(2)$  unfolding  $\text{qi.Ring-Congruence-def}$  by
auto
qed
have  $(a' \cdot b', a \cdot b) \in \text{qi.Ring-Congruence}$ 
  using  $ca \ cb$  by (rule  $\text{qi.multiplicative-congruence}$ )
then have  $(a' \cdot b', a \cdot b) \in \text{Ji-Cong } i$  using  $\text{Ji-Cong-eq}[OF \ i]$  by simp
then have  $\text{Ji-Class } i \ (a' \cdot b') = \text{Ji-Class } i \ (a \cdot b)$ 
  using  $a'R \ b'R \ ab(1) \ ab(2)$  by (intro  $\text{Ji-Class-cong}[OF \ i]$ ) auto
then show  $C = \text{Ji-Class } i \ (a \cdot b)$  using  $Ceq$  by simp
qed
finally show  $\text{ncanon } (a \cdot b) \ i = \text{prod.Pmult } (\text{ncanon } a) (\text{ncanon } b) \ i$ 
  using  $lhs$  by simp
qed
next
show  $\text{ncanon } 0 = \text{prod.Pzero}$ 
proof (rule  $\text{prod.Pcarrier-eqI}$ )
  show  $\text{ncanon } 0 \in \text{prod.Pcarrier}$  by (simp add:  $\text{ncanon-closed}$ )
  show  $\text{prod.Pzero} \in \text{prod.Pcarrier}$  by simp
next
fix i assume  $i: i \in S$ 
show  $\text{ncanon } 0 \ i = \text{prod.Pzero } i$ 
  using  $i$  by (simp add:  $\text{ncanon-apply prod.Pzero-apply quot-zero-def ncaset-def}$ )
qed
next

```

```

show ncanon 1 = prod.Pone
proof (rule prod.Pcarrier-eqI)
  show ncanon 1 ∈ prod.Pcarrier by (simp add: ncanon-closed)
  show prod.Pone ∈ prod.Pcarrier by simp
next
fix i assume i: i ∈ S
show ncanon 1 i = prod.Pone i
  using i by (simp add: ncanon-apply prod.Pone-apply quot-one-def ncset-def)
qed
qed

interpretation ncanon: ring-homomorphism ncanon R (+) (·) 0 1
  prod.Pcarrier prod.Padd prod.Pmult prod.Pzero prod.Pone
  by (rule ncanon-hom)

interpretation ncanon: ring-homomorphism-fundamental ncanon R (+) (·) 0 1
  prod.Pcarrier prod.Padd prod.Pmult prod.Pzero prod.Pone
  by unfold-locales

```

65.3 Kernel and surjectivity

The kernel of the canonical projection is $\bigcap (J \text{ ' } S)$.

theorem *ncanon-kernel*: $\text{ncanon.additive.Ker} = \bigcap (J \text{ ' } S)$

proof (rule set-eqI)

fix a

show $a \in \text{ncanon.additive.Ker} \longleftrightarrow a \in \bigcap (J \text{ ' } S)$

proof

assume ker: $a \in \text{ncanon.additive.Ker}$

then have aR: $a \in R$ and eq: $\text{ncanon } a = \text{prod.Pzero}$

unfolding *ncanon.additive.Ker-def* by auto

show $a \in \bigcap (J \text{ ' } S)$

proof (rule InterI)

fix X assume X ∈ J ' S

then obtain i where i: $i \in S$ and Xi: $X = J \text{ } i$ by auto

interpret qi: quotient-ring J i R (+) (·) 0 1 by (rule ideals[OF i])

have ncanon a i = prod.Pzero i using eq by simp

then have cls: *Ji-Class* i a = *Ji-Class* i 0

using i aR by (simp add: ncanon-apply prod.Pzero-apply ncset-def quot-zero-def)

have a ∈ *Ji-Class* i a

using aR qi.additive.sub-unit-closed unfolding *Ji-Class-def* *Ji-Cong-def* by

simp

then have a ∈ *Ji-Class* i 0 using cls by simp

then have (0, a) ∈ *Ji-Cong* i unfolding *Ji-Class-def* by simp

then have (a, 0) ∈ *Ji-Cong* i by (rule *Ji-Cong-sym*[OF i])

then have $a - 0 \in J \text{ } i$ using aR unfolding *Ji-Cong-def* by simp

then have a ∈ J i using aR by simp

then show a ∈ X using Xi by simp

qed

next

```

assume aInt:  $a \in \bigcap (J \text{ ' } S)$ 
then have aI:  $\bigwedge i. i \in S \implies a \in J \ i$  by auto
have aR:  $a \in R$ 
proof -
  obtain i where  $i \in S$  using nonempty by blast
  then interpret qi: quotient-ring  $J \ i \ R \ (+) \ (\cdot) \ 0 \ 1$  by (rule ideals)
  show  $a \in R$  using aI[OF  $\langle i \in S \rangle$ ] qi.additive.subset by blast
qed
have ncanon  $a = \text{prod.Pzero}$ 
proof (rule prod.Pcarrier-eqI)
  show ncanon  $a \in \text{prod.Pcarrier}$  by (rule ncanon-closed[OF aR])
  show prod.Pzero  $\in \text{prod.Pcarrier}$  by simp
next
fix i assume i:  $i \in S$ 
interpret qi: quotient-ring  $J \ i \ R \ (+) \ (\cdot) \ 0 \ 1$  by (rule ideals[OF i])
have  $a \in J \ i$  using aI[OF i] .
then have  $a - 0 \in J \ i$  using aR by simp
then have cong:  $(a, 0) \in \text{Ji-Cong } i$  using aR unfolding Ji-Cong-def by auto
then have Ji-Class  $i \ a = \text{Ji-Class } i \ 0$ 
  by (intro Ji-Class-cong[OF i aR - cong]) simp
then show ncanon  $a \ i = \text{prod.Pzero } i$ 
  using i aR by (simp add: ncanon-apply prod.Pzero-apply ncaset-def quot-zero-def)
qed
then show  $a \in \text{ncanon.additive.Ker}$ 
  unfolding ncanon.additive.Ker-def using aR by auto
qed
qed

```

The *Ji-Class* version of $\llbracket \text{ideal-in-comm-ring } ?I \ ?R \ ?\text{addition} \ ?\text{multiplication} \ ?\text{zero} \ ?\text{unit}; \ ?x \in ?R; \ ?y \in ?R; \ ?i \in ?I; \ ?j \in ?R; \ ?\text{addition} \ ?i \ ?j = ?\text{unit} \rrbracket \implies \text{Ideal.coset } ?I \ ?R \ ?\text{addition} \ ?\text{zero} \ (? \text{addition} \ (? \text{multiplication} \ ?x \ ?j) \ (? \text{multiplication} \ ?y \ ?i)) = \text{Ideal.coset } ?I \ ?R \ ?\text{addition} \ ?\text{zero} \ ?x$: if $m + n = 1$ with $m \in J \ i$, then $x \cdot n + y \cdot m$ and x have the same class modulo $J \ i$.

lemma *Ji-Class-comaximal*:

```

assumes i:  $i \in S$  and x:  $x \in R$  and y:  $y \in R$ 
  and m:  $m \in J \ i$  and nR:  $n \in R$  and mn:  $m + n = 1$ 
shows Ji-Class  $i \ (x \cdot n + y \cdot m) = \text{Ji-Class } i \ x$ 
proof -
  interpret qi: quotient-ring  $J \ i \ R \ (+) \ (\cdot) \ 0 \ 1$  by (rule ideals[OF i])
  interpret Ji: ideal-in-comm-ring  $J \ i \ R \ (+) \ (\cdot) \ 0 \ 1$  by (rule Ji-ideal-comm[OF i])
  have bridge:  $\bigwedge c. c \in R \implies \text{Ji-Class } i \ c = \text{qi.coset } c$ 
    using qi.additive.symmetric by (auto simp: Ji-Class-def Ji-Cong-eq i qi.coset-def)
  have mR:  $m \in R$  using m qi.additive.subset by blast
  have xnymR:  $x \cdot n + y \cdot m \in R$  using x y mR nR by simp
  have qi.coset  $(x \cdot n + y \cdot m) = \text{qi.coset } x$ 
    using Ji.coset-comaximal[OF x y m nR mn] by (simp add: qi.coset-def)
  then show ?thesis using bridge[OF xnymR] bridge[OF x] by simp
qed

```

Surjectivity of the canonical map. We first prove, by induction on a nonempty

finite subset $T \subseteq S$, that any family of target classes over T is realised simultaneously by a single element of R . The inductive step splits off one index k and combines a solution over the rest via the comaximality of $J \ k$ with $\bigcap (J \ ' \ F)$ ($\llbracket ?i \in S; ?T \subseteq S; ?T \neq \{\}; ?i \notin ?T; \bigwedge j. j \in ?T \implies \exists a \in J \ ?i. \exists b \in J \ j. a + b = \mathbf{1} \rrbracket \implies \exists a \in J \ ?i. \exists b \in \bigcap (J \ ' \ ?T). a + b = \mathbf{1}$).

lemma *partial-solution*:

assumes $T \subseteq S$ **and** $T \neq \{\}$ **and** $\bigwedge i. i \in T \implies x \ i \in R$

shows $\exists a \in R. \forall i \in T. \text{Ji-Class } i \ a = \text{Ji-Class } i \ (x \ i)$

proof –

have $\text{fin}T$: *finite* T **using** *assms*(1) *fin finite-subset* **by** *blast*

have $T \neq \{\} \implies T \subseteq S \implies (\forall i \in T. x \ i \in R) \implies$

$(\exists a \in R. \forall i \in T. \text{Ji-Class } i \ a = \text{Ji-Class } i \ (x \ i))$

using $\text{fin}T$

proof (*induction* T *rule*: *finite-induct*)

case *empty*

then show $?case$ **by** *simp*

next

case (*insert* $k \ F$)

have kS : $k \in S$ **using** *insert.prem*s **by** *auto*

have xkR : $x \ k \in R$ **using** *insert.prem*s **by** *simp*

show $?case$

proof (*cases* $F = \{\}$)

case *True*

— Singleton: $x \ k$ itself realises the single component.

have $\forall i \in \text{insert } k \ F. \text{Ji-Class } i \ (x \ k) = \text{Ji-Class } i \ (x \ i)$ **using** *True* **by** *simp*

then show $?thesis$ **using** xkR **by** *blast*

next

case Fne : *False*

have FS : $F \subseteq S$ **using** *insert.prem*s **by** *auto*

have kF : $k \notin F$ **by** (*rule* *insert.hyps*)

— Solve over F by the induction hypothesis.

obtain y **where** yR : $y \in R$ **and** yF : $\bigwedge i. i \in F \implies \text{Ji-Class } i \ y = \text{Ji-Class } i \ (x$

$i)$

using *insert.IH* $Fne \ FS$ *insert.prem*s(3) **by** *auto*

— Comaximality of $J \ k$ with $\bigcap (J \ ' \ F)$.

have $\exists m \in J \ k. \exists n \in \bigcap (J \ ' \ F). m + n = \mathbf{1}$

proof (*rule* *comaximal-with-Inter*[*OF* $kS \ FS \ Fne \ kF$])

fix j **assume** $j: j \in F$

then have $j \in S$ **using** FS **by** *auto*

then show $\exists a \in J \ k. \exists b \in J \ j. a + b = \mathbf{1}$

using *pairwise*[*OF* kS] $kF \ j$ **by** *auto*

qed

then obtain $m \ n$ **where** m : $m \in J \ k$ **and** n : $n \in \bigcap (J \ ' \ F)$ **and** mn : $m + n$

$= \mathbf{1}$ **by** *blast*

have mR : $m \in R$

proof –

interpret Jk : *Ideal* $J \ k \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}$ **by** (*rule* *Ji-ideal*[*OF* kS])

show $?thesis$ **using** $m \ Jk$.*additive.subset* **by** *blast*

qed

```

have nR:  $n \in R$ 
proof -
  obtain j where j:  $j \in F$  using Fne by blast
  then interpret Jj: Ideal J j R (+) ( $\cdot$ ) 0 1 using Ji-ideal FS by blast
  have  $n \in J j$  using n j by blast
  then show ?thesis using Jj.additive.subset by blast
qed
define a where  $a = x k \cdot n + y \cdot m$ 
have aR:  $a \in R$  unfolding a-def using xkR yR mR nR by simp
have  $\forall i \in \text{insert } k F. \text{Ji-Class } i a = \text{Ji-Class } i (x i)$ 
proof
  fix i assume  $i \in \text{insert } k F$ 
  then consider  $i = k \mid i \in F$  by auto
  then show  $\text{Ji-Class } i a = \text{Ji-Class } i (x i)$ 
  proof cases
    case 1
    have  $\text{Ji-Class } k a = \text{Ji-Class } k (x k)$ 
    unfolding a-def by (rule Ji-Class-comaximal[OF kS xkR yR m nR mn])
    then show ?thesis using 1 by simp
  next
    case 2
    have iS:  $i \in S$  using 2 FS by auto
    have nJi:  $n \in J i$  using n 2 by blast
    have nm:  $n + m = 1$  using mn mR nR by (simp add: additive.commutative)
    have  $\text{Ji-Class } i (y \cdot m + x k \cdot n) = \text{Ji-Class } i y$ 
    by (rule Ji-Class-comaximal[OF iS yR xkR nJi mR nm])
    moreover have  $y \cdot m + x k \cdot n = a$ 
    unfolding a-def using xkR yR mR nR by (simp add: additive.commutative)
    ultimately have  $\text{Ji-Class } i a = \text{Ji-Class } i y$  by simp
    then show ?thesis using yF[OF 2] by simp
  qed
qed
then show ?thesis using aR by blast
qed
then show ?thesis using assms by blast
qed

```

```

theorem ncanon-surjective:  $\text{ncanon} \text{ ' } R = \text{prod.Pcarrier}$ 
proof
  show  $\text{ncanon} \text{ ' } R \subseteq \text{prod.Pcarrier}$  using ncanon-closed by blast
next
  show  $\text{prod.Pcarrier} \subseteq \text{ncanon} \text{ ' } R$ 
  proof
    fix f assume  $f: f \in \text{prod.Pcarrier}$ 
    — Choose a representative  $x i \in R$  of each component  $f i$ .
    have  $\forall i \in S. \exists xi \in R. f i = \text{Ji-Class } i xi$ 
    proof
      fix i assume  $i: i \in S$ 
      have  $f i \in \text{quot-carrier } i$  using f i by (simp add: prod.Pcarrier-component)
    
```

```

    then show  $\exists xi \in R. f\ i = Ji\text{-}Class\ i\ xi$  unfolding quot-carrier-def by auto
qed
then obtain  $x$  where  $xR: \bigwedge i. i \in S \implies x\ i \in R$ 
  and  $fx: \bigwedge i. i \in S \implies f\ i = Ji\text{-}Class\ i\ (x\ i)$  by metis
have  $\exists a \in R. \forall i \in S. Ji\text{-}Class\ i\ a = Ji\text{-}Class\ i\ (x\ i)$ 
  by (rule partial-solution[OF subset-refl nonempty]) (rule xR)
then obtain  $a$  where  $aR: a \in R$  and  $aeq: \bigwedge i. i \in S \implies Ji\text{-}Class\ i\ a = Ji\text{-}Class$ 
 $i\ (x\ i)$ 
  by blast
have  $ncanon\ a = f$ 
proof (rule prod.Pcarrier-eqI)
  show  $ncanon\ a \in prod.Pcarrier$  by (rule ncanon-closed[OF aR])
  show  $f \in prod.Pcarrier$  by (rule f)
next
  fix  $i$  assume  $i: i \in S$ 
  have  $ncanon\ a\ i = Ji\text{-}Class\ i\ a$  using  $aR\ i$  by (simp add: ncanon-apply ncaset-def)
  also have  $\dots = Ji\text{-}Class\ i\ (x\ i)$  using  $aeq[OF\ i]$  .
  also have  $\dots = f\ i$  using  $fx[OF\ i]$  by simp
  finally show  $ncanon\ a\ i = f\ i$  .
qed
then show  $f \in ncanon\ 'R$  using  $aR$  by blast
qed
qed

```

65.4 The n-ary Chinese Remainder Theorem

theorem *chinese-remainder-general*:

```

( $R / (subgroup\text{-}of\text{-}additive\text{-}group\text{-}of\text{-}ring.Ring\text{-}Congruence\ (\bigcap\ (J\ 'S))\ R\ (+)\ \mathbf{0})$ ,
   $ncanon.kernel.additive.quotient\text{-}composition$ ,  $ncanon.kernel.multiplicative.quotient\text{-}composition$ ,
   $ncanon.kernel.additive.Class\ \mathbf{0}$ ,  $ncanon.kernel.additive.Class\ \mathbf{1}$ )
 $\cong_R (prod.Pcarrier, prod.Padd, prod.Pmult, prod.Pzero, prod.Pone)$ 
proof –
  have  $ker: ncanon.additive.Ker = \bigcap\ (J\ 'S)$  by (rule ncanon-kernel)
  have ring-isomorphism  $ncanon.additive.induced$ 
    ( $R / (subgroup\text{-}of\text{-}additive\text{-}group\text{-}of\text{-}ring.Ring\text{-}Congruence\ (\bigcap\ (J\ 'S))\ R\ (+)\ \mathbf{0})$ )
     $ncanon.kernel.additive.quotient\text{-}composition\ ncanon.kernel.multiplicative.quotient\text{-}composition$ 
    ( $ncanon.kernel.additive.Class\ \mathbf{0}$ ) ( $ncanon.kernel.additive.Class\ \mathbf{1}$ )
     $prod.Pcarrier\ prod.Padd\ prod.Pmult\ prod.Pzero\ prod.Pone$ 
  unfolding  $ker[symmetric]$ 
proof (rule ring-isomorphism.intro)
  show ring-homomorphism  $ncanon.additive.induced$ 
    ( $R / (subgroup\text{-}of\text{-}additive\text{-}group\text{-}of\text{-}ring.Ring\text{-}Congruence\ ncanon.additive.Ker$ 
 $R\ (+)\ \mathbf{0})$ )
     $ncanon.kernel.additive.quotient\text{-}composition\ ncanon.kernel.multiplicative.quotient\text{-}composition$ 
    ( $ncanon.kernel.additive.Class\ \mathbf{0}$ ) ( $ncanon.kernel.additive.Class\ \mathbf{1}$ )
     $prod.Pcarrier\ prod.Padd\ prod.Pmult\ prod.Pzero\ prod.Pone$ 
  by (rule ncanon.induced.ring-homomorphism-axioms)
  show bijective-map  $ncanon.additive.induced$ 
    ( $R / (subgroup\text{-}of\text{-}additive\text{-}group\text{-}of\text{-}ring.Ring\text{-}Congruence\ ncanon.additive.Ker$ 
 $R\ (+)\ \mathbf{0})$ )

```

```

      prod.Pcarrier
    proof
      show bij-betw ncanon.additive.induced
        (R / (subgroup-of-additive-group-of-ring.Ring-Congruence ncanon.additive.Ker
R (+) 0))
      prod.Pcarrier
      using ncanon.additive.induced-image ncanon.additive.induced-inj-on ncanon-surjective
      by (simp add: bij-betw-def)
    qed
  qed
  then show ?thesis
  by (auto simp: isomorphic-as-rings-def)
qed

end

end

```

66 The field of fractions of an integral domain

```

theory Field-Of-Fractions
  imports Divisibility-Theory
begin

```

Given an integral domain R we construct its field of fractions as a carrier-set object, mirroring the elementary construction of $\mathbb{Q}$ from $\mathbb{Z}$: fractions are pairs (a, b) with $b \neq 0$, identified when they cross-multiply equal, and the field operations are the usual ones on numerators and denominators. We reuse the *Equivalence* machinery of *Set-Theory* for the quotient, then interpret *Field* on it. The canonical map $a \mapsto a / 1$ is an injective ring homomorphism, giving the universal embedding of the domain into a field.

```

context integral-domain
begin

```

66.1 Fractions and their equivalence

A fraction is a pair of ring elements with nonzero denominator.

```

definition frac-set :: ('a  $\times$  'a) set
  where frac-set = {(a, b). a  $\in$  R  $\wedge$  b  $\in$  R  $\wedge$  b  $\neq$  0}

```

```

lemma frac-setI [intro]:  $\llbracket a \in R; b \in R; b \neq 0 \rrbracket \implies (a, b) \in \text{frac-set}$ 
  unfolding frac-set-def by blast

```

```

lemma frac-setD [dest]:  $(a, b) \in \text{frac-set} \implies a \in R \wedge b \in R \wedge b \neq 0$ 
  unfolding frac-set-def by blast

```

The denominator of a fraction is a nonzero ring element; the product of two nonzero denominators is again nonzero (no zero divisors).

lemma *frac-mult-denom-nonzero* [intro]:
 $\llbracket b \in R; d \in R; b \neq 0; d \neq 0 \rrbracket \implies b \cdot d \neq 0$
using *no-zero-divisors* **by** *blast*

Cross-multiplication equivalence: $(a, b) \sim (c, d)$ iff $a \cdot d = c \cdot b$.

definition *frac-rel* :: $((a \times 'a) \times (c \times 'a))$ set
where *frac-rel* = $\{((a, b), (c, d)). (a, b) \in \text{frac-set} \wedge (c, d) \in \text{frac-set} \wedge a \cdot d = c \cdot b\}$

lemma *frac-relI* [intro]:
 $\llbracket (a, b) \in \text{frac-set}; (c, d) \in \text{frac-set}; a \cdot d = c \cdot b \rrbracket \implies ((a, b), (c, d)) \in \text{frac-rel}$
unfolding *frac-rel-def* **by** *blast*

lemma *frac-relE*:
assumes $((a, b), (c, d)) \in \text{frac-rel}$
obtains $(a, b) \in \text{frac-set} \ (c, d) \in \text{frac-set} \ a \cdot d = c \cdot b$
using *assms* **unfolding** *frac-rel-def* **by** *blast*

Transitivity is where cancellation — hence the domain hypothesis — is used.

lemma *frac-rel-trans*:
assumes 1: $((a, b), (c, d)) \in \text{frac-rel}$ **and** 2: $((c, d), (e, f)) \in \text{frac-rel}$
shows $((a, b), (e, f)) \in \text{frac-rel}$

proof —

from 1 **obtain** *ab*: $(a, b) \in \text{frac-set}$ **and** *cd*: $(c, d) \in \text{frac-set}$ **and** *e1*: $a \cdot d = c \cdot b$
by (rule *frac-relE*)

from 2 **obtain** *ef*: $(e, f) \in \text{frac-set}$ **and** *e2*: $c \cdot f = e \cdot d$ **by** (rule *frac-relE*)

from *ab cd ef* **have** [*simp*]: $a \in R \ b \in R \ c \in R \ d \in R \ e \in R \ f \in R$
and *dnz*: $d \neq 0$ **by** *auto*

— From $a \cdot d = c \cdot b$ and $c \cdot f = e \cdot d$: multiply and cancel d to reach $a \cdot f = e \cdot b$.

have $d \cdot (a \cdot f) = (a \cdot d) \cdot f$ **by** (*simp add: mult-ac*)

also have $\dots = (c \cdot b) \cdot f$ **by** (*simp add: e1*)

also have $\dots = (c \cdot f) \cdot b$ **by** (*simp add: mult-ac*)

also have $\dots = (e \cdot d) \cdot b$ **by** (*simp add: e2*)

also have $\dots = d \cdot (e \cdot b)$ **by** (*simp add: mult-ac*)

finally have *eq*: $d \cdot (a \cdot f) = d \cdot (e \cdot b)$.

have $a \cdot f = e \cdot b$

by (rule *mult-cancel-left*[**where** $c = d$]) (use *dnz eq* **in** *simp-all*)

with *ab ef* **show** *?thesis* **by** (rule *frac-relI*)

qed

The relation is an equivalence on the set of fractions.

interpretation *frac*: *Equivalence frac-set frac-rel*

proof (rule *equivalenceI*)

show *frac-rel* $\subseteq \text{frac-set} \times \text{frac-set}$ **by** (auto *simp: frac-rel-def*)

next

fix *x* **assume** $x \in \text{frac-set}$

then show $(x, x) \in \text{frac-rel}$ **by** (*cases x*) (auto *simp: frac-rel-def*)

next

fix *x y* **assume** $(x, y) \in \text{frac-rel}$

```

    then show  $(y, x) \in \text{frac-rel}$  by (cases x; cases y) (auto simp: frac-rel-def)
next
  fix x y z assume  $(x, y) \in \text{frac-rel}$   $(y, z) \in \text{frac-rel}$ 
  then show  $(x, z) \in \text{frac-rel}$ 
    by (cases x; cases y; cases z) (blast intro: frac-rel-trans)
qed

```

66.2 The carrier and class equality

The carrier of the field of fractions: the set of equivalence classes.

abbreviation *Fractions* :: $('a \times 'a)$ set set
where *Fractions* $\equiv$ *frac-set* / *frac-rel*

Class equality is cross-multiplication.

lemma *frac-eqI*:
 assumes $(a, b) \in \text{frac-set}$ $(c, d) \in \text{frac-set}$ $a \cdot d = c \cdot b$
 shows *frac.Class* $(a, b) = \text{frac.Class}$ (c, d)
 using *assms* by (intro *frac.Class-eq* *frac-relI*)

lemma *frac-eqD*:
 assumes $(a, b) \in \text{frac-set}$ $(c, d) \in \text{frac-set}$ *frac.Class* $(a, b) = \text{frac.Class}$ (c, d)
 shows $a \cdot d = c \cdot b$
 using *assms* *frac.Class-equivalence* by (blast elim: *frac-relE*)

A class lies in the carrier exactly when it is the class of a fraction.

lemma *Class-in-Fractions* [*intro*, *simp*]: $(a, b) \in \text{frac-set} \implies \text{frac.Class}$ $(a, b) \in$
Fractions
 by (rule *frac.Class-in-Partition*)

lemma *Fractions-cases*:
 assumes $A \in \text{Fractions}$ obtains $a\ b$ where $(a, b) \in \text{frac-set}$ $A = \text{frac.Class}$ (a, b)
 using *assms* *frac.representant-exists* by (metis *surj-pair*)

A class equals the zero-class *Class* $(\mathbf{0}, \mathbf{1})$ iff any (equivalently every) numerator is zero. Later, we take *Class* $(\mathbf{0}, \mathbf{1})$ as the additive unit and *Class* $(\mathbf{1}, \mathbf{1})$ as the multiplicative unit of *Fractions*.

lemma *frac-Class-zero-iff*:
 assumes $ab: (a, b) \in \text{frac-set}$
 shows *frac.Class* $(a, b) = \text{frac.Class}$ $(\mathbf{0}, \mathbf{1}) \longleftrightarrow a = \mathbf{0}$

proof
 from *ab* have *aR*: $a \in R$ and *bR*: $b \in R$ by *auto*
 have *zero-in*: $(\mathbf{0}, \mathbf{1}) \in \text{frac-set}$ by (auto intro!: *frac-setI* *simp*: *nontrivial*)
 assume *frac.Class* $(a, b) = \text{frac.Class}$ $(\mathbf{0}, \mathbf{1})$
 from *frac-eqD*[*OF* *ab zero-in this*] show $a = \mathbf{0}$ using *aR bR* by *simp*
next
 assume $a = \mathbf{0}$
 with *ab* show *frac.Class* $(a, b) = \text{frac.Class}$ $(\mathbf{0}, \mathbf{1})$
 by (intro *frac-eqI*) (auto intro!: *frac-setI* *simp*: *nontrivial*)
qed

66.3 The operations on classes

We define addition, multiplication, negation, and inversion on classes by picking any representative and applying the elementary fraction formulas. For each operation we prove a congruence lemma (well-definedness under the equivalence) and a *Class*-formula giving the computation on a canonical representative.

definition *frac-add* :: (*'a* × *'a*) *set* ⇒ (*'a* × *'a*) *set* ⇒ (*'a* × *'a*) *set*

(**infixl** $\langle [+]\rangle$ 65)

where *A* $\langle [+]\rangle$ *B* =

(*let* (*a*, *b*) = *SOME* *p*. *p* ∈ *A* ∧ *p* ∈ frac-set;

(*c*, *d*) = *SOME* *p*. *p* ∈ *B* ∧ *p* ∈ frac-set

in frac.Class (*a* · *d* + *c* · *b*, *b* · *d*))

definition *frac-mult* :: (*'a* × *'a*) *set* ⇒ (*'a* × *'a*) *set* ⇒ (*'a* × *'a*) *set*

(**infixl** $\langle [\cdot]\rangle$ 70)

where *A* $\langle [\cdot]\rangle$ *B* =

(*let* (*a*, *b*) = *SOME* *p*. *p* ∈ *A* ∧ *p* ∈ frac-set;

(*c*, *d*) = *SOME* *p*. *p* ∈ *B* ∧ *p* ∈ frac-set

in frac.Class (*a* · *c*, *b* · *d*))

definition *frac-neg* :: (*'a* × *'a*) *set* ⇒ (*'a* × *'a*) *set* ($\langle [-]\rangle \rightarrow [80] 80$)

where $\langle [-]\rangle$ *A* = (*let* (*a*, *b*) = *SOME* *p*. *p* ∈ *A* ∧ *p* ∈ frac-set *in* frac.Class (*-* *a*, *b*))

Inversion is only used on nonzero classes.

definition *frac-inv* :: (*'a* × *'a*) *set* ⇒ (*'a* × *'a*) *set*

where *frac-inv* *A* = (*let* (*a*, *b*) = *SOME* *p*. *p* ∈ *A* ∧ *p* ∈ frac-set *in* frac.Class (*b*, *a*))

The zero and unit of the fraction field.

abbreviation *frac-zero* :: (*'a* × *'a*) *set* ($\langle [0]\rangle$)

where $[0] \equiv$ frac.Class (**0**, **1**)

abbreviation *frac-one* :: (*'a* × *'a*) *set* ($\langle [1]\rangle$)

where $[1] \equiv$ frac.Class (**1**, **1**)

lemma *one-in-frac-set*: (**1**, **1**) ∈ frac-set

by (rule frac-setI) (auto simp: nontrivial)

lemma *zero-in-frac-set*: (**0**, **1**) ∈ frac-set

by (rule frac-setI) (auto simp: nontrivial)

lemma *frac-zero-in* [*intro*, *simp*]: $[0] \in$ Fractions

using zero-in-frac-set **by** (rule Class-in-Fractions)

lemma *frac-one-in* [*intro*, *simp*]: $[1] \in$ Fractions

using one-in-frac-set **by** (rule Class-in-Fractions)

lemma *frac-one-neg-zero*: $[1] \neq [0]$

proof

```

    assume eq: [1] = [0]
    from frac-eqD[OF one-in-frac-set zero-in-frac-set eq] show False using nontrivial
  by simp
qed

```

66.3.1 Representative selection

Any class of a fraction contains that fraction.

```

lemma rep-in-frac-set:
  assumes (a, b) ∈ frac-set
  shows (SOME p. p ∈ frac.Class (a, b) ∧ p ∈ frac-set) ∈ frac.Class (a, b) ∧
        (SOME p. p ∈ frac.Class (a, b) ∧ p ∈ frac-set) ∈ frac-set
proof (rule someI-ex)
  from assms have (a, b) ∈ frac.Class (a, b) by (intro frac.Class-self)
  with assms show ∃ p. p ∈ frac.Class (a, b) ∧ p ∈ frac-set by blast
qed

```

66.3.2 Congruence: well-definedness under the equivalence

```

lemma frac-add-cong:
  assumes r1: ((a, b), (a', b')) ∈ frac-rel and r2: ((c, d), (c', d')) ∈ frac-rel
  shows ((a · d + c · b, b · d), (a' · d' + c' · b', b' · d')) ∈ frac-rel
proof -
  from r1 obtain ab: (a, b) ∈ frac-set and ab': (a', b') ∈ frac-set and e1: a · b' =
a' · b
  by (rule frac-relE)
  from r2 obtain cd: (c, d) ∈ frac-set and cd': (c', d') ∈ frac-set and e2: c · d' =
c' · d
  by (rule frac-relE)
  from ab ab' cd cd' have [simp]:
    a ∈ R b ∈ R a' ∈ R b' ∈ R c ∈ R d ∈ R c' ∈ R d' ∈ R
    and bnz: b ≠ 0 and b'nz: b' ≠ 0 and dnz: d ≠ 0 and d'nz: d' ≠ 0 by auto
  have step1: (a · d + c · b) · (b' · d') = (a · b') · (d · d') + (c · d') · (b · b')
  proof -
    have (a · d + c · b) · (b' · d') = (a · d) · (b' · d') + (c · b) · (b' · d')
    by (simp add: distributive)
    also have (a · d) · (b' · d') = (a · b') · (d · d') by (simp add: mult-ac)
    also have (c · b) · (b' · d') = (c · d') · (b · b') by (simp add: mult-ac)
    finally show ?thesis .
  qed
  also have (a · b') · (d · d') + (c · d') · (b · b') = (a' · b) · (d · d') + (c' · d) · (b ·
b')
  by (simp add: e1 e2)
  also have ... = (a' · d' + c' · b') · (b · d)
  proof -
    have (a' · d' + c' · b') · (b · d) = (a' · d') · (b · d) + (c' · b') · (b · d)
    by (simp add: distributive)
    also have (a' · d') · (b · d) = (a' · b) · (d · d') by (simp add: mult-ac)
    also have (c' · b') · (b · d) = (c' · d) · (b · b') by (simp add: mult-ac)

```


finally show ?thesis by simp
 qed
 finally have eq: $(a \cdot d + c \cdot b) \cdot (b' \cdot d') = (a' \cdot d' + c' \cdot b') \cdot (b \cdot d)$.
 have s1: $(a \cdot d + c \cdot b, b \cdot d) \in \text{frac-set}$
 using bnz dnz frac-mult-denom-nonzero[of b d] by (auto intro!: frac-setI)
 have s2: $(a' \cdot d' + c' \cdot b', b' \cdot d') \in \text{frac-set}$
 using b'nz d'nz frac-mult-denom-nonzero[of b' d'] by (auto intro!: frac-setI)
 from s1 s2 eq show ?thesis by (rule frac-relI)
 qed

lemma frac-mult-cong:

assumes r1: $((a, b), (a', b')) \in \text{frac-rel}$ and r2: $((c, d), (c', d')) \in \text{frac-rel}$
 shows $((a \cdot c, b \cdot d), (a' \cdot c', b' \cdot d')) \in \text{frac-rel}$
 proof -
 from r1 obtain ab: $(a, b) \in \text{frac-set}$ and ab': $(a', b') \in \text{frac-set}$ and e1: $a \cdot b' = a' \cdot b$
 by (rule frac-relE)
 from r2 obtain cd: $(c, d) \in \text{frac-set}$ and cd': $(c', d') \in \text{frac-set}$ and e2: $c \cdot d' = c' \cdot d$
 by (rule frac-relE)
 from ab ab' cd cd' have [simp]:
 $a \in R \ b \in R \ a' \in R \ b' \in R \ c \in R \ d \in R \ c' \in R \ d' \in R$
 and bnz: $b \neq 0$ and b'nz: $b' \neq 0$ and dnz: $d \neq 0$ and d'nz: $d' \neq 0$ by auto
 have $(a \cdot c) \cdot (b' \cdot d') = (a \cdot b') \cdot (c \cdot d')$ by (simp add: mult-ac)
 also have $\dots = (a' \cdot b) \cdot (c' \cdot d)$ by (simp add: e1 e2)
 also have $\dots = (a' \cdot c') \cdot (b \cdot d)$ by (simp add: mult-ac)
 finally have eq: $(a \cdot c) \cdot (b' \cdot d') = (a' \cdot c') \cdot (b \cdot d)$.
 have s1: $(a \cdot c, b \cdot d) \in \text{frac-set}$
 using bnz dnz frac-mult-denom-nonzero[of b d] by (auto intro!: frac-setI)
 have s2: $(a' \cdot c', b' \cdot d') \in \text{frac-set}$
 using b'nz d'nz frac-mult-denom-nonzero[of b' d'] by (auto intro!: frac-setI)
 from s1 s2 eq show ?thesis by (rule frac-relI)
 qed

lemma frac-neg-cong:

assumes r: $((a, b), (a', b')) \in \text{frac-rel}$
 shows $((-a, b), (-a', b')) \in \text{frac-rel}$
 proof -
 from r obtain ab: $(a, b) \in \text{frac-set}$ and ab': $(a', b') \in \text{frac-set}$ and e: $a \cdot b' = a' \cdot b$
 by (rule frac-relE)
 from ab ab' have [simp]: $a \in R \ b \in R \ a' \in R \ b' \in R$
 and bnz: $b \neq 0$ and b'nz: $b' \neq 0$ by auto
 have $(-a) \cdot b' = -(a \cdot b')$ by (simp add: left-minus)
 also have $\dots = -(a' \cdot b)$ by (simp add: e)
 also have $\dots = (-a') \cdot b$ by (simp add: left-minus)
 finally have eq: $(-a) \cdot b' = (-a') \cdot b$.
 have s1: $(-a, b) \in \text{frac-set}$ using bnz by (auto intro!: frac-setI)
 have s2: $(-a', b') \in \text{frac-set}$ using b'nz by (auto intro!: frac-setI)
 from s1 s2 eq show ?thesis by (rule frac-relI)
 qed

qed

Inversion: well-defined provided the numerators are nonzero (i.e. the class is nonzero).

lemma *frac-inv-cong*:

assumes $r: ((a, b), (a', b')) \in \text{frac-rel}$ **and** $\text{anz}: a \neq 0$
shows $((b, a), (b', a')) \in \text{frac-rel}$

proof –

from r **obtain** $ab: (a, b) \in \text{frac-set}$ **and** $ab': (a', b') \in \text{frac-set}$ **and** $e: a \cdot b' = a' \cdot b$

by (*rule frac-relE*)

from ab ab' **have** $[simp]: a \in R \ b \in R \ a' \in R \ b' \in R$

and $\text{bnz}: b \neq 0$ **and** $b'\text{nz}: b' \neq 0$ **by** *auto*

have $a'\text{nz}: a' \neq 0$

proof

assume $a' = 0$

with e **have** $a \cdot b' = 0$ **by** *simp*

then **have** $a = 0 \vee b' = 0$ **using** *no-zero-divisors* **by** *simp*

with anz $b'\text{nz}$ **show** *False* **by** *auto*

qed

have $s1: (b, a) \in \text{frac-set}$ **using** anz **by** (*auto intro!: frac-setI*)

have $s2: (b', a') \in \text{frac-set}$ **using** $a'\text{nz}$ **by** (*auto intro!: frac-setI*)

have $b \cdot a' = a' \cdot b$ **by** (*simp add: mult-ac*)

also **have** $\dots = a \cdot b'$ **by** (*simp add: e*)

also **have** $\dots = b' \cdot a$ **by** (*simp add: mult-ac*)

finally **have** $b \cdot a' = b' \cdot a$.

with $s1$ $s2$ **show** *?thesis* **by** (*rule frac-relI*)

qed

66.3.3 The Class-formulas

Key auxiliary: if a class is chosen from a fraction, the SOME-representative is equivalent to the given fraction.

lemma *some-rep-equiv*:

assumes $ab: (a, b) \in \text{frac-set}$

defines $p \equiv \text{SOME } p. p \in \text{frac.Class } (a, b) \wedge p \in \text{frac-set}$

shows $(p, (a, b)) \in \text{frac-rel}$ **and** $p \in \text{frac-set}$

proof –

have $\text{rep-prop}: p \in \text{frac.Class } (a, b) \wedge p \in \text{frac-set}$

unfolding $p\text{-def}$ **using** $\text{rep-in-frac-set}[OF\ ab]$ **by** *simp*

then **have** $p \in \text{frac.Class } (a, b)$ **and** $p \in \text{frac-set}$ **by** *auto*

from $\text{frac.ClassD}[OF\ \text{this}(1)\ ab]$ **show** $(p, (a, b)) \in \text{frac-rel}$.

from rep-prop **show** $p \in \text{frac-set}$ **by** *simp*

qed

lemma *frac-add-Class*:

assumes $ab: (a, b) \in \text{frac-set}$ **and** $cd: (c, d) \in \text{frac-set}$

shows $\text{frac.Class } (a, b) \ [+]\ \text{frac.Class } (c, d) = \text{frac.Class } (a \cdot d + c \cdot b, b \cdot d)$

proof –

```

let ?p = SOME p. p ∈ frac.Class (a, b) ∧ p ∈ frac-set
let ?q = SOME p. p ∈ frac.Class (c, d) ∧ p ∈ frac-set
have p-equiv: (?p, (a, b)) ∈ frac-rel using some-rep-equiv(1)[OF ab] .
have q-equiv: (?q, (c, d)) ∈ frac-rel using some-rep-equiv(1)[OF cd] .
obtain a1 b1 where p-eq: ?p = (a1, b1) by force
obtain c1 d1 where q-eq: ?q = (c1, d1) by force
from p-eq p-equiv have ((a1, b1), (a, b)) ∈ frac-rel by simp
moreover from q-eq q-equiv have ((c1, d1), (c, d)) ∈ frac-rel by simp
ultimately have ((a1 · d1 + c1 · b1, b1 · d1), (a · d + c · b, b · d)) ∈ frac-rel
  by (rule frac-add-cong)
then have frac.Class (a1 · d1 + c1 · b1, b1 · d1) = frac.Class (a · d + c · b, b ·
d)
  by (rule frac.Class-eq)
then show ?thesis
  unfolding frac-add-def by (simp add: p-eq q-eq)
qed

```

lemma frac-mult-Class:

```

assumes ab: (a, b) ∈ frac-set and cd: (c, d) ∈ frac-set
shows frac.Class (a, b) [·] frac.Class (c, d) = frac.Class (a · c, b · d)
proof -
let ?p = SOME p. p ∈ frac.Class (a, b) ∧ p ∈ frac-set
let ?q = SOME p. p ∈ frac.Class (c, d) ∧ p ∈ frac-set
have p-equiv: (?p, (a, b)) ∈ frac-rel using some-rep-equiv(1)[OF ab] .
have q-equiv: (?q, (c, d)) ∈ frac-rel using some-rep-equiv(1)[OF cd] .
obtain a1 b1 where p-eq: ?p = (a1, b1) by force
obtain c1 d1 where q-eq: ?q = (c1, d1) by force
from p-eq p-equiv have ((a1, b1), (a, b)) ∈ frac-rel by simp
moreover from q-eq q-equiv have ((c1, d1), (c, d)) ∈ frac-rel by simp
ultimately have ((a1 · c1, b1 · d1), (a · c, b · d)) ∈ frac-rel
  by (rule frac-mult-cong)
then have frac.Class (a1 · c1, b1 · d1) = frac.Class (a · c, b · d)
  by (rule frac.Class-eq)
then show ?thesis
  unfolding frac-mult-def by (simp add: p-eq q-eq)
qed

```

lemma frac-neg-Class:

```

assumes ab: (a, b) ∈ frac-set
shows [-] frac.Class (a, b) = frac.Class (- a, b)
proof -
let ?p = SOME p. p ∈ frac.Class (a, b) ∧ p ∈ frac-set
have p-equiv: (?p, (a, b)) ∈ frac-rel using some-rep-equiv(1)[OF ab] .
obtain a1 b1 where p-eq: ?p = (a1, b1) by force
from p-eq p-equiv have ((a1, b1), (a, b)) ∈ frac-rel by simp
then have ((- a1, b1), (- a, b)) ∈ frac-rel by (rule frac-neg-cong)
then have frac.Class (- a1, b1) = frac.Class (- a, b) by (rule frac.Class-eq)
then show ?thesis unfolding frac-neg-def by (simp add: p-eq)
qed

```

lemma *frac-inv-Class*:

assumes *ab*: $(a, b) \in \text{frac-set}$ **and** *anz*: $a \neq 0$
shows *frac-inv* (*frac.Class* (a, b)) = *frac.Class* (b, a)

proof –

let *?p* = *SOME* *p*. $p \in \text{frac.Class } (a, b) \wedge p \in \text{frac-set}$
have *p-equiv*: $(?p, (a, b)) \in \text{frac-rel}$ **using** *some-rep-equiv*(1)[*OF ab*] .
obtain *a1 b1* **where** *p-eq*: $?p = (a1, b1)$ **by** *force*
from *p-eq* *p-equiv* **have** *r*: $((a1, b1), (a, b)) \in \text{frac-rel}$ **by** *simp*
— Symmetry gives us $((a,b),(a1,b1)) \in \text{frac-rel}$, whose inversion congruence needs
 $a \neq 0$.
from *r* **have** *r'*: $((a, b), (a1, b1)) \in \text{frac-rel}$ **by** (*rule frac.symmetric*)
then **have** $((b, a), (b1, a1)) \in \text{frac-rel}$ **using** *anz* **by** (*rule frac-inv-cong*)
then **have** $((b1, a1), (b, a)) \in \text{frac-rel}$ **by** (*rule frac.symmetric*)
then **have** *frac.Class* $(b1, a1) = \text{frac.Class } (b, a)$ **by** (*rule frac.Class-eq*)
then **show** *?thesis* **unfolding** *frac-inv-def* **by** (*simp add: p-eq*)

qed

66.3.4 Closure under the operations

lemma *frac-add-closed* [*intro, simp*]:

$A \in \text{Fractions} \implies B \in \text{Fractions} \implies A [+] B \in \text{Fractions}$

proof –

assume $A \in \text{Fractions}$ $B \in \text{Fractions}$

then obtain *a b c d*

where *A*: $A = \text{frac.Class } (a, b)$ $(a, b) \in \text{frac-set}$
and *B*: $B = \text{frac.Class } (c, d)$ $(c, d) \in \text{frac-set}$
by (*metis Fractions-cases*)

then have [*simp*]: $b \in R$ $d \in R$ $b \neq 0$ $d \neq 0$ **and** *abR*: $a \in R$ $c \in R$ **by** *auto*
have $(a \cdot d + c \cdot b, b \cdot d) \in \text{frac-set}$ **using** *abR frac-mult-denom-nonzero*[*of b d*] **by** (*auto intro!: frac-setI*)

then show $A [+] B \in \text{Fractions}$ **using** *A B frac-add-Class*[*OF A(2) B(2)*] **by** *auto*

qed

lemma *frac-mult-closed* [*intro, simp*]:

$A \in \text{Fractions} \implies B \in \text{Fractions} \implies A [\cdot] B \in \text{Fractions}$

proof –

assume $A \in \text{Fractions}$ $B \in \text{Fractions}$

then obtain *a b c d*

where *A*: $A = \text{frac.Class } (a, b)$ $(a, b) \in \text{frac-set}$
and *B*: $B = \text{frac.Class } (c, d)$ $(c, d) \in \text{frac-set}$
by (*metis Fractions-cases*)

then have [*simp*]: $b \in R$ $d \in R$ $b \neq 0$ $d \neq 0$ **and** *abR*: $a \in R$ $c \in R$ **by** *auto*
have $(a \cdot c, b \cdot d) \in \text{frac-set}$ **using** *abR frac-mult-denom-nonzero*[*of b d*] **by** (*auto intro!: frac-setI*)

then show $A [\cdot] B \in \text{Fractions}$ **using** *A B frac-mult-Class*[*OF A(2) B(2)*] **by** *auto*

qed

lemma *frac-neg-closed* [*intro, simp*]: $A \in \text{Fractions} \implies [-] A \in \text{Fractions}$

proof –

assume $A \in \text{Fractions}$

```

then obtain  $a\ b$  where  $A$ :  $A = \text{frac.Class } (a, b) \ (a, b) \in \text{frac-set}$  by (metis
Fractions-cases)
then have  $(- \ a, b) \in \text{frac-set}$  by (auto intro!: frac-setI)
then show  $[-] \ A \in \text{Fractions}$  using  $A \ \text{frac-neg-Class}[OF \ A(2)]$  by auto
qed

```

66.4 The field of fractions

We assemble the additive abelian group and the multiplicative monoid as separate top-level lemmas, then compose them via *Ring.intro* to obtain the ring structure. This avoids taking on the (locale-qualified) *Monoid.invertible* obligation mid-interpretation: we discharge it once, at the level of *GroupI*, which takes the raw $\exists v \in G. u \cdot v = 1 \wedge v \cdot u = 1$ form directly.

66.4.1 The additive abelian group of *Fractions*

The additive group. Associativity, unit, invertibility, and commutativity are established on fraction representatives. Invertibility supplies the explicit inverse *Class* $(- \ a, b)$ of *Class* (a, b) .

lemma *Frac-add-group*: *Group Fractions* $([+])$ $[0]$

proof (*rule GroupI*)

— Closure.

fix $A\ B$ **assume** $A \in \text{Fractions}$ $B \in \text{Fractions}$

then show $A \ [+]\ B \in \text{Fractions}$ **by** *simp*

next

show $[0] \in \text{Fractions}$ **by** *simp*

next

— Associativity of $[+]$, on representatives.

fix $A\ B\ C$ **assume** A : $A \in \text{Fractions}$ **and** B : $B \in \text{Fractions}$ **and** C : $C \in \text{Fractions}$

obtain $a\ b$ **where** A' : $A = \text{frac.Class } (a, b) \ (a, b) \in \text{frac-set}$ **using** A **by** (*metis* *Fractions-cases*)

obtain $c\ d$ **where** B' : $B = \text{frac.Class } (c, d) \ (c, d) \in \text{frac-set}$ **using** B **by** (*metis* *Fractions-cases*)

obtain $e\ f$ **where** C' : $C = \text{frac.Class } (e, f) \ (e, f) \in \text{frac-set}$ **using** C **by** (*metis* *Fractions-cases*)

from $A'\ B'\ C'$ **have** $[simp]$:

$a \in R\ b \in R\ c \in R\ d \in R\ e \in R\ f \in R$

$b \neq 0\ d \neq 0\ f \neq 0$ **by** *auto*

have $s1$: $(a \cdot d + c \cdot b, b \cdot d) \in \text{frac-set}$

using *frac-mult-denom-nonzero* $[of \ b \ d]$ **by** (*auto intro!*: *frac-setI*)

have $s2$: $(c \cdot f + e \cdot d, d \cdot f) \in \text{frac-set}$

using *frac-mult-denom-nonzero* $[of \ d \ f]$ **by** (*auto intro!*: *frac-setI*)

have $(A \ [+]\ B) \ [+]\ C = \text{frac.Class } ((a \cdot d + c \cdot b) \cdot f + e \cdot (b \cdot d), (b \cdot d) \cdot f)$

using $A'\ B'\ C'$ **by** (*simp add*: *frac-add-Class* $[OF \ A'(2) \ B'(2)]$ *frac-add-Class* $[OF \ s1 \ C'(2)]$)

moreover have $A \ [+]\ (B \ [+]\ C) = \text{frac.Class } (a \cdot (d \cdot f) + (c \cdot f + e \cdot d) \cdot b, b \cdot (d \cdot f))$

```

    using A' B' C' by (simp add: frac-add-Class[OF B'(2) C'(2)] frac-add-Class[OF
A'(2) s2])
  moreover have  $(a \cdot d + c \cdot b) \cdot f + e \cdot (b \cdot d) = a \cdot (d \cdot f) + (c \cdot f + e \cdot d) \cdot b$ 
  proof -
    have LHS:  $(a \cdot d + c \cdot b) \cdot f + e \cdot (b \cdot d) = a \cdot (d \cdot f) + c \cdot f \cdot b + e \cdot d \cdot b$ 
      by (simp add: distributive mult-ac add-ac)
    have RHS:  $a \cdot (d \cdot f) + (c \cdot f + e \cdot d) \cdot b = a \cdot (d \cdot f) + c \cdot f \cdot b + e \cdot d \cdot b$ 
      by (simp add: distributive mult-ac add-ac)
    from LHS RHS show ?thesis by simp
  qed
  moreover have  $(b \cdot d) \cdot f = b \cdot (d \cdot f)$  by (simp add: mult-ac)
  ultimately show  $(A \text{ [} + \text{]} B) \text{ [} + \text{]} C = A \text{ [} + \text{]} (B \text{ [} + \text{]} C)$  by simp
next
  — Left unit of  $\text{[} + \text{]}$  is  $\text{[} 0 \text{]}$ .
  fix A assume A:  $A \in \text{Fractions}$ 
  obtain a b where A':  $A = \text{frac.Class } (a, b) \ (a, b) \in \text{frac-set}$  using A by (metis
Fractions-cases)
  from A' have [simp]:  $a \in R \ b \in R \ b \neq 0$  by auto
  have  $\text{[} 0 \text{]} \text{ [} + \text{]} A = \text{frac.Class } (0 \cdot b + a \cdot 1, 1 \cdot b)$ 
    using A' by (simp add: frac-add-Class[OF zero-in-fraction-set A'(2)])
  also have  $\dots = \text{frac.Class } (a, b)$  by simp
  finally show  $\text{[} 0 \text{]} \text{ [} + \text{]} A = A$  using A' by simp
next
  — Right unit of  $\text{[} + \text{]}$ .
  fix A assume A:  $A \in \text{Fractions}$ 
  obtain a b where A':  $A = \text{frac.Class } (a, b) \ (a, b) \in \text{frac-set}$  using A by (metis
Fractions-cases)
  from A' have [simp]:  $a \in R \ b \in R \ b \neq 0$  by auto
  have  $A \text{ [} + \text{]} \text{[} 0 \text{]} = \text{frac.Class } (a \cdot 1 + 0 \cdot b, b \cdot 1)$ 
    using A' by (simp add: frac-add-Class[OF A'(2) zero-in-fraction-set])
  also have  $\dots = \text{frac.Class } (a, b)$  by simp
  finally show  $A \text{ [} + \text{]} \text{[} 0 \text{]} = A$  using A' by simp
next
  — Invertibility for  $\text{[} + \text{]}$ : supply the explicit two-sided inverse  $\text{[} - \text{]}$  A. This is the pinch
point that motivates the whole refactor — GroupI takes the raw  $\exists v \in G. u \cdot v = 1 \wedge v \cdot u = 1$  form, avoiding the locale-qualified Monoid.invertible obligation that would
otherwise be raised inside a bare interpretation of Ring.
  fix A assume A:  $A \in \text{Fractions}$ 
  obtain a b where A':  $A = \text{frac.Class } (a, b) \ (a, b) \in \text{frac-set}$  using A by (metis
Fractions-cases)
  from A' have abR:  $a \in R \ b \in R$  and bnz:  $b \neq 0$  by auto
  have bbR:  $b \cdot b \in R$  and bbnz:  $b \cdot b \neq 0$ 
    using abR bnz frac-mult-denom-nonzero[of b b] by auto
  have negA-in-fraction-set:  $(-a, b) \in \text{frac-set}$  using abR bnz by (auto intro!: fraction-setI)
  have negA:  $\text{[} - \text{]} A = \text{frac.Class } (-a, b)$  using A' by (simp add: frac-neg-Class[OF
A'(2)])
  have zero-bb-in:  $(0, b \cdot b) \in \text{frac-set}$  using bbR bbnz by (auto intro!: fraction-setI)
  have zero-bb-eq-zero:  $\text{frac.Class } (0, b \cdot b) = \text{frac.Class } (0, 1)$ 
    using bbR by (intro frac-eqI[OF zero-bb-in zero-in-fraction-set]) simp
  have right:  $A \text{ [} + \text{]} (\text{[} - \text{]} A) = \text{[} 0 \text{]}$ 

```

```

proof -
  have  $A \text{ } [+]$  ( $[-]$   $A$ ) =  $\text{frac.Class } (a \cdot b + (-a) \cdot b, b \cdot b)$ 
    using  $A' \text{ neg}A$  by (simp add: frac-add-Class[OF A'(2) negA-in-fraction-set])
  also have  $a \cdot b + (-a) \cdot b = 0$  using  $abR$  by (simp add: left-minus)
  finally have  $A \text{ } [+]$  ( $[-]$   $A$ ) =  $\text{frac.Class } (0, b \cdot b)$  .
  then show ?thesis using zero-bb-eq-zero by simp
qed
have left:  $[-]$   $A \text{ } [+]$   $A = [0]$ 
proof -
  have  $[-]$   $A \text{ } [+]$   $A = \text{frac.Class } ((-a) \cdot b + a \cdot b, b \cdot b)$ 
    using  $A' \text{ neg}A$  by (simp add: frac-add-Class[OF negA-in-fraction-set A'(2)])
  also have  $(-a) \cdot b + a \cdot b = 0$  using  $abR$  by (simp add: left-minus)
  finally have  $[-]$   $A \text{ } [+]$   $A = \text{frac.Class } (0, b \cdot b)$  .
  then show ?thesis using zero-bb-eq-zero by simp
qed
have negA-class:  $[-]$   $A \in \text{Fractions}$  using  $A$  by simp
from right left negA-class show  $\exists v \in \text{Fractions}. A \text{ } [+]$   $v = [0] \wedge v \text{ } [+]$   $A = [0]$  by
blast
qed

lemma Frac-add-commutative:  $\llbracket A \in \text{Fractions}; B \in \text{Fractions} \rrbracket \implies A \text{ } [+]$   $B = B \text{ } [+]$   $A$ 
proof -
  fix  $A B$  assume  $A: A \in \text{Fractions}$  and  $B: B \in \text{Fractions}$ 
  obtain  $a b$  where  $A': A = \text{frac.Class } (a, b)$   $(a, b) \in \text{frac-set}$  using  $A$  by (metis Fractions-cases)
  obtain  $c d$  where  $B': B = \text{frac.Class } (c, d)$   $(c, d) \in \text{frac-set}$  using  $B$  by (metis Fractions-cases)
  from  $A' B'$  have [simp]:  $a \in R \ b \in R \ c \in R \ d \in R \ b \neq 0 \ d \neq 0$  by auto
  have  $A \text{ } [+]$   $B = \text{frac.Class } (a \cdot d + c \cdot b, b \cdot d)$ 
    using  $A' B'$  by (simp add: frac-add-Class[OF A'(2) B'(2)])
  moreover have  $B \text{ } [+]$   $A = \text{frac.Class } (c \cdot b + a \cdot d, d \cdot b)$ 
    using  $A' B'$  by (simp add: frac-add-Class[OF B'(2) A'(2)])
  moreover have  $a \cdot d + c \cdot b = c \cdot b + a \cdot d$  by (simp add: add-ac)
  moreover have  $b \cdot d = d \cdot b$  by (simp add: mult-ac)
  ultimately show  $A \text{ } [+]$   $B = B \text{ } [+]$   $A$  by simp
qed

lemma Frac-add-commutative-monoid: commutative-monoid Fractions ( $[+]$ )  $[0]$ 
proof -
  interpret  $G:$  Group Fractions ( $[+]$ )  $[0]$  by (rule Frac-add-group)
  show ?thesis
    by unfold-locales (simp-all add: Frac-add-commutative)
qed

lemma Frac-abelian-group: Abelian-Group Fractions ( $[+]$ )  $[0]$ 
  by (rule Abelian-Group.intro[OF Frac-add-group Frac-add-commutative-monoid])

```

66.4.2 The multiplicative monoid of *Fractions*

```

lemma Frac-mult-monoid: Monoid Fractions ([·]) [1]
proof
  fix A B assume A ∈ Fractions B ∈ Fractions
  then show A [·] B ∈ Fractions by simp
next
  show [1] ∈ Fractions by simp
next
  — Associativity of [·], on representatives.
  fix A B C assume A: A ∈ Fractions and B: B ∈ Fractions and C: C ∈ Fractions
  obtain a b where A': A = frac.Class (a, b) (a, b) ∈ frac-set using A by (metis
Fractions-cases)
  obtain c d where B': B = frac.Class (c, d) (c, d) ∈ frac-set using B by (metis
Fractions-cases)
  obtain e f where C': C = frac.Class (e, f) (e, f) ∈ frac-set using C by (metis
Fractions-cases)
  from A' B' C' have [simp]:
    a ∈ R b ∈ R c ∈ R d ∈ R e ∈ R f ∈ R
    b ≠ 0 d ≠ 0 f ≠ 0 by auto
  have s1: (a · c, b · d) ∈ frac-set
    using frac-mult-denom-nonzero[of b d] by (auto intro!: frac-setI)
  have s2: (c · e, d · f) ∈ frac-set
    using frac-mult-denom-nonzero[of d f] by (auto intro!: frac-setI)
  have (A [·] B) [·] C = frac.Class ((a · c) · e, (b · d) · f)
    using A' B' C' by (simp add: frac-mult-Class[OF A' (2) B' (2)] frac-mult-Class[OF
s1 C' (2)])
  moreover have A [·] (B [·] C) = frac.Class (a · (c · e), b · (d · f))
    using A' B' C' by (simp add: frac-mult-Class[OF B' (2) C' (2)] frac-mult-Class[OF
A' (2) s2])
  moreover have (a · c) · e = a · (c · e) by (simp add: mult-ac)
  moreover have (b · d) · f = b · (d · f) by (simp add: mult-ac)
  ultimately show (A [·] B) [·] C = A [·] (B [·] C) by simp
next
  — Left unit.
  fix A assume A: A ∈ Fractions
  obtain a b where A': A = frac.Class (a, b) (a, b) ∈ frac-set using A by (metis
Fractions-cases)
  from A' have [simp]: a ∈ R b ∈ R b ≠ 0 by auto
  have [1] [·] A = frac.Class (1 · a, 1 · b)
    using A' by (simp add: frac-mult-Class[OF one-in-frac-set A' (2)])
  also have ... = frac.Class (a, b) by simp
  finally show [1] [·] A = A using A' by simp
next
  — Right unit.
  fix A assume A: A ∈ Fractions
  obtain a b where A': A = frac.Class (a, b) (a, b) ∈ frac-set using A by (metis
Fractions-cases)
  from A' have [simp]: a ∈ R b ∈ R b ≠ 0 by auto
  have A [·] [1] = frac.Class (a · 1, b · 1)

```


using A' by (simp add: frac-mult-Class[OF $A'(2)$ one-in-frac-set])
 also have $\dots = \text{frac.Class } (a, b)$ by simp
 finally show $A \cdot [1] = A$ using A' by simp
 qed

66.4.3 Distributivity and the ring axiom

lemma *Frac-Ring-axioms: Ring-axioms Fractions* ($[+]$) ($[\cdot]$)

proof

— Left distributivity.

fix $A B C$ **assume** $A: A \in \text{Fractions}$ **and** $B: B \in \text{Fractions}$ **and** $C: C \in \text{Fractions}$
obtain $a b$ **where** $A': A = \text{frac.Class } (a, b)$ $(a, b) \in \text{frac-set}$ **using** A **by** (metis
Fractions-cases)

obtain $c d$ **where** $B': B = \text{frac.Class } (c, d)$ $(c, d) \in \text{frac-set}$ **using** B **by** (metis
Fractions-cases)

obtain $e f$ **where** $C': C = \text{frac.Class } (e, f)$ $(e, f) \in \text{frac-set}$ **using** C **by** (metis
Fractions-cases)

from $A' B' C'$ **have** [simp]:

$a \in R \ b \in R \ c \in R \ d \in R \ e \in R \ f \in R$

$b \neq 0 \ d \neq 0 \ f \neq 0$ **by** auto

have *BC-in*: $(c \cdot f + e \cdot d, d \cdot f) \in \text{frac-set}$

using frac-mult-denom-nonzero[of $d f$] **by** (auto intro!: frac-setI)

have *AB-in*: $(a \cdot c, b \cdot d) \in \text{frac-set}$

using frac-mult-denom-nonzero[of $b d$] **by** (auto intro!: frac-setI)

have *AC-in*: $(a \cdot e, b \cdot f) \in \text{frac-set}$

using frac-mult-denom-nonzero[of $b f$] **by** (auto intro!: frac-setI)

have $A \cdot [\cdot] (B \cdot [+] C) = \text{frac.Class } (a \cdot (c \cdot f + e \cdot d), b \cdot (d \cdot f))$

using $A' B' C'$ **by** (simp add: frac-add-Class[OF $B'(2)$ $C'(2)$] frac-mult-Class[OF
 $A'(2)$ *BC-in*])

moreover have $A \cdot [\cdot] B \cdot [+] A \cdot [\cdot] C =$

$\text{frac.Class } ((a \cdot c) \cdot (b \cdot f) + (a \cdot e) \cdot (b \cdot d), (b \cdot d) \cdot (b \cdot f))$

using $A' B' C'$

by (simp add: frac-mult-Class[OF $A'(2)$ $B'(2)$] frac-mult-Class[OF $A'(2)$ $C'(2)$]
 frac-add-Class[OF *AB-in* *AC-in*])

moreover have

$(a \cdot (c \cdot f + e \cdot d)) \cdot ((b \cdot d) \cdot (b \cdot f)) =$

$((a \cdot c) \cdot (b \cdot f) + (a \cdot e) \cdot (b \cdot d)) \cdot (b \cdot (d \cdot f))$

proof —

have $a \cdot (c \cdot f + e \cdot d) = a \cdot (c \cdot f) + a \cdot (e \cdot d)$ **by** (simp add: distributive)

then have $L: (a \cdot (c \cdot f + e \cdot d)) \cdot ((b \cdot d) \cdot (b \cdot f))$

$= a \cdot (c \cdot f) \cdot ((b \cdot d) \cdot (b \cdot f)) + a \cdot (e \cdot d) \cdot ((b \cdot d) \cdot (b \cdot f))$

by (simp add: distributive)

have $((a \cdot c) \cdot (b \cdot f) + (a \cdot e) \cdot (b \cdot d)) \cdot (b \cdot (d \cdot f))$

$= (a \cdot c) \cdot (b \cdot f) \cdot (b \cdot (d \cdot f)) + (a \cdot e) \cdot (b \cdot d) \cdot (b \cdot (d \cdot f))$

by (simp add: distributive)

also have $(a \cdot c) \cdot (b \cdot f) \cdot (b \cdot (d \cdot f)) = a \cdot (c \cdot f) \cdot ((b \cdot d) \cdot (b \cdot f))$

by (simp add: mult-ac)

also have $(a \cdot e) \cdot (b \cdot d) \cdot (b \cdot (d \cdot f)) = a \cdot (e \cdot d) \cdot ((b \cdot d) \cdot (b \cdot f))$

by (simp add: mult-ac)

finally show ?thesis **using** L **by** simp

```

qed
moreover have  $(a \cdot (c \cdot f + e \cdot d), b \cdot (d \cdot f)) \in \text{frac-set}$ 
  using frac-mult-denom-nonzero[of d f] frac-mult-denom-nonzero[of b d · f]
  by (auto intro!: frac-setI)
moreover have  $((a \cdot c) \cdot (b \cdot f) + (a \cdot e) \cdot (b \cdot d), (b \cdot d) \cdot (b \cdot f)) \in \text{frac-set}$ 
  using frac-mult-denom-nonzero[of b d] frac-mult-denom-nonzero[of b f]
    frac-mult-denom-nonzero[of b · d b · f]
  by (auto intro!: frac-setI)
ultimately show  $A [\cdot] (B [+] C) = A [\cdot] B [+] A [\cdot] C$ 
  by (metis frac-eqI)
next
— Right distributivity.
fix A B C assume A:  $A \in \text{Fractions}$  and B:  $B \in \text{Fractions}$  and C:  $C \in \text{Fractions}$ 
obtain a b where A':  $A = \text{frac.Class } (a, b)$   $(a, b) \in \text{frac-set}$  using A by (metis
Fractions-cases)
obtain c d where B':  $B = \text{frac.Class } (c, d)$   $(c, d) \in \text{frac-set}$  using B by (metis
Fractions-cases)
obtain e f where C':  $C = \text{frac.Class } (e, f)$   $(e, f) \in \text{frac-set}$  using C by (metis
Fractions-cases)
from A' B' C' have [simp]:
   $a \in R \ b \in R \ c \in R \ d \in R \ e \in R \ f \in R$ 
   $b \neq 0 \ d \neq 0 \ f \neq 0$  by auto
have BC-in:  $(c \cdot f + e \cdot d, d \cdot f) \in \text{frac-set}$ 
  using frac-mult-denom-nonzero[of d f] by (auto intro!: frac-setI)
have BA-in:  $(c \cdot a, d \cdot b) \in \text{frac-set}$ 
  using frac-mult-denom-nonzero[of d b] by (auto intro!: frac-setI)
have CA-in:  $(e \cdot a, f \cdot b) \in \text{frac-set}$ 
  using frac-mult-denom-nonzero[of f b] by (auto intro!: frac-setI)
have  $(B [+] C) [\cdot] A = \text{frac.Class } ((c \cdot f + e \cdot d) \cdot a, (d \cdot f) \cdot b)$ 
  using A' B' C' by (simp add: frac-add-Class[OF B'(2) C'(2)] frac-mult-Class[OF
BC-in A'(2)])
moreover have  $B [\cdot] A [+] C [\cdot] A =$ 
   $\text{frac.Class } ((c \cdot a) \cdot (f \cdot b) + (e \cdot a) \cdot (d \cdot b), (d \cdot b) \cdot (f \cdot b))$ 
  using A' B' C'
  by (simp add: frac-mult-Class[OF B'(2) A'(2)] frac-mult-Class[OF C'(2) A'(2)]
    frac-add-Class[OF BA-in CA-in])
moreover have
   $((c \cdot f + e \cdot d) \cdot a) \cdot ((d \cdot b) \cdot (f \cdot b)) =$ 
   $((c \cdot a) \cdot (f \cdot b) + (e \cdot a) \cdot (d \cdot b)) \cdot ((d \cdot f) \cdot b)$ 
proof —
  have  $(c \cdot f + e \cdot d) \cdot a = (c \cdot f) \cdot a + (e \cdot d) \cdot a$  by (simp add: distributive)
  then have L:  $((c \cdot f + e \cdot d) \cdot a) \cdot ((d \cdot b) \cdot (f \cdot b))$ 
     $= ((c \cdot f) \cdot a) \cdot ((d \cdot b) \cdot (f \cdot b)) + ((e \cdot d) \cdot a) \cdot ((d \cdot b) \cdot (f \cdot b))$ 
    by (simp add: distributive)
  have  $((c \cdot a) \cdot (f \cdot b) + (e \cdot a) \cdot (d \cdot b)) \cdot ((d \cdot f) \cdot b)$ 
     $= (c \cdot a) \cdot (f \cdot b) \cdot ((d \cdot f) \cdot b) + (e \cdot a) \cdot (d \cdot b) \cdot ((d \cdot f) \cdot b)$ 
    by (simp add: distributive)
  also have  $(c \cdot a) \cdot (f \cdot b) \cdot ((d \cdot f) \cdot b) = ((c \cdot f) \cdot a) \cdot ((d \cdot b) \cdot (f \cdot b))$ 
    by (simp add: mult-ac)
  also have  $(e \cdot a) \cdot (d \cdot b) \cdot ((d \cdot f) \cdot b) = ((e \cdot d) \cdot a) \cdot ((d \cdot b) \cdot (f \cdot b))$ 

```

```

    by (simp add: mult-ac)
  finally show ?thesis using L by simp
qed
moreover have  $((c \cdot f + e \cdot d) \cdot a, (d \cdot f) \cdot b) \in \text{frac-set}$ 
  using frac-mult-denom-nonzero[of d f] frac-mult-denom-nonzero[of d f b]
  by (auto intro!: frac-setI)
moreover have  $((c \cdot a) \cdot (f \cdot b) + (e \cdot a) \cdot (d \cdot b), (d \cdot b) \cdot (f \cdot b)) \in \text{frac-set}$ 
  using frac-mult-denom-nonzero[of d b] frac-mult-denom-nonzero[of f b]
    frac-mult-denom-nonzero[of d b f b]
  by (auto intro!: frac-setI)
ultimately show  $(B \text{ [ + ] } C) \text{ [ \cdot ] } A = B \text{ [ \cdot ] } A \text{ [ + ] } C \text{ [ \cdot ] } A$ 
  by (metis frac-eqI)
qed

```

Assemble the additive abelian group, multiplicative monoid, and the distributive laws into the *Ring* interpretation.

```

interpretation Frac: Ring Fractions ([+]) ([\cdot]) [0] [1]
  by (rule Ring.intro[OF Frac-abelian-group Frac-mult-monoid Frac-Ring-axioms])

```

```

interpretation Frac: commutative-ring Fractions ([+]) ([\cdot]) [0] [1]
proof
  fix A B assume A:  $A \in \text{Fractions}$  and B:  $B \in \text{Fractions}$ 
  obtain a b where A':  $A = \text{frac.Class } (a, b)$   $(a, b) \in \text{frac-set}$  using A by (metis
    Fractions-cases)
  obtain c d where B':  $B = \text{frac.Class } (c, d)$   $(c, d) \in \text{frac-set}$  using B by (metis
    Fractions-cases)
  from A' B' have [simp]:  $a \in R$   $b \in R$   $c \in R$   $d \in R$   $b \neq 0$   $d \neq 0$  by auto
  have A [\cdot]  $B = \text{frac.Class } (a \cdot c, b \cdot d)$ 
    using A' B' by (simp add: frac-mult-Class[OF A'(2) B'(2)])
  moreover have B [\cdot]  $A = \text{frac.Class } (c \cdot a, d \cdot b)$ 
    using A' B' by (simp add: frac-mult-Class[OF B'(2) A'(2)])
  moreover have  $a \cdot c = c \cdot a$  by (simp add: mult-ac)
  moreover have  $b \cdot d = d \cdot b$  by (simp add: mult-ac)
  ultimately show  $A \text{ [ \cdot ] } B = B \text{ [ \cdot ] } A$  by simp
qed

```

```

interpretation Frac: Field Fractions ([+]) ([\cdot]) [0] [1]
proof
  — Nontrivial.
  show [1]  $\neq$  [0] using frac-one-neq-zero .
next
  — Every nonzero class is invertible.
  fix A assume A:  $A \in \text{Fractions}$  and Anz:  $A \neq [0]$ 
  obtain a b where A':  $A = \text{frac.Class } (a, b)$   $(a, b) \in \text{frac-set}$  using A by (metis
    Fractions-cases)
  from A' have abR:  $a \in R$   $b \in R$  and bnz:  $b \neq 0$  by auto
  have anz:  $a \neq 0$  using Anz A' frac-Class-zero-iff[OF A'(2)] by blast

```

```

have inv-in: (b, a) ∈ frac-set using abR anz by (auto intro!: frac-setI)
have inv-class: frac-inv A = frac.Class (b, a)
  using A' by (simp add: frac-inv-Class[OF A'(2) anz])
have A [·] frac-inv A = frac.Class (a · b, b · a)
  using A' inv-class by (simp add: frac-mult-Class[OF A'(2) inv-in])
moreover have frac.Class (a · b, b · a) = [1]
proof -
  have (a · b, b · a) ∈ frac-set
    using abR anz bnz frac-mult-denom-nonzero[of b a] by (auto intro!: frac-setI)
  moreover have (1, 1) ∈ frac-set by (auto simp: nontrivial)
  moreover have (a · b) · 1 = 1 · (b · a) using abR by (simp add: mult-ac)
  ultimately show ?thesis by (rule frac-eqI)
qed
ultimately have right-inv: A [·] frac-inv A = [1] by simp

have frac-inv A [·] A = frac.Class (b · a, a · b)
  using A' inv-class by (simp add: frac-mult-Class[OF inv-in A'(2)])
moreover have frac.Class (b · a, a · b) = [1]
proof -
  have (b · a, a · b) ∈ frac-set
    using abR anz bnz frac-mult-denom-nonzero[of a b] by (auto intro!: frac-setI)
  moreover have (1, 1) ∈ frac-set by (auto simp: nontrivial)
  moreover have (b · a) · 1 = 1 · (a · b) using abR by (simp add: mult-ac)
  ultimately show ?thesis by (rule frac-eqI)
qed
ultimately have left-inv: frac-inv A [·] A = [1] by simp

have inv-in-carrier: frac-inv A ∈ Fractions using inv-class inv-in by simp
show Frac.multiplicative.invertible A using right-inv left-inv A inv-in-carrier
  by (blast intro: Frac.multiplicative.invertibleI)
qed

```

66.5 The canonical embedding $R \rightarrow \text{Fractions}$

The canonical map $a \mapsto \langle \text{Class}(a, 1) \rangle$ embeds R into its field of fractions. It is an injective ring homomorphism. Together with the field structure, this is the universal embedding of an integral domain into a field: every nonzero element of R acquires a multiplicative inverse in Fractions .

definition *to-frac* :: 'a $\Rightarrow$ ('a $\times$ 'a) set
 where *to-frac* = ($\lambda a \in R. \text{frac.Class } (a, 1)$)

lemma *to-frac-apply* [simp]: $a \in R \implies \text{to-frac } a = \text{frac.Class } (a, 1)$
 unfolding *to-frac-def* by simp

lemma *to-frac-undefined* [simp]: $a \notin R \implies \text{to-frac } a = \text{undefined}$
 unfolding *to-frac-def* by simp

lemma *to-frac-pair*: $a \in R \implies (a, 1) \in \text{frac-set}$
 by (auto intro!: frac-setI simp: nontrivial)

lemma *to-frac-in-Fractions* [*intro*, *simp*]: $a \in R \implies \text{to-frac } a \in \text{Fractions}$
using *to-frac-pair* **by** (*simp add: to-frac-apply*)

lemma *to-frac-zero*: $\text{to-frac } 0 = [0]$ **by** *simp*
lemma *to-frac-one*: $\text{to-frac } 1 = [1]$ **by** *simp*

lemma *to-frac-add*:
assumes $a: a \in R$ **and** $b: b \in R$
shows $\text{to-frac } (a + b) = \text{to-frac } a \text{ } [+] \text{ to-frac } b$
proof –
have $\text{to-frac } a \text{ } [+] \text{ to-frac } b = \text{frac.Class } (a \cdot 1 + b \cdot 1, 1 \cdot 1)$
using $a \text{ } b$ **by** (*simp add: frac-add-Class[OF to-frac-pair[OF a] to-frac-pair[OF b]]*)
also have $\dots = \text{frac.Class } (a + b, 1)$ **using** $a \text{ } b$ **by** *simp*
finally show *?thesis* **using** $a \text{ } b$ **by** *simp*
qed

lemma *to-frac-mult*:
assumes $a: a \in R$ **and** $b: b \in R$
shows $\text{to-frac } (a \cdot b) = \text{to-frac } a \text{ } [\cdot] \text{ to-frac } b$
proof –
have $\text{to-frac } a \text{ } [\cdot] \text{ to-frac } b = \text{frac.Class } (a \cdot b, 1 \cdot 1)$
using $a \text{ } b$ **by** (*simp add: frac-mult-Class[OF to-frac-pair[OF a] to-frac-pair[OF b]]*)
also have $\dots = \text{frac.Class } (a \cdot b, 1)$ **by** *simp*
finally show *?thesis* **using** $a \text{ } b$ **by** *simp*
qed

lemma *to-frac-inj*:
assumes $a: a \in R$ **and** $b: b \in R$ **and** $\text{eq: to-frac } a = \text{to-frac } b$
shows $a = b$
proof –
have $a \cdot 1 = b \cdot 1$
by (*rule frac-eqD[OF to-frac-pair[OF a] to-frac-pair[OF b]]*) (*use eq a b in simp*)
with $a \text{ } b$ **show** *?thesis* **by** *simp*
qed

interpretation *to-frac*: *ring-homomorphism to-frac R addition* $(\cdot) \ 0 \ 1$
Fractions $([+]) \ ([\cdot]) \ [0] \ [1]$

proof
have $\bigwedge x. x \in R \implies \text{to-frac } x \in \text{Fractions}$ **by** (*rule to-frac-in-Fractions*)
moreover have $\bigwedge x. x \notin R \implies \text{to-frac } x = \text{undefined}$ **by** (*rule to-frac-undefined*)
ultimately show $\text{to-frac} \in R \rightarrow_E \text{Fractions}$
unfolding *PiE-def extensional-def* **by** *blast*
next
fix $a \text{ } b$ **assume** $a \in R \text{ } b \in R$
then show $\text{to-frac } (a + b) = \text{to-frac } a \text{ } [+] \text{ to-frac } b$ **by** (*rule to-frac-add*)
from $\langle a \in R \rangle \langle b \in R \rangle$ **show** $\text{to-frac } (a \cdot b) = \text{to-frac } a \text{ } [\cdot] \text{ to-frac } b$ **by** (*rule to-frac-mult*)
next

```

  show to-frac 0 = [0] by (rule to-frac-zero)
  show to-frac 1 = [1] by (rule to-frac-one)
qed

end

end

```

67 The polynomial ring over a field is a Euclidean domain

```

theory Poly-Divisibility
  imports Poly-Ring Divisibility-Theory Poly-Ideal
begin

```

Over a field F , the polynomial ring $F[X]$ is a Euclidean domain, with the polynomial degree as Euclidean function: this is exactly the division algorithm of *Poly-Ring* (*Field.poly-divide*) packaged into the abstract *Euclidean-Domain* of *Divisibility-Theory*. The interpretation is the bridge that endows $F[X]$ — for free — with the whole divisibility tower: it is a principal ideal domain and a factorial (unique factorization) domain, in which irreducible polynomials are prime and factorizations into irreducibles exist and are unique up to the number of factors.

```

context Field
begin

```

The carrier-set polynomial ring satisfies the abstract Euclidean-domain interface. This named theorem keeps the division-algorithm proof reusable by the exported sublocale below.

```

theorem poly-euclidean-domain:
  Euclidean-Domain poly-carrier ( $\oplus_P$ ) ( $\otimes_P$ ) 0_P 1_P degree
proof -
  show Euclidean-Domain poly-carrier ( $\oplus_P$ ) ( $\otimes_P$ ) 0_P 1_P degree
proof (rule Euclidean-Domain.intro)
  show integral-domain poly-carrier ( $\oplus_P$ ) ( $\otimes_P$ ) 0_P 1_P
    by (rule poly-integral-domain)
next
  show Euclidean-Domain-axioms poly-carrier ( $\oplus_P$ ) ( $\otimes_P$ ) 0_P degree
proof
  fix a b assume a:  $a \in \text{poly-carrier}$  and b:  $b \in \text{poly-carrier}$ 
    and anz:  $a \neq 0_P$  and bnz:  $b \neq 0_P$ 
  show  $\exists q \in \text{poly-carrier}. \exists r \in \text{poly-carrier}.$ 
     $a = (b \otimes_P q) \oplus_P r \wedge (r = 0_P \vee \text{degree } r < \text{degree } b)$ 
    using poly-divide[OF a b bnz] by blast
qed
qed
qed

```

end

sublocale *Field* $\subseteq$ *FX*: *Euclidean-Domain*

poly-carrier $(\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$ *degree*

proof –

show *Euclidean-Domain* *poly-carrier* $(\oplus_P) (\otimes_P) \mathbf{0}_P \mathbf{1}_P$ *degree*
by (*rule poly-euclidean-domain*)

qed

context *Field*

begin

Consequences, transported from the abstract tower. Every nonzero non-unit polynomial factors into irreducibles (*FX.factorization-exists*), an irreducible polynomial is prime (*FX.irreducible-imp-prime*), and any two prime factorizations of associated polynomials have the same length (*FX.prime-factorization-length-unique*).

theorem *poly-factorization-exists*:

$\llbracket p \in \text{poly-carrier}; p \neq \mathbf{0}_P; \neg \text{FX.is-unit } p \rrbracket \implies \exists fs. \text{FX.factorization } fs \ p$
by (*rule FX.factorization-exists*)

theorem *poly-irreducible-imp-prime*:

FX.irreducible-elem $p \implies \text{FX.prime-elem } p$
by (*rule FX.irreducible-imp-prime*)

end

Poly-Divisibility interprets the carrier-set polynomial ring as an abstract Euclidean domain, while *Poly-Ideal* exposes the same multiples through *Field.poly-dvd* and *Field.poly-pideal*. The two interfaces are intentionally kept in their original theories so that each remains independently usable. This theory is the explicit seam between them.

context *Field*

begin

The concrete polynomial divisibility relation and the abstract relation differ only by the orientation of the product. Commutativity of the polynomial ring supplies the bridge.

lemma *FX-divides-iff-poly-dvd*:

assumes $d: d \in \text{poly-carrier}$
shows $\text{FX.divides } d \ x \longleftrightarrow \text{poly-dvd } d \ x$

proof

assume $\text{dvd}: \text{FX.divides } d \ x$

obtain q **where** $q: q \in \text{poly-carrier}$ **and** $xeq: x = d \otimes_P q$

using *FX.dividesE*[*OF dvd*] **by** *blast*

have $x = q \otimes_P d$

proof –

have $d \otimes_P q = q \otimes_P d$ **using** $d \ q$ **by** (*rule poly-mult-comm*)

```

    then show ?thesis using xeq by simp
qed
then show poly-dvd d x
  unfolding poly-dvd-def using q by blast
next
assume dvd: poly-dvd d x
obtain q where q: q ∈ poly-carrier and xeq: x = q ⊗P d
  using dvd unfolding poly-dvd-def by blast
have x = d ⊗P q
proof -
  have q ⊗P d = d ⊗P q using q d by (rule poly-mult-comm)
  then show ?thesis using xeq by simp
qed
show FX.divides d x
proof (rule FX.dividesI)
  show q ∈ poly-carrier using q .
  show x = d ⊗P q using ⟨x = d ⊗P q⟩ .
qed
qed

```

The concrete principal ideal and the abstract divisibility relation now have a common polynomial-level formulation.

lemma *FX-divides-iff-mem-poly-pideal*:
assumes *d*: $d \in \text{poly-carrier}$
shows $\text{FX.divides } d \ x \longleftrightarrow x \in \text{poly-pideal } d$
using *FX-divides-iff-poly-dvd*[*OF* *d*] *poly-dvd-iff-mem-pideal* **by** *blast*

lemma *poly-dvd-iff-mem-FX-principal*:
assumes *d*: $d \in \text{poly-carrier}$ **and** *x*: $x \in \text{poly-carrier}$
shows $\text{poly-dvd } d \ x \longleftrightarrow x \in \text{FX.principal-ideal } d$
using *FX-divides-iff-poly-dvd*[*OF* *d*] *FX.divides-iff-mem-principal*[*OF* *d* *x*] **by** *blast*

lemma *FX-divides-iff-pmod-zero*:
assumes *d*: $d \in \text{poly-carrier}$ **and** *dnz*: $d \neq \mathbf{0}_P$
and *x*: $x \in \text{poly-carrier}$
shows $\text{FX.divides } d \ x \longleftrightarrow \text{pmod } x \ d = \mathbf{0}_P$
using *FX-divides-iff-poly-dvd*[*OF* *d*] *poly-dvd-iff-pmod-zero*[*OF* *d* *dnz* *x*] **by** *blast*

67.1 Units and irreducibility across the seam

The same seam again one notion up. *poly-unit* spells out invertibility in the polynomial ring directly, while *FX.is-unit* is the abstract *Monoid.invertible* at that ring. *They are not quite the same predicate*: *Monoid.invertible* does not require its argument to lie in the carrier, whereas *poly-unit* does, so the two agree only on *poly-carrier*. That is enough for the notion actually wanted, because *poly-irreducible* and *FX.irreducible-elem* both demand carrier membership outright and apply their unit test only to carrier elements. Their agreement is what lets a factor produced by *poly-factorization-exists* be fed to *poly-quotient-field*.

lemma *FX-is-unit-iff-poly-unit*:
assumes $p: p \in \text{poly-carrier}$ **shows** $\text{FX.is-unit } p \longleftrightarrow \text{poly-unit } p$
proof
assume $\text{FX.is-unit } p$
then obtain q **where** $q \in \text{poly-carrier}$ **and** $p \otimes_P q = \mathbf{1}_P$ **and** $q \otimes_P p = \mathbf{1}_P$ **by**
blast
with p **show** $\text{poly-unit } p$ **by** (*rule poly-unitI*)
next
assume $\text{poly-unit } p$
then obtain q **where** $q: q \in \text{poly-carrier}$ **and** $pq: p \otimes_P q = \mathbf{1}_P$ **and** $qp: q \otimes_P p$
 $= \mathbf{1}_P$
unfolding *poly-unit-def* **by** *blast*
show $\text{FX.is-unit } p$ **by** (*rule FX.is-unitI* [*OF* pq qp q])
qed

lemma *FX-irreducible-iff-poly-irreducible*:
 $\text{FX.irreducible-elem } p \longleftrightarrow \text{poly-irreducible } p$
unfolding *FX.irreducible-elem-def* *poly-irreducible-def*
by (*auto simp: FX-is-unit-iff-poly-unit*)

Every polynomial that is neither zero nor a unit has an irreducible factor: factor it and take the head of the list, which is nonempty because the empty product is $\mathbf{1}_P$. This is the form in which the adjunction of a root consumes factorization — a root of an irreducible factor of p is a root of p — and it is stated with *poly-irreducible* so that *poly-quotient-field* applies directly.

theorem *exists-poly-irreducible-factor*:
assumes $p: p \in \text{poly-carrier}$ **and** $\text{pnz}: p \neq \mathbf{0}_P$ **and** $\text{pnu}: \neg \text{poly-unit } p$
obtains q **where** $\text{poly-irreducible } q$ **and** $\text{poly-dvd } q \ p$
proof –
have $\neg \text{FX.is-unit } p$ **using** pnu **by** (*simp add: FX-is-unit-iff-poly-unit* [*OF* p])
then obtain fs **where** $fs: \text{FX.factorization } fs \ p$
using *poly-factorization-exists* [*OF* p pnz] **by** *blast*
have $\text{prod}: \text{FX.list-prod } fs = p$ **by** (*rule FX.factorization-prod* [*OF* fs])
have $fs \neq []$
proof
assume $fs = []$
with prod **have** $\text{peq}: p = \mathbf{1}_P$ **by** *simp*
have $\text{poly-unit } \mathbf{1}_P$
by (*rule poly-unitI* [*OF* *poly-one-closed* *poly-one-closed*])
(*simp-all add: poly-mult-one-left poly-one-closed*)
with $\text{peq } \text{pnu}$ **show** *False* **by** *simp*
qed
then obtain $f \ gs$ **where** $\text{feq}: fs = f \ \# \ gs$ **by** (*cases* fs) *auto*
have $\text{irr}: \text{FX.irreducible-elem } f$ **using** $fs \ \text{feq}$ **by** (*simp add: FX.factorization-irreducible*)
have $\text{fc}: f \in \text{poly-carrier}$ **using** irr **by** (*rule FX.irreducible-elemD*(1))
have $\text{gsR}: \text{set } gs \subseteq \text{poly-carrier}$ **using** *FX.factorization-set-R* [*OF* fs] feq **by** *simp*
have $p = f \otimes_P \text{FX.list-prod } gs$ **using** $\text{prod } \text{feq}$ **by** *simp*
then have $\text{poly-dvd } f \ p$
using *FX.list-prod-closed* [*OF* gsR] fc **by** (*auto simp: poly-dvd-def poly-mult-comm*)

```

with irr show thesis
  using that FX-irreducible-iff-poly-irreducible by blast
qed

```

Each member of a list divides the product of the list. Needed to feed a single irreducible factor of a factorisation to the adjunction of a root.

```

lemma poly-dvd-poly-prod:
  assumes fs:  $\bigwedge g. g \in \text{set } fs \implies g \in \text{poly-carrier}$  and f:  $f \in \text{set } fs$ 
  shows poly-dvd f (poly-prod fs)
  using fs f
proof (induct fs)
  case (Cons g gs)
  have gc:  $g \in \text{poly-carrier}$  and gsc:  $\bigwedge h. h \in \text{set } gs \implies h \in \text{poly-carrier}$ 
    using Cons.prem1 by auto
  have Pgs: poly-prod gs  $\in \text{poly-carrier}$  by (rule poly-prod-closed [OF gsc])
  show ?case
proof (cases f = g)
  case True
  have poly-prod (g # gs) = poly-prod gs  $\otimes_P$  g
    using poly-mult-comm [OF gc Pgs] by simp
  then show ?thesis unfolding poly-dvd-def using True Pgs by blast
next
  case False
  then have f  $\in \text{set } gs$  using Cons.prem2 by simp
  then obtain q where q:  $q \in \text{poly-carrier}$  and eq: poly-prod gs =  $q \otimes_P f$ 
    using Cons.hyps [OF gsc] unfolding poly-dvd-def by blast
  have fc:  $f \in \text{poly-carrier}$  using  $\langle f \in \text{set } gs \rangle$  by (rule gsc)
  have poly-prod (g # gs) =  $(g \otimes_P q) \otimes_P f$ 
    using eq poly-mult-assoc [OF gc q fc] by simp
  then show ?thesis unfolding poly-dvd-def using gc q by (blast intro: poly-mult-closed)
qed
qed simp

```

67.2 The field of remainders modulo an irreducible polynomial

Kronecker's theorem in *normal-form* presentation: rather than the cosets of (p) , take the polynomials of degree below p , with addition unchanged and multiplication followed by reduction. This is the same field as *poly-quotient-field* exhibits, but its carrier is a set of polynomials rather than a set of cosets, so it stays inside the coefficient type. That is what a construction adjoining roots inside a fixed ambient type needs, and it is also the form in which one computes.

Invertibility is the only substantial part, and it does not need gcds: a nonzero remainder is not a multiple of p , so it lies outside the maximal ideal (p) , and extending that ideal by it gives the whole ring — which is to say $\mathbf{1}_P$ is a combination $m \oplus_P r \otimes_P S$ with m a multiple of p . Reducing that equation is exactly the inverse.

theorem *remainder-field*:

assumes *irr*: *poly-irreducible* *p*
shows *Field* (*low-poly* (*degree* *p*)) ($\oplus_P$) ($\lambda S\ T. \text{pmod } (S \otimes_P T) \ p$) $\mathbf{0}_P \ \mathbf{1}_P$
proof –
have *p*: $p \in \text{poly-carrier}$ **using** *irr* **by** (*rule poly-irreducibleD-carrier*)
have *pnz*: $p \neq \mathbf{0}_P$ **using** *irr* **by** (*rule poly-irreducibleD-nonzero*)
have *dpos*: $0 < \text{degree } p$ **using** *irr* **by** (*rule poly-irreducible-degree-pos*)
define *K* **where** $K = \text{low-poly } (\text{degree } p)$
define *mult* **where** $\text{mult} = (\lambda S\ T. \text{pmod } (S \otimes_P T) \ p)$
— Basic facts about the carrier.
have *Kc*: $S \in \text{poly-carrier}$ **if** $S \in K$ **for** S
using *that* **unfolding** *K-def* **by** (*rule low-poly-closed*)
have *Ksmall*: $\text{pmod } S \ p = S$ **if** $S \in K$ **for** S
using *that* *low-poly-iff-degree* [*OF* *Kc* [*OF that*]] *pmod-small* [*OF* *p pnz Kc* [*OF that*]]
unfolding *K-def* **by** *blast*
have *Kadd*: $S \oplus_P T \in K$ **if** $S \in K \ T \in K$ **for** $S \ T$
using *that* **unfolding** *K-def* **by** (*rule low-poly-add-closed*)
have *Kneg*: $\ominus_P S \in K$ **if** $S \in K$ **for** S
using *that* **unfolding** *K-def* **by** (*rule low-poly-neg-closed*)
have *Kzero*: $\mathbf{0}_P \in K$ **unfolding** *K-def* **by** (*rule low-poly-zero*)
have *Kone*: $\mathbf{1}_P \in K$ **unfolding** *K-def* **using** *dpos* **by** (*rule low-poly-one*)
have *Kmult*: $\text{mult } S \ T \in K$ **if** $S \in K \ T \in K$ **for** $S \ T$
unfolding *mult-def* *K-def*
using *poly-mult-closed* [*OF* *Kc* [*OF that*(1)] *Kc* [*OF that*(2)]] *p pnz* **by** (*rule pmod-low-poly*)
— Reduction is a homomorphism, so it carries the ring laws over from $F[X]$.
have *reduce*: $\text{pmod } (S \otimes_P T) \ p = \text{mult } S \ T$ **for** $S \ T$ **unfolding** *mult-def* **by** *rule*
have *mult-assoc*: $\text{mult } (\text{mult } S \ T) \ U = \text{mult } S \ (\text{mult } T \ U)$ **if** $K: S \in K \ T \in K \ U \in K$ **for** $S \ T \ U$
proof –
have *c*: $S \in \text{poly-carrier} \ T \in \text{poly-carrier} \ U \in \text{poly-carrier}$
using *K* **by** (*blast intro: Kc*)
have $\text{mult } (\text{mult } S \ T) \ U = \text{pmod } (\text{pmod } (S \otimes_P T) \ p \otimes_P \text{pmod } U \ p) \ p$
unfolding *mult-def* **using** *Ksmall* [*OF* *K*(3)] **by** *simp*
also have $\dots = \text{pmod } ((S \otimes_P T) \otimes_P U) \ p$
by (*rule pmod-mult* [*OF poly-mult-closed* [*OF* *c*(1) *c*(2)] *c*(3) *p pnz, symmetric*])
also have $\dots = \text{pmod } (S \otimes_P (T \otimes_P U)) \ p$ **using** *poly-mult-assoc* [*OF* *c*] **by** *simp*
also have $\dots = \text{pmod } (\text{pmod } S \ p \otimes_P \text{pmod } (T \otimes_P U) \ p) \ p$
by (*rule pmod-mult* [*OF* *c*(1) *poly-mult-closed* [*OF* *c*(2) *c*(3)] *p pnz*])
also have $\dots = \text{mult } S \ (\text{mult } T \ U)$
unfolding *mult-def* **using** *Ksmall* [*OF* *K*(1)] **by** *simp*
finally show *?thesis* .
qed
have *mult-comm*: $\text{mult } S \ T = \text{mult } T \ S$ **if** $S \in K \ T \in K$ **for** $S \ T$
unfolding *mult-def* **using** *poly-mult-comm* [*OF* *Kc* [*OF that*(1)] *Kc* [*OF that*(2)]]
by *simp*
have *mult-one*: $\text{mult } \mathbf{1}_P \ S = S$ **if** $S \in K$ **for** S
unfolding *mult-def*
using *poly-mult-one-left* [*OF* *Kc* [*OF that*]] *Ksmall* [*OF that*] **by** *simp*
have *distrib*: $\text{mult } S \ (T \oplus_P U) = \text{mult } S \ T \oplus_P \text{mult } S \ U$ **if** $K: S \in K \ T \in K \ U \in K$

K for $S \ T \ U$
proof –
 have $c: S \in \text{poly-carrier } T \in \text{poly-carrier } U \in \text{poly-carrier}$
 using K **by** (blast intro: Kc)
 have $\text{mult } S \ (T \oplus_P U) = \text{pmod } ((S \otimes_P T) \oplus_P (S \otimes_P U)) \ p$
 unfolding mult-def **using** $\text{poly-mult-add-distrib-left}$ $[OF \ c]$ **by** simp
 also have $\dots = \text{pmod } (S \otimes_P T) \ p \oplus_P \text{pmod } (S \otimes_P U) \ p$
 by (rule pmod-add $[OF \ \text{poly-mult-closed } [OF \ c(1) \ c(2)] \ \text{poly-mult-closed } [OF \ c(1) \ c(3)] \ p \ \text{pnz}]$)
 finally show $?thesis$ **unfolding** mult-def .
qed
— A nonzero remainder is invertible: it escapes the maximal ideal (p).
have $\text{inverse: } \exists T \in K. \ \text{mult } S \ T = 1_P$ **if** $S: S \in K$ **and** $\text{Snz: } S \neq 0_P$ **for** S
proof –
 interpret $I: \text{ideal-in-comm-ring poly-pideal } p \ \text{poly-carrier } (\oplus_P) \ (\otimes_P) \ 0_P \ 1_P$
 by (rule $\text{poly-pideal-in-comm-ring}$ $[OF \ p]$)
 have $Sc: S \in \text{poly-carrier}$ **using** S **by** (rule Kc)
 have $\text{pmod } S \ p \neq 0_P$ **using** $K\text{small}$ $[OF \ S]$ Snz **by** simp
 then have $\text{SnotI: } S \notin \text{poly-pideal } p$ **using** $\text{poly-pideal-iff-pmod}$ $[OF \ p \ \text{pnz } Sc]$ **by**
 simp
 have $1_P \in I.\text{ext-ideal } S$
 using $I.\text{maximal-ext-ideal-eq-whole}$ $[OF \ \text{poly-irreducible-maximal } [OF \ \text{irr}] \ Sc$
 $\text{SnotI}]$
 poly-one-closed **by** simp
 then obtain $m \ r$ **where** $m: m \in \text{poly-pideal } p$ **and** $r: r \in \text{poly-carrier}$
 and one: $1_P = m \oplus_P (r \otimes_P S)$
 unfolding $I.\text{ext-ideal-def}$ **by** blast
 have $mc: m \in \text{poly-carrier}$
proof –
 obtain u **where** $u: u \in \text{poly-carrier}$ **and** $\text{meq: } m = u \otimes_P p$
 using m **unfolding** poly-pideal-def **by** blast
 show $?thesis$ **using** $\text{meq poly-mult-closed}$ $[OF \ u \ p]$ **by** simp
qed
 have $mz: \text{pmod } m \ p = 0_P$ **using** m $\text{poly-pideal-iff-pmod}$ $[OF \ p \ \text{pnz } mc]$ **by** simp
— Reduce the combination: the multiple of p disappears.
 have $1_P = \text{pmod } 1_P \ p$ **using** pmod-small $[OF \ p \ \text{pnz } \text{poly-one-closed}] \ Kone \ K\text{small}$
by simp
 also have $\dots = \text{pmod } m \ p \oplus_P \text{pmod } (r \otimes_P S) \ p$
 using one pmod-add $[OF \ mc \ \text{poly-mult-closed } [OF \ r \ Sc] \ p \ \text{pnz}]$ **by** simp
 also have $\dots = \text{pmod } (r \otimes_P S) \ p$
 using mz pmod-closed $[OF \ \text{poly-mult-closed } [OF \ r \ Sc] \ p \ \text{pnz}]$ **by** (simp add:
 poly-add-zero)
 also have $\dots = \text{pmod } (\text{pmod } r \ p \otimes_P \text{pmod } S \ p) \ p$
 by (rule pmod-mult $[OF \ r \ Sc \ p \ \text{pnz}]$)
 also have $\dots = \text{mult } S \ (\text{pmod } r \ p)$
 unfolding mult-def **using** $K\text{small}$ $[OF \ S]$ poly-mult-comm $[OF \ Sc \ \text{pmod-closed}$
 $[OF \ r \ p \ \text{pnz}]]$ **by** simp
 finally have $\text{mult } S \ (\text{pmod } r \ p) = 1_P$ **by** simp
 moreover have $\text{pmod } r \ p \in K$ **unfolding** $K\text{-def}$ **using** $r \ p \ \text{pnz}$ **by** (rule
 pmod-low-poly)

```

ultimately show ?thesis by blast
qed
— Assemble.
have grp: Group K ( $\oplus_P$ )  $\mathbf{0}_P$ 
proof (rule GroupI)
  show  $\bigwedge x y. \llbracket x \in K; y \in K \rrbracket \implies x \oplus_P y \in K$  by (rule Kadd)
  show  $\mathbf{0}_P \in K$  by (rule Kzero)
  show  $\bigwedge x y z. \llbracket x \in K; y \in K; z \in K \rrbracket \implies (x \oplus_P y) \oplus_P z = x \oplus_P (y \oplus_P z)$ 
    using Kc by (blast intro: poly-add-assoc)
  show  $\bigwedge x. x \in K \implies \mathbf{0}_P \oplus_P x = x$  using Kc by (blast intro: poly-add-zero)
  show  $\bigwedge x. x \in K \implies x \oplus_P \mathbf{0}_P = x$ 
    using Kc poly-add-comm poly-zero-closed by (metis poly-add-zero)
  show  $\bigwedge x. x \in K \implies \exists y \in K. x \oplus_P y = \mathbf{0}_P \wedge y \oplus_P x = \mathbf{0}_P$ 
    using Kneg Kc poly-add-neg poly-add-neg-right by blast
qed
interpret A: Group K ( $\oplus_P$ )  $\mathbf{0}_P$  by (rule grp)
interpret A: Abelian-Group K ( $\oplus_P$ )  $\mathbf{0}_P$ 
  by unfold-locales (use Kc poly-add-comm in blast)
have mult-one': mult S  $\mathbf{1}_P = S$  if S:  $S \in K$  for S
proof —
  have mult S  $\mathbf{1}_P = \text{mult } \mathbf{1}_P S$  by (rule mult-comm [OF S Kone])
  also have  $\dots = S$  by (rule mult-one [OF S])
  finally show ?thesis .
qed
interpret M: commutative-monoid K mult  $\mathbf{1}_P$ 
proof unfold-locales
  show  $\bigwedge a b. \llbracket a \in K; b \in K \rrbracket \implies \text{mult } a b \in K$  by (rule Kmult)
  show  $\mathbf{1}_P \in K$  by (rule Kone)
  show  $\bigwedge a b c. \llbracket a \in K; b \in K; c \in K \rrbracket$ 
     $\implies \text{mult } (\text{mult } a b) c = \text{mult } a (\text{mult } b c)$  by (rule mult-assoc)
  show  $\bigwedge a. a \in K \implies \text{mult } \mathbf{1}_P a = a$  by (rule mult-one)
  show  $\bigwedge a. a \in K \implies \text{mult } a \mathbf{1}_P = a$  by (rule mult-one')
  show  $\bigwedge x y. \llbracket x \in K; y \in K \rrbracket \implies \text{mult } x y = \text{mult } y x$  by (rule mult-comm)
qed
show ?thesis
  unfolding sym [OF K-def] sym [OF mult-def]
proof unfold-locales
  show  $\bigwedge x y z. \llbracket x \in K; y \in K; z \in K \rrbracket$ 
     $\implies \text{mult } x (y \oplus_P z) = \text{mult } x y \oplus_P \text{mult } x z$  by (rule distrib)
  show  $\bigwedge x y z. \llbracket x \in K; y \in K; z \in K \rrbracket$ 
     $\implies \text{mult } (y \oplus_P z) x = \text{mult } y x \oplus_P \text{mult } z x$ 
proof —
  fix x y z assume K:  $x \in K y \in K z \in K$ 
  have mult  $(y \oplus_P z) x = \text{mult } x (y \oplus_P z)$ 
    by (rule mult-comm [OF Kadd [OF K(2) K(3)] K(1)])
  also have  $\dots = \text{mult } x y \oplus_P \text{mult } x z$  by (rule distrib [OF K])
  also have  $\dots = \text{mult } y x \oplus_P \text{mult } z x$ 
    using mult-comm [OF K(1) K(2)] mult-comm [OF K(1) K(3)] by simp
  finally show mult  $(y \oplus_P z) x = \text{mult } y x \oplus_P \text{mult } z x$  .
qed

```

```

show  $1_P \neq 0_P$  by (metis nontrivial poly-one-def poly-zero-def)
show  $\bigwedge x. \llbracket x \in K; x \neq 0_P \rrbracket \implies M.invertible\ x$ 
proof –
  fix  $x$  assume  $x: x \in K$  and  $xnz: x \neq 0_P$ 
  obtain  $T$  where  $T: T \in K$  and  $xT: mult\ x\ T = 1_P$  using inverse [OF  $x\ xnz$ ]
by blast
  have  $mult\ T\ x = 1_P$  using  $xT\ mult-comm$  [OF  $x\ T$ ] by simp
  with  $xT\ T$  show  $M.invertible\ x$  by (blast intro: M.invertibleI)
qed
qed
qed

```

The adjoined root really is a root. Evaluating in the field of remainders, at the variable and with the coefficients embedded as constants, reduces the polynomial modulo the modulus. So the class of q is the value of q at the variable, and when the modulus divides q that value is zero.

Reduction is a ring homomorphism from $F[X]$ onto the remainders, by *pmod-add* and *pmod-mult*, and it fixes the variable and the constants because those are already of low degree — which is where $2 \leq degree\ p$ is needed. So *eval-hom* applies, and the source evaluation is the polynomial itself by *eval-const-lift-at-var*.

theorem *eval-remainder-at-var*:

assumes *irr*: *poly-irreducible* p **and** $d2: 2 \leq degree\ p$ **and** $q: q \in poly-carrier$

shows $Ring.eval\ (low-poly\ (degree\ p))\ (\oplus_P)\ (\lambda S\ T. pmod\ (S \otimes_P T)\ p)$

$0_P\ 1_P\ X_P\ (\lambda k. poly-const\ (q\ k)) = pmod\ q\ p$

proof –

have $pc: p \in poly-carrier$ **using** *irr* **by** (*rule poly-irreducibleD-carrier*)

have $pnz: p \neq 0_P$ **using** *irr* **by** (*rule poly-irreducibleD-nonzero*)

have $RB: Ring\ (low-poly\ (degree\ p))\ (\oplus_P)\ (\lambda S\ T. pmod\ (S \otimes_P T)\ p)\ 0_P\ 1_P$

using *remainder-field* [*OF irr*] **by** (*simp add: Field-def commutative-ring-def*)

have $coeffR: q\ k \in R$ **for** k **using** q **by** (*rule poly-carrier-coeff-closed*)

have $constc: poly-const\ (q\ k) \in poly-carrier$ **for** k **using** $coeffR$ **by** (*rule poly-const-closed*)

— Reduction fixes the constants and the variable, both being of low degree.

have $fix-const: pmod\ (poly-const\ (q\ k))\ p = poly-const\ (q\ k)$ **for** k

proof (*rule pmod-small* [*OF pc pnz constc*])

have $degree\ (poly-const\ (q\ k)) = 0$ **using** *degree-const-le* **by** (*simp add: le-zero-eq*)

then show $poly-const\ (q\ k) = 0_P \vee degree\ (poly-const\ (q\ k)) < degree\ p$

using $d2$ **by** *simp*

qed

have $fix-var: pmod\ X_P\ p = X_P$

by (*rule pmod-small* [*OF pc pnz var-closed*])

(*use d2 degree-var* [*OF nontrivial*] **in** *simp*)

have $Ring.eval\ (low-poly\ (degree\ p))\ (\oplus_P)\ (\lambda S\ T. pmod\ (S \otimes_P T)\ p)$

$0_P\ 1_P\ (pmod\ X_P\ p)\ (\lambda k. pmod\ (poly-const\ (q\ k))\ p)$

$= pmod\ (Ring.eval\ poly-carrier\ (\oplus_P)\ (\otimes_P)\ 0_P\ 1_P\ X_P\ (\lambda k. poly-const\ (q\ k)))\ p$

proof (*rule eval-hom* [*OF poly-ring RB*])

show $\bigwedge S. S \in poly-carrier \implies pmod\ S\ p \in low-poly\ (degree\ p)$

using $pc\ pnz$ **by** (*blast intro: pmod-low-poly*)

show $pmod\ 0_P\ p = 0_P$ **by** (*rule pmod-small* [*OF pc pnz poly-zero-closed*]) *simp*

show $pmod\ 1_P\ p = 1_P$

```

    by (rule pmod-small [OF pc pnz poly-one-closed]) (use d2 degree-one in simp)
  show  $\bigwedge S T. \llbracket S \in \text{poly-carrier}; T \in \text{poly-carrier} \rrbracket$ 
     $\implies \text{pmod } (S \oplus_P T) p = \text{pmod } S p \oplus_P \text{pmod } T p$ 
    using pc pnz by (blast intro: pmod-add)
  show  $\bigwedge S T. \llbracket S \in \text{poly-carrier}; T \in \text{poly-carrier} \rrbracket$ 
     $\implies \text{pmod } (S \otimes_P T) p = \text{pmod } (\text{pmod } S p \otimes_P \text{pmod } T p) p$ 
    using pc pnz by (blast intro: pmod-mult)
  show  $X_P \in \text{poly-carrier}$  by (rule var-closed)
  show  $\bigwedge k. \text{poly-const } (q k) \in \text{poly-carrier}$  by (rule constc)
  show  $\text{Ring.degree } \mathbf{0}_P (\lambda k. \text{pmod } (\text{poly-const } (q k)) p)$ 
     $= \text{Ring.degree } \mathbf{0}_P (\lambda k. \text{poly-const } (q k))$ 
    using fix-const by simp
qed
then show ?thesis
  using fix-var fix-const eval-const-lift-at-var [OF q] by simp
qed

end

end

```

68 Polynomials in an Indexed Family of Indeterminates

```

theory Indexed-Poly
  imports Poly-Ring HOL-Library.Multiset
begin

```

Poly-Ring builds $R[X]$ from finite-support functions $\text{nat} \Rightarrow 'a$, the exponent of the single indeterminate being the index. Here the indeterminates form an arbitrary family, indexed by a type $'i$, and a monomial is therefore not a single exponent but a *multiset* over $'i$: the monomial $X_i^2 X_j$ is $\{\#i, i, j\}$. A polynomial is a finite-support function $'i \text{ multiset} \Rightarrow 'a$ with coefficients in the carrier, so the representation choice is the same one *Poly-Ring* makes, with the exponent *nat* replaced by a monomial. Addition is pointwise and multiplication is the convolution $(p \otimes q) m = (\sum n \subseteq \# m. p n \cdot q (m - n))$, the sum running over the submultisets of m — the monomial factorisations $m = n + (m - n)$ — exactly as the univariate convolution runs over $\{..k\}$, the factorisations $k = i + (k - i)$.

Why an indexed family, rather than iterating *Poly-Ring*. Iteration gives $R[X][Y]$ and so on, which is *finitely* many indeterminates. The construction of an algebraic closure needs one indeterminate for every polynomial over the base ring, all present at once, so that a maximal extension has room for a root of each. That is what this theory supplies.

Three facts used below — *fincomp-Sigma*, *fincomp-mult-distrib-left* and *fincomp-mult-distrib-right* — live in *Poly-Ring* but are generic in the index type and say nothing about polynomials.

68.1 Submultisets

The convolution sums over the submultisets of a monomial, so that index set must be finite. Each submultiset of $\text{add-mset } x \ m$ either omits x , and is a submultiset of m , or contains it, and is x added to one.

```

lemma finite-submultisets:  $\text{finite } \{n. n \subseteq\# (m :: 'i \text{ multiset})\}$ 
proof (induct m rule: multiset-induct)
  case empty
    have  $\{n. n \subseteq\# \{\#\}\} = \{\{\#\}\}$  by auto
    then show ?case by simp
next
  case (add x m)
  have  $\{n. n \subseteq\# \text{add-mset } x \ m\} \subseteq \{n. n \subseteq\# m\} \cup \text{add-mset } x \ ' \{n. n \subseteq\# m\}$ 
  proof
    fix  $n$  assume  $n \in \{n. n \subseteq\# \text{add-mset } x \ m\}$ 
    then have  $n: \bigwedge y. \text{count } n \ y \leq \text{count } (\text{add-mset } x \ m) \ y$ 
      by (simp add: subseteq-mset-def)
    show  $n \in \{n. n \subseteq\# m\} \cup \text{add-mset } x \ ' \{n. n \subseteq\# m\}$ 
    proof (cases x ∈# n)
      case True
        have  $\text{count } (n - \{\#x\# \}) \ y \leq \text{count } m \ y$  for  $y$ 
          using  $n$  [of y] True by (cases y = x) auto
        moreover from True have  $n = \text{add-mset } x \ (n - \{\#x\# \})$  by simp
        ultimately show ?thesis by (auto simp: subseteq-mset-def)
      next
      case False
        have  $\text{count } n \ y \leq \text{count } m \ y$  for  $y$ 
          using  $n$  [of y] False by (cases y = x) (auto simp: not-in-iff)
        then show ?thesis by (simp add: subseteq-mset-def)
    qed
  qed
  moreover have  $\text{finite } (\{n. n \subseteq\# m\} \cup \text{add-mset } x \ ' \{n. n \subseteq\# m\})$ 
    using add by blast
  ultimately show ?case by (rule finite-subset)
qed

```

Monomial arithmetic for the convolution, all at the level of coefficient counts. The last two are the identities that make the associativity reindexing below a bijection.

```

lemma msub-diff-mono:  $\llbracket a \subseteq\# b; b \subseteq\# m \rrbracket \implies b - a \subseteq\# m - a$ 
  using subset-eq-diff-conv by fastforce

```

```

lemma msub-add-closed:  $\llbracket a \subseteq\# m; b \subseteq\# m - a \rrbracket \implies a + b \subseteq\# m$ 
  by (simp add: add.commute subset-mset.le-diff-conv2)

```

```

lemma msub-diff-diff:  $\llbracket a \subseteq\# b; b \subseteq\# m \rrbracket \implies m - a - (b - a) = m - b$ 
  by (simp add: subseteq-mset-def multiset-eq-iff)

```

Complementation is an involution of the submultisets of a monomial; this is

what makes the convolution symmetric when the coefficients commute.

lemma *msub-double-diff*: $n \subseteq \# m \implies m - (m - n) = n$
by (*simp add: subseteq-mset-def multiset-eq-iff*)

context *Ring*
begin

68.2 Carrier and coefficients

The polynomials over R in the indeterminates indexed by $'i$: finite support on monomials, coefficients in the carrier.

definition *ipoly-carrier* :: $('i \text{ multiset} \Rightarrow 'a) \text{ set}$
where *ipoly-carrier* $\equiv \{p. \text{finite } \{m. p \ m \neq \mathbf{0}\} \wedge (\forall m. p \ m \in R)\}$

lemma *ipoly-carrierI*:
 $\llbracket \text{finite } \{m. p \ m \neq \mathbf{0}\}; \bigwedge m. p \ m \in R \rrbracket \implies p \in \text{ipoly-carrier}$
unfolding *ipoly-carrier-def* **by** *blast*

lemma *ipoly-carrier-finite*: $p \in \text{ipoly-carrier} \implies \text{finite } \{m. p \ m \neq \mathbf{0}\}$
unfolding *ipoly-carrier-def* **by** *blast*

lemma *ipoly-carrier-coeff-closed*: $p \in \text{ipoly-carrier} \implies p \ m \in R$
unfolding *ipoly-carrier-def* **by** *blast*

68.3 Operations

definition *ipoly-zero* :: $'i \text{ multiset} \Rightarrow 'a \ (\langle \mathbf{0}_I \rangle)$
where *ipoly-zero* $\equiv (\lambda m. \mathbf{0})$

The unit is the constant $\mathbf{1}$, carried by the empty monomial.

definition *ipoly-one* :: $'i \text{ multiset} \Rightarrow 'a \ (\langle \mathbf{1}_I \rangle)$
where *ipoly-one* $\equiv (\lambda m. \text{if } m = \{\#\} \text{ then } \mathbf{1} \text{ else } \mathbf{0})$

definition *ipoly-add* :: $('i \text{ multiset} \Rightarrow 'a) \Rightarrow ('i \text{ multiset} \Rightarrow 'a) \Rightarrow ('i \text{ multiset} \Rightarrow 'a)$
 $(\text{infixl } \langle \oplus_I \rangle \ 65)$
where *ipoly-add* $p \ q \equiv (\lambda m. p \ m + q \ m)$

definition *ipoly-neg* :: $('i \text{ multiset} \Rightarrow 'a) \Rightarrow ('i \text{ multiset} \Rightarrow 'a) \ (\langle \ominus_I \rightarrow [66] \ 65)$
where *ipoly-neg* $p \equiv (\lambda m. - p \ m)$

The convolution: a coefficient of the product sums over the factorisations $m = n + (m - n)$ of the monomial.

definition *ipoly-mult* :: $('i \text{ multiset} \Rightarrow 'a) \Rightarrow ('i \text{ multiset} \Rightarrow 'a) \Rightarrow ('i \text{ multiset} \Rightarrow 'a)$
 $(\text{infixl } \langle \otimes_I \rangle \ 70)$
where *ipoly-mult* $p \ q \equiv (\lambda m. \text{additive.fincomp } (\lambda n. p \ n \cdot q \ (m - n)) \ \{n. n \subseteq \# m\})$

68.4 Closure under the operations

lemma *ipoly-zero-closed*: $\mathbf{0}_I \in \text{ipoly-carrier}$
by (*rule ipoly-carrierI*) (*auto simp: ipoly-zero-def*)

lemma *ipoly-one-closed*: $\mathbf{1}_I \in \text{ipoly-carrier}$
by (*intro ipoly-carrierI*) (*auto simp: ipoly-one-def*)

lemma *ipoly-add-closed*:
assumes $p \in \text{ipoly-carrier}$ $q \in \text{ipoly-carrier}$
shows $p \oplus_I q \in \text{ipoly-carrier}$
proof (*rule ipoly-carrierI*)
have $\{m. (p \oplus_I q) m \neq \mathbf{0}\} \subseteq \{m. p m \neq \mathbf{0}\} \cup \{m. q m \neq \mathbf{0}\}$
using *additive.left-unit* **by** (*fastforce simp: additive.right-unit additive.unit-closed ipoly-add-def*)
moreover have *finite* $(\{m. p m \neq \mathbf{0}\} \cup \{m. q m \neq \mathbf{0}\})$
using *assms* **by** (*simp add: ipoly-carrier-finite*)
ultimately show *finite* $\{m. (p \oplus_I q) m \neq \mathbf{0}\}$ **by** (*rule finite-subset*)
show $\bigwedge m. (p \oplus_I q) m \in R$
using *assms* **by** (*simp add: ipoly-add-def ipoly-carrier-coeff-closed*)
qed

lemma *ipoly-neg-closed*:
assumes $p \in \text{ipoly-carrier}$ **shows** $\ominus_I p \in \text{ipoly-carrier}$
proof (*rule ipoly-carrierI*)
have $\{m. (\ominus_I p) m \neq \mathbf{0}\} \subseteq \{m. p m \neq \mathbf{0}\}$
unfolding *ipoly-neg-def* **using** *additive.inverse-unit* **by** *force*
then show *finite* $\{m. (\ominus_I p) m \neq \mathbf{0}\}$
using $p \in \text{ipoly-carrier-finite}$ *finite-subset* **by** *blast*
show $\bigwedge m. (\ominus_I p) m \in R$
using $p \in \text{ipoly-carrier}$ **by** (*simp add: ipoly-neg-def ipoly-carrier-coeff-closed*)
qed

lemma *ipoly-mult-coeff-closed*:
assumes $p \in \text{ipoly-carrier}$ $q \in \text{ipoly-carrier}$
shows $(p \otimes_I q) m \in R$
by (*simp add: assms ipoly-carrier-coeff-closed ipoly-mult-def*)

A product coefficient can be nonzero only at a monomial that factors as a support monomial of p times one of q , and there are finitely many such products. This replaces the degree bound used in the univariate case, where the support is bounded by a single number.

lemma *ipoly-mult-support-bound*:
assumes $p \in \text{ipoly-carrier}$ **and** $q \in \text{ipoly-carrier}$
shows $\{m. (p \otimes_I q) m \neq \mathbf{0}\} \subseteq (\lambda(a, b). a + b) ' (\{m. p m \neq \mathbf{0}\} \times \{m. q m \neq \mathbf{0}\})$
proof
fix m **assume** $m \in \{m. (p \otimes_I q) m \neq \mathbf{0}\}$
then have $\text{nz}: (p \otimes_I q) m \neq \mathbf{0}$ **by** *simp*
show $m \in (\lambda(a, b). a + b) ' (\{m. p m \neq \mathbf{0}\} \times \{m. q m \neq \mathbf{0}\})$
proof (*rule ccontr*)

```

assume  $m \notin (\lambda(a, b). a + b) \text{ ' } (\{m. p \ m \neq \mathbf{0}\} \times \{m. q \ m \neq \mathbf{0}\})$ 
then have  $p \cdot n \cdot q \ (m - n) = \mathbf{0}$  if  $n \in \{n. n \subseteq \# \ m\}$  for  $n$ 
  using that assms
  apply (simp add: image-iff ipoly-carrier-coeff-closed)
by (metis ipoly-carrier-coeff-closed left-zero right-zero subset-mset.add-diff-inverse)
then have  $(p \otimes_I q) \ m = \mathbf{0}$ 
  unfolding ipoly-mult-def by (rule additive.fincomp-unit-eqI)
with nz show False by simp
qed
qed

```

```

lemma ipoly-mult-closed:
  assumes  $p \in \text{ipoly-carrier}$  and  $q \in \text{ipoly-carrier}$ 
  shows  $p \otimes_I q \in \text{ipoly-carrier}$ 
proof (rule ipoly-carrierI)
  have finite  $(\{m. p \ m \neq \mathbf{0}\} \times \{m. q \ m \neq \mathbf{0}\})$ 
    using assms by (simp add: ipoly-carrier-finite)
  then show finite  $\{m. (p \otimes_I q) \ m \neq \mathbf{0}\}$ 
    using ipoly-mult-support-bound [OF assms] by (blast intro: finite-subset)
qed (simp add: ipoly-mult-coeff-closed assms)

```

68.5 Additive structure

Addition is pointwise, so the abelian-group laws are those of R coefficientwise.

```

lemma ipoly-add-assoc:
   $\llbracket p \in \text{ipoly-carrier}; q \in \text{ipoly-carrier}; r \in \text{ipoly-carrier} \rrbracket$ 
   $\implies (p \oplus_I q) \oplus_I r = p \oplus_I (q \oplus_I r)$ 
  by (auto simp: ipoly-add-def additive.associative ipoly-carrier-coeff-closed)

```

```

lemma ipoly-add-comm:
   $\llbracket p \in \text{ipoly-carrier}; q \in \text{ipoly-carrier} \rrbracket \implies p \oplus_I q = q \oplus_I p$ 
  by (auto simp: ipoly-add-def additive.commutative ipoly-carrier-coeff-closed)

```

```

lemma ipoly-add-zero:  $p \in \text{ipoly-carrier} \implies \mathbf{0}_I \oplus_I p = p$ 
  by (auto simp: ipoly-add-def ipoly-zero-def ipoly-carrier-coeff-closed)

```

```

lemma ipoly-add-neg:  $p \in \text{ipoly-carrier} \implies (\ominus_I p) \oplus_I p = \mathbf{0}_I$ 
  by (auto simp: ipoly-add-def ipoly-neg-def ipoly-zero-def ipoly-carrier-coeff-closed)

```

```

lemma ipoly-add-neg-right:  $p \in \text{ipoly-carrier} \implies p \oplus_I (\ominus_I p) = \mathbf{0}_I$ 
  using ipoly-add-comm ipoly-add-neg ipoly-neg-closed by force

```

68.6 Multiplicative unit

Only the empty monomial contributes to a product with the unit.

```

lemma ipoly-mult-one-left:
  assumes  $p: p \in \text{ipoly-carrier}$ 
  shows  $\mathbf{1}_I \otimes_I p = p$ 

```

```

proof
  fix  $m$ 
  have  $(1_I \otimes_I p) m = \text{additive.fincomp } (\lambda n. 1_I n \cdot p (m - n)) \{n. n \subseteq\# m\}$ 
    by (simp add: ipoly-mult-def)
  also have  $\dots = \text{additive.fincomp } (\lambda n. \text{if } n = \{\#\} \text{ then } (\lambda-. p m) n \text{ else } \mathbf{0}) \{n. n \subseteq\# m\}$ 
    using  $p$  by (intro additive.fincomp-cong')
      (auto simp: ipoly-one-def ipoly-carrier-coeff-closed)
  also have  $\dots = p m$ 
    by (intro additive.fincomp-singleton-swap) (auto simp: finite-submultisets ipoly-carrier-coeff-closed)
  finally show  $(1_I \otimes_I p) m = p m$  .
qed

```

lemma *ipoly-mult-one-right*:
assumes $p: p \in \text{ipoly-carrier}$
shows $p \otimes_I 1_I = p$

```

proof
  fix  $m$ 
  have  $(p \otimes_I 1_I) m = \text{additive.fincomp } (\lambda n. p n \cdot 1_I (m - n)) \{n. n \subseteq\# m\}$ 
    by (simp add: ipoly-mult-def)
  also have  $\dots = \text{additive.fincomp } (\lambda n. \text{if } n = m \text{ then } (\lambda-. p m) n \text{ else } \mathbf{0}) \{n. n \subseteq\# m\}$ 
    using  $p$ 
    by (intro additive.fincomp-cong')
      (auto simp: ipoly-one-def ipoly-carrier-coeff-closed Diff-eq-empty-iff-mset)
  also have  $\dots = p m$ 
    by (intro additive.fincomp-singleton-swap) (auto simp: finite-submultisets ipoly-carrier-coeff-closed)
  finally show  $(p \otimes_I 1_I) m = p m$  .
qed

```

68.7 Distributivity

lemma *ipoly-mult-add-distrib-left*:
assumes $p: p \in \text{ipoly-carrier}$ **and** $q: q \in \text{ipoly-carrier}$ **and** $r: r \in \text{ipoly-carrier}$
shows $p \otimes_I (q \oplus_I r) = (p \otimes_I q) \oplus_I (p \otimes_I r)$

```

proof
  fix  $m$ 
  have  $pi: \bigwedge n. p n \in R$  and  $qi: \bigwedge n. q n \in R$  and  $ri: \bigwedge n. r n \in R$ 
    using  $p q r$  by (auto simp: ipoly-carrier-coeff-closed)
  have  $(p \otimes_I (q \oplus_I r)) m$ 
     $= \text{additive.fincomp } (\lambda n. p n \cdot (q (m - n) + r (m - n))) \{n. n \subseteq\# m\}$ 
    by (simp add: ipoly-mult-def ipoly-add-def)
  also have  $\dots = \text{additive.fincomp } (\lambda n. p n \cdot q (m - n) + p n \cdot r (m - n)) \{n. n \subseteq\# m\}$ 
    using distributive(1)  $pi qi ri$  by presburger
  also have  $\dots = \text{additive.fincomp } (\lambda n. p n \cdot q (m - n)) \{n. n \subseteq\# m\}$ 
     $+ \text{additive.fincomp } (\lambda n. p n \cdot r (m - n)) \{n. n \subseteq\# m\}$ 
    by (rule additive.fincomp-comp) (use pi qi ri in auto)

```

also have $\dots = ((p \otimes_I q) \oplus_I (p \otimes_I r)) m$
 by (simp add: ipoly-mult-def ipoly-add-def)
 finally show $(p \otimes_I (q \oplus_I r)) m = ((p \otimes_I q) \oplus_I (p \otimes_I r)) m$.
 qed

lemma ipoly-mult-add-distrib-right:

assumes $p: p \in \text{ipoly-carrier}$ and $q: q \in \text{ipoly-carrier}$ and $r: r \in \text{ipoly-carrier}$
 shows $(q \oplus_I r) \otimes_I p = (q \otimes_I p) \oplus_I (r \otimes_I p)$
 proof
 fix m
 have $pi: \bigwedge n. p \ n \in R$ and $qi: \bigwedge n. q \ n \in R$ and $ri: \bigwedge n. r \ n \in R$
 using $p \ q \ r$ by (auto simp: ipoly-carrier-coeff-closed)
 have $((q \oplus_I r) \otimes_I p) m = \text{additive.fincomp } (\lambda n. (q \ n + r \ n) \cdot p \ (m - n)) \{n. n \subseteq\# m\}$
 by (simp add: ipoly-mult-def ipoly-add-def)
 also have $\dots = \text{additive.fincomp } (\lambda n. q \ n \cdot p \ (m - n) + r \ n \cdot p \ (m - n)) \{n. n \subseteq\# m\}$
 using distributive(2) $pi \ qi \ ri$ by presburger
 also have $\dots = \text{additive.fincomp } (\lambda n. q \ n \cdot p \ (m - n)) \{n. n \subseteq\# m\}$
 $\quad + \text{additive.fincomp } (\lambda n. r \ n \cdot p \ (m - n)) \{n. n \subseteq\# m\}$
 by (rule additive.fincomp-comp) (use $pi \ qi \ ri$ in auto)
 also have $\dots = ((q \otimes_I p) \oplus_I (r \otimes_I p)) m$
 by (simp add: ipoly-mult-def ipoly-add-def)
 finally show $((q \oplus_I r) \otimes_I p) m = ((q \otimes_I p) \oplus_I (r \otimes_I p)) m$.
 qed

68.8 Associativity of the convolution

Both sides are the sum of $p \ a \cdot q \ b \cdot r \ c$ over the factorisations $m = a + b + c$. The two nested sums present that set differently — as pairs $(b + a, a)$ on the left and (a, b) on the right — so the proof is a reindexing along $(b, a) \mapsto (a, b - a)$, exactly as in the univariate case, with monomial arithmetic in place of arithmetic on degrees.

lemma ipoly-mult-assoc:

fixes $p \ q \ r :: 'i \text{ multiset} \Rightarrow 'a$
 assumes $p: p \in \text{ipoly-carrier}$ and $q: q \in \text{ipoly-carrier}$ and $r: r \in \text{ipoly-carrier}$
 shows $(p \otimes_I q) \otimes_I r = p \otimes_I (q \otimes_I r)$
 proof (rule ext)
 fix $m :: 'i \text{ multiset}$
 have $pi: \bigwedge n. p \ n \in R$ and $qi: \bigwedge n. q \ n \in R$ and $ri: \bigwedge n. r \ n \in R$
 using $p \ q \ r$ by (auto simp: ipoly-carrier-coeff-closed)
 let $?S1 = \text{Sigma } \{b. b \subseteq\# m\} (\lambda b. 'i \text{ multiset. } \{a. a \subseteq\# b\})$
 let $?S2 = \text{Sigma } \{a. a \subseteq\# m\} (\lambda a. 'i \text{ multiset. } \{b. b \subseteq\# m - a\})$
 — The left side, as a sum over pairs (b, a) with $a \subseteq\# b \subseteq\# m$.
 have $((p \otimes_I q) \otimes_I r) m$
 $\quad = \text{additive.fincomp}$
 $\quad \quad (\lambda b. \text{additive.fincomp } (\lambda a. p \ a \cdot q \ (b - a)) \{a. a \subseteq\# b\} \cdot r \ (m - b))$
 $\quad \quad \{b. b \subseteq\# m\}$
 by (simp add: ipoly-mult-def)

```

also have ... = additive.fincomp
  (λb. additive.fincomp (λa. (p a · q (b - a)) · r (m - b)) {a. a ⊆# b})
    {b. b ⊆# m}
  using pi qi ri
  by (intro additive.fincomp-cong' fincomp-mult-distrib-right) auto
also have ... = additive.fincomp
  (λx. (p (snd x) · q (fst x - snd x)) · r (m - fst x)) ?S1
  by (rule fincomp-Sigma) (use pi qi ri finite-submultisets in auto)
finally have LHS: ((p ⊗I q) ⊗I r) m
  = additive.fincomp (λx. (p (snd x) · q (fst x - snd x)) · r (m - fst x)) ?S1 .
— The right side, as a sum over pairs (a, b) with a ⊆# m and b ⊆# m - a.
have (p ⊗I (q ⊗I r)) m
  = additive.fincomp
    (λa. p a · additive.fincomp (λb. q b · r (m - a - b)) {b. b ⊆# m - a})
      {a. a ⊆# m}
  by (simp add: ipoly-mult-def)
also have ... = additive.fincomp
  (λa. additive.fincomp (λb. p a · (q b · r (m - a - b))) {b. b ⊆# m
- a})
    {a. a ⊆# m}
  using pi qi ri
  by (intro additive.fincomp-cong' fincomp-mult-distrib-left) auto
also have ... = additive.fincomp
  (λx. p (fst x) · (q (snd x) · r (m - fst x - snd x))) ?S2
  by (rule fincomp-Sigma) (use pi qi ri finite-submultisets in auto)
finally have RHS: (p ⊗I (q ⊗I r)) m
  = additive.fincomp (λx. p (fst x) · (q (snd x) · r (m - fst x - snd x))) ?S2 .
— Reindex: (b, a) ↦ (a, b - a), with inverse (a, b) ↦ (a + b, a).
let ?h = λx :: 'i multiset × 'i multiset. (snd x, fst x - snd x)
let ?h' = λx :: 'i multiset × 'i multiset. (fst x + snd x, fst x)
have bij: bij-betw ?h ?S1 ?S2
proof (rule bij-betw-byWitness [where f' = ?h'])
  show ?h ' ?S1 ⊆ ?S2
  by (auto simp: msub-diff-mono intro: subset-mset.order-trans)
qed (auto simp: subset-mset.add-diff-inverse msub-add-closed)
let ?F = λx. p (fst x) · (q (snd x) · r (m - fst x - snd x))
have additive.fincomp ?F ?S2 = additive.fincomp ?F (?h ' ?S1)
  using bij bij-betw-imp-surj-on by fastforce
also have ... = additive.fincomp (λx. ?F (?h x)) ?S1
proof (rule additive.fincomp-reindex)
  show ?F ∈ ?h ' ?S1 → R using pi qi ri by auto
  show inj-on ?h ?S1 using bij bij-betw-imp-inj-on by blast
qed
also have ... = additive.fincomp (λx. (p (snd x) · q (fst x - snd x)) · r (m - fst
x)) ?S1
  using pi qi ri multiplicative.associative
  by (intro additive.fincomp-cong') (auto simp: msub-diff-diff)
finally show ((p ⊗I q) ⊗I r) m = (p ⊗I (q ⊗I r)) m
  using LHS RHS by simp
qed

```

68.9 The indexed polynomial ring

```

theorem ipoly-ring: Ring ipoly-carrier  $(\oplus_I)$   $(\otimes_I)$   $\mathbf{0}_I$   $\mathbf{1}_I$ 
proof –
  have add-grp: Group ipoly-carrier  $(\oplus_I)$   $\mathbf{0}_I$ 
  proof (rule GroupI)
    show  $\bigwedge p. p \in \text{ipoly-carrier} \implies p \oplus_I \mathbf{0}_I = p$ 
    by (metis ipoly-add-comm ipoly-add-zero ipoly-zero-closed)
    show  $\bigwedge p. p \in \text{ipoly-carrier} \implies \exists q \in \text{ipoly-carrier}. p \oplus_I q = \mathbf{0}_I \wedge q \oplus_I p = \mathbf{0}_I$ 
    using ipoly-add-neg ipoly-add-neg-right ipoly-neg-closed by blast
  qed (auto simp: ipoly-add-closed ipoly-zero-closed ipoly-add-assoc ipoly-add-zero)
  interpret add: Group ipoly-carrier  $(\oplus_I)$   $\mathbf{0}_I$  by (rule add-grp)
  interpret add: Abelian-Group ipoly-carrier  $(\oplus_I)$   $\mathbf{0}_I$ 
    by unfold-locales (rule ipoly-add-comm)
  interpret mult: Monoid ipoly-carrier  $(\otimes_I)$   $\mathbf{1}_I$ 
    by (simp add: Monoid.intro ipoly-mult-assoc ipoly-mult-closed ipoly-mult-one-left
      ipoly-mult-one-right ipoly-one-closed)
  show ?thesis
  proof qed (use ipoly-mult-add-distrib-left ipoly-mult-add-distrib-right in auto)
qed

```

68.10 Constants and indeterminates

A constant carries its value on the empty monomial. This is the embedding of the coefficient ring, and it is what lets an extension of R be built inside this ring.

```

definition ipoly-const :: 'a  $\Rightarrow$  ('i multiset  $\Rightarrow$  'a)
  where ipoly-const a  $\equiv$  ( $\lambda m.$  if  $m = \{\#\}$  then a else  $\mathbf{0}$ )

```

```

lemma ipoly-const-coeff [simp]: ipoly-const a  $\{\#\} = a$ 
  by (simp add: ipoly-const-def)

```

```

lemma ipoly-const-closed:  $a \in R \implies \text{ipoly-const } a \in \text{ipoly-carrier}$ 
  by (auto simp: ipoly-mult-def ipoly-const-def ipoly-carrierI)

```

```

lemma ipoly-const-zero [simp]: ipoly-const  $\mathbf{0} = \mathbf{0}_I$ 
  by (simp add: ipoly-const-def ipoly-zero-def fun-eq-iff)

```

```

lemma ipoly-const-one [simp]: ipoly-const  $\mathbf{1} = \mathbf{1}_I$ 
  by (simp add: ipoly-const-def ipoly-one-def fun-eq-iff)

```

```

lemma ipoly-const-add:
  assumes  $a \in R$  and  $b \in R$ 
  shows ipoly-const (a + b) = (ipoly-const a :: 'i multiset  $\Rightarrow$  'a)  $\oplus_I$  ipoly-const b
  using assms by (auto simp: ipoly-const-def ipoly-add-def fun-eq-iff additive.right-unit)

```

```

lemma ipoly-const-mult:
  assumes  $a \in R$   $b \in R$ 
  shows ipoly-const (a · b) = (ipoly-const a :: 'i multiset  $\Rightarrow$  'a)  $\otimes_I$  ipoly-const b

```

using *assms* **by** (*auto simp: ipoly-mult-def ipoly-const-def additive.fincomp-unit-eqI*)

lemma *ipoly-const-inj-on*: *inj-on (ipoly-const :: 'a $\Rightarrow$ 'i multiset $\Rightarrow$ 'a) R*
using *ipoly-const-coeff* **by** (*metis inj-on-def*)

The indeterminate X_i is the monomial $\{\#i\#$ with coefficient $\mathbf{1}$. Distinct indices give distinct indeterminates whenever the ring is nontrivial: note that *Ring* does not itself assume $\mathbf{1} \neq \mathbf{0}$.

definition *ivar* :: *'i $\Rightarrow$ ('i multiset $\Rightarrow$ 'a)*
where *ivar* *i* $\equiv (\lambda m. \text{if } m = \{\#i\# \text{ then } \mathbf{1} \text{ else } \mathbf{0})$

lemma *ivar-closed*: *ivar* *i* $\in$ *ipoly-carrier*
by (*rule ipoly-carrierI*) (*auto simp: ivar-def*)

lemma *ivar-inj*:
assumes $\mathbf{1} \neq \mathbf{0}$
shows *inj (ivar :: 'i $\Rightarrow$ 'i multiset $\Rightarrow$ 'a)*
using *assms* **by** (*auto simp: ivar-def inj-on-def fun-eq-iff*)

68.11 Indeterminates a polynomial does not involve

$\mathcal{P}$ does not involve the indeterminate j when no monomial containing j carries a nonzero coefficient. An index that *no* element of an extension involves is what supplies room for a new root: adjoining one is then guaranteed not to disturb what is there already. This replaces the cardinality argument that a construction over an abstract type of labels would need, and is the reason for choosing indexed polynomials as the ambient type.

definition *index-free* :: *('i multiset $\Rightarrow$ 'a) $\Rightarrow$ 'i $\Rightarrow$ bool*
where *index-free* $\mathcal{P}$ *j* $\equiv (\forall m. j \in \# m \longrightarrow \mathcal{P} m = \mathbf{0})$

lemma *index-freeI*: $(\bigwedge m. j \in \# m \implies \mathcal{P} m = \mathbf{0}) \implies \text{index-free } \mathcal{P} j$
by (*simp add: index-free-def*)

lemma *index-freeD*: $\llbracket \text{index-free } \mathcal{P} j; j \in \# m \rrbracket \implies \mathcal{P} m = \mathbf{0}$
by (*simp add: index-free-def*)

Constants involve no indeterminate at all.

lemma *index-free-const* [*simp*]: *index-free (ipoly-const a) j*
by (*rule index-freeI*) (*auto simp: ipoly-const-def*)

lemma *index-free-zero* [*simp*]: *index-free* $\mathbf{0}_I$ *j*
by (*rule index-freeI*) (*simp add: ipoly-zero-def*)

The indeterminate *ivar* j does involve j , provided the coefficient ring is nontrivial — *Ring* does not itself assume $\mathbf{1} \neq \mathbf{0}$.

lemma *not-index-free-ivar*:
assumes $\mathbf{1} \neq \mathbf{0}$ **shows** $\neg \text{index-free (ivar } j) j$
using *assms* *index-freeD* *ivar-def* **by** (*metis multi-member-last*)

68.12 Monomials

The monomial *imonom* $a\ n$ carries the coefficient a on the monomial n . Constants and indeterminates are the special cases $n = \{\#\}$ and $a = \mathbf{1}$ with n a singleton.

definition *imonom* $:: 'a \Rightarrow 'i\ multiset \Rightarrow ('i\ multiset \Rightarrow 'a)$
where *imonom* $a\ n \equiv (\lambda m. \text{if } m = n \text{ then } a \text{ else } \mathbf{0})$

lemma *imonom-apply* [simp]: *imonom* $a\ n\ n = a$
by (simp add: *imonom-def*)

lemma *imonom-closed*: $a \in R \implies \text{imonom } a\ n \in \text{ipoly-carrier}$

proof (rule *ipoly-carrierI*)
show *finite* $\{m. \text{imonom } a\ m \neq \mathbf{0}\}$
by (rule *finite-subset* [where $B = \{n\}$]) (auto simp: *imonom-def*)
qed (simp add: *imonom-def*)

lemma *ipoly-const-eq-imonom*: *ipoly-const* $a = \text{imonom } a\ \{\#\}$
by (simp add: *ipoly-const-def imonom-def*)

lemma *ivar-eq-imonom*: *ivar* $j = \text{imonom } \mathbf{1}\ \{\#j\#\}$
by (simp add: *ivar-def imonom-def*)

lemma *imonom-mult*:
fixes $n\ n' :: 'i\ multiset$
assumes $a: a \in R$ **and** $b: b \in R$
shows *imonom* $a\ n \otimes_I \text{imonom } b\ n' = \text{imonom } (a \cdot b)\ (n + n')$
proof
fix m
show $(\text{imonom } a\ n \otimes_I \text{imonom } b\ n')\ m = \text{imonom } (a \cdot b)\ (n + n')\ m$
proof (cases $m = n + n'$)
case *True*
have $n_{\text{sub}}: n \subseteq\# m$ **using** *True* **by** *simp*
have $(\text{imonom } a\ n \otimes_I \text{imonom } b\ n')\ m$
 $= \text{additive.fincomp } (\lambda p. \text{if } p = n \text{ then } (\lambda-. a \cdot b)\ p \text{ else } \mathbf{0})\ \{p. p \subseteq\# m\}$
unfolding *ipoly-mult-def*
using $a\ b\ \text{True}$ **by** (intro *additive.fincomp-cong'*) (auto simp: *imonom-def*)
also have $\dots = a \cdot b$
using $n_{\text{sub}}\ a\ b$
by (intro *finite-submultisets additive.fincomp-singleton-swap*) *auto*
finally show *?thesis* **using** *True* **by** *simp*
next
case *False*
with $a\ b$ **have** *imonom* $a\ n\ p \cdot \text{imonom } b\ n'\ (m - p) = \mathbf{0}$ **if** $p \in \{p. p \subseteq\# m\}$
for p
using *that a b* **by** (auto simp: *imonom-def subset-mset.add-diff-inverse*)
then have $(\text{imonom } a\ n \otimes_I \text{imonom } b\ n')\ m = \mathbf{0}$
unfolding *ipoly-mult-def* **by** (rule *additive.fincomp-unit-eqI*)
then show *?thesis* **using** *False* **by** (simp add: *imonom-def*)

```

qed
qed

lemma imonom-shift:
  fixes  $n\ m :: 'i\ multiset$ 
  assumes  $c: c \in ipoly\text{-}carrier$ 
  shows  $(c \otimes_I imonom\ 1\ n)\ m = (if\ n \subseteq\# m\ then\ c\ (m - n)\ else\ 0)$ 
proof (cases  $n \subseteq\# m$ )
  case True
  — For  $p \subseteq\# m$  the second factor is 1 exactly at  $p = m - n$ .
  then have unique:  $(m - p = n) = (p = m - n)$  if  $p \subseteq\# m$  for  $p$ 
    using that by (auto simp: msub-double-diff)
  have §:  $\forall m. c\ m \in R$ 
    by (meson Ring.ipoly-carrier-coeff-closed Ring-axioms c)
  have  $(c \otimes_I imonom\ 1\ n)\ m = additive.fincomp\ (\lambda p. if\ p = m - n\ then\ (\lambda p. c\ p)\ p\ else\ 0)\ \{p. p \subseteq\# m\}$ 
    unfolding ipoly-mult-def
    using c by (intro additive.fincomp-cong')
    (auto simp: imonom-def unique ipoly-carrier-coeff-closed)
  also have  $\dots = c\ (m - n)$ 
    using § by (simp add: additive.fincomp-singleton-swap finite-submultisets)
  finally show ?thesis using True by simp
next
  case False
  then have  $c\ p \cdot imonom\ 1\ n\ (m - p) = 0$  if  $p \in \{p. p \subseteq\# m\}$  for  $p$ 
    by (metis c imonom-def ipoly-carrier-coeff-closed right-zero diff-subset-eq-self)
  then have  $(c \otimes_I imonom\ 1\ n)\ m = 0$ 
    unfolding ipoly-mult-def by (rule additive.fincomp-unit-eqI)
  then show ?thesis using False by simp
qed

```

A power of an indeterminate is the monomial with that exponent.

```

lemma ivar-pow:
  fixes  $j :: 'i$ 
  shows  $Ring.rpow\ (\otimes_I)\ 1_I\ (ivar\ j)\ k = imonom\ 1\ (replicate\ mset\ k\ j)$ 
proof (induct  $k$ )
  case 0
  have  $Ring.rpow\ (\otimes_I)\ 1_I\ (ivar\ j)\ 0 = 1_I$ 
    by (rule Ring.rpow-0 [OF ipoly-ring])
  then show ?case
    by (simp add: ipoly-one-def imonom-def)
next
  case (Suc  $k$ )
  have  $Ring.rpow\ (\otimes_I)\ 1_I\ (ivar\ j)\ (Suc\ k) = ivar\ j \otimes_I Ring.rpow\ (\otimes_I)\ 1_I\ (ivar\ j)\ k$ 
    by (rule Ring.rpow-Suc [OF ipoly-ring])
  then show ?case
    using Suc by (simp add: ivar-eq-imonom imonom-mult)
qed

```

68.13 Reading a polynomial off its value at an indeterminate

The point of the monomial lemmas above. A sum $\sum_k c_k X_j^k$ whose coefficients do not involve X_j determines those coefficients: its value on a monomial m is the coefficient indexed by the multiplicity of j in m , evaluated at what remains of m . So such a sum is injective in the coefficients, which is what makes the realisation of a quotient $M[X]/(Q)$ inside this ring faithful.

lemma *ipoly-add-cmonoid: commutative-monoid ipoly-carrier $(\oplus_I) \mathbf{0}_I$*
using *ipoly-ring by (meson Abelian-Group-def Ring-def)*

Addition is pointwise, so a finite sum may be evaluated coefficientwise.

lemma *ipoly-fincomp-apply:*
fixes $m :: 'i \text{ multiset}$
assumes $f: \bigwedge k. f\ k \in \text{ipoly-carrier}$
shows $\text{commutative-monoid.fincomp ipoly-carrier } (\oplus_I) \mathbf{0}_I f\ A\ m = \text{additive.fincomp } (\lambda k. f\ k\ m)\ A$
proof (*induction A rule: infinite-finite-induct*)
case empty
have $\text{commutative-monoid.fincomp ipoly-carrier } (\oplus_I) \mathbf{0}_I f\ \{\} = \mathbf{0}_I$
by (*rule commutative-monoid.fincomp-empty [OF ipoly-add-cmonoid]*)
then show $?case$ **by** (*simp add: ipoly-zero-def*)
next
case (insert a A)
have $f \in A \rightarrow \text{ipoly-carrier } f\ a \in \text{ipoly-carrier}$ **using** f **by** *auto*
then have $\text{commutative-monoid.fincomp ipoly-carrier } (\oplus_I) \mathbf{0}_I f\ (\text{insert } a\ A)\ m$
 $= f\ a\ m + \text{commutative-monoid.fincomp ipoly-carrier } (\oplus_I) \mathbf{0}_I f\ A\ m$
using $\text{commutative-monoid.fincomp-insert [OF ipoly-add-cmonoid]}$ *insert.hyps*
by (*smt (verit) ipoly-add-def*)
also have $\dots = \text{additive.fincomp } (\lambda k. f\ k\ m)\ (\text{insert } a\ A)$
using *insert.IH* **by** (*simp add: f insert.hyps ipoly-carrier-coeff-closed*)
finally show $?case$ **.**
next
case infinite
then show $?case$
using *additive.commutative-monoid-axioms ipoly-add-cmonoid ipoly-zero-def*
by (*metis commutative-monoid.fincomp-def*)
qed

lemma *replicate-mset-subseteq-count: replicate-mset $k\ j \subseteq\# m \longleftrightarrow k \leq \text{count } m\ j$*
by (*auto simp: subseteq-mset-def split: if-split-asm*)

The closed form: only the term whose exponent matches the multiplicity of j in m survives, because every earlier coefficient is evaluated at a monomial that still involves j , where it vanishes.

theorem *ieval-ivar:*
fixes $j :: 'i$ **and** $m :: 'i \text{ multiset}$
assumes $c: \bigwedge k. c\ k \in \text{ipoly-carrier}$ **and** $\text{free}: \bigwedge k. \text{index-free } (c\ k)\ j$

```

shows commutative-monoid.fincomp ipoly-carrier ( $\oplus_I$ )  $\mathbf{0}_I$ 
  ( $\lambda k. c\ k \otimes_I \text{Ring.rpow } (\otimes_I) \mathbf{1}_I (\text{ivar } j)\ k) \{..d\}\ m$ 
  = (if count  $m\ j \leq d$  then  $c\ (\text{count } m\ j)\ (m - \text{replicate-mset } (\text{count } m\ j)\ j)$  else
0)
proof –
  have  $cl: c\ k \otimes_I \text{Ring.rpow } (\otimes_I) \mathbf{1}_I (\text{ivar } j)\ k \in \text{ipoly-carrier}$  for  $k$ 
    using  $c$  by (simp add: ivar-pow ipoly-mult-closed imonom-closed)
  have term-eq:  $(c\ k \otimes_I \text{Ring.rpow } (\otimes_I) \mathbf{1}_I (\text{ivar } j)\ k)\ m$ 
    = (if  $k = \text{count } m\ j$ 
      then  $c\ (\text{count } m\ j)\ (m - \text{replicate-mset } (\text{count } m\ j)\ j)$  else  $\mathbf{0}$ ) for  $k$ 
  proof –
    have  $(c\ k \otimes_I \text{Ring.rpow } (\otimes_I) \mathbf{1}_I (\text{ivar } j)\ k)\ m$ 
      = (if  $\text{replicate-mset } k\ j \subseteq\# m$  then  $c\ k\ (m - \text{replicate-mset } k\ j)$  else  $\mathbf{0}$ )
    by (simp add: ivar-pow imonom-shift [OF c])
    also have ... = (if  $k = \text{count } m\ j$  then  $c\ (\text{count } m\ j)\ (m - \text{replicate-mset } (\text{count } m\ j)\ j)$  else  $\mathbf{0}$ )
      using index-freeD [OF free]
      by (cases  $k < \text{count } m\ j$  (auto simp: in-diff-count replicate-mset-subseteq-count))
    finally show ?thesis .
  qed
  have commutative-monoid.fincomp ipoly-carrier ( $\oplus_I$ )  $\mathbf{0}_I$ 
    ( $\lambda k. c\ k \otimes_I \text{Ring.rpow } (\otimes_I) \mathbf{1}_I (\text{ivar } j)\ k) \{..d\}\ m$ 
    = additive.fincomp ( $\lambda k. (c\ k \otimes_I \text{Ring.rpow } (\otimes_I) \mathbf{1}_I (\text{ivar } j)\ k)\ m) \{..d\}$ 
    by (rule ipoly-fincomp-apply [OF cl])
  also have ... = additive.fincomp
    ( $\lambda k. \text{if } k = \text{count } m\ j$ 
      then  $(\lambda-. c\ (\text{count } m\ j)\ (m - \text{replicate-mset } (\text{count } m\ j)\ j))\ k$ 
      else  $\mathbf{0}$ )  $\{..d\}$ 
    using term-eq by simp
  also have ... = (if count  $m\ j \leq d$ 
    then  $c\ (\text{count } m\ j)\ (m - \text{replicate-mset } (\text{count } m\ j)\ j)$  else  $\mathbf{0}$ )
  proof (cases count  $m\ j \leq d$ )
    case True
    then show ?thesis
      using assms(1)
      by (auto intro!: ipoly-carrier-coeff-closed additive.fincomp-singleton-swap)
    next
    case False then show ?thesis
      by (force intro!: additive.fincomp-unit-eqI)
  qed
  finally show ?thesis .
qed

```

What the closed form is for: the coefficients can be recovered, so a polynomial in *ivar j* with index-free coefficients determines them. Reading *ieval-ivar* at the monomial $n + \text{replicate-mset } k\ j$ isolates the k th coefficient at n — provided n itself avoids j , and where it does not, both coefficients vanish anyway. This is the injectivity that lets a quotient be realised by normal forms: distinct remainders go to distinct elements of the ambient ring.

```

lemma ieval-ivar-inj:
  fixes  $j :: 'i$  and  $D \ k :: nat$ 
  assumes  $c: \bigwedge k. c \ k \in ipoly\text{-}carrier$  and  $cfree: \bigwedge k. index\text{-}free \ (c \ k) \ j$ 
  and  $d: \bigwedge k. d \ k \in ipoly\text{-}carrier$  and  $dfree: \bigwedge k. index\text{-}free \ (d \ k) \ j$ 
  and  $eq: commutative\text{-}monoid.fincomp \ ipoly\text{-}carrier \ (\oplus_I) \ \mathbf{0}_I$ 
     $(\lambda k. c \ k \otimes_I Ring.rpow \ (\otimes_I) \ \mathbf{1}_I \ (ivar \ j) \ k) \ \{..D\}$ 
     $= commutative\text{-}monoid.fincomp \ ipoly\text{-}carrier \ (\oplus_I) \ \mathbf{0}_I$ 
     $(\lambda k. d \ k \otimes_I Ring.rpow \ (\otimes_I) \ \mathbf{1}_I \ (ivar \ j) \ k) \ \{..D\}$ 
  and  $k: k \leq D$ 
  shows  $c \ k = d \ k$ 
proof (rule ext)
  fix  $n :: 'i$  multiset
  show  $c \ k \ n = d \ k \ n$ 
proof (cases  $j \in \# \ n$ )
  case True
  — Both coefficients are index-free at  $j$ , so both vanish here.
  show ?thesis
    using  $index\text{-}freeD \ [OF \ cfree \ True] \ index\text{-}freeD \ [OF \ dfree \ True]$  by simp
next
  case False
  define  $m$  where  $m \equiv n + replicate\text{-}mset \ k \ j$ 
  have  $cm: count \ m \ j = k$  using False by (simp add: m-def not-in-iff)
  have  $mn: m - replicate\text{-}mset \ k \ j = n$  by (simp add: m-def)
  have  $ceq: commutative\text{-}monoid.fincomp \ ipoly\text{-}carrier \ (\oplus_I) \ \mathbf{0}_I$ 
     $(\lambda k. c \ k \otimes_I Ring.rpow \ (\otimes_I) \ \mathbf{1}_I \ (ivar \ j) \ k) \ \{..D\} \ m$ 
     $= (if \ count \ m \ j \leq D$ 
       $then \ c \ (count \ m \ j) \ (m - replicate\text{-}mset \ (count \ m \ j) \ j) \ else \ \mathbf{0})$ 
    by (ruleieval-ivar [OF  $c \ cfree$ ])
  have  $deq: commutative\text{-}monoid.fincomp \ ipoly\text{-}carrier \ (\oplus_I) \ \mathbf{0}_I$ 
     $(\lambda k. d \ k \otimes_I Ring.rpow \ (\otimes_I) \ \mathbf{1}_I \ (ivar \ j) \ k) \ \{..D\} \ m$ 
     $= (if \ count \ m \ j \leq D$ 
       $then \ d \ (count \ m \ j) \ (m - replicate\text{-}mset \ (count \ m \ j) \ j) \ else \ \mathbf{0})$ 
    by (ruleieval-ivar [OF  $d \ dfree$ ])
  from  $ceq \ deq \ eq \ cm \ mn \ k$  show ?thesis by simp
qed
qed
end

```

69 Polynomials over a Commutative Ring

```

context commutative-ring
begin

```

When the coefficients commute so do the polynomials: the convolution is symmetric under the involution $n \mapsto m - n$ of the submultisets of m .

```

lemma ipoly-mult-comm:
  fixes  $p \ q :: 'i$  multiset  $\Rightarrow 'a$ 
  assumes  $p: p \in ipoly\text{-}carrier$  and  $q: q \in ipoly\text{-}carrier$ 

```

```

shows  $p \otimes_I q = q \otimes_I p$ 
proof
  fix  $m :: 'i \text{ multiset}$ 
  have  $pi: \bigwedge n. p \ n \in R$  and  $qi: \bigwedge n. q \ n \in R$ 
    using  $p \ q$  by (auto simp: ipoly-carrier-coeff-closed)
  have  $bij: bij\text{-}betw \ (\lambda n :: 'i \text{ multiset}. m - n) \ \{n. n \subseteq\# m\} \ \{n. n \subseteq\# m\}$ 
    by (rule  $bij\text{-}betw\text{-}byWitness$  [where  $f' = \lambda n :: 'i \text{ multiset}. m - n$ ])
      (auto simp:  $m\text{-}sub\text{-}double\text{-}diff \ diff\text{-}subset\text{-}eq\text{-}self$ )
  have  $(p \otimes_I q) \ m = additive.\text{fincomp} \ (\lambda n. q \ (m - n) \cdot p \ n) \ \{n. n \subseteq\# m\}$ 
    by (simp add:  $ipoly\text{-}mult\text{-}def \ multiplicative.\text{commutative} \ pi \ qi$ )
  also have  $\dots = additive.\text{fincomp} \ (\lambda n. q \ (m - n) \cdot p \ n)$ 
    (( $\lambda n :: 'i \text{ multiset}. m - n$ ) '  $\{n. n \subseteq\# m\}$ )
    using  $bij \ bij\text{-}betw\text{-}imp\text{-}surj\text{-}on$  by fastforce
  also have  $\dots = additive.\text{fincomp} \ (\lambda n. q \ (m - (m - n)) \cdot p \ (m - n)) \ \{n. n \subseteq\# m\}$ 
    by (rule  $additive.\text{fincomp}\text{-}reindex$ )
      (use  $pi \ qi \ bij \ bij\text{-}betw\text{-}imp\text{-}inj\text{-}on$  in auto)
  also have  $\dots = additive.\text{fincomp} \ (\lambda n. q \ n \cdot p \ (m - n)) \ \{n. n \subseteq\# m\}$ 
    using  $pi \ qi$  by (intro  $additive.\text{fincomp}\text{-}cong'$ ) (auto simp:  $m\text{-}sub\text{-}double\text{-}diff$ )
  also have  $\dots = (q \otimes_I p) \ m$  by (simp add:  $ipoly\text{-}mult\text{-}def$ )
  finally show  $(p \otimes_I q) \ m = (q \otimes_I p) \ m$  .
qed

theorem ipoly-commutative-ring:
  commutative-ring (ipoly-carrier :: ('i multiset  $\Rightarrow$  'a) set) ( $\oplus_I$ ) ( $\otimes_I$ )  $\mathbf{0}_I \ \mathbf{1}_I$ 
proof -
  interpret  $P$ : Ring ipoly-carrier :: ('i multiset  $\Rightarrow$  'a) set ( $\oplus_I$ ) ( $\otimes_I$ )  $\mathbf{0}_I \ \mathbf{1}_I$ 
    by (rule ipoly-ring)
  show ?thesis
  proof qed (simp add: ipoly-mult-comm)
qed

end

```

70 The Univariate Case, as a Check on the Convolution

With a single indeterminate — index type *unit* — a monomial is determined by its size, so the submultisets of X^k are exactly X^i for $i \leq k$, and the indexed convolution must reduce to the univariate one of *Poly-Ring*. This is an independent check on both the index set of the convolution and the placement of $m - n$ within it: a sum over the wrong factorisations of a monomial would still build a ring, and would still satisfy every law proved above, but it would not agree with *Poly-Ring* here.

lemma *unit-mset-eq-replicate*: $(n :: unit \text{ multiset}) = replicate\text{-}mset \ (count \ n \ ()) \ ()$
 by (simp add: *multiset-eq-iff*)

```

lemma submultisets-replicate-unit:
  {n :: unit multiset. n ⊆# replicate-mset k ()} = (λi. replicate-mset i ()) ‘{..k}
using unit-mset-eq-replicate
by (auto simp: subseteq-mset-def image-iff)

lemma replicate-unit-diff:
  replicate-mset k () - replicate-mset i () = replicate-mset (k - i) ()
by (simp add: multiset-eq-iff)

lemma inj-on-replicate-unit: inj-on (λi. replicate-mset i ()) {..k}
by (rule inj-on-inverseI [where g = λn. count n ()]) simp

context Ring
begin

theorem ipoly-mult-unit-index:
  fixes p q :: unit multiset ⇒ 'a
  assumes p: p ∈ ipoly-carrier and q: q ∈ ipoly-carrier
  shows (p ⊗I q) (replicate-mset k ())
    = ((λi. p (replicate-mset i ())) ⊗P (λi. q (replicate-mset i ()))) k
proof -
  have pi: ∧n. p n ∈ R and qi: ∧n. q n ∈ R
    using p q by (auto simp: ipoly-carrier-coeff-closed)
  have (p ⊗I q) (replicate-mset k ())
    = additive.fincomp (λn. p n · q (replicate-mset k () - n))
      ((λi. replicate-mset i ()) ‘{..k})
    by (simp add: ipoly-mult-def submultisets-replicate-unit)
  also have ... = additive.fincomp
    (λi. p (replicate-mset i ())
      · q (replicate-mset k () - replicate-mset i ())) {..k}
    by (rule additive.fincomp-reindex) (use pi qi inj-on-replicate-unit in auto)
  also have ... = ((λi. p (replicate-mset i ())) ⊗P (λi. q (replicate-mset i ()))) k
    by (simp add: replicate-unit-diff poly-mult-def)
  finally show ?thesis .
qed

end

end

```

71 Extensions of a Field inside the Indexed Polynomial Ring

```

theory Closure-Extension
imports Indexed-Poly Poly-Divisibility HOL.Zorn
begin

```

Groundwork for the existence of an algebraic closure. Every extension needed for that is generated by adjoining roots of polynomials over the base field R ,

so all of them can be built inside *one* fixed type: the indexed polynomials of *Indexed-Poly* whose indeterminates are indexed by pairs (P, i) — a polynomial over R together with a natural number, so that each polynomial has as many indeterminates available as it could have roots.

Fixing the type in advance is what makes Zorn’s lemma applicable: the extensions then form a *set*, which can be ordered and exhausted. The alternative, iterating a quotient construction to adjoin one root at a time, changes the type at every step — a quotient of a ring over $'a$ has carrier of type $'a$ *set* — and so cannot be iterated by induction at all.

An extension is a triple $(K, \oplus, \otimes)$: a carrier with its own addition and multiplication. The operations must be allowed to vary, because an extension realises a quotient $M[X]/(Q)$ concretely, by normal forms, and the modulus Q differs from step to step. The zero and the unit, by contrast, are the same throughout: the constants $\mathbf{0}_I$ and $\mathbf{1}_I$.

Why the operations are required to be undefined off the carrier. Two extensions with the same carrier and the same operations *on* that carrier are the same extension mathematically, but need not be equal as triples if the operations disagree elsewhere — and then the ordering below fails antisymmetry, so it is not a partial order and Zorn does not apply. HOL-Algebra repairs this after the fact with a *law-restrict* operator and a family of lemmas transporting field-ness, homomorphisms and isomorphisms across it. Here it is part of what an extension is, which costs one condition and no lemmas: in the carrier-set architecture the operations are parameters that can be constrained, rather than record fields that are total by construction.

type-synonym $'a$ *root-index* $= (\text{nat} \Rightarrow 'a) \times \text{nat}$

type-synonym $'a$ *ipoly* $= 'a$ *root-index* *multiset* $\Rightarrow 'a$

type-synonym $'a$ *iop* $= 'a$ *ipoly* $\Rightarrow 'a$ *ipoly* $\Rightarrow 'a$ *ipoly*

type-synonym $'a$ *iext* $= 'a$ *ipoly set* $\times 'a$ *iop* $\times 'a$ *iop*

abbreviation $\text{ecarrier} :: 'a$ *iext* $\Rightarrow 'a$ *ipoly set* **where** $\text{ecarrier } E \equiv \text{fst } E$

abbreviation $\text{eadd} :: 'a$ *iext* $\Rightarrow 'a$ *iop* **where** $\text{eadd } E \equiv \text{fst } (\text{snd } E)$

abbreviation $\text{emult} :: 'a$ *iext* $\Rightarrow 'a$ *iop* **where** $\text{emult } E \equiv \text{snd } (\text{snd } E)$

71.1 Operations defined only on their carrier

An operation is *restricted* to K when it is *undefined* unless both arguments lie in K . This is the condition discussed above; the lemma after it is the only thing the condition is ever used for, and replaces HOL-Algebra’s *law-restrict* apparatus.

definition *op-restricted* $:: 'a$ *ipoly set* $\Rightarrow 'a$ *iop* $\Rightarrow \text{bool}$

where *op-restricted* $K f \iff (\forall x y. (x \notin K \vee y \notin K) \longrightarrow f x y = \text{undefined})$

lemma *op-restrictedI*:

$(\bigwedge x y. x \notin K \vee y \notin K \implies f x y = \text{undefined}) \implies \text{op-restricted } K f$

by (*simp add: op-restricted-def*)

lemma *op-restrictedD*:

$\llbracket \text{op-restricted } K f; x \notin K \vee y \notin K \rrbracket \implies f x y = \text{undefined}$
by (*simp add: op-restricted-def*)

Two operations restricted to the same carrier and agreeing on it are equal.

lemma *restricted-op-eqI*:

assumes *f*: *op-restricted* *K f* **and** *g*: *op-restricted* *K g*

and *eq*: $\bigwedge x y. \llbracket x \in K; y \in K \rrbracket \implies f x y = g x y$

shows $f = g$

— Two applications of *ext*, not *intro*: the result type *'a ipoly* is itself a function type, so a greedy *intro* strips a third argument.

proof (*rule ext, rule ext*)

fix *x y*

show $f x y = g x y$

proof (*cases* $x \in K \wedge y \in K$)

case *True* **then show** *?thesis* **using** *eq* **by** *simp*

next

case *False*

then have $x \notin K \vee y \notin K$ **by** *blast*

then show *?thesis* **using** *op-restrictedD* [*OF f*] *op-restrictedD* [*OF g*] **by** *simp*

qed

qed

context *Ring*

begin

71.2 Lifted polynomials

A polynomial over *R* read as a polynomial over the ambient ring, coefficientwise. Where HOL-Algebra maps a coefficient *list* through its constant embedding, a polynomial here is already a function on exponents, so the lift is a composition.

definition *lift-poly* :: $(\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a \text{ ipoly})$

where *lift-poly* *P* = $(\lambda k. \text{ipoly-const } (P k))$

lemma *lift-poly-apply* [*simp*]: *lift-poly* *P* *k* = *ipoly-const* (*P* *k*)

by (*simp add: lift-poly-def*)

lemma *lift-poly-coeff-closed*: $P \in \text{poly-carrier} \implies \text{lift-poly } P k \in \text{ipoly-carrier}$

by (*simp add: ipoly-const-closed poly-carrier-coeff-closed*)

71.3 What it is to be an extension

Evaluation in an extension, at that extension's own operations.

abbreviation *eeval* :: $'a \text{ iext} \Rightarrow 'a \text{ ipoly} \Rightarrow (\text{nat} \Rightarrow 'a \text{ ipoly}) \Rightarrow 'a \text{ ipoly}$

where *eeval* *E* $\equiv \text{Ring.eval } (\text{ecarrier } E) (\text{eadd } E) (\text{emult } E) \mathbf{0}_I \mathbf{1}_I$

Evaluation in the *ambient* indexed polynomial ring. A locale prefix is not available here: we are inside *Ring* already, and interpreting the indexed polynomial

ring as a *Ring* at this point would be self-referential. An abbreviation costs nothing and hides the five parameters just as well.

abbreviation *ieval* :: 'a ipoly $\Rightarrow$ (nat $\Rightarrow$ 'a ipoly) $\Rightarrow$ 'a ipoly
where *ieval* $\equiv$ Ring.eval ipoly-carrier ($\oplus_I$) ($\otimes_I$) **0_I** **1_I**

An extension is closed under the roots it mentions: whenever one of its elements involves the indeterminate (P, i), that indeterminate belongs to the extension, and is a root of P there. This is the condition that forces a *maximal* extension to split every polynomial — an unsplit polynomial leaves some index unused, and adjoining a root at it yields a strictly larger extension.

definition *root-closed* :: 'a iext $\Rightarrow$ bool
where *root-closed* $E \longleftrightarrow$
 $(\forall \mathcal{P} \in \text{ecarrier } E. \forall P \in \text{poly-carrier}. \forall i.$
 $\neg \text{index-free } \mathcal{P} (P, i) \longrightarrow$
 $\text{ivar } (P, i) \in \text{ecarrier } E \wedge \text{eval } E (\text{ivar } (P, i)) (\text{lift-poly } P) = \mathbf{0}_I)$

lemma *root-closedI*:
 $(\bigwedge \mathcal{P} P i. [\mathcal{P} \in \text{ecarrier } E; P \in \text{poly-carrier}; \neg \text{index-free } \mathcal{P} (P, i)])$
 $\implies \text{ivar } (P, i) \in \text{ecarrier } E \wedge \text{eval } E (\text{ivar } (P, i)) (\text{lift-poly } P) = \mathbf{0}_I$
 $\implies \text{root-closed } E$
unfolding *root-closed-def* **by** *blast*

lemma *root-closed-mem*:
 $[\text{root-closed } E; \mathcal{P} \in \text{ecarrier } E; P \in \text{poly-carrier}; \neg \text{index-free } \mathcal{P} (P, i)]$
 $\implies \text{ivar } (P, i) \in \text{ecarrier } E$
unfolding *root-closed-def* **by** *blast*

lemma *root-closed-eval*:
 $[\text{root-closed } E; \mathcal{P} \in \text{ecarrier } E; P \in \text{poly-carrier}; \neg \text{index-free } \mathcal{P} (P, i)]$
 $\implies \text{eval } E (\text{ivar } (P, i)) (\text{lift-poly } P) = \mathbf{0}_I$
unfolding *root-closed-def* **by** *blast*

An extension of R inside the indexed polynomial ring: a field of indexed polynomials over R that contains the constants, induces on them the ring operations of R , and is closed under the roots it mentions.

definition *is-extension* :: 'a iext $\Rightarrow$ bool
where *is-extension* $E \longleftrightarrow$
 $\text{Field } (\text{ecarrier } E) (\text{eadd } E) (\text{emult } E) \mathbf{0}_I \mathbf{1}_I$
 $\wedge \text{ecarrier } E \subseteq \text{ipoly-carrier}$
 $\wedge (\forall a \in R. \text{ipoly-const } a \in \text{ecarrier } E)$
 $\wedge (\forall a \in R. \forall b \in R. \text{eadd } E (\text{ipoly-const } a) (\text{ipoly-const } b) = \text{ipoly-const } (a + b))$
 $\wedge (\forall a \in R. \forall b \in R. \text{emult } E (\text{ipoly-const } a) (\text{ipoly-const } b) = \text{ipoly-const } (a \cdot b))$
 $\wedge \text{root-closed } E$
 $\wedge \text{op-restricted } (\text{ecarrier } E) (\text{eadd } E)$
 $\wedge \text{op-restricted } (\text{ecarrier } E) (\text{emult } E)$

definition *extensions* :: 'a iext set
where *extensions* = $\{E. \text{is-extension } E\}$

lemma *extensions-iff* [simp]: $E \in \text{extensions} \longleftrightarrow \text{is-extension } E$
by (simp add: extensions-def)

The component facts, so that clients never unfold *is-extension*.

lemma *is-extension-field*: $\text{is-extension } E \implies \text{Field } (\text{ecarrier } E) (\text{eadd } E) (\text{emult } E)$
 $\mathbf{0}_I \ \mathbf{1}_I$
by (simp add: is-extension-def)

lemma *is-extension-subset*: $\text{is-extension } E \implies \text{ecarrier } E \subseteq \text{ipoly-carrier}$
by (simp add: is-extension-def)

lemma *is-extension-carrier-closed*: $\llbracket \text{is-extension } E; x \in \text{ecarrier } E \rrbracket \implies x \in \text{ipoly-carrier}$
using *is-extension-subset* **by** blast

lemma *is-extension-const-mem*: $\llbracket \text{is-extension } E; a \in R \rrbracket \implies \text{ipoly-const } a \in \text{ecarrier } E$
by (simp add: is-extension-def)

lemma *is-extension-const-add*:
 $\llbracket \text{is-extension } E; a \in R; b \in R \rrbracket$
 $\implies \text{eadd } E (\text{ipoly-const } a) (\text{ipoly-const } b) = \text{ipoly-const } (a + b)$
by (simp add: is-extension-def)

lemma *is-extension-const-mult*:
 $\llbracket \text{is-extension } E; a \in R; b \in R \rrbracket$
 $\implies \text{emult } E (\text{ipoly-const } a) (\text{ipoly-const } b) = \text{ipoly-const } (a \cdot b)$
by (simp add: is-extension-def)

lemma *is-extension-root-closed*: $\text{is-extension } E \implies \text{root-closed } E$
by (simp add: is-extension-def)

lemma *is-extension-restricted*:
 $\text{is-extension } E \implies \text{op-restricted } (\text{ecarrier } E) (\text{eadd } E)$
 $\text{is-extension } E \implies \text{op-restricted } (\text{ecarrier } E) (\text{emult } E)$
by (simp-all add: is-extension-def)

lemma *is-extensionI*:
assumes $\text{Field } (\text{ecarrier } E) (\text{eadd } E) (\text{emult } E) \ \mathbf{0}_I \ \mathbf{1}_I$
and $\text{ecarrier } E \subseteq \text{ipoly-carrier}$
and $\bigwedge a. a \in R \implies \text{ipoly-const } a \in \text{ecarrier } E$
and $\bigwedge a \ b. \llbracket a \in R; b \in R \rrbracket$
 $\implies \text{eadd } E (\text{ipoly-const } a) (\text{ipoly-const } b) = \text{ipoly-const } (a + b)$
and $\bigwedge a \ b. \llbracket a \in R; b \in R \rrbracket$
 $\implies \text{emult } E (\text{ipoly-const } a) (\text{ipoly-const } b) = \text{ipoly-const } (a \cdot b)$
and $\text{root-closed } E$
and $\text{op-restricted } (\text{ecarrier } E) (\text{eadd } E)$
and $\text{op-restricted } (\text{ecarrier } E) (\text{emult } E)$
shows $\text{is-extension } E$
unfolding *is-extension-def* **using** *assms* **by** blast

71.4 The ordering by inclusion

One extension extends another when its carrier is contained in the other's and its operations agree with the other's wherever the smaller one is defined.

definition *ext-le* :: 'a iext $\Rightarrow$ 'a iext $\Rightarrow$ bool
where *ext-le* $E\ F \longleftrightarrow$ $ecarrier\ E \subseteq ecarrier\ F$
 $\wedge (\forall x \in ecarrier\ E. \forall y \in ecarrier\ E.$
 $\quad eadd\ E\ x\ y = eadd\ F\ x\ y \wedge emult\ E\ x\ y = emult\ F\ x\ y)$

lemma *ext-leI*:
 $\llbracket ecarrier\ E \subseteq ecarrier\ F;$
 $\quad \bigwedge x\ y. \llbracket x \in ecarrier\ E; y \in ecarrier\ E \rrbracket \Longrightarrow eadd\ E\ x\ y = eadd\ F\ x\ y;$
 $\quad \bigwedge x\ y. \llbracket x \in ecarrier\ E; y \in ecarrier\ E \rrbracket \Longrightarrow emult\ E\ x\ y = emult\ F\ x\ y \rrbracket$
 $\Longrightarrow ext-le\ E\ F$
unfolding *ext-le-def* **by** *blast*

lemma *ext-le-carrier*: $ext-le\ E\ F \Longrightarrow ecarrier\ E \subseteq ecarrier\ F$
by (*simp add: ext-le-def*)

lemma *ext-le-add*: $\llbracket ext-le\ E\ F; x \in ecarrier\ E; y \in ecarrier\ E \rrbracket \Longrightarrow eadd\ E\ x\ y =$
 $eadd\ F\ x\ y$
unfolding *ext-le-def* **by** *blast*

lemma *ext-le-mult*: $\llbracket ext-le\ E\ F; x \in ecarrier\ E; y \in ecarrier\ E \rrbracket \Longrightarrow emult\ E\ x\ y =$
 $emult\ F\ x\ y$
unfolding *ext-le-def* **by** *blast*

lemma *ext-le-refl*: $ext-le\ E\ E$
by (*simp add: ext-le-def*)

lemma *ext-le-trans*:
assumes EF : $ext-le\ E\ F$ **and** FG : $ext-le\ F\ G$ **shows** $ext-le\ E\ G$
proof (*rule ext-leI*)
show sub : $ecarrier\ E \subseteq ecarrier\ G$
by (*rule subset-trans* [$OF\ ext-le-carrier\ [OF\ EF]\ ext-le-carrier\ [OF\ FG]$])
fix $x\ y$ **assume** xy : $x \in ecarrier\ E\ y \in ecarrier\ E$
then have xyF : $x \in ecarrier\ F\ y \in ecarrier\ F$
using *ext-le-carrier* [$OF\ EF$] **by** *auto*
show $eadd\ E\ x\ y = eadd\ G\ x\ y$
using *ext-le-add* [$OF\ EF\ xy(1)\ xy(2)$] *ext-le-add* [$OF\ FG\ xyF(1)\ xyF(2)$] **by** *simp*
show $emult\ E\ x\ y = emult\ G\ x\ y$
using *ext-le-mult* [$OF\ EF\ xy(1)\ xy(2)$] *ext-le-mult* [$OF\ FG\ xyF(1)\ xyF(2)$] **by** *simp*
qed

Antisymmetry is where restrictedness earns its place: equal carriers and agreement on them settle the operations there, and off the carrier both are *undefined*.

lemma *ext-le-antisym*:
assumes E : *is-extension* E **and** F : *is-extension* F

and EF : $ext\text{-}le\ E\ F$ **and** FE : $ext\text{-}le\ F\ E$
shows $E = F$
proof –
have K : $ecarrier\ E = ecarrier\ F$
by (*rule subset-antisym* [$OF\ ext\text{-}le\text{-}carrier\ [OF\ EF]\ ext\text{-}le\text{-}carrier\ [OF\ FE]$])
have rF : $op\text{-}restricted\ (ecarrier\ E)\ (eadd\ F)\ op\text{-}restricted\ (ecarrier\ E)\ (emult\ F)$
using $is\text{-}extension\text{-}restricted\ [OF\ F]\ K$ **by** *simp-all*
have $eadd\ E = eadd\ F$
by (*rule restricted-op-eqI* [$OF\ is\text{-}extension\text{-}restricted(1)\ [OF\ E]\ rF(1)\ ext\text{-}le\text{-}add\ [OF\ EF]$])
moreover **have** $emult\ E = emult\ F$
by (*rule restricted-op-eqI* [$OF\ is\text{-}extension\text{-}restricted(2)\ [OF\ E]\ rF(2)\ ext\text{-}le\text{-}mult\ [OF\ EF]$])
ultimately **have** $(ecarrier\ E, eadd\ E, emult\ E) = (ecarrier\ F, eadd\ F, emult\ F)$
using K **by** *simp*
then show *?thesis* **by** *simp*
qed

theorem *ext-partial-order: partial-order-on extensions (relation-of ext-le extensions)*
proof (*rule partial-order-on-relation-ofI*)
show $\bigwedge E. E \in extensions \implies ext\text{-}le\ E\ E$ **by** (*rule ext-le-refl*)
show $\bigwedge E\ F\ G. \llbracket E \in extensions; F \in extensions; G \in extensions; ext\text{-}le\ E\ F; ext\text{-}le\ F\ G \rrbracket$
 $\implies ext\text{-}le\ E\ G$
by (*rule ext-le-trans*)
show $\bigwedge E\ F. \llbracket E \in extensions; F \in extensions; ext\text{-}le\ E\ F; ext\text{-}le\ F\ E \rrbracket \implies E = F$
by (*metis extensions-iff ext-le-antisym*)
qed

71.5 Powers and evaluation transfer along the ordering

A power or an evaluation of an element of the smaller extension is the same whether computed there or in the larger one. Nothing beyond ring-ness of the two, carrier inclusion, and agreement of the operations on the smaller carrier is used — that is, exactly *ext-le* — so the same two lemmas serve the union of a chain and the adjunction of a root.

There is no such fact one level down: *commutative-monoid.fincomp* depends on the *carrier* as well as the operation, through its *M-ify*, so *commutative-monoid.fincomp-cong'*, which stays inside one monoid, does not apply. *fincomp-transfer* is what bridges the two carriers.

lemma *ext-le-rpow*:
assumes RE : *Ring* $(ecarrier\ E)\ (eadd\ E)\ (emult\ E)\ \mathbf{0}_I\ \mathbf{1}_I$
and RF : *Ring* $(ecarrier\ F)\ (eadd\ F)\ (emult\ F)\ \mathbf{0}_I\ \mathbf{1}_I$
and le : $ext\text{-}le\ E\ F$ **and** a : $a \in ecarrier\ E$
shows $Ring.rpow\ (emult\ F)\ \mathbf{1}_I\ a\ n = Ring.rpow\ (emult\ E)\ \mathbf{1}_I\ a\ n$
proof (*induct n*)
case 0
show *?case* **by** (*simp add: Ring.rpow-0* [$OF\ RE$] *Ring.rpow-0* [$OF\ RF$])

```

next
  case (Suc n)
  have cl: Ring.rpow (emult E) 1_I a n ∈ ecarrier E by (rule Ring.rpow-closed [OF RE a])
  have Ring.rpow (emult F) 1_I a (Suc n) = emult F a (Ring.rpow (emult F) 1_I a n)
    by (rule Ring.rpow-Suc [OF RF])
  also have ... = emult F a (Ring.rpow (emult E) 1_I a n) using Suc by simp
  also have ... = emult E a (Ring.rpow (emult E) 1_I a n)
    by (rule ext-le-mult [OF le a cl, symmetric])
  also have ... = Ring.rpow (emult E) 1_I a (Suc n) by (rule Ring.rpow-Suc [OF RE, symmetric])
  finally show ?case .
qed

```

lemma ext-le-eval:

```

assumes RE: Ring (ecarrier E) (eadd E) (emult E) 0_I 1_I
  and RF: Ring (ecarrier F) (eadd F) (emult F) 0_I 1_I
  and le: ext-le E F and a: a ∈ ecarrier E and p:  $\bigwedge k. p k \in ecarrier E$ 
shows eeval F a p = eeval E a p
proof -
  interpret E: Ring ecarrier E eadd E emult E 0_I 1_I by (rule RE)
  have cl: Ring.rpow (emult E) 1_I a i ∈ ecarrier E for i by (rule Ring.rpow-closed [OF RE a])
  have summand: emult F (p i) (Ring.rpow (emult F) 1_I a i)
    = emult E (p i) (Ring.rpow (emult E) 1_I a i) for i
    using ext-le-rpow [OF RE RF le a] ext-le-mult [OF le p cl] by simp
  have inE: ( $\lambda i. emult E (p i) (Ring.rpow (emult E) 1_I a i)$ ) ∈ A → ecarrier E for
  A
    using p cl by auto
  have eeval F a p
    = commutative-monoid.fincomp (ecarrier F) (eadd F) 0_I
      ( $\lambda i. emult F (p i) (Ring.rpow (emult F) 1_I a i)$ ) {..Ring.degree 0_I p}
    by (rule Ring.eval-def [OF RF])
  also have ... = commutative-monoid.fincomp (ecarrier F) (eadd F) 0_I
      ( $\lambda i. emult E (p i) (Ring.rpow (emult E) 1_I a i)$ ) {..Ring.degree 0_I p}
    using summand by simp
  also have ... = commutative-monoid.fincomp (ecarrier E) (eadd E) 0_I
      ( $\lambda i. emult E (p i) (Ring.rpow (emult E) 1_I a i)$ ) {..Ring.degree 0_I p}
  proof (rule fincomp-transfer [symmetric])
    show commutative-monoid (ecarrier E) (eadd E) 0_I
      using RE by (simp add: Ring-def Abelian-Group-def)
    show commutative-monoid (ecarrier F) (eadd F) 0_I
      using RF by (simp add: Ring-def Abelian-Group-def)
    show ecarrier E ⊆ ecarrier F by (rule ext-le-carrier [OF le])
    show  $\bigwedge x y. \llbracket x \in ecarrier E; y \in ecarrier E \rrbracket \implies eadd E x y = eadd F x y$ 
      by (rule ext-le-add [OF le])
    show ( $\lambda i. emult E (p i) (Ring.rpow (emult E) 1_I a i)$ ) ∈ {..Ring.degree 0_I p} →
      ecarrier E
      by (rule inE)
  qed
qed

```

also have $\dots = \text{eval } E \text{ a } p$ by (rule *Ring.eval-def* [*OF RE, symmetric*])
 finally show ?thesis .
 qed

71.6 The base extension

The constants form an extension: a copy of R inside the indexed polynomial ring, carrying the ambient operations restricted to it. Besides supplying the upper bound Zorn needs for the *empty* chain, this is the check that *is-extension* is satisfiable at all — a condition as encoded as *root-closed* could easily be vacuous, and then everything proved from it would be too. Here *root-closed* holds vacuously *by design*: a constant involves no indeterminate, so its hypothesis is never met.

definition *base-ext* :: 'a iext

where *base-ext* =

(*ipoly-const* ' R ,
 $(\lambda x y. \text{if } x \in \text{ipoly-const ' } R \wedge y \in \text{ipoly-const ' } R \text{ then } x \oplus_I y \text{ else undefined}),$
 $(\lambda x y. \text{if } x \in \text{ipoly-const ' } R \wedge y \in \text{ipoly-const ' } R \text{ then } x \otimes_I y \text{ else undefined}))$

lemma *base-ext-carrier* [*simp*]: *ecarrier base-ext* = *ipoly-const ' R*
by (*simp add: base-ext-def*)

lemma *base-ext-add*:

assumes $a \in R$ **and** $b \in R$

shows *eadd base-ext* (*ipoly-const a*) (*ipoly-const b*) = *ipoly-const* ($a + b$)

using *assms* **by** (*simp add: base-ext-def ipoly-const-add*)

lemma *base-ext-mult*:

assumes $a \in R$ **and** $b \in R$

shows *emult base-ext* (*ipoly-const a*) (*ipoly-const b*) = *ipoly-const* ($a \cdot b$)

using *assms* **by** (*simp add: base-ext-def ipoly-const-mult*)

lemma *base-ext-restricted*:

op-restricted (*ecarrier base-ext*) (*eadd base-ext*)

op-restricted (*ecarrier base-ext*) (*emult base-ext*)

by (*auto simp: base-ext-def intro!: op-restrictedI*)

71.7 Unions of chains

A chain of extensions: a subset of *extensions* any two of whose members are comparable. Its union is again an extension, and is an upper bound — which is what Zorn's lemma asks for.

definition *ext-chain* :: 'a iext set $\Rightarrow$ bool

where *ext-chain* $C \longleftrightarrow C \subseteq \text{extensions} \wedge (\forall E \in C. \forall F \in C. \text{ext-le } E F \vee \text{ext-le } F E)$

lemma *ext-chainD-extension*: $\llbracket \text{ext-chain } C; E \in C \rrbracket \Longrightarrow \text{is-extension } E$
unfolding *ext-chain-def* **by** *auto*

lemma *ext-chainD-comparable*: $\llbracket \text{ext-chain } C; E \in C; F \in C \rrbracket \implies \text{ext-le } E F \vee \text{ext-le } F E$
unfolding *ext-chain-def* **by** *blast*

The union carries, on each pair of elements, the operations of any member containing both; the chain condition makes the choice immaterial.

definition *ext-Union* :: 'a iext set $\Rightarrow$ 'a iext
where *ext-Union* $C =$
 $(\bigcup (\text{ecarrier } 'C),$
 $(\lambda x y. \text{if } x \in \bigcup (\text{ecarrier } 'C) \wedge y \in \bigcup (\text{ecarrier } 'C)$
 $\text{then eadd (SOME } E. E \in C \wedge x \in \text{ecarrier } E \wedge y \in \text{ecarrier } E) \ x \ y \text{ else}$
 $\text{undefined}),$
 $(\lambda x y. \text{if } x \in \bigcup (\text{ecarrier } 'C) \wedge y \in \bigcup (\text{ecarrier } 'C)$
 $\text{then emult (SOME } E. E \in C \wedge x \in \text{ecarrier } E \wedge y \in \text{ecarrier } E) \ x \ y \text{ else}$
 $\text{undefined}))$

lemma *ext-Union-carrier* [simp]: $\text{ecarrier } (\text{ext-Union } C) = \bigcup (\text{ecarrier } 'C)$
by (simp add: *ext-Union-def*)

lemma *ext-Union-restricted*:
 $\text{op-restricted } (\text{ecarrier } (\text{ext-Union } C)) (\text{eadd } (\text{ext-Union } C))$
 $\text{op-restricted } (\text{ecarrier } (\text{ext-Union } C)) (\text{emult } (\text{ext-Union } C))$
by (auto simp: *ext-Union-def* intro!: *op-restrictedI*)

context
fixes $C :: 'a \text{ iext set}$
assumes *chain*: *ext-chain* C
begin

lemma *chain-field*: $E \in C \implies \text{Field } (\text{ecarrier } E) (\text{eadd } E) (\text{emult } E) \mathbf{0}_I \mathbf{1}_I$
using *chain* **by** (simp add: *ext-chainD-extension is-extension-field*)

Two elements of the union lie in a common member: the larger of two members containing them.

lemma *chain-common2*:
assumes $x \in \text{ecarrier } (\text{ext-Union } C)$ **and** $y \in \text{ecarrier } (\text{ext-Union } C)$
shows $\exists E \in C. x \in \text{ecarrier } E \wedge y \in \text{ecarrier } E$
proof –
from *assms* **obtain** $E F$ **where** $E: E \in C \ x \in \text{ecarrier } E$ **and** $F: F \in C \ y \in \text{ecarrier } F$
by *auto*
from *ext-chainD-comparable* [OF *chain* $E(1)$ $F(1)$] **show** ?thesis
proof
assume *ext-le* $E F$
then have $x \in \text{ecarrier } F$ **using** $E(2)$ *ext-le-carrier* **by** *blast*
then show ?thesis **using** F **by** *blast*
next


```

    assume ext-le F E
    then have  $y \in \text{ecarrier } E$  using  $F(2)$  ext-le-carrier by blast
    then show ?thesis using E by blast
qed
qed

lemma chain-common3:
  assumes  $x \in \text{ecarrier } (\text{ext-Union } C)$  and  $y \in \text{ecarrier } (\text{ext-Union } C)$ 
    and  $z \in \text{ecarrier } (\text{ext-Union } C)$ 
  shows  $\exists E \in C. x \in \text{ecarrier } E \wedge y \in \text{ecarrier } E \wedge z \in \text{ecarrier } E$ 
proof -
  from chain-common2 [OF assms(1,2)] obtain E
    where  $E: E \in C \wedge x \in \text{ecarrier } E \wedge y \in \text{ecarrier } E$  by blast
  from assms(3) obtain F where  $F: F \in C \wedge z \in \text{ecarrier } F$  by auto
  from ext-chainD-comparable [OF chain E(1) F(1)] show ?thesis
proof
  assume ext-le E F
  then have  $x \in \text{ecarrier } F \wedge y \in \text{ecarrier } F$  using E ext-le-carrier by blast+
  then show ?thesis using F by blast
next
  assume ext-le F E
  then have  $z \in \text{ecarrier } E$  using  $F(2)$  ext-le-carrier by blast
  then show ?thesis using E by blast
qed
qed

```

The union's operations agree with those of any member containing both arguments.

```

lemma ext-Union-ops:
  assumes  $E: E \in C$  and  $x: x \in \text{ecarrier } E$  and  $y: y \in \text{ecarrier } E$ 
  shows eadd (ext-Union C) x y = eadd E x y
    and emult (ext-Union C) x y = emult E x y
proof -
  have mem:  $x \in \bigcup (\text{ecarrier } 'C) \wedge y \in \bigcup (\text{ecarrier } 'C)$  using E x y by auto
  have ex:  $\exists F. F \in C \wedge x \in \text{ecarrier } F \wedge y \in \text{ecarrier } F$  using E x y by blast
  let ?F = SOME F.  $F \in C \wedge x \in \text{ecarrier } F \wedge y \in \text{ecarrier } F$ 
  have  $F: ?F \in C$  and  $xF: x \in \text{ecarrier } ?F$  and  $yF: y \in \text{ecarrier } ?F$ 
    using someI-ex [OF ex] by auto
  have or: ext-le E ?F  $\vee$  ext-le ?F E by (rule ext-chainD-comparable [OF chain E
F])
  have a: eadd E x y = eadd ?F x y
  proof (cases ext-le E ?F)
    case True then show ?thesis using x y by (simp add: ext-le-add)
  next
    case False
    then have ext-le ?F E using or by blast
    then show ?thesis using xF yF by (simp add: ext-le-add)
  qed
  have m: emult E x y = emult ?F x y

```

```

proof (cases ext-le E ?F)
  case True then show ?thesis using x y by (simp add: ext-le-mult)
next
  case False
  then have ext-le ?F E using or by blast
  then show ?thesis using xF yF by (simp add: ext-le-mult)
qed
show eadd (ext-Union C) x y = eadd E x y
  using mem a by (simp add: ext-Union-def)
show emult (ext-Union C) x y = emult E x y
  using mem m by (simp add: ext-Union-def)
qed

```

lemma ext-Union-upper-bound: $E \in C \implies \text{ext-le } E \text{ (ext-Union } C)$
by (rule ext-leI) (auto simp: ext-Union-ops)

Every member's carrier contains $\mathbf{0}_I$ and $\mathbf{1}_I$, being an extension of R .

lemma chain-zero-one-mem:

assumes E: $E \in C$ **shows** $\mathbf{0}_I \in \text{ecarrier } E$ **and** $\mathbf{1}_I \in \text{ecarrier } E$

proof –

have ext: is-extension E **using** chain E **by** (rule ext-chainD-extension)

have ipoly-const $\mathbf{0} \in \text{ecarrier } E$

by (rule is-extension-const-mem [OF ext additive.unit-closed])

moreover have ipoly-const $\mathbf{1} \in \text{ecarrier } E$

by (rule is-extension-const-mem [OF ext multiplicative.unit-closed])

ultimately show $\mathbf{0}_I \in \text{ecarrier } E$ **and** $\mathbf{1}_I \in \text{ecarrier } E$ **by** simp-all

qed

The field laws of a single member, extracted through one interpretation so that the union's proof below can cite them without re-interpreting for every axiom.

lemma chain-member:

assumes E: $E \in C$

shows chain-add-closed: $\bigwedge x y. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E \rrbracket \implies \text{eadd } E x y \in \text{ecarrier } E$

and chain-mult-closed: $\bigwedge x y. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E \rrbracket \implies \text{emult } E x y \in \text{ecarrier } E$

and chain-add-assoc: $\bigwedge x y z. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E; z \in \text{ecarrier } E \rrbracket \implies \text{eadd } E (\text{eadd } E x y) z = \text{eadd } E x (\text{eadd } E y z)$

and chain-mult-assoc: $\bigwedge x y z. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E; z \in \text{ecarrier } E \rrbracket \implies \text{emult } E (\text{emult } E x y) z = \text{emult } E x (\text{emult } E y z)$

and chain-add-zero: $\bigwedge x. x \in \text{ecarrier } E \implies \text{eadd } E \mathbf{0}_I x = x$

and chain-add-zero': $\bigwedge x. x \in \text{ecarrier } E \implies \text{eadd } E x \mathbf{0}_I = x$

and chain-mult-one: $\bigwedge x. x \in \text{ecarrier } E \implies \text{emult } E \mathbf{1}_I x = x$

and chain-mult-one': $\bigwedge x. x \in \text{ecarrier } E \implies \text{emult } E x \mathbf{1}_I = x$

and chain-add-comm: $\bigwedge x y. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E \rrbracket \implies \text{eadd } E x y = \text{eadd } E y x$

and chain-mult-comm: $\bigwedge x y. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E \rrbracket \implies \text{emult } E x y = \text{emult } E y x$

and chain-distrib: $\bigwedge x y z. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E; z \in \text{ecarrier } E \rrbracket$

$\implies \text{emult } E \ x \ (\text{eadd } E \ y \ z) = \text{eadd } E \ (\text{emult } E \ x \ y) \ (\text{emult } E \ x \ z)$
and *chain-distrib'*: $\bigwedge x \ y \ z. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E; z \in \text{ecarrier } E \rrbracket$
 $\implies \text{emult } E \ (\text{eadd } E \ y \ z) \ x = \text{eadd } E \ (\text{emult } E \ y \ x) \ (\text{emult } E \ z \ x)$
and *chain-add-inv*: $\bigwedge x. x \in \text{ecarrier } E$
 $\implies \exists y \in \text{ecarrier } E. \text{eadd } E \ x \ y = \mathbf{0}_I \wedge \text{eadd } E \ y \ x = \mathbf{0}_I$
and *chain-mult-inv*: $\bigwedge x. \llbracket x \in \text{ecarrier } E; x \neq \mathbf{0}_I \rrbracket$
 $\implies \exists y \in \text{ecarrier } E. \text{emult } E \ x \ y = \mathbf{1}_I \wedge \text{emult } E \ y \ x = \mathbf{1}_I$

proof –

interpret *E*: *Field* *ecarrier* *E* *eadd* *E* *emult* *E* $\mathbf{0}_I$ $\mathbf{1}_I$
by (*rule chain-field* [*OF* *E*])

show $\bigwedge x \ y. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E \rrbracket \implies \text{eadd } E \ x \ y \in \text{ecarrier } E$ **by** *simp*
show $\bigwedge x \ y. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E \rrbracket \implies \text{emult } E \ x \ y \in \text{ecarrier } E$ **by** *simp*

show $\bigwedge x \ y \ z. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E; z \in \text{ecarrier } E \rrbracket$
 $\implies \text{eadd } E \ (\text{eadd } E \ x \ y) \ z = \text{eadd } E \ x \ (\text{eadd } E \ y \ z)$
by (*simp add*: *E.additive.associative*)

show $\bigwedge x \ y \ z. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E; z \in \text{ecarrier } E \rrbracket$
 $\implies \text{emult } E \ (\text{emult } E \ x \ y) \ z = \text{emult } E \ x \ (\text{emult } E \ y \ z)$
by (*simp add*: *E.multiplicative.associative*)

show $\bigwedge x. x \in \text{ecarrier } E \implies \text{eadd } E \ \mathbf{0}_I \ x = x$ **by** *simp*
show $\bigwedge x. x \in \text{ecarrier } E \implies \text{eadd } E \ x \ \mathbf{0}_I = x$ **by** *simp*
show $\bigwedge x. x \in \text{ecarrier } E \implies \text{emult } E \ \mathbf{1}_I \ x = x$ **by** *simp*
show $\bigwedge x. x \in \text{ecarrier } E \implies \text{emult } E \ x \ \mathbf{1}_I = x$ **by** *simp*

show $\bigwedge x \ y. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E \rrbracket \implies \text{eadd } E \ x \ y = \text{eadd } E \ y \ x$
by (*simp add*: *E.additive.commutative*)

show $\bigwedge x \ y. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E \rrbracket \implies \text{emult } E \ x \ y = \text{emult } E \ y \ x$
by (*simp add*: *E.multiplicative.commutative*)

show $\bigwedge x \ y \ z. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E; z \in \text{ecarrier } E \rrbracket$
 $\implies \text{emult } E \ x \ (\text{eadd } E \ y \ z) = \text{eadd } E \ (\text{emult } E \ x \ y) \ (\text{emult } E \ x \ z)$
by (*simp add*: *E.distributive(1)*)

show $\bigwedge x \ y \ z. \llbracket x \in \text{ecarrier } E; y \in \text{ecarrier } E; z \in \text{ecarrier } E \rrbracket$
 $\implies \text{emult } E \ (\text{eadd } E \ y \ z) \ x = \text{eadd } E \ (\text{emult } E \ y \ x) \ (\text{emult } E \ z \ x)$
by (*simp add*: *E.distributive(2)*)

show $\bigwedge x. x \in \text{ecarrier } E \implies \exists y \in \text{ecarrier } E. \text{eadd } E \ x \ y = \mathbf{0}_I \wedge \text{eadd } E \ y \ x = \mathbf{0}_I$

proof –

fix *x* **assume** *x*: *x* $\in \text{ecarrier } E$
have *E.additive.inverse* *x* $\in \text{ecarrier } E$ **using** *x* **by** *simp*
moreover have *eadd* *E* *x* (*E.additive.inverse* *x*) = $\mathbf{0}_I$
using *x* **by** (*simp add*: *E.additive.invertible-right-inverse*)
moreover have *eadd* *E* (*E.additive.inverse* *x*) *x* = $\mathbf{0}_I$
using *x* **by** (*simp add*: *E.additive.invertible-left-inverse*)
ultimately show $\exists y \in \text{ecarrier } E. \text{eadd } E \ x \ y = \mathbf{0}_I \wedge \text{eadd } E \ y \ x = \mathbf{0}_I$ **by** *blast*

qed

show $\bigwedge x. \llbracket x \in \text{ecarrier } E; x \neq \mathbf{0}_I \rrbracket$
 $\implies \exists y \in \text{ecarrier } E. \text{emult } E \ x \ y = \mathbf{1}_I \wedge \text{emult } E \ y \ x = \mathbf{1}_I$

proof –

fix *x* **assume** *x*: *x* $\in \text{ecarrier } E$ **and** *xnz*: *x* $\neq \mathbf{0}_I$
have *inv*: *E.multiplicative.invertible* *x* **by** (*rule* *E.field-inverse* [*OF* *x xnz*])

```

have  $E.\text{multiplicative.inverse } x \in \text{ecarrier } E$ 
  by (rule  $E.\text{multiplicative.invertible-inverse-closed}$  [OF inv x])
moreover have  $\text{emult } E \ x \ (E.\text{multiplicative.inverse } x) = \mathbf{1}_I$ 
  by (rule  $E.\text{multiplicative.invertible-right-inverse}$  [OF inv x])
moreover have  $\text{emult } E \ (E.\text{multiplicative.inverse } x) \ x = \mathbf{1}_I$ 
  by (rule  $E.\text{multiplicative.invertible-left-inverse}$  [OF inv x])
ultimately show  $\exists y \in \text{ecarrier } E. \text{emult } E \ x \ y = \mathbf{1}_I \wedge \text{emult } E \ y \ x = \mathbf{1}_I$  by blast
qed
qed

```

The union of a nonempty chain is a field. Each axiom involves at most three elements, which a common member accommodates; there the axiom holds, and the union's operations agree with that member's.

lemma *ext-Union-field*:

```

assumes  $ne: C \neq \{\}$  and  $nz: \mathbf{1} \neq \mathbf{0}$ 
shows  $\text{Field } (\text{ecarrier } (\text{ext-Union } C)) \ (\text{eadd } (\text{ext-Union } C)) \ (\text{emult } (\text{ext-Union } C))$ 
 $\mathbf{0}_I \ \mathbf{1}_I$ 
proof -
  obtain  $E0$  where  $E0: E0 \in C$  using  $ne$  by blast
  have  $zmem: \mathbf{0}_I \in \text{ecarrier } (\text{ext-Union } C)$ 
    using  $E0 \text{chain-zero-one-mem}(1)$  [OF  $E0$ ] unfolding  $\text{ext-Union-carrier}$  by blast
  have  $omem: \mathbf{1}_I \in \text{ecarrier } (\text{ext-Union } C)$ 
    using  $E0 \text{chain-zero-one-mem}(2)$  [OF  $E0$ ] unfolding  $\text{ext-Union-carrier}$  by blast
  have  $grp: \text{Group } (\text{ecarrier } (\text{ext-Union } C)) \ (\text{eadd } (\text{ext-Union } C)) \ \mathbf{0}_I$ 
  proof (rule  $\text{GroupI}$ )
    fix  $x \ y$  assume  $x \in \text{ecarrier } (\text{ext-Union } C)$  and  $y \in \text{ecarrier } (\text{ext-Union } C)$ 
    then obtain  $E$  where  $E: E \in C \ x \in \text{ecarrier } E \ y \in \text{ecarrier } E$ 
      using  $\text{chain-common2}$  by blast
    then show  $\text{eadd } (\text{ext-Union } C) \ x \ y \in \text{ecarrier } (\text{ext-Union } C)$ 
      using  $\text{ext-Union-ops}(1)$  [OF  $E$ ]  $\text{chain-add-closed}$  [OF  $E(1)$ ] by auto
  next
    show  $\mathbf{0}_I \in \text{ecarrier } (\text{ext-Union } C)$  by (rule  $zmem$ )
  next
    fix  $x \ y \ z$ 
    assume  $x \in \text{ecarrier } (\text{ext-Union } C) \ y \in \text{ecarrier } (\text{ext-Union } C) \ z \in \text{ecarrier } (\text{ext-Union } C)$ 
    then obtain  $E$  where  $E: E \in C \ x \in \text{ecarrier } E \ y \in \text{ecarrier } E \ z \in \text{ecarrier } E$ 
      using  $\text{chain-common3}$  by blast
    have  $\text{eadd } E \ x \ y \in \text{ecarrier } E$  and  $\text{eadd } E \ y \ z \in \text{ecarrier } E$ 
      using  $\text{chain-add-closed}$  [OF  $E(1)$ ]  $E$  by auto
    then show  $\text{eadd } (\text{ext-Union } C) \ (\text{eadd } (\text{ext-Union } C) \ x \ y) \ z$ 
      =  $\text{eadd } (\text{ext-Union } C) \ x \ (\text{eadd } (\text{ext-Union } C) \ y \ z)$ 
      using  $\text{ext-Union-ops}(1)$  [OF  $E(1)$ ]  $E \text{chain-add-assoc}$  [OF  $E(1)$ ] by simp
  next
    fix  $x$  assume  $x \in \text{ecarrier } (\text{ext-Union } C)$ 
    then obtain  $E$  where  $E: E \in C \ x \in \text{ecarrier } E$  by auto
    then show  $\text{eadd } (\text{ext-Union } C) \ \mathbf{0}_I \ x = x$ 
      using  $\text{ext-Union-ops}(1)$  [OF  $E(1) \text{chain-zero-one-mem}(1)$  [OF  $E(1)$ ]  $E(2)$ ]
       $\text{chain-add-zero}$  [OF  $E(1) \ E(2)$ ] by simp

```

```

next
  fix x assume x ∈ ecarrier (ext-Union C)
  then obtain E where E: E ∈ C x ∈ ecarrier E by auto
  then show eadd (ext-Union C) x 0I = x
    using ext-Union-ops(1) [OF E(1) E(2) chain-zero-one-mem(1) [OF E(1)]]
      chain-add-zero' [OF E(1) E(2)] by simp
next
  fix x assume x ∈ ecarrier (ext-Union C)
  then obtain E where E: E ∈ C x ∈ ecarrier E by auto
  from chain-add-inv [OF E(1) E(2)] obtain y
    where y: y ∈ ecarrier E eadd E x y = 0I eadd E y x = 0I by blast
  then show ∃ v ∈ ecarrier (ext-Union C).
    eadd (ext-Union C) x v = 0I ∧ eadd (ext-Union C) v x = 0I
    using ext-Union-ops(1) [OF E(1) E(2) y(1)] ext-Union-ops(1) [OF E(1) y(1)
E(2)] E by auto
qed
interpret Uadd: Group ecarrier (ext-Union C) eadd (ext-Union C) 0I by (rule grp)
interpret Uadd: Abelian-Group ecarrier (ext-Union C) eadd (ext-Union C) 0I
proof unfold-locales
  fix x y assume x ∈ ecarrier (ext-Union C) and y ∈ ecarrier (ext-Union C)
  then obtain E where E: E ∈ C x ∈ ecarrier E y ∈ ecarrier E
    using chain-common2 by blast
  then show eadd (ext-Union C) x y = eadd (ext-Union C) y x
    using ext-Union-ops(1) [OF E(1) E(2) E(3)] ext-Union-ops(1) [OF E(1) E(3)
E(2)]
      chain-add-comm [OF E(1) E(2) E(3)] by simp
qed
interpret Umult: commutative-monoid ecarrier (ext-Union C) emult (ext-Union C)
1I
proof unfold-locales
  fix x y assume x ∈ ecarrier (ext-Union C) and y ∈ ecarrier (ext-Union C)
  then obtain E where E: E ∈ C x ∈ ecarrier E y ∈ ecarrier E
    using chain-common2 by blast
  then show emult (ext-Union C) x y ∈ ecarrier (ext-Union C)
    using ext-Union-ops(2) [OF E] chain-mult-closed [OF E(1)] by auto
next
  show 1I ∈ ecarrier (ext-Union C) by (rule omem)
next
  fix x y z
    assume x ∈ ecarrier (ext-Union C) y ∈ ecarrier (ext-Union C) z ∈ ecarrier
(ext-Union C)
  then obtain E where E: E ∈ C x ∈ ecarrier E y ∈ ecarrier E z ∈ ecarrier E
    using chain-common3 by blast
  have emult E x y ∈ ecarrier E and emult E y z ∈ ecarrier E
    using chain-mult-closed [OF E(1)] E by auto
  then show emult (ext-Union C) (emult (ext-Union C) x y) z
    = emult (ext-Union C) x (emult (ext-Union C) y z)
    using ext-Union-ops(2) [OF E(1)] E chain-mult-assoc [OF E(1)] by simp
next
  fix x assume x ∈ ecarrier (ext-Union C)

```

```

then obtain  $E$  where  $E: E \in C \ x \in \text{ecarrier } E$  by auto
then show emult (ext-Union  $C$ )  $\mathbf{1}_I \ x = x$ 
  using ext-Union-ops(2) [OF  $E(1)$  chain-zero-one-mem(2) [OF  $E(1)$ ]  $E(2)$ ]
    chain-mult-one [OF  $E(1)$   $E(2)$ ] by simp
next
  fix  $x$  assume  $x \in \text{ecarrier } (\text{ext-Union } C)$ 
  then obtain  $E$  where  $E: E \in C \ x \in \text{ecarrier } E$  by auto
  then show emult (ext-Union  $C$ )  $x \ \mathbf{1}_I = x$ 
    using ext-Union-ops(2) [OF  $E(1)$   $E(2)$  chain-zero-one-mem(2) [OF  $E(1)$ ]]
      chain-mult-one' [OF  $E(1)$   $E(2)$ ] by simp
next
  fix  $x \ y$  assume  $x \in \text{ecarrier } (\text{ext-Union } C)$  and  $y \in \text{ecarrier } (\text{ext-Union } C)$ 
  then obtain  $E$  where  $E: E \in C \ x \in \text{ecarrier } E \ y \in \text{ecarrier } E$ 
    using chain-common2 by blast
  then show emult (ext-Union  $C$ )  $x \ y = \text{emult } (\text{ext-Union } C) \ y \ x$ 
    using ext-Union-ops(2) [OF  $E(1)$   $E(2)$   $E(3)$ ] ext-Union-ops(2) [OF  $E(1)$   $E(3)$ 
 $E(2)$ ]
      chain-mult-comm [OF  $E(1)$   $E(2)$   $E(3)$ ] by simp
qed
show ?thesis
proof unfold-locales
  fix  $x \ y \ z$ 
    assume  $x \in \text{ecarrier } (\text{ext-Union } C) \ y \in \text{ecarrier } (\text{ext-Union } C) \ z \in \text{ecarrier } (\text{ext-Union } C)$ 
    then obtain  $E$  where  $E: E \in C \ x \in \text{ecarrier } E \ y \in \text{ecarrier } E \ z \in \text{ecarrier } E$ 
      using chain-common3 by blast
    have  $cl: \text{eadd } E \ y \ z \in \text{ecarrier } E \ \text{emult } E \ x \ y \in \text{ecarrier } E \ \text{emult } E \ x \ z \in \text{ecarrier } E$ 
      emult  $E \ y \ x \in \text{ecarrier } E \ \text{emult } E \ z \ x \in \text{ecarrier } E$ 
      using chain-add-closed [OF  $E(1)$ ] chain-mult-closed [OF  $E(1)$ ]  $E$  by auto
    show emult (ext-Union  $C$ )  $x \ (\text{eadd } (\text{ext-Union } C) \ y \ z)$ 
       $= \text{eadd } (\text{ext-Union } C) \ (\text{emult } (\text{ext-Union } C) \ x \ y) \ (\text{emult } (\text{ext-Union } C) \ x \ z)$ 
      using  $cl \ E$  ext-Union-ops(1) [OF  $E(1)$ ] ext-Union-ops(2) [OF  $E(1)$ ]
        chain-distrib [OF  $E(1)$ ] by simp
    show emult (ext-Union  $C$ )  $(\text{eadd } (\text{ext-Union } C) \ y \ z) \ x$ 
       $= \text{eadd } (\text{ext-Union } C) \ (\text{emult } (\text{ext-Union } C) \ y \ x) \ (\text{emult } (\text{ext-Union } C) \ z \ x)$ 
      using  $cl \ E$  ext-Union-ops(1) [OF  $E(1)$ ] ext-Union-ops(2) [OF  $E(1)$ ]
        chain-distrib' [OF  $E(1)$ ] by simp
next
  — Taken through the definitions rather than from a member's nontrivial: the two
  statements are interchangeable, and Ring does not itself assume  $\mathbf{1} \neq \mathbf{0}$ .
  show  $\mathbf{1}_I \neq \mathbf{0}_I$ 
    using nz by (auto simp: ipoly-one-def ipoly-zero-def fun-eq-iff)
next
  fix  $x$  assume  $x: x \in \text{ecarrier } (\text{ext-Union } C)$  and  $xnz: x \neq \mathbf{0}_I$ 
  from  $x$  obtain  $E$  where  $E: E \in C \ x \in \text{ecarrier } E$  by auto
  from chain-mult-inv [OF  $E(1)$   $E(2)$   $xnz$ ] obtain  $y$ 
    where  $y: y \in \text{ecarrier } E \ \text{emult } E \ x \ y = \mathbf{1}_I \ \text{emult } E \ y \ x = \mathbf{1}_I$  by blast
  have emult (ext-Union  $C$ )  $x \ y = \mathbf{1}_I$  and emult (ext-Union  $C$ )  $y \ x = \mathbf{1}_I$ 
    using ext-Union-ops(2) [OF  $E(1)$   $E(2)$   $y(1)$ ] ext-Union-ops(2) [OF  $E(1)$   $y(1)$ ]

```

```

E(2)] y by simp-all
  moreover have y ∈ ecarrier (ext-Union C) using E(1) y(1) by auto
  ultimately show Umult.invertible x
    by (blast intro: Umult.invertibleI)
qed
qed

```

71.8 Evaluation is the same in a member and in the union

The remaining extension condition, *root-closed*, asserts that an indeterminate is a *root*, which is an equation about evaluation. Evaluation does not transfer between a member and the union for free: *commutative-monoid.fcomp* depends on the carrier as well as the operation (through *commutative-monoid.M-ify*), so *fcomp-transfer* is what makes the two agree, given that every value involved stays inside the member.

lemma *chain-Ring*:

```

assumes E: E ∈ C shows Ring (ecarrier E) (eadd E) (emult E) 0I 1I
using chain-field [OF E] by (simp add: Field-def commutative-ring-def)

```

lemma *ext-Union-Ring*:

```

assumes ne: C ≠ {} and nz: 1 ≠ 0
shows Ring (ecarrier (ext-Union C)) (eadd (ext-Union C)) (emult (ext-Union C))
0I 1I
using ext-Union-field [OF ne nz] by (simp add: Field-def commutative-ring-def)

```

The chain case is now just the ordering case at *ext-Union-upper-bound*.

lemma *rpow-transfer*:

```

assumes ne: C ≠ {} and nz: 1 ≠ 0 and E: E ∈ C and a: a ∈ ecarrier E
shows Ring.rpow (emult (ext-Union C)) 1I a n = Ring.rpow (emult E) 1I a n
by (rule ext-le-rpow [OF chain-Ring [OF E] ext-Union-Ring [OF ne nz]
    ext-Union-upper-bound [OF E] a])

```

lemma *ext-Union-eval*:

```

assumes ne: C ≠ {} and nz: 1 ≠ 0 and E: E ∈ C
and a: a ∈ ecarrier E and p: ⋀k. p k ∈ ecarrier E
shows eval (ext-Union C) a p = eval E a p
by (rule ext-le-eval [OF chain-Ring [OF E] ext-Union-Ring [OF ne nz]
    ext-Union-upper-bound [OF E] a p])

```

71.9 The union of a nonempty chain is an extension

lemma *ext-Union-is-extension*:

```

assumes ne: C ≠ {} and nz: 1 ≠ 0
shows is-extension (ext-Union C)

```

proof (rule is-extensionI)

```

obtain E0 where E0: E0 ∈ C using ne by blast
have ext0: is-extension E0 using chain E0 by (rule ext-chainD-extension)
show Field (ecarrier (ext-Union C)) (eadd (ext-Union C)) (emult (ext-Union C))
0I 1I

```

```

    by (rule ext-Union-field [OF ne nz])
  show ecarrier (ext-Union C)  $\subseteq$  ipoly-carrier
    using chain ext-chainD-extension is-extension-subset by (fastforce simp: ext-Union-def)
  show  $\bigwedge a. a \in R \implies \text{ipoly-const } a \in \text{ecarrier } (\text{ext-Union } C)$ 
    using E0 is-extension-const-mem [OF ext0] by auto
  show  $\bigwedge a b. \llbracket a \in R; b \in R \rrbracket$ 
     $\implies \text{eadd } (\text{ext-Union } C) (\text{ipoly-const } a) (\text{ipoly-const } b) = \text{ipoly-const } (a +$ 
b)
  proof -
    fix a b assume ab:  $a \in R \ b \in R$ 
    have mem:  $\text{ipoly-const } a \in \text{ecarrier } E0 \ \text{ipoly-const } b \in \text{ecarrier } E0$ 
      using ab is-extension-const-mem [OF ext0] by simp-all
    show  $\text{eadd } (\text{ext-Union } C) (\text{ipoly-const } a) (\text{ipoly-const } b) = \text{ipoly-const } (a + b)$ 
      using ext-Union-ops(1) [OF E0 mem(1) mem(2)]
      is-extension-const-add [OF ext0 ab(1) ab(2)] by simp
  qed
  show  $\bigwedge a b. \llbracket a \in R; b \in R \rrbracket$ 
     $\implies \text{emult } (\text{ext-Union } C) (\text{ipoly-const } a) (\text{ipoly-const } b) = \text{ipoly-const } (a \cdot$ 
b)
  proof -
    fix a b assume ab:  $a \in R \ b \in R$ 
    have mem:  $\text{ipoly-const } a \in \text{ecarrier } E0 \ \text{ipoly-const } b \in \text{ecarrier } E0$ 
      using ab is-extension-const-mem [OF ext0] by simp-all
    show  $\text{emult } (\text{ext-Union } C) (\text{ipoly-const } a) (\text{ipoly-const } b) = \text{ipoly-const } (a \cdot b)$ 
      using ext-Union-ops(2) [OF E0 mem(1) mem(2)]
      is-extension-const-mult [OF ext0 ab(1) ab(2)] by simp
  qed
  show root-closed (ext-Union C)
  proof (rule root-closedI)
    fix  $\mathcal{P} \ P \ i$ 
    assume Pm:  $\mathcal{P} \in \text{ecarrier } (\text{ext-Union } C)$  and Pc:  $P \in \text{poly-carrier}$ 
    and nif:  $\neg \text{index-free } \mathcal{P} \ (P, i)$ 
    from Pm obtain E where  $E: E \in C \ \mathcal{P} \in \text{ecarrier } E$  by auto
    have extE:  $\text{is-extension } E$  using chain E(1) by (rule ext-chainD-extension)
    have rc:  $\text{root-closed } E$  by (rule is-extension-root-closed [OF extE])
    have mem:  $\text{ivar } (P, i) \in \text{ecarrier } E$  by (rule root-closed-mem [OF rc E(2) Pc
nif])
    have ev:  $\text{eval } E \ (\text{ivar } (P, i)) \ (\text{lift-poly } P) = \mathbf{0}_I$ 
      by (rule root-closed-eval [OF rc E(2) Pc nif])
    have coeff:  $\text{lift-poly } P \ k \in \text{ecarrier } E$  for k
      using Pc extE by (simp add: is-extension-const-mem poly-carrier-coeff-closed)
    have eval:  $\text{eval } (\text{ext-Union } C) \ (\text{ivar } (P, i)) \ (\text{lift-poly } P) = \text{eval } E \ (\text{ivar } (P, i)) \ (\text{lift-poly}$ 
P)
      by (rule ext-Union-eval [OF ne nz E(1) mem coeff])
    then show  $\text{ivar } (P, i) \in \text{ecarrier } (\text{ext-Union } C)$ 
       $\wedge \text{eval } (\text{ext-Union } C) \ (\text{ivar } (P, i)) \ (\text{lift-poly } P) = \mathbf{0}_I$ 
      using mem E(1) ev by auto
  qed
  show op-restricted (ecarrier (ext-Union C)) (eadd (ext-Union C))
    by (rule ext-Union-restricted(1))

```



```

    show op-restricted (ecarrier (ext-Union C)) (emult (ext-Union C))
      by (rule ext-Union-restricted(2))
qed

end

end

```

72 Transporting a Field Structure along an Injection

```

context Field
begin

```

A field structure transports along a map injective on the carrier, given operations on the image that make the map a homomorphism there. Only the behaviour *on the image* is constrained, which is what makes the restricted operations of an extension free of charge: no separate lemma is needed to see that restricting a field's operations to its own carrier leaves a field. The same lemma will serve when a root is adjoined, since that too realises a field as the image of an injection.

```

lemma field-image-transport:
  fixes h :: 'a  $\Rightarrow$  'b and hadd hmult :: 'b  $\Rightarrow$  'b  $\Rightarrow$  'b
  assumes inj: inj-on h R
    and ha:  $\bigwedge a b. \llbracket a \in R; b \in R \rrbracket \Longrightarrow \text{hadd } (h a) (h b) = h (a + b)$ 
    and hm:  $\bigwedge a b. \llbracket a \in R; b \in R \rrbracket \Longrightarrow \text{hmult } (h a) (h b) = h (a \cdot b)$ 
  shows Field (h ' R) hadd hmult (h 0) (h 1)
proof -
  have grp: Group (h ' R) hadd (h 0)
  proof (rule GroupI)
    show  $\bigwedge x y. \llbracket x \in h ' R; y \in h ' R \rrbracket \Longrightarrow \text{hadd } x y \in h ' R$ 
      using ha by force
    show h 0  $\in h ' R$  by simp
    show  $\bigwedge x y z. \llbracket x \in h ' R; y \in h ' R; z \in h ' R \rrbracket$ 
       $\Longrightarrow \text{hadd } (\text{hadd } x y) z = \text{hadd } x (\text{hadd } y z)$ 
      using ha by (clarsimp simp: additive.associative)
    show  $\bigwedge x. x \in h ' R \Longrightarrow \text{hadd } (h 0) x = x$  using ha by force
    show  $\bigwedge x. x \in h ' R \Longrightarrow \text{hadd } x (h 0) = x$  using ha by force
    show  $\bigwedge x. x \in h ' R \Longrightarrow \exists y \in h ' R. \text{hadd } x y = h 0 \wedge \text{hadd } y x = h 0$ 
  proof -
    fix x assume x  $\in h ' R$ 
    then obtain a where a:  $a \in R$  and xa:  $x = h a$  by blast
    then have na:  $- a \in R$  by (simp add: additive.invertible-inverse-closed)
    have hadd (h a) (h (- a)) = h 0
      using ha [OF a na] a by (simp add: additive.invertible-right-inverse)
    moreover have hadd (h (- a)) (h a) = h 0
      using ha [OF na a] a by (simp add: additive.invertible-left-inverse)
    ultimately show  $\exists y \in h ' R. \text{hadd } x y = h 0 \wedge \text{hadd } y x = h 0$ 
      using na xa by blast
  end
end

```

```

qed
qed
interpret hadd: Group h ' R hadd h 0 by (rule grp)
interpret hadd: Abelian-Group h ' R hadd h 0
proof unfold-locales
  fix x y assume x ∈ h ' R and y ∈ h ' R
  then show hadd x y = hadd y x
    using ha by (clarsimp simp: additive.commutative)
qed
interpret hmult: commutative-monoid h ' R hmult h 1
proof unfold-locales
  show  $\bigwedge x y. \llbracket x \in h ' R; y \in h ' R \rrbracket \implies hmult\ x\ y \in h ' R$ 
    using hm by force
  show h 1 ∈ h ' R by simp
  show  $\bigwedge x y z. \llbracket x \in h ' R; y \in h ' R; z \in h ' R \rrbracket$ 
     $\implies hmult\ (hmult\ x\ y)\ z = hmult\ x\ (hmult\ y\ z)$ 
    using hm by (clarsimp simp: multiplicative.associative)
  show  $\bigwedge x. x \in h ' R \implies hmult\ (h\ 1)\ x = x$  using hm by force
  show  $\bigwedge x. x \in h ' R \implies hmult\ x\ (h\ 1) = x$  using hm by force
  show  $\bigwedge x y. \llbracket x \in h ' R; y \in h ' R \rrbracket \implies hmult\ x\ y = hmult\ y\ x$ 
    using hm by (clarsimp simp: multiplicative.commutative)
qed
have distrib:  $\bigwedge x y z. \llbracket x \in h ' R; y \in h ' R; z \in h ' R \rrbracket$ 
   $\implies hmult\ x\ (hadd\ y\ z) = hadd\ (hmult\ x\ y)\ (hmult\ x\ z)$ 
  using ha hm by (clarsimp simp: distributive)
have distrib':  $\bigwedge x y z. \llbracket x \in h ' R; y \in h ' R; z \in h ' R \rrbracket$ 
   $\implies hmult\ (hadd\ y\ z)\ x = hadd\ (hmult\ y\ x)\ (hmult\ z\ x)$ 
  using ha hm by (clarsimp simp: distributive)
have nz: h 1 ≠ h 0
  using inj nontrivial by (simp add: inj-on-eq-iff)
show ?thesis
proof unfold-locales
  show  $\bigwedge x y z. \llbracket x \in h ' R; y \in h ' R; z \in h ' R \rrbracket$ 
     $\implies hmult\ x\ (hadd\ y\ z) = hadd\ (hmult\ x\ y)\ (hmult\ x\ z)$ 
    by (rule distrib)
  show  $\bigwedge x y z. \llbracket x \in h ' R; y \in h ' R; z \in h ' R \rrbracket$ 
     $\implies hmult\ (hadd\ y\ z)\ x = hadd\ (hmult\ y\ x)\ (hmult\ z\ x)$ 
    by (rule distrib')
  show h 1 ≠ h 0 by (rule nz)
  show  $\bigwedge x. \llbracket x \in h ' R; x \neq h\ 0 \rrbracket \implies hmult.invertible\ x$ 
  proof -
    fix x assume x: x ∈ h ' R and xnz: x ≠ h 0
    then obtain a where a: a ∈ R and xa: x = h a by blast
    with xnz have anz: a ≠ 0 by blast
    have inv: multiplicative.invertible a by (rule field-inverse [OF a anz])
    have ia: multiplicative.inverse a ∈ R
      by (rule multiplicative.invertible-inverse-closed [OF inv a])
    have hmult (h a) (h (multiplicative.inverse a)) = h 1
      using hm [OF a ia] multiplicative.invertible-right-inverse [OF inv a] by simp
    moreover have hmult (h (multiplicative.inverse a)) (h a) = h 1

```

```

      using hm [OF ia a] multiplicative.invertible-left-inverse [OF inv a] by simp
    moreover have h (multiplicative.inverse a) ∈ h ‘ R using ia by simp
    ultimately show hmult.invertible x
      using xa by (blast intro: hmult.invertibleI)
  qed
qed
qed

```

72.1 The base extension is an extension

theorem *is-extension-base-ext*: *is-extension base-ext*

proof (rule *is-extensionI*)

have *Field (ipoly-const ‘ R) (eadd base-ext) (emult base-ext) (ipoly-const 0) (ipoly-const 1)*

```

  by (rule field-image-transport [OF ipoly-const-inj-on base-ext-add base-ext-mult])
then show Field (ecarrier base-ext) (eadd base-ext) (emult base-ext) 0I 1I by simp
show ecarrier base-ext ⊆ ipoly-carrier
  using ipoly-const-closed by auto
show  $\bigwedge a. a \in R \implies \text{ipoly-const } a \in \text{ecarrier base-ext}$  by simp
show  $\bigwedge a b. \llbracket a \in R; b \in R \rrbracket$ 
   $\implies \text{eadd base-ext (ipoly-const } a) (\text{ipoly-const } b) = \text{ipoly-const } (a + b)$ 
  by (rule base-ext-add)
show  $\bigwedge a b. \llbracket a \in R; b \in R \rrbracket$ 
   $\implies \text{emult base-ext (ipoly-const } a) (\text{ipoly-const } b) = \text{ipoly-const } (a \cdot b)$ 
  by (rule base-ext-mult)

```

show *root-closed base-ext*

— Vacuous: every element is a constant, and a constant is index-free.

by (rule *root-closedI*) auto

show *op-restricted (ecarrier base-ext) (eadd base-ext)* **by** (rule *base-ext-restricted(1)*)

show *op-restricted (ecarrier base-ext) (emult base-ext)* **by** (rule *base-ext-restricted(2)*)

qed

corollary *extensions-non-empty*: *extensions ≠ {}*

using *is-extension-base-ext* **by** auto

72.2 A maximal extension exists

Zorn’s lemma applies: the extensions form a set, partially ordered by *ext-le*, in which every chain has an upper bound — the union of a nonempty chain, and *base-ext* for the empty one. Maximality is what will force every polynomial over *R* to split, since an unsplit polynomial leaves an index free at which a root could be adjoined.

theorem *exists-maximal-extension*:

$\exists M \in \text{extensions}. \forall L \in \text{extensions}. \text{ext-le } M L \longrightarrow L = M$

proof (rule *predicate-Zorn [OF ext-partial-order]*)

fix *D* **assume** *D*: *D ∈ Chains (relation-of ext-le extensions)*

then have *sub*: *D ⊆ extensions* **by** (rule *Chains-relation-of*)

have *ch*: *ext-chain D*

using *D sub unfolding ext-chain-def Chains-def relation-of-def* **by** auto

```

show  $\exists u \in \text{extensions}. \forall a \in D. \text{ext-le } a \ u$ 
proof (cases  $D = \{\}$ )
  case True
    have  $\text{base-ext} \in \text{extensions}$  by (simp add: is-extension-base-ext)
    with True show ?thesis by blast
  next
  case False
    have is-extension (ext-Union  $D$ )
      by (rule ext-Union-is-extension [OF ch False nontrivial])
    moreover have  $\forall a \in D. \text{ext-le } a \ (\text{ext-Union } D)$ 
      using ext-Union-upper-bound [OF ch] by blast
    ultimately show ?thesis by auto
qed
qed

```

72.3 A fresh index always exists

Towards *exists-root*: for any nonzero polynomial P over R there is an index i that no element of a given extension involves. This is the step that makes room for a new root, and it is where the choice of indexed polynomials as the ambient type pays off — no cardinality argument is needed, only the observation that an extension *mentioning* every index would contain a root of P for each of them, and a nonzero polynomial over a field has only finitely many roots.

lemma *exists-fresh-index*:

```

assumes  $M$ : is-extension  $M$  and  $P$ :  $P \in \text{poly-carrier}$  and  $\text{Pnz}$ :  $P \neq \mathbf{0}_P$ 
shows  $\exists i. \forall \mathcal{P} \in \text{ecarrier } M. \text{index-free } \mathcal{P} \ (P, i)$ 
proof (rule ccontr)
  assume  $\neg (\exists i. \forall \mathcal{P} \in \text{ecarrier } M. \text{index-free } \mathcal{P} \ (P, i))$ 
  then have bad:  $\exists \mathcal{P} \in \text{ecarrier } M. \neg \text{index-free } \mathcal{P} \ (P, i)$  for  $i$  by blast
  interpret  $M$ : Field  $\text{ecarrier } M \ \text{eadd } M \ \text{emult } M \ \mathbf{0}_I \ \mathbf{1}_I$ 
    by (rule is-extension-field [OF  $M$ ])
  have  $\text{rc}$ : root-closed  $M$  by (rule is-extension-root-closed [OF  $M$ ])
  — Each index is realised, and its indeterminate is then a root of  $P$  in  $M$ .
  have root:  $\text{ivar } (P, i) \in \text{ecarrier } M \wedge M.\text{eval } (\text{ivar } (P, i)) \ (\text{lift-poly } P) = \mathbf{0}_I$  for  $i$ 
  proof —
    obtain  $\mathcal{P}$  where  $\mathcal{P} \in \text{ecarrier } M$  and  $\neg \text{index-free } \mathcal{P} \ (P, i)$  using bad by blast
    then show ?thesis
      using root-closed-mem [OF  $\text{rc} \ - \ P$ ] root-closed-eval [OF  $\text{rc} \ - \ P$ ] by blast
  qed
  — The lifted polynomial is a nonzero polynomial over  $M$ .
  have  $\text{lc}$ : lift-poly  $P \in M.\text{poly-carrier}$ 
  proof (rule  $M.\text{poly-carrierI}$ )
    have  $\{k. \text{lift-poly } P \ k \neq \mathbf{0}_I\} \subseteq \{k. P \ k \neq \mathbf{0}\}$ 
      by (auto simp: lift-poly-def ipoly-const-def ipoly-zero-def)
    then show finite  $\{k. \text{lift-poly } P \ k \neq \mathbf{0}_I\}$ 
      using poly-carrier-finite [OF  $P$ ] by (rule finite-subset)
    show  $\bigwedge k. \text{lift-poly } P \ k \in \text{ecarrier } M$ 
      using  $P \ M$  by (simp add: is-extension-const-mem poly-carrier-coeff-closed)
  qed

```

```

have lnz: lift-poly P ≠ M.poly-zero
proof -
  obtain k where k: P k ≠ 0 using Pnz by (auto simp: poly-zero-def fun-eq-iff)
  then have ipoly-const (P k) ≠ 0I
    by (metis ipoly-const-coeff ipoly-zero-def)
  then show ?thesis by (auto simp: lift-poly-def M.poly-zero-def fun-eq-iff)
qed
have fin: finite {a ∈ ecarrier M. M.eval a (lift-poly P) = 0I}
  using M.card-roots-le-degree [OF lc lnz] by blast
— Yet the root set contains an injective image of nat.
have inj: inj (λi. ivar (P, i) :: 'a ipoly)
  using ivar-inj [OF nontrivial] by (auto simp: inj-on-def)
have range (λi. ivar (P, i) :: 'a ipoly)
  ⊆ {a ∈ ecarrier M. M.eval a (lift-poly P) = 0I}
  using root by blast
with fin inj show False
  by (meson finite-subset finite-imageD infinite-UNIV-nat)
qed

end

```

72.4 Transporting a field onto its image, with restricted operations

An injective image of a field carries a field structure, and the operations obtained by *transport* — undo the injection, operate, redo it — are automatically the restricted ones an extension requires. Nothing is assumed about how *h* interacts with the ambient operations, which is the point: an extension's addition need not be the ambient addition, and *Ring.is-extension* does not require it to be. Choosing the operations by transport sidesteps the question instead of forcing a stronger definition.

lemma *field-transport-restricted*:

```

fixes A :: 'b set and add mult :: 'b ⇒ 'b ⇒ 'b and z u :: 'b
  and h :: 'b ⇒ 'a ipoly
assumes F: Field A add mult z u and inj: inj-on h A
defines K ≡ h ' A
  and hadd ≡ (λx y. if x ∈ h ' A ∧ y ∈ h ' A
    then h (add (inv-into A h x) (inv-into A h y)) else undefined)
  and hmult ≡ (λx y. if x ∈ h ' A ∧ y ∈ h ' A
    then h (mult (inv-into A h x) (inv-into A h y)) else undefined)
shows field-transport-field: Field K hadd hmult (h z) (h u)
  and field-transport-add-restricted: op-restricted K hadd
  and field-transport-mult-restricted: op-restricted K hmult
proof -
  have undo: inv-into A h (h a) = a if a ∈ A for a
    using inj that by (rule inv-into-f-f)
  have ha: hadd (h a) (h b) = h (add a b) if a ∈ A b ∈ A for a b
    unfolding hadd-def using that undo by simp

```

```

have hm: hmult (h a) (h b) = h (mult a b) if a ∈ A b ∈ A for a b
  unfolding hmult-def using that undo by simp
show Field K hadd hmult (h z) (h u)
  unfolding K-def by (rule Field.field-image-transport [OF F inj ha hm])
show op-restricted K hadd
  by (rule op-restrictedI) (auto simp: hadd-def K-def)
show op-restricted K hmult
  by (rule op-restrictedI) (auto simp: hmult-def K-def)
qed

```

73 Working Inside a Fixed Extension

Adjoining a root needs the polynomial machinery of the extension's own coefficient field — *pmod*, *poly-carrier*, *degree* at *ecarrier* *M*. Fixing *M* in a locale and discharging *Field* for it once, by *Ring.is-extension-field*, makes all of it available under the *M* prefix throughout, including *inside definitions*. **interpret** would not serve, being a proof-body command, but **sublocale** is its declaration-level counterpart, so no definition here has to spell out the five locale parameters.

```

locale field-extension = Field +
  fixes M :: 'a iext
  assumes is-ext: is-extension M

```

```

sublocale field-extension ⊆ M: Field ecarrier M eadd M emult M 0_I 1_I
  by (rule is-extension-field [OF is-ext])

```

```

context field-extension
begin

```

The realisation map: reduce a polynomial over *M* modulo *Q*, then read the remainder off at the fresh indeterminate, using the *ambient* operations. Both prefixes now read naturally.

```

definition realise :: 'a root-index ⇒ (nat ⇒ 'a ipoly) ⇒ (nat ⇒ 'a ipoly) ⇒ 'a ipoly
  where realise j Q S = ieval (ivar j) (M.pmod S Q)

```

A polynomial over *M* is also one over the ambient ring, its carrier being contained there. Note that the two rings share their zero, so they also share *Ring.degree* and *Ring.poly-zero*: no translation of degrees is needed anywhere below.

```

lemma coeff-ipoly-closed: S ∈ M.poly-carrier ⟹ S k ∈ ipoly-carrier
  using is-extension-subset [OF is-ext] M.poly-carrier-coeff-closed by blast

```

```

lemma low-poly-ambient:
  assumes S: S ∈ M.low-poly d shows S ∈ Ring.poly-carrier ipoly-carrier 0_I
proof (rule Ring.poly-carrierI [OF ipoly-ring])
  have Sc: S ∈ M.poly-carrier using S by (rule M.low-poly-closed)
  show finite {i. S i ≠ 0_I} using Sc by (rule M.poly-carrier-finite)
  show ∧i. S i ∈ ipoly-carrier using Sc by (rule coeff-ipoly-closed)

```

qed

On the remainders the realisation is a sum over the *fixed* range $\{..d\}$ rather than over each polynomial's own degree. A common range is what *ieval-ivar-inj* needs, and enlarging the range past the degree is harmless.

lemma *ieval-low-fincomp*:

assumes $S: S \in M.\text{low-poly } d$

shows $\text{ieval } (\text{ivar } j) S$

$= \text{commutative-monoid.fincomp } \text{ipoly-carrier } (\oplus_I) \mathbf{0}_I$
 $(\lambda k. S k \otimes_I \text{Ring.rpow } (\otimes_I) \mathbf{1}_I (\text{ivar } j) k) \{..d\}$

proof (rule *Ring.eval-fincomp-le* [OF *ipoly-ring low-poly-ambient* [OF *S*] *ivar-closed*])

have $S = M.\text{poly-zero} \vee \text{Ring.degree } \mathbf{0}_I S < d$

using $S M.\text{low-poly-iff-degree}$ [OF $M.\text{low-poly-closed}$ [OF *S*]] **by** *blast*

then show $\text{Ring.degree } \mathbf{0}_I S \leq d$

using $M.\text{degree-zero}$ **by** *fastforce*

qed

The realisation is injective on the remainders. This is the point of a *fresh* index: every coefficient lies in M and so is index-free at j , which is exactly the hypothesis under which *ieval-ivar-inj* reads the coefficients back off. Above the range the two remainders agree trivially, both coefficients being zero.

lemma *ieval-inj-on-low*:

assumes *fresh*: $\bigwedge \mathcal{P}. \mathcal{P} \in \text{ecarrier } M \implies \text{index-free } \mathcal{P} j$

shows $\text{inj-on } (\text{ieval } (\text{ivar } j)) (M.\text{low-poly } d)$

proof (rule *inj-onI*)

fix $S T$

assume $S: S \in M.\text{low-poly } d$ **and** $T: T \in M.\text{low-poly } d$

and $\text{eq}: \text{ieval } (\text{ivar } j) S = \text{ieval } (\text{ivar } j) T$

have $\text{Sc}: S \in M.\text{poly-carrier}$ **and** $\text{Tc}: T \in M.\text{poly-carrier}$

using $S T$ **by** (*blast intro: M.low-poly-closed*)**+**

have $\text{Sip}: S k \in \text{ipoly-carrier}$ **and** $\text{Tip}: T k \in \text{ipoly-carrier}$ **for** k

using Sc Tc **by** (*blast intro: coeff-ipoly-closed*)**+**

have $\text{Sf}: \text{index-free } (S k) j$ **and** $\text{Tf}: \text{index-free } (T k) j$ **for** k

using *fresh Sc Tc* **by** (*blast intro: M.poly-carrier-coeff-closed*)**+**

have $\text{sums}: \text{commutative-monoid.fincomp } \text{ipoly-carrier } (\oplus_I) \mathbf{0}_I$

$(\lambda k. S k \otimes_I \text{Ring.rpow } (\otimes_I) \mathbf{1}_I (\text{ivar } j) k) \{..d\}$

$= \text{commutative-monoid.fincomp } \text{ipoly-carrier } (\oplus_I) \mathbf{0}_I$

$(\lambda k. T k \otimes_I \text{Ring.rpow } (\otimes_I) \mathbf{1}_I (\text{ivar } j) k) \{..d\}$

using *ieval-low-fincomp* [OF *S*] *ieval-low-fincomp* [OF *T*] *eq* **by** *simp*

show $S = T$

proof (rule *ext*)

fix k

show $S k = T k$

proof (*cases* $k \leq d$)

case *True*

show *?thesis* **by** (rule *ieval-ivar-inj* [OF *Sip Sf Tip Tf sums True*])

next

case *False*

then have $S k = \mathbf{0}_I$ **and** $T k = \mathbf{0}_I$

```

    using  $S$   $T$  unfolding  $M.\text{low-poly-def}$  by auto
    then show ?thesis by simp
qed
qed
qed

```

The extension obtained by adjoining a root. The carrier is the image of the remainders modulo Q under evaluation at the fresh indeterminate; the operations are the transported ones. Reduction modulo Q has to live in the multiplication, since the realisation of Q itself is not zero in the ambient ring — which is exactly why an extension carries its own operations rather than inheriting the ambient ones.

definition $\text{adjoin-carrier} :: 'a \text{ root-index} \Rightarrow (\text{nat} \Rightarrow 'a \text{ ipoly}) \Rightarrow 'a \text{ ipoly set}$
where $\text{adjoin-carrier } j \ Q = \text{ieval } (\text{ivar } j) \ `M.\text{low-poly } (M.\text{degree } Q)$

definition $\text{adjoin} :: 'a \text{ root-index} \Rightarrow (\text{nat} \Rightarrow 'a \text{ ipoly}) \Rightarrow 'a \text{ iext}$
where $\text{adjoin } j \ Q =$
 $(\text{adjoin-carrier } j \ Q,$
 $(\lambda x \ y. \text{if } x \in \text{adjoin-carrier } j \ Q \wedge y \in \text{adjoin-carrier } j \ Q$
 $\text{then } \text{ieval } (\text{ivar } j)$
 $\quad (M.\text{poly-add } (\text{inv-into } (M.\text{low-poly } (M.\text{degree } Q)) (\text{ieval } (\text{ivar } j)) \ x)$
 $\quad (\text{inv-into } (M.\text{low-poly } (M.\text{degree } Q)) (\text{ieval } (\text{ivar } j)) \ y))$
 $\text{else undefined}),$
 $(\lambda x \ y. \text{if } x \in \text{adjoin-carrier } j \ Q \wedge y \in \text{adjoin-carrier } j \ Q$
 $\text{then } \text{ieval } (\text{ivar } j)$
 $\quad (M.\text{pmod } (M.\text{poly-mult}$
 $\quad (\text{inv-into } (M.\text{low-poly } (M.\text{degree } Q)) (\text{ieval } (\text{ivar } j)) \ x)$
 $\quad (\text{inv-into } (M.\text{low-poly } (M.\text{degree } Q)) (\text{ieval } (\text{ivar } j)) \ y)) \ Q)$
 $\text{else undefined}))$

lemma adjoin-carrier-eq [simp]: $\text{ecarrier } (\text{adjoin } j \ Q) = \text{adjoin-carrier } j \ Q$
by (simp add: adjoin-def)

The field structure and the restrictedness of both operations come together, by transporting $\text{Field.remainder-field}$ along the realisation, which ieval-inj-on-low shows to be injective on the remainders. The zero and the unit land on $\mathbf{0}_I$ and $\mathbf{1}_I$ because evaluation sends the zero and unit polynomials there.

The realisation sends the zero, the unit and the variable of the polynomial ring over M to $\mathbf{0}_I$, $\mathbf{1}_I$ and the fresh indeterminate. The first two are what make the transported field structure have the zero and unit that *is-extension* demands; the third is what exhibits $\text{ivar } j$ as an element of the adjoined carrier.

lemma ieval-poly-zero : $\text{ieval } (\text{ivar } j) \ M.\text{poly-zero} = \mathbf{0}_I$
by (rule Ring.eval-zero [OF $\text{ipoly-ring ivar-closed}$])

lemma ieval-poly-one : $\text{ieval } (\text{ivar } j) \ M.\text{poly-one} = \mathbf{1}_I$
by (rule Ring.eval-one [OF $\text{ipoly-ring ivar-closed}$])

lemma ieval-poly-var : $\text{ieval } (\text{ivar } j) \ M.\text{var} = \text{ivar } j$

by (rule *Ring.eval-var* [*OF ipoly-ring ivar-closed*])

Nontriviality of the ambient indexed ring, taken from R rather than from M : a statement about $\mathbf{1}_I$ and $\mathbf{0}_I$ alone is generalised over the index type and so cannot be derived from a hypothesis where that type is pinned.

lemma *ipoly-one-neq-zero*: ($\mathbf{1}_I :: 'a \text{ root-index multiset} \Rightarrow 'a) \neq \mathbf{0}_I$

proof –

have *one*: $\mathbf{1} \in R$ and *zero*: $\mathbf{0} \in R$ by *simp-all*

have (*ipoly-const* $\mathbf{1} :: 'a \text{ root-index multiset} \Rightarrow 'a) \neq \text{ipoly-const } \mathbf{0}$

using *inj-on-eq-iff* [*OF ipoly-const-inj-on one zero*] **nontrivial** by *simp*

then show *?thesis* by *simp*

qed

lemma *var-low*:

assumes *d2*: $2 \leq M.\text{degree } Q$ **shows** $M.\text{var} \in M.\text{low-poly } (M.\text{degree } Q)$

proof –

have $M.\text{degree } M.\text{var} = 1$ by (rule *M.degree-var* [*OF ipoly-one-neq-zero*])

with *d2* **show** *?thesis* using *M.var-closed* by (*simp add: M.low-poly-iff-degree*)

qed

The constants of M sit inside the adjoined carrier, realised by the constant polynomials, and the transported operations restrict to M 's own there. So M lies below the adjoined extension in the ordering — which is what carries M 's root witnesses upwards, by *Ring.ext-le-eval*.

lemma *M-Ring*: *Ring* (*ecarrier* M) (*eadd* M) (*emult* M) $\mathbf{0}_I$ $\mathbf{1}_I$

using *is-extension-field* [*OF is-ext*] by (*simp add: Field-def commutative-ring-def*)

lemma *adjoin-rep*:

assumes *fresh*: $\bigwedge \mathcal{P}. \mathcal{P} \in \text{ecarrier } M \Rightarrow \text{index-free } \mathcal{P} \ j$

and *S*: $S \in M.\text{low-poly } (M.\text{degree } Q)$

shows *inv-into* ($M.\text{low-poly } (M.\text{degree } Q)$) (*ieval* (*ivar* j)) (*ieval* (*ivar* j) S) = S

using *ieval-inj-on-low* [*OF fresh*] S by (rule *inv-into-f-f*)

lemma *ieval-poly-const*:

assumes *y*: $y \in \text{ipoly-carrier}$ **shows** *ieval* (*ivar* j) ($M.\text{poly-const } y$) = y

by (rule *Ring.eval-const* [*OF ipoly-ring ivar-closed y*])

lemma *const-low*:

assumes *irr*: $M.\text{poly-irreducible } Q$ and *y*: $y \in \text{ecarrier } M$

shows $M.\text{poly-const } y \in M.\text{low-poly } (M.\text{degree } Q)$

proof –

have *dpos*: $0 < M.\text{degree } Q$ using *irr* by (rule *M.poly-irreducible-degree-pos*)

have *c*: $M.\text{poly-const } y \in M.\text{poly-carrier}$ using *y* by (rule *M.poly-const-closed*)

have $M.\text{degree } (M.\text{poly-const } y) = 0$ using *M.degree-const-le* by (*simp add: le-zero-eq*)

with *dpos* **show** *?thesis* using *c* by (*simp add: M.low-poly-iff-degree*)

qed

lemma *const-rep*:

assumes *fresh*: $\bigwedge \mathcal{P}. \mathcal{P} \in \text{ecarrier } M \Rightarrow \text{index-free } \mathcal{P} \ j$

and $\text{irr}: M.\text{poly-irreducible } Q$ and $y: y \in \text{ecarrier } M$
 shows $\text{inv-into } (M.\text{low-poly } (M.\text{degree } Q)) (\text{ieval } (\text{ivar } j)) y = M.\text{poly-const } y$
 proof –
 have $\text{yip}: y \in \text{ipoly-carrier}$ using $\text{is-ext } y$ by (rule $\text{is-extension-carrier-closed}$)
 have $\text{inv-into } (M.\text{low-poly } (M.\text{degree } Q)) (\text{ieval } (\text{ivar } j)) (\text{ieval } (\text{ivar } j) (M.\text{poly-const } y))$
 $= M.\text{poly-const } y$
 by (rule $\text{adjoin-rep } [OF \text{ fresh const-low } [OF \text{ irr } y]]$)
 then show $?thesis$ using $\text{ieval-poly-const } [OF \text{ yip}]$ by simp
 qed

lemma $\text{ecarrier-subset-adjoin}$:
 assumes $\text{fresh}: \bigwedge \mathcal{P}. \mathcal{P} \in \text{ecarrier } M \implies \text{index-free } \mathcal{P} \ j$
 and $\text{irr}: M.\text{poly-irreducible } Q$
 shows $\text{ecarrier } M \subseteq \text{adjoin-carrier } j \ Q$
 proof
 fix y assume $y: y \in \text{ecarrier } M$
 have $\text{yip}: y \in \text{ipoly-carrier}$ using $\text{is-ext } y$ by (rule $\text{is-extension-carrier-closed}$)
 have $y = \text{ieval } (\text{ivar } j) (M.\text{poly-const } y)$ using $\text{ieval-poly-const } [OF \text{ yip}]$ by simp
 then show $y \in \text{adjoin-carrier } j \ Q$
 unfolding $\text{adjoin-carrier-def}$ using $\text{const-low } [OF \text{ irr } y]$ by blast
 qed

lemma ext-le-adjoin :
 assumes $\text{fresh}: \bigwedge \mathcal{P}. \mathcal{P} \in \text{ecarrier } M \implies \text{index-free } \mathcal{P} \ j$
 and $\text{irr}: M.\text{poly-irreducible } Q$
 shows $\text{ext-le } M \ (\text{adjoin } j \ Q)$
 proof (rule ext-leI)
 show $\text{ecarrier } M \subseteq \text{ecarrier } (\text{adjoin } j \ Q)$
 using $\text{ecarrier-subset-adjoin } [OF \text{ fresh irr}]$ by simp
 next
 fix $y \ y'$ assume $y: y \in \text{ecarrier } M$ and $y': y' \in \text{ecarrier } M$
 have $K: y \in \text{adjoin-carrier } j \ Q \ y' \in \text{adjoin-carrier } j \ Q$
 using $\text{ecarrier-subset-adjoin } [OF \text{ fresh irr}] \ y \ y'$ by blast+
 have $\text{ry}: \text{inv-into } (M.\text{low-poly } (M.\text{degree } Q)) (\text{ieval } (\text{ivar } j)) y = M.\text{poly-const } y$
 and $\text{ry}': \text{inv-into } (M.\text{low-poly } (M.\text{degree } Q)) (\text{ieval } (\text{ivar } j)) y' = M.\text{poly-const } y'$
 using $\text{const-rep } [OF \text{ fresh irr } y] \ \text{const-rep } [OF \text{ fresh irr } y']$ by blast+
 have $\text{sum}: \text{eadd } M \ y \ y' \in \text{ecarrier } M$ using $M.\text{additive.composition-closed } y \ y'$ by simp
 have $\text{eadd } (\text{adjoin } j \ Q) \ y \ y'$
 $= \text{ieval } (\text{ivar } j) (M.\text{poly-add } (M.\text{poly-const } y) (M.\text{poly-const } y'))$
 unfolding adjoin-def using $K \ \text{ry} \ \text{ry}'$ by simp
 also have $M.\text{poly-add } (M.\text{poly-const } y) (M.\text{poly-const } y') = M.\text{poly-const } (\text{eadd } M \ y \ y')$
 by (rule $M.\text{poly-const-add } [OF \text{ y } y']$)
 also have $\text{ieval } (\text{ivar } j) (M.\text{poly-const } (\text{eadd } M \ y \ y')) = \text{eadd } M \ y \ y'$
 using $\text{is-extension-carrier-closed } [OF \text{ is-ext sum}]$ by (rule ieval-poly-const)
 finally show $\text{eadd } M \ y \ y' = \text{eadd } (\text{adjoin } j \ Q) \ y \ y'$ by simp
 next
 fix $y \ y'$ assume $y: y \in \text{ecarrier } M$ and $y': y' \in \text{ecarrier } M$

```

have K: y ∈ adjoin-carrier j Q y' ∈ adjoin-carrier j Q
  using ecarrier-subset-adjoin [OF fresh irr] y y' by blast+
have ry: inv-into (M.low-poly (M.degree Q)) (ieval (ivar j)) y = M.poly-const y
  and ry': inv-into (M.low-poly (M.degree Q)) (ieval (ivar j)) y' = M.poly-const y'
  using const-rep [OF fresh irr y] const-rep [OF fresh irr y'] by blast+
have prod: emult M y y' ∈ ecarrier M using M.multiplicative.composition-closed y
y' by simp
have Qc: Q ∈ M.poly-carrier using irr by (rule M.poly-irreducibleD-carrier)
have Qnz: Q ≠ M.poly-zero using irr by (rule M.poly-irreducibleD-nonzero)
have emult (adjoin j Q) y y'
  = ieval (ivar j)
    (M.pmod (M.poly-mult (M.poly-const y) (M.poly-const y')) Q)
  unfolding adjoin-def using K ry ry' by simp
also have M.poly-mult (M.poly-const y) (M.poly-const y') = M.poly-const (emult
M y y')
  by (rule M.poly-const-mult [OF y y'])
also have M.pmod (M.poly-const (emult M y y')) Q = M.poly-const (emult M y y')
proof (rule M.pmod-small [OF Qc Qnz M.poly-const-closed [OF prod]])
  have M.degree (M.poly-const (emult M y y')) = 0
  using M.degree-const-le by (simp add: le-zero-eq)
  then show M.poly-const (emult M y y') = M.poly-zero
    ∨ M.degree (M.poly-const (emult M y y')) < M.degree Q
  using M.poly-irreducible-degree-pos [OF irr] by simp
qed
also have ieval (ivar j) (M.poly-const (emult M y y')) = emult M y y'
  using is-extension-carrier-closed [OF is-ext prod] by (rule ieval-poly-const)
finally show emult M y y' = emult (adjoin j Q) y y' by simp
qed

```

lemma *adjoin-field*:

```

assumes fresh:  $\bigwedge \mathcal{P}. \mathcal{P} \in \text{ecarrier } M \implies \text{index-free } \mathcal{P} \ j$ 
  and irr: M.poly-irreducible Q
shows Field (ecarrier (adjoin j Q)) (eadd (adjoin j Q)) (emult (adjoin j Q))
  (ieval (ivar j) M.poly-zero) (ieval (ivar j) M.poly-one)
  and op-restricted (ecarrier (adjoin j Q)) (eadd (adjoin j Q))
  and op-restricted (ecarrier (adjoin j Q)) (emult (adjoin j Q))
using field-transport-restricted
  [OF M.remainer-field [OF irr] ieval-inj-on-low [OF fresh]]
by (simp-all add: adjoin-def adjoin-carrier-def)

```

The adjoined field, with the zero and unit *is-extension* demands.

lemma *adjoin-field-zero-one*:

```

assumes fresh:  $\bigwedge \mathcal{P}. \mathcal{P} \in \text{ecarrier } M \implies \text{index-free } \mathcal{P} \ j$ 
  and irr: M.poly-irreducible Q
shows Field (ecarrier (adjoin j Q)) (eadd (adjoin j Q)) (emult (adjoin j Q))
  0I 1I
  using adjoin-field (1) [OF fresh irr] ieval-poly-zero ieval-poly-one by simp

```

lemma *adjoin-Ring*:

```

assumes fresh:  $\bigwedge \mathcal{P}. \mathcal{P} \in \text{ecarrier } M \implies \text{index-free } \mathcal{P} \ j$ 

```

and *irr*: $M.\text{poly-irreducible } Q$
shows $\text{Ring } (e\text{carrier } (\text{adjoin } j \ Q)) \ (e\text{add } (\text{adjoin } j \ Q)) \ (e\text{mult } (\text{adjoin } j \ Q))$
 $\mathbf{0}_I \ \mathbf{1}_I$
using *adjoin-field-zero-one* [*OF fresh irr*] **by** (*simp add: Field-def commutative-ring-def*)

lemma *lift-poly-M-carrier*:

assumes $P: P \in \text{poly-carrier}$ **shows** $\text{lift-poly } P \in M.\text{poly-carrier}$
proof (*rule M.poly-carrierI*)
have $\{k. \text{lift-poly } P \ k \neq \mathbf{0}_I\} \subseteq \{k. P \ k \neq \mathbf{0}\}$
by (*auto simp: lift-poly-def ipoly-const-def ipoly-zero-def*)
then show $\text{finite } \{k. \text{lift-poly } P \ k \neq \mathbf{0}_I\}$
using *poly-carrier-finite* [*OF P*] **by** (*rule finite-subset*)
show $\bigwedge k. \text{lift-poly } P \ k \in e\text{carrier } M$
using *P is-ext* **by** (*simp add: is-extension-const-mem poly-carrier-coeff-closed*)
qed

Root-closedness at the adjoined index itself. The fresh indeterminate is the realisation of the variable, so it lies in the carrier once the modulus has degree at least two. And it is a root: *eval-transport* moves the evaluation into the field of remainders, where *eval-remainder-at-var* identifies it with the reduction of the polynomial — which vanishes precisely because the modulus divides it.

lemma *root-closed-adjoin-self*:

assumes *fresh*: $\bigwedge \mathcal{P}. \mathcal{P} \in e\text{carrier } M \implies \text{index-free } \mathcal{P} \ j$
and *irr*: $M.\text{poly-irreducible } Q$
and *dvd*: $M.\text{poly-dvd } Q \ (\text{lift-poly } (\text{fst } j))$
and *Pj*: $\text{fst } j \in \text{poly-carrier}$
and *d2*: $2 \leq M.\text{degree } Q$
shows $\text{ivar } j \in e\text{carrier } (\text{adjoin } j \ Q)$
 $\wedge e\text{eval } (\text{adjoin } j \ Q) \ (\text{ivar } j) \ (\text{lift-poly } (\text{fst } j)) = \mathbf{0}_I$
proof —
have *varlow*: $M.\text{var} \in M.\text{low-poly } (M.\text{degree } Q)$ **by** (*rule var-low* [*OF d2*])
have $\text{ivar } j = \text{ieval } (\text{ivar } j) \ M.\text{var}$ **by** (*rule ieval-poly-var* [*symmetric*])
then have $\text{ivar } j \in \text{ieval } (\text{ivar } j) \ M.\text{low-poly } (M.\text{degree } Q)$ **using** *varlow* **by** *blast*
then have *mem*: $\text{ivar } j \in e\text{carrier } (\text{adjoin } j \ Q)$ **by** (*simp add: adjoin-carrier-def*)
have *RA*: $\text{Ring } (M.\text{low-poly } (M.\text{degree } Q)) \ (M.\text{poly-add})$
 $(\lambda S \ T. M.\text{pmod } (M.\text{poly-mult } S \ T) \ Q) \ M.\text{poly-zero } M.\text{poly-one}$
using *M.remainer-field* [*OF irr*] **by** (*simp add: Field-def commutative-ring-def*)
have *RK*: $\text{Ring } (\text{ieval } (\text{ivar } j) \ M.\text{low-poly } (M.\text{degree } Q))$
 $(e\text{add } (\text{adjoin } j \ Q)) \ (e\text{mult } (\text{adjoin } j \ Q))$
 $(\text{ieval } (\text{ivar } j) \ M.\text{poly-zero}) \ (\text{ieval } (\text{ivar } j) \ M.\text{poly-one})$
using *adjoin-Ring* [*OF fresh irr*] *ieval-poly-zero* *ieval-poly-one*
by (*simp add: adjoin-carrier-def*)
— The transported operations agree with the adjoined ones on realisations.
have *ha*: $\text{ieval } (\text{ivar } j) \ (M.\text{poly-add } S \ T)$
 $= e\text{add } (\text{adjoin } j \ Q) \ (\text{ieval } (\text{ivar } j) \ S) \ (\text{ieval } (\text{ivar } j) \ T)$
if *ST*: $S \in M.\text{low-poly } (M.\text{degree } Q) \ T \in M.\text{low-poly } (M.\text{degree } Q)$ **for** $S \ T$
unfolding *adjoin-def*
using *ST adjoin-rep* [*OF fresh ST(1)*] *adjoin-rep* [*OF fresh ST(2)*]
by (*simp add: adjoin-carrier-def*)

```

have hm: ieval (ivar j) (M.pmod (M.poly-mult S T) Q)
      = emult (adjoin j Q) (ieval (ivar j S) (ieval (ivar j T)))
if ST: S ∈ M.low-poly (M.degree Q) T ∈ M.low-poly (M.degree Q) for S T
unfolding adjoin-def
using ST adjoin-rep [OF fresh ST(1)] adjoin-rep [OF fresh ST(2)]
by (simp add: adjoin-carrier-def)
have liftc: lift-poly (fst j) ∈ M.poly-carrier by (rule lift-poly-M-carrier [OF Pj])
have coeffM: lift-poly (fst j) k ∈ ecarrier M for k
using liftc by (rule M.poly-carrier-coeff-closed)
have constlow: M.poly-const (lift-poly (fst j) k) ∈ M.low-poly (M.degree Q) for k
using const-low [OF irr coeffM] .
have hconst: ieval (ivar j) (M.poly-const (lift-poly (fst j) k)) = lift-poly (fst j) k for
k
using ieval-poly-const [OF is-extension-carrier-closed [OF is-ext coeffM]] .
have hfun: ( $\lambda k. \text{ieval } (\text{ivar } j) (M.\text{poly-const } (\text{lift-poly } (\text{fst } j) k))$ ) = lift-poly (fst j)
by (rule ext) (rule hconst)
note transport =
      eval-transport [OF RA RK ha hm ieval-inj-on-low [OF fresh] varlow constlow]
note evalrem = M.eval-remainder-at-var [OF irr d2 liftc]
have pmodz: M.pmod (lift-poly (fst j)) Q = M.poly-zero
using dvd M.poly-dvd-iff-pmod-zero
      [OF M.poly-irreducibleD-carrier [OF irr] M.poly-irreducibleD-nonzero [OF
irr] liftc]
by simp
— Move the evaluation into the remainder field, reduce, and come back.
have step1: eeval (adjoin j Q) (ivar j) (lift-poly (fst j))
      = Ring.eval (ieval (ivar j) ‘M.low-poly (M.degree Q)’)
      (eadd (adjoin j Q) (emult (adjoin j Q)))
      (ieval (ivar j) M.poly-zero) (ieval (ivar j) M.poly-one)
      (ieval (ivar j) M.var)
      ( $\lambda k. \text{ieval } (\text{ivar } j) (M.\text{poly-const } (\text{lift-poly } (\text{fst } j) k))$ )
unfolding adjoin-carrier-eq adjoin-carrier-def hfun
      ieval-poly-zero ieval-poly-one ieval-poly-var
by rule
have eeval (adjoin j Q) (ivar j) (lift-poly (fst j))
      = ieval (ivar j)
      (Ring.eval (M.low-poly (M.degree Q)) (M.poly-add)
      ( $\lambda S T. M.\text{pmod } (M.\text{poly-mult } S T) Q$ ) M.poly-zero M.poly-one
      M.var ( $\lambda k. M.\text{poly-const } (\text{lift-poly } (\text{fst } j) k)$ )))
unfolding step1 by (rule transport)
also have ... = ieval (ivar j) M.poly-zero using evalrem pmodz by simp
also have ... = 0I by (rule ieval-poly-zero)
finally have eeval (adjoin j Q) (ivar j) (lift-poly (fst j)) = 0I .
with mem show ?thesis by blast
qed

```

Root-closedness at an index other than the adjoined one. An element of the adjoined carrier that involves such an index must have a coefficient involving it — removing carrier copies of the adjoined index cannot remove a different one — and that coefficient lies in *M*, where root-closedness applies. *Ring.ext-le-eval* then

carries the root upwards.

lemma *root-closed-adjoin-other*:

assumes *fresh*: $\bigwedge \mathcal{P}. \mathcal{P} \in \text{ecarrier } M \implies \text{index-free } \mathcal{P} \ j$

and *irr*: $M.\text{poly-irreducible } Q$

and *S*: $S \in M.\text{low-poly } (M.\text{degree } Q)$

and *P'*: $P' \in \text{poly-carrier}$ **and** *ne*: $(P', i') \neq j$

and *nf*: $\neg \text{index-free } (\text{ieval } (\text{ivar } j) S) (P', i')$

shows $\text{ivar } (P', i') \in \text{ecarrier } (\text{adjoin } j Q)$

$\wedge \text{eval } (\text{adjoin } j Q) (\text{ivar } (P', i')) (\text{lift-poly } P') = \mathbf{0}_I$

proof –

have *Sc*: $S \in M.\text{poly-carrier}$ **using** *S* **by** (rule *M.low-poly-closed*)

have *Sip*: $S \ k \in \text{ipoly-carrier}$ **for** *k* **using** *Sc* **by** (rule *coeff-ipoly-closed*)

have *SM*: $S \ k \in \text{ecarrier } M$ **for** *k* **using** *Sc* **by** (rule *M.poly-carrier-coeff-closed*)

have *Sf*: $\text{index-free } (S \ k) \ j$ **for** *k* **using** *fresh SM* **by** *blast*

obtain *m* **where** *m*: $(P', i') \in \# \ m$ **and** *nz*: $\text{ieval } (\text{ivar } j) S \ m \neq \mathbf{0}$

using *nf* **unfolding** *index-free-def* **by** *blast*

have *val*: $\text{ieval } (\text{ivar } j) S \ m$

$= (\text{if count } m \ j \leq M.\text{degree } Q$

$\text{then } S \ (\text{count } m \ j) \ (m - \text{replicate-mset } (\text{count } m \ j) \ j) \text{ else } \mathbf{0})$

using *ieval-low-fincomp* [*OF S*] *ieval-ivar* [*OF Sip Sf*] **by** *simp*

with *nz* **have** *cnz*: $S \ (\text{count } m \ j) \ (m - \text{replicate-mset } (\text{count } m \ j) \ j) \neq \mathbf{0}$

by (*auto split: if-split-asm*)

have *inm*: $(P', i') \in \# \ m - \text{replicate-mset } (\text{count } m \ j) \ j$

using *m ne* **by** (*simp add: in-diff-count count-replicate-mset*)

then **have** *nfS*: $\neg \text{index-free } (S \ (\text{count } m \ j)) (P', i')$

using *cnz* **unfolding** *index-free-def* **by** *blast*

have *rc*: *root-closed* *M* **using** *is-ext* **by** (rule *is-extension-root-closed*)

have *mem*: $\text{ivar } (P', i') \in \text{ecarrier } M$ **by** (rule *root-closed-mem* [*OF rc SM P' nfS*])

have *rootM*: $\text{eval } M \ (\text{ivar } (P', i')) (\text{lift-poly } P') = \mathbf{0}_I$

by (rule *root-closed-eval* [*OF rc SM P' nfS*])

have $\text{eval } (\text{adjoin } j Q) (\text{ivar } (P', i')) (\text{lift-poly } P')$

$= \text{eval } M \ (\text{ivar } (P', i')) (\text{lift-poly } P')$

proof (rule *ext-le-eval* [*OF M-Ring adjoin-Ring*] [*OF fresh irr*] *ext-le-adjoin* [*OF fresh irr*] *mem*])

show $\bigwedge k. \text{lift-poly } P' \ k \in \text{ecarrier } M$

using *P'* *is-extension-const-mem* [*OF is-ext*] *poly-carrier-coeff-closed* **by** *simp*

qed

with *rootM mem* *ecarrier-subset-adjoin* [*OF fresh irr*] **show** *?thesis* **by** *auto*

qed

The adjoined extension is root-closed. The two cases above, joined by the observation that the adjoined index can only be involved at all when the modulus is nonlinear: were its degree at most one the remainders would be constants, so every element of the carrier would lie in *M* and be index-free at the fresh index by freshness.

lemma *root-closed-adjoin*:

assumes *fresh*: $\bigwedge \mathcal{P}. \mathcal{P} \in \text{ecarrier } M \implies \text{index-free } \mathcal{P} \ j$

and *irr*: $M.\text{poly-irreducible } Q$

and *dvd*: $M.\text{poly-dvd } Q \ (\text{fst } j)$

```

    and Pj: fst j ∈ poly-carrier
  shows root-closed (adjoin j Q)
proof (rule root-closedI)
  fix P P' i'
  assume Pmem: P ∈ ecarrier (adjoin j Q) and P': P' ∈ poly-carrier
  and nf: ¬ index-free P (P', i')
  obtain S where S: S ∈ M.low-poly (M.degree Q) and Peq: P = ieval (ivar j) S
  using Pmem unfolding adjoin-carrier-eq adjoin-carrier-def by blast
  show ivar (P', i') ∈ ecarrier (adjoin j Q)
    ∧ eeval (adjoin j Q) (ivar (P', i')) (lift-poly P') = 0I
  proof (cases (P', i') = j)
    case False
    show ?thesis
      using root-closed-adjoin-other [OF fresh irr S P' False] nf Peq by simp
  next
  case True
  — A linear modulus leaves only constants, which are index-free at the fresh index.
  have d2: 2 ≤ M.degree Q
  proof (rule ccontr)
    assume ¬ 2 ≤ M.degree Q
    then have d1: M.degree Q ≤ 1 by simp
    have Sc: S ∈ M.poly-carrier using S by (rule M.low-poly-closed)
    have S = M.poly-zero ∨ M.degree S < M.degree Q
      using S M.low-poly-iff-degree [OF Sc] by blast
    then have dS: M.degree S = 0 using d1 M.degree-zero by fastforce
    have S = M.poly-const (S 0) by (rule M.degree-zero-imp-const [OF Sc dS])
    moreover have S 0 ∈ ecarrier M using Sc by (rule M.poly-carrier-coeff-closed)
    ultimately have P = S 0
      using Peq ieval-poly-const [OF is-extension-carrier-closed [OF is-ext]] by metis
    then have index-free P j using fresh ⟨S 0 ∈ ecarrier M⟩ by simp
    with nf True show False by simp
  qed
  have Pfst: P' = fst j using True by (metis fst-conv)
  show ?thesis
    using root-closed-adjoin-self [OF fresh irr dvd Pj d2] True Pfst by simp
  qed
qed

```

The adjoined triple is an extension. Everything is now in place: the field structure and the restrictedness of both operations come from the transport, root-closedness from the two cases above, and the conditions on the constants from the fact that M lies below, its own agreement with R on constants carrying over.

theorem *is-extension-adjoin*:
assumes *fresh*: $\bigwedge P. P \in \text{ecarrier } M \implies \text{index-free } P \ j$
and *irr*: $M.\text{poly-irreducible } Q$
and *dvd*: $M.\text{poly-dvd } Q \ (\text{lift-poly } (\text{fst } j))$
and *Pj*: $\text{fst } j \in \text{poly-carrier}$
shows *is-extension* (adjoin j Q)

```

proof (rule is-extensionI)
  show Field (ecarrier (adjoin j Q)) (eadd (adjoin j Q)) (emult (adjoin j Q)) 0I 1I
    by (rule adjoin-field-zero-one [OF fresh irr])
  show ecarrier (adjoin j Q)  $\subseteq$  ipoly-carrier
proof
  fix y assume y  $\in$  ecarrier (adjoin j Q)
  then obtain S where S: S  $\in$  M.low-poly (M.degree Q) and yeq: y = ieval (ivar
j) S
    unfolding adjoin-carrier-eq adjoin-carrier-def by blast
    show y  $\in$  ipoly-carrier
    using yeq Ring.eval-closed [OF ipoly-ring low-poly-ambient [OF S] ivar-closed]
by simp
qed
next
  fix a assume a: a  $\in$  R
  have ipoly-const a  $\in$  ecarrier M using is-ext a by (rule is-extension-const-mem)
  then have ipoly-const a  $\in$  adjoin-carrier j Q
    using ecarrier-subset-adjoin [OF fresh irr] by blast
  then show ipoly-const a  $\in$  ecarrier (adjoin j Q) by simp
next
  fix a b assume a: a  $\in$  R and b: b  $\in$  R
  have mem: ipoly-const a  $\in$  ecarrier M ipoly-const b  $\in$  ecarrier M
    using is-ext a b by (blast intro: is-extension-const-mem)+
  have eadd (adjoin j Q) (ipoly-const a) (ipoly-const b)
    = eadd M (ipoly-const a) (ipoly-const b)
    by (rule ext-le-add [OF ext-le-adjoin [OF fresh irr] mem(1) mem(2), symmetric])
  also have ... = ipoly-const (a + b) using is-ext a b by (rule is-extension-const-add)
  finally show eadd (adjoin j Q) (ipoly-const a) (ipoly-const b) = ipoly-const (a + b)
  .
next
  fix a b assume a: a  $\in$  R and b: b  $\in$  R
  have mem: ipoly-const a  $\in$  ecarrier M ipoly-const b  $\in$  ecarrier M
    using is-ext a b by (blast intro: is-extension-const-mem)+
  have emult (adjoin j Q) (ipoly-const a) (ipoly-const b)
    = emult M (ipoly-const a) (ipoly-const b)
    by (rule ext-le-mult [OF ext-le-adjoin [OF fresh irr] mem(1) mem(2), symmetric])
  also have ... = ipoly-const (a · b) using is-ext a b by (rule is-extension-const-mult)
  finally show emult (adjoin j Q) (ipoly-const a) (ipoly-const b) = ipoly-const (a · b)
  .
next
  show root-closed (adjoin j Q) by (rule root-closed-adjoin [OF fresh irr dvd Pj])
next
  show op-restricted (ecarrier (adjoin j Q)) (eadd (adjoin j Q))
    by (rule adjoin-field (2) [OF fresh irr])
next
  show op-restricted (ecarrier (adjoin j Q)) (emult (adjoin j Q))
    by (rule adjoin-field (3) [OF fresh irr])
qed

```

And it is *strictly* larger: the fresh indeterminate involves its own index, whereas

every element of M is index-free there.

lemma *adjoin-strictly-larger*:

assumes *fresh*: $\bigwedge \mathcal{P}. \mathcal{P} \in \text{ecarrier } M \implies \text{index-free } \mathcal{P} \ j$
and *irr*: $M.\text{poly-irreducible } Q$ **and** *d2*: $2 \leq M.\text{degree } Q$
shows $\text{adjoin } j \ Q \neq M$

proof –

have $\text{ivar } j = \text{ieval } (\text{ivar } j) \ M.\text{var}$ **by** (rule *ieval-poly-var* [*symmetric*])
then have $\text{ivar } j \in \text{ieval } (\text{ivar } j) \ M.\text{low-poly } (M.\text{degree } Q)$
using *var-low* [*OF d2*] **by** *blast*
then have $\text{inC}: \text{ivar } j \in \text{ecarrier } (\text{adjoin } j \ Q)$ **by** (*simp add: adjoin-carrier-def*)
have $\text{notinM}: \text{ivar } j \notin \text{ecarrier } M$
proof
assume $\text{ivar } j \in \text{ecarrier } M$
then have $\text{index-free } (\text{ivar } j) \ j$ **by** (rule *fresh*)
with *not-index-free-ivar* [*OF nontrivial, of j*] **show** *False* **by** *simp*

qed

show *?thesis*

proof

assume $\text{eq}: \text{adjoin } j \ Q = M$
have $\text{ivar } j \in \text{ecarrier } M$ **using** *inC eq* **by** *simp*
with *notinM* **show** *False* ..

qed

qed

73.1 Every polynomial over the base field splits in a maximal extension

The point of maximality. Factor the lifted polynomial into irreducibles over M . A factor of degree two or more would let a root of *it* be adjoined at a fresh index — giving a strictly larger extension, which maximality forbids. So every factor is linear, and a product of linear factors splits.

Note where each earlier piece enters: *exists-fresh-index* supplies the index, *is-extension-adjoin* and *ext-le-adjoin* make the enlargement an extension above M , *adjoin-strictly-larger* contradicts maximality, and *Field.degree-one-splits* with *Field.splits-poly-prod* assembles the conclusion.

theorem *exists-root*:

assumes *max*: $\bigwedge L. \llbracket \text{is-extension } L; \text{ext-le } M \ L \rrbracket \implies L = M$
and $P: P \in \text{poly-carrier}$
shows $M.\text{splits } (\text{lift-poly } P)$

proof –

have $\text{liftc}: \text{lift-poly } P \in M.\text{poly-carrier}$ **by** (rule *lift-poly-M-carrier* [*OF P*])
show *?thesis*
proof (*cases lift-poly P = M.poly-zero*)
case *True*
then show *?thesis* **by** (*simp add: M.splits-zero*)

next

case *nz: False*
show *?thesis*

```

proof (cases  $M.poly-unit$  ( $lift-poly\ P$ ))
  case True
    then have  $lift-poly\ P = M.poly-const\ (lift-poly\ P\ 0)$ 
      using  $M.poly-unit-imp-const$  by blast
    moreover have  $lift-poly\ P\ 0 \in ecarrier\ M$ 
      using liftc by (rule  $M.poly-carrier-coeff-closed$ )
    ultimately show ?thesis using  $M.splits-const$  by metis
  next
    case nu: False
    obtain  $fs$  where  $firr: \bigwedge f. f \in set\ fs \implies M.poly-irreducible\ f$ 
      and  $prod: lift-poly\ P = M.poly-prod\ fs$ 
      using  $M.factorization-into-irreducibles$  [OF liftc nz nu] by blast
    have  $fc: f \in M.poly-carrier$  if  $f \in set\ fs$  for  $f$ 
      using firr [OF that] by (rule  $M.poly-irreducibleD-carrier$ )
    have  $Pnz: P \neq 0_P$ 
    proof
      assume  $P = 0_P$ 
      then have  $lift-poly\ P = M.poly-zero$ 
        by (simp add: lift-poly-def poly-zero-def M.poly-zero-def)
      with nz show False ..
    qed
    — No factor can have degree two or more.
    have  $deg1: M.degree\ f = 1$  if  $f \in set\ fs$  for  $f$ 
    proof (rule ccontr)
      assume  $M.degree\ f \neq 1$ 
      moreover have  $0 < M.degree\ f$ 
        using firr [OF f] by (rule  $M.poly-irreducible-degree-pos$ )
      ultimately have  $d2: 2 \leq M.degree\ f$  by simp
      obtain  $i$  where  $fresh: \bigwedge \mathcal{P}. \mathcal{P} \in ecarrier\ M \implies index-free\ \mathcal{P}\ (P, i)$ 
        using exists-fresh-index [OF is-ext P Pnz] by blast
      have  $dvd: M.poly-dvd\ f\ (lift-poly\ (fst\ (P, i)))$ 
        using prod M.poly-dvd-poly-prod [OF fc f] by simp
      have  $Pj: fst\ (P, i) \in poly-carrier$  using  $P$  by simp
      have is-extension (adjoin ( $P, i$ )  $f$ )
        by (rule is-extension-adjoin [OF fresh firr] [OF f] dvd Pj)
      moreover have ext-le  $M$  (adjoin ( $P, i$ )  $f$ )
        by (rule ext-le-adjoin [OF fresh firr] [OF f])
      ultimately have adjoin ( $P, i$ )  $f = M$  by (rule max)
      moreover have adjoin ( $P, i$ )  $f \neq M$ 
        by (rule adjoin-strictly-larger [OF fresh firr] [OF f] d2)
      ultimately show False by simp
    qed
    have  $M.splits\ f$  if  $f \in set\ fs$  for  $f$ 
      by (rule  $M.degree-one-splits$  [OF fc] [OF that] deg1 [OF that])
    then show ?thesis using prod M.splits-poly-prod by simp
  qed
qed
qed
end

```

74 An Extension in which Every Base Polynomial Splits

context *Field*
begin

Zorn gave a maximal extension; maximality makes every polynomial over R split in it. Putting the two together is the whole of this step, exactly as it is in HOL-Algebra.

theorem *exists-extension-with-roots:*

obtains M **where** *is-extension* M
and $\bigwedge P. P \in \text{poly-carrier}$
 $\implies \text{Field.splits } (\text{ecarrier } M) (\text{eadd } M) (\text{emult } M) \mathbf{0}_I \mathbf{1}_I (\text{lift-poly } P)$

proof –

obtain M **where** *Mext*: $M \in \text{extensions}$
and *Mmax*: $\forall L \in \text{extensions}. \text{ext-le } M L \longrightarrow L = M$
using *exists-maximal-extension* **by** *blast*
have *ext*: *is-extension* M **using** *Mext* **by** *simp*
interpret *FE*: *field-extension* R *addition multiplication zero unit* M
by *unfold-locales* (*rule ext*)
have *max*: $\bigwedge L. [\text{is-extension } L; \text{ext-le } M L] \implies L = M$
using *Mmax* **by** *simp*
show *thesis*
proof (*rule that* [*OF ext*])
fix P **assume** $P: P \in \text{poly-carrier}$
show $\text{Field.splits } (\text{ecarrier } M) (\text{eadd } M) (\text{emult } M) \mathbf{0}_I \mathbf{1}_I (\text{lift-poly } P)$
by (*rule FE.exists-root* [*OF max P*])

qed

qed

end

end

75 Linking the carrier-set polynomial ring to Isabelle's polynomial type

theory *Poly-Typeclass*

imports *Poly-Ring Ring-Theory HOL-Computational-Algebra.Polynomial*

begin

Poly-Ring represents a polynomial as a finite-support function $\text{nat} \Rightarrow 'a$ with coefficients in a carrier R , which is exactly the representing set of HOL's *'a poly* once the carrier is the whole type. So for a type-class field the two developments describe the same objects, and the dictionary between them is little more than *Abs-poly*: this theory writes it out, so that the ring-theoretic results proved for *poly-carrier* — irreducibility, division, unique factorisation — become available

to the extension and Galois theory, which is phrased throughout in terms of $'a$ *poly*.

75.1 Finite composites are sums and products

The carrier-set *commutative-monoid.fincomp* at the whole type is HOL's *sum* or *prod*; the guard *commutative-monoid.M-ify* in its definition is vacuous there.

lemma *fincomp-TC-sum*:

commutative-monoid.fincomp (*UNIV* :: $'a$:: *comm-monoid-add set*) (+) 0 *f A* =
sum f A

proof –

interpret *M*: *commutative-monoid UNIV* :: $'a$ *set* (+) 0

by *unfold-locales* (*auto simp: ac-simps*)

show *?thesis*

by (*induct A rule: infinite-finite-induct*) *auto*

qed

lemma *fincomp-TC-prod*:

commutative-monoid.fincomp (*UNIV* :: $'a$:: *comm-monoid-mult set*) (*) 1 *f A* =
prod f A

proof –

interpret *M*: *commutative-monoid UNIV* :: $'a$ *set* (*) 1

by *unfold-locales* (*auto simp: ac-simps*)

show *?thesis*

by (*induct A rule: infinite-finite-induct*) *auto*

qed

75.2 The two representations of a polynomial

A finite-support function read as a polynomial. Off the finite-support functions the value is arbitrary; 0 is chosen so that the map is total.

definition *poly-of-fun* :: (*nat* $\Rightarrow$ $'a$:: *zero*) $\Rightarrow$ $'a$ *poly*

where *poly-of-fun p* $\equiv$ (*if finite* {*i. p i* $\neq$ 0} *then Abs-poly p* *else 0*)

lemma *coeff-poly-of-fun [simp]*:

finite {*i. p i* $\neq$ 0} $\Longrightarrow$ *Polynomial.coeff* (*poly-of-fun p*) = *p*

by (*simp add: poly-of-fun-def Abs-poly-inverse MOST-iff-cofinite*)

lemma *finite-support-coeff [simp, intro]*: *finite* {*i. Polynomial.coeff p i* $\neq$ 0}

using *MOST-coeff-eq-0 [of p]* **by** (*simp add: MOST-iff-cofinite*)

lemma *poly-of-fun-coeff [simp]*: *poly-of-fun* (*Polynomial.coeff p*) = *p*

by (*simp add: poly-of-fun-def coeff-inverse*)

lemma *inj-on-poly-of-fun*: *inj-on poly-of-fun* {*p. finite* {*i. p i* $\neq$ 0}}

by (*rule inj-onI*) (*metis coeff-poly-of-fun mem-Collect-eq*)

75.3 The dictionary at a type-class field

Fixing the carrier to be the whole type turns every constant of *Poly-Ring* into an operation on finite-support functions over a type-class field; the locale below exists only to make those constants available unqualified.

locale *poly-TC* = *Field UNIV* :: 'a :: field set (+) (*) 0 1

interpretation *poly-TC*
by (*simp add: poly-TC-def field-TC.Field-axioms*)

context *poly-TC*
begin

Membership of the carrier is finite support alone: every coefficient lies in the carrier because the carrier is everything.

lemma *poly-carrier-TC*: *poly-carrier* = {*p* :: nat $\Rightarrow$ 'a. finite {*i*. *p* *i* $\neq$ 0}}
by (*auto simp: poly-carrier-def*)

lemma *poly-carrier-iff*: *p* $\in$ *poly-carrier* $\longleftrightarrow$ finite {*i*. *p* *i* $\neq$ 0}
by (*simp add: poly-carrier-TC*)

lemma *coeff-in-poly-carrier* [*simp*, *intro*]: *Polynomial.coeff* *q* $\in$ *poly-carrier*
by (*simp add: poly-carrier-iff*)

lemma *coeff-poly-of-fun-carrier* [*simp*]:
p $\in$ *poly-carrier* $\implies$ *Polynomial.coeff* (*poly-of-fun* *p*) = *p*
by (*simp add: poly-carrier-iff*)

The additive inverse supplied by the group structure is HOL's, though the two are distinct constants and print alike; without this the coefficientwise proofs below cannot close.

lemma *additive-inverse-TC* [*simp*]: *additive.inverse* *a* = *uminus* *a*
by (*intro additive.inverse-equality simp-all*)

On the carrier the dictionary is injective, so an identity between carrier-set polynomials may be checked in 'a *poly* and conversely.

lemma *poly-of-fun-eq-iff* [*simp*]:
 $\llbracket p \in \text{poly-carrier}; q \in \text{poly-carrier} \rrbracket \implies \text{poly-of-fun } p = \text{poly-of-fun } q \longleftrightarrow p = q$
by (*metis coeff-poly-of-fun-carrier*)

75.4 The ring operations correspond

lemma *poly-of-fun-zero* [*simp*]: *poly-of-fun* *poly-zero* = 0
by (*simp add: poly-zero-def poly-of-fun-def zero-poly-def*)

lemma *poly-of-fun-one* [*simp*]: *poly-of-fun* *poly-one* = 1
by (*intro poly-eql*) (*simp add: poly-one-def*)

lemma *poly-of-fun-add* [simp]:
assumes $p \in \text{poly-carrier}$ **and** $q \in \text{poly-carrier}$
shows $\text{poly-of-fun } (\text{poly-add } p \ q) = \text{poly-of-fun } p + \text{poly-of-fun } q$
using *assms poly-add-closed*
by (*intro poly-eqI*) (*simp add: poly-add-def*)

lemma *poly-of-fun-neg* [simp]:
assumes $p \in \text{poly-carrier}$
shows $\text{poly-of-fun } (\text{poly-neg } p) = - \text{poly-of-fun } p$
using *assms poly-neg-closed*
by (*intro poly-eqI*) (*simp add: poly-neg-def*)

The convolution defining *poly-mult* is literally HOL's *coeff-mult*, once the finite composite is read as a sum.

lemma *poly-of-fun-mult* [simp]:
assumes $p \in \text{poly-carrier}$ **and** $q \in \text{poly-carrier}$
shows $\text{poly-of-fun } (\text{poly-mult } p \ q) = \text{poly-of-fun } p * \text{poly-of-fun } q$
using *assms poly-mult-closed*
by (*intro poly-eqI*) (*simp add: poly-mult-def coeff-mult fincomp-TC-sum*)

75.5 Constants, the indeterminate, degree and evaluation

lemma *poly-of-fun-const* [simp]: $\text{poly-of-fun } (\text{poly-const } c) = [:c:]$
unfolding *poly-const-def*
by (*intro poly-eqI*) (*auto simp: poly-carrier-iff coeff-pCons split: nat.split*)

lemma *poly-of-fun-monom* [simp]: $\text{poly-of-fun } (\text{monom } c \ k) = \text{Polynomial.monom } c \ k$
using *local.monom-def* **by** (*intro poly-eqI*) *simp*

lemma *poly-of-fun-var* [simp]: $\text{poly-of-fun } \text{var} = [:0, 1:]$
by (*simp add: var-def monom-altdef*)

Both notions of degree are pinned by the same two inequalities: no coefficient above the degree, and a nonzero coefficient at it.

lemma *degree-poly-of-fun* [simp]:
assumes $p \in \text{poly-carrier}$
shows $\text{Polynomial.degree } (\text{poly-of-fun } p) = \text{degree } p$
proof –
have $c: \text{Polynomial.coeff } (\text{poly-of-fun } p) = p$ **using** p **by** *simp*
show *?thesis*
proof (*intro antisym*)
show $\text{Polynomial.degree } (\text{poly-of-fun } p) \leq \text{degree } p$
using $p \ c$ **by** (*intro degree-le*) (*auto dest: coeff-gt-degree*)
show $\text{degree } p \leq \text{Polynomial.degree } (\text{poly-of-fun } p)$
using c *coeff-degree-nonzero degree-le-iff p poly-zero-def* **by** (*metis le-degree*)
qed
qed

Powers in the multiplicative monoid of the type are ordinary powers.

lemma *rpow-TC* [*simp*]: $\text{rpow } a \ n = a \wedge n$
by (*induct* *n*) *simp-all*

Evaluation matches *poly-altdef*, the coefficientwise form of *poly*.

lemma *poly-poly-of-fun* [*simp*]:
assumes $p \in \text{poly-carrier}$
shows $\text{Polynomial.poly} (\text{poly-of-fun } p) \ x = \text{eval } x \ p$
using *assms* **by** (*simp add: poly-altdef eval-def fincomp-TC-sum*)

75.6 Splitting into linear factors

lemma *poly-of-fun-root-factor* [*simp*]: $\text{poly-of-fun} (\text{root-factor } a) = [\text{:uminus } a, 1:]$
unfolding *root-factor-def*
by (*simp add: var-closed poly-const-closed poly-neg-closed*)

lemma *poly-of-fun-poly-prod*:
assumes $\bigwedge f. f \in \text{set } fs \implies f \in \text{poly-carrier}$
shows $\text{poly-of-fun} (\text{poly-prod } fs) = (\prod f \leftarrow fs. \text{poly-of-fun } f)$
using *assms* **by** (*induct* *fs*) (*auto simp: poly-prod-closed*)

The witnessing product, translated once for both directions of the theorem below.

lemma *poly-of-fun-split-form*:
 $\text{poly-of-fun} (\text{poly-mult} (\text{poly-const } c) (\text{poly-prod} (\text{List.map } \text{root-factor } as)))$
 $= [\text{:c:}] * (\prod a \leftarrow as. [\text{:uminus } a, 1:])$
proof –
have $\text{rf: } \bigwedge f. f \in \text{set} (\text{List.map } \text{root-factor } as) \implies f \in \text{poly-carrier}$
by (*auto intro: root-factor-closed*)
have $\text{poly-of-fun} (\text{poly-mult} (\text{poly-const } c) (\text{poly-prod} (\text{List.map } \text{root-factor } as)))$
 $= [\text{:c:}] * \text{poly-of-fun} (\text{poly-prod} (\text{List.map } \text{root-factor } as))$
using *rf* *poly-of-fun-mult* [*OF poly-const-closed poly-prod-closed*] **by** *fastforce*
also have $\text{poly-of-fun} (\text{poly-prod} (\text{List.map } \text{root-factor } as)) = (\prod f \leftarrow \text{List.map } \text{root-factor}$
 $as. \text{poly-of-fun } f)$
using *rf* **by** (*rule poly-of-fun-poly-prod*)
also have $\dots = (\prod a \leftarrow as. [\text{:uminus } a, 1:])$
by (*induct* *as*) *simp-all*
finally show *?thesis* .
qed

lemma *poly-mult-const-prod-carrier*: $\text{poly-mult} (\text{poly-const } c) (\text{poly-prod} (\text{List.map } \text{root-factor } as)) \in \text{poly-carrier}$
by (*metis splitsI splits-closed UNIV-I top-greatest*)

splits says exactly that the corresponding polynomial of '*a poly*' is a constant times a product of linear factors. Both directions are the dictionary applied to the defining equation, the reverse using injectivity on the carrier.

theorem *splits-iff-linear-factors*:
assumes $p \in \text{poly-carrier}$
shows $\text{splits } p \longleftrightarrow (\exists c \ as. \text{poly-of-fun } p = [\text{:c:}] * (\prod a \leftarrow as. [\text{:uminus } a, 1:]))$

```

proof
  assume splits p
  then show  $\exists c \text{ as. } \text{poly-of-fun } p = [:c:] * (\prod a \leftarrow \text{as. } [:u\text{minus } a, 1:])$ 
    by (force simp: splits-def poly-of-fun-split-form)
next
  assume  $\exists c \text{ as. } \text{poly-of-fun } p = [:c:] * (\prod a \leftarrow \text{as. } [:u\text{minus } a, 1:])$ 
  then obtain c as where q: poly-of-fun p = [:c:] * (\prod a \leftarrow as. [:u\text{minus } a, 1:]) by
    blast
  have  $\text{poly-of-fun } p = \text{poly-of-fun } (\text{poly-mult } (\text{poly-const } c) (\text{poly-prod } (\text{List.map}$ 
     $\text{root-factor as})))$ 
  by (simp add: q poly-of-fun-split-form)
  with p poly-mult-const-prod-carrier show splits p
  by (force simp: splits-def)
qed

```

A splitting polynomial of positive degree has a root: the factor list cannot be empty.

theorem *splits-imp-root*:

assumes *p: p* $\in$ *poly-carrier* **and** *sp: splits p* **and** *d: 0 < degree p*
obtains *a* **where** *eval a p = 0*

proof –

from *sp* **obtain** *c as* **where** *q: poly-of-fun p = [:c:] * (\prod a \leftarrow as. [:u\text{minus } a, 1:])*
using *splits-iff-linear-factors [OF p]* **by** *blast*
with *assms degree-poly-of-fun[of p]* **have** *as* $\neq []$
by (*auto split: if-splits*)
then obtain *a bs* **where** *as: as = a # bs*
by (*metis list.exhaust*)
have *Polynomial.poly (poly-of-fun p) a = 0*
unfolding *q as* **by** *simp*
with *p* **that** **show** *thesis* **by** *simp*

qed

75.7 The dictionary read backwards

Every polynomial of '*a poly*' is the reading of its own coefficient function, so each correspondence above serves in the other direction too.

lemma *eval-coeff [simp]: eval x (Polynomial.coeff q) = Polynomial.poly q x*
by (*metis coeff-in-poly-carrier poly-of-fun-coeff poly-poly-of-fun*)

lemma *splits-coeff-iff*:

*splits (Polynomial.coeff q) $\longleftrightarrow$ ($\exists c \text{ as. } q = [:c:] * (\prod a \leftarrow \text{as. } [:u\text{minus } a, 1:])$)*
using *splits-iff-linear-factors [OF coeff-in-poly-carrier]* **by** *simp*

corollary *splits-coeff-imp-root*:

assumes *splits (Polynomial.coeff q)* **and** $0 < \text{Polynomial.degree } q$
obtains *a* **where** *Polynomial.poly q a = 0*
by (*metis assms coeff-in-poly-carrier degree-poly-of-fun eval-coeff poly-of-fun-coeff splits-imp-root*)

end

Outside the locale the carrier-set degree must be named in full, its only parameter being the zero of the ring; that also makes it usable as a simplification rule on ordinary polynomials.

```
lemma degree-coeff [simp]:
  fixes q :: 'a :: field poly
  shows Ring.degree 0 (Polynomial.coeff q) = Polynomial.degree q
  by (metis coeff-in-poly-carrier degree-poly-of-fun poly-of-fun-coeff)
```

end

76 Degree of a simple algebraic extension

```
theory Extension-Degree
  imports Simple-Extension Extension-Vector-Space
begin
```

This theory begins the *field side* of phase R2. For an element a algebraic over a subfield K , it turns the *predicate* *is-minpoly* into a genuine *function* *minpoly* $K a$ (the minimal polynomial is unique) and defines the degree of the simple extension $K a$ over K as *degree* (*minpoly* $K a$).

76.1 Uniqueness of the minimal polynomial

```
context Subfield
begin
```

The minimal polynomial is unique: two minimal polynomials of a over K divide each other (each is a K -annihilator of a , so a multiple of the other by *minpoly-dvd*), and being monic they must be equal.

```
lemma minpoly-unique:
  assumes m1: is-minpoly K a m1 and m2: is-minpoly K a m2
  shows m1 = m2
proof -
  have k1: m1 ∈ poly-over K lead-coeff m1 = 1 poly m1 a = 0
    using m1 by (auto simp: is-minpoly-def)
  have k2: m2 ∈ poly-over K lead-coeff m2 = 1 poly m2 a = 0
    using m2 by (auto simp: is-minpoly-def)
  have d12: m1 dvd m2 using minpoly-dvd[OF m1 k2(1) k2(3)] .
  have d21: m2 dvd m1 using minpoly-dvd[OF m2 k1(1) k1(3)] .
  from d12 obtain u where u: m2 = m1 * u ..
  from d21 obtain v where v: m1 = m2 * v ..
  have m1nz: m1 ≠ 0 using k1(2) by auto
  have m1 = m2 * v by (rule v)
  also have ... = m1 * (u * v) using u by (simp add: mult.assoc)
  finally have uv: u * v = 1 using m1nz by simp
  have unv: u ≠ 0 using uv by auto
```

```

obtain  $c$  where  $uc: u = [:c:]$ 
  by (metis dvd-triv-left is-unit-poly-iff uv)
have  $lead-coeff\ m2 = lead-coeff\ m1 * lead-coeff\ u$  using  $u$  by (simp add: lead-coeff-mult)
then have  $c = 1$  using  $k1(2)\ k2(2)\ uc$  by simp
then show ?thesis using  $u\ uc$  by simp
qed

end

```

76.2 The minimal polynomial as a function

The minimal polynomial of a over K . For a algebraic over K this is the unique monic polynomial of least degree over K vanishing at a ; otherwise it is an unspecified default.

definition $minpoly :: 'a :: field\ set \Rightarrow 'a \Rightarrow 'a\ poly$
where $minpoly\ K\ a = (SOME\ m.\ is-minpoly\ K\ a\ m)$

The degree of the simple extension $K\ a$ over K .

definition $ext-degree :: 'a :: field\ set \Rightarrow 'a \Rightarrow nat$
where $ext-degree\ K\ a = degree\ (minpoly\ K\ a)$

context *Subfield*
begin

For an algebraic element, $minpoly$ genuinely is a minimal polynomial.

lemma *is-minpoly-minpoly*:
assumes *algebraic-over* $K\ a$
shows $is-minpoly\ K\ a\ (minpoly\ K\ a)$
unfolding *minpoly-def* **by** (*rule someI-ex[OF minpoly-exists[OF assms]]*)

... and it is *the* minimal polynomial: any minimal polynomial equals it.

lemma *minpoly-eq*:
assumes *algebraic-over* $K\ a$ **and** $is-minpoly\ K\ a\ m$
shows $minpoly\ K\ a = m$
using *minpoly-unique*[*OF is-minpoly-minpoly*[*OF assms*(1)] *assms*(2)] .

lemma *minpoly-over*: $algebraic-over\ K\ a \Longrightarrow minpoly\ K\ a \in poly-over\ K$
using *is-minpoly-minpoly* **by** (*simp add: is-minpoly-def*)

lemma *minpoly-monic*: $algebraic-over\ K\ a \Longrightarrow lead-coeff\ (minpoly\ K\ a) = 1$
using *is-minpoly-minpoly* **by** (*simp add: is-minpoly-def*)

lemma *minpoly-root*: $algebraic-over\ K\ a \Longrightarrow poly\ (minpoly\ K\ a)\ a = 0$
using *is-minpoly-minpoly* **by** (*simp add: is-minpoly-def*)

lemma *minpoly-nonzero*:
assumes *algebraic-over* $K\ a$ **shows** $minpoly\ K\ a \neq 0$
using *assms minpoly-monic* **by** *fastforce*

76.3 The extension degree

The degree of a simple algebraic extension is positive.

```

lemma ext-degree-pos:
  assumes algebraic-over  $K$   $a$ 
  shows ext-degree  $K$   $a > 0$ 
  unfolding ext-degree-def using minpoly-degree-pos[OF is-minpoly-minpoly[OF assms]]
  .

```

For an element already in K , its minimal polynomial is the linear $[-a, 1:]$.

```

lemma is-minpoly-linear:
  assumes  $a: a \in K$ 
  shows is-minpoly  $K$   $a$   $[-a, 1:]$ 
  unfolding is-minpoly-def
proof (intro conjI ballI impI)
  show  $[-a, 1:] \in \text{poly-over } K$ 
    using  $a$  by (auto intro!: poly-over-pCons poly-over-1 uminus-closed)
  show lead-coeff  $[-a, 1:] = 1$  by simp
  show poly  $[-a, 1:]$   $a = 0$  by simp
  fix  $r$  assume  $r: r \in \text{poly-over } K$  and  $\text{rnz}: r \neq 0 \wedge \text{poly } r \ a = 0$ 
  show degree  $[-a, 1:] \leq \text{degree } r$ 
    by (meson divides-degree poly-eq-0-iff-dvd rnz)
qed

```

Consequently an element of K is algebraic and has extension degree 1.

```

lemma algebraic-over-self:
  assumes  $a \in K$ 
  shows algebraic-over  $K$   $a$ 
  using is-minpoly-linear[OF assms]
  by (auto simp: algebraic-over-def is-minpoly-def intro! bexI[of - [-a, 1:]])

```

```

lemma minpoly-in-field:
  assumes  $a \in K$ 
  shows minpoly  $K$   $a = [-a, 1:]$ 
  using minpoly-eq[OF algebraic-over-self[OF assms] is-minpoly-linear[OF assms]] .

```

```

lemma ext-degree-in-field:
  assumes  $a \in K$ 
  shows ext-degree  $K$   $a = 1$ 
  unfolding ext-degree-def by (simp add: minpoly-in-field[OF assms])

```

Degree one characterises membership. An algebraic element has extension degree 1 exactly when it already lies in K : a monic degree-one minimal polynomial is $[-c, 1:]$ with $c \in K$, and vanishing at a forces $a = -c \in K$.

```

lemma ext-degree-eq-1-iff:
  assumes algebraic-over  $K$   $a$ 
  shows ext-degree  $K$   $a = 1 \longleftrightarrow a \in K$ 
proof

```

```

assume ext-degree  $K\ a = 1$ 
then have deg: degree (minpoly  $K\ a$ ) = 1 by (simp add: ext-degree-def)
have monic: lead-coeff (minpoly  $K\ a$ ) = 1 using minpoly-monic[OF assms] .
have root: poly (minpoly  $K\ a$ )  $a = 0$  using minpoly-root[OF assms] .
— Expand the root condition coefficientwise: poly (minpoly  $K\ a$ )  $a$  is the sum over
{..1}, i.e. coeff (minpoly  $K\ a$ ) 0 + coeff (minpoly  $K\ a$ ) 1 *  $a$ .
have lc1: coeff (minpoly  $K\ a$ ) 1 = 1 using deg monic by simp
have c0K: coeff (minpoly  $K\ a$ ) 0  $\in K$ 
  by (simp add: assms minpoly-over poly-over-coeff)
have coeff (minpoly  $K\ a$ ) 0 +  $a = 0$ 
  using deg lc1 root by (simp add: poly-altdef)
then show  $a \in K$ 
  by (metis add-eq-0-iff c0K Subfield.uminus-closed Subfield-axioms)
next
  assume  $a \in K$ 
  then show ext-degree  $K\ a = 1$  by (rule ext-degree-in-field)
qed

```

poly-over is closed under finite sums (needed to build a polynomial from a coordinate function).

lemma *poly-over-sum*:
assumes $\bigwedge i. i \in A \implies f\ i \in \text{poly-over } K$
shows $(\sum i \in A. f\ i) \in \text{poly-over } K$
using *assms* **by** (*induction A rule: infinite-finite-induct*) (*auto intro: poly-over-add*)

76.4 The power basis of a simple algebraic extension

Spanning. Every element of the simple extension *eval-img* $K\ a$ is a K -linear combination of the powers 1, a , ..., $a^{\text{ext-degree } K\ a - 1}$. Given $y = \text{poly } p\ a$, divide p by the (monic) minimal polynomial *minpoly* $K\ a$; the quotient term vanishes at a (as *poly* (*minpoly* $K\ a$) $a = 0$), leaving a remainder of degree $< d$ whose coefficients supply the combination.

lemma *power-basis-span*:
assumes *alg*: *algebraic-over* $K\ a$ **and** *y*: $y \in \text{eval-img } K\ a$
shows $\exists c. (\forall i < \text{ext-degree } K\ a. c\ i \in K) \wedge y = (\sum i < \text{ext-degree } K\ a. c\ i * a^i)$

proof —

```

define m where  $m = \text{minpoly } K\ a$ 
define d where  $d = \text{ext-degree } K\ a$ 
have mK:  $m \in \text{poly-over } K$  using minpoly-over[OF alg] by (simp add: m-def)
have monic: lead-coeff  $m = 1$  using minpoly-monic[OF alg] by (simp add: m-def)
have mdeg: degree  $m = d$  by (simp add: d-def ext-degree-def m-def)
have ddeg: degree  $m > 0$  using ext-degree-pos[OF alg] mdeg by (simp add: d-def)
have mroot: poly  $m\ a = 0$  using minpoly-root[OF alg] by (simp add: m-def)
obtain p where  $p \in \text{poly-over } K$  and yp:  $y = \text{poly } p\ a$ 
  using eval-imgE[OF y] by blast
obtain q r where  $qr: q \in \text{poly-over } K\ r \in \text{poly-over } K$ 
   $p = q * m + r$  degree  $r < \text{degree } m$ 

```

```

    using poly-over-divmod-exists[OF mK monic ddeg p] by blast
  have rdeg: degree r < d using qr(4) mdeg by simp
  have y-r: y = poly r a
    using yp qr(3) mroot by simp
  have poly r a = ( $\sum i \leq \text{degree } r. \text{coeff } r \ i * a^i$ ) by (rule poly-altdef)
  also have ... = ( $\sum i < d. \text{coeff } r \ i * a^i$ )
    by (rule sum.mono-neutral-left) (use rdeg in ⟨auto simp: coeff-eq-0⟩)
  finally have y = ( $\sum i < d. \text{coeff } r \ i * a^i$ ) using y-r by simp
  moreover have  $\forall i < d. \text{coeff } r \ i \in K$  using qr(2) by (auto simp: poly-over-coeff)
  ultimately show ?thesis unfolding d-def by blast
qed

```

Independence. The powers $1, a, \dots, a^{\text{ext-degree } K \ a - 1}$ are K -linearly independent: a vanishing combination has all coefficients 0 . The combination is $\text{poly } r \ a$ for $r = (\sum i < d. \text{monom } (c \ i) \ i)$, a polynomial over K of degree $< d$; were it nonzero it would annihilate a below the minimal degree, so $r = 0$ and every coefficient $c \ i$ vanishes.

lemma *power-basis-indep:*

```

  assumes alg: algebraic-over K a
    and cK:  $\bigwedge i. i < \text{ext-degree } K \ a \implies c \ i \in K$ 
    and zero: ( $\sum i < \text{ext-degree } K \ a. c \ i * a^i$ ) = 0
  shows  $\forall i < \text{ext-degree } K \ a. c \ i = 0$ 
proof -
  define d where d = ext-degree K a
  have dpos: d > 0 using ext-degree-pos[OF alg] by (simp add: d-def)
  define r where r = ( $\sum i < d. \text{monom } (c \ i) \ i$ )
  — r lies over K, has degree below d, and annihilates a.
  have rK: r ∈ poly-over K
    unfolding r-def using cK
    by (auto intro!: poly-over-sum poly-over-monom simp: d-def)
  have degree r ≤ d - 1
    unfolding r-def
    by (rule degree-sum-le) (auto intro: order-trans[OF degree-monom-le])
  then have rdeg: degree r < d
    by (simp add: dpos order-le-less-trans)
  have coeff-r: coeff r j = c j if j < d for j
    using that by (simp add: coeff-sum r-def)
  have rroot: poly r a = 0
    using zero by (force simp add: poly-sum poly-monom r-def d-def)
  — Below the minimal degree a nonzero annihilator is impossible, so r = 0.
  have r0: r = 0
    using is-minpoly-minpoly[OF alg] rK rroot unfolding is-minpoly-def
    by (metis d-def ext-degree-def leD rdeg)
  show ?thesis
    using coeff-r d-def r0 by auto
qed

```

end

context *subfield-tower*
begin

For a simple algebraic extension, the vector-space dimension is the existing minimal-polynomial degree. The proof turns the power-basis spanning and independence results into the coordinate-map basis used by *vs.dimension*; the explicit equality is the API bridge between the two presentations of extension degree.

theorem *simple-extension-dimension-eq-ext-degree*:

assumes *alg*: algebraic-over K a **and** L -def: $L = \text{eval-img } K \ a$
shows $\text{vs.dimension} = \text{ext-degree } K \ a$

proof –

define d **where** $d = \text{ext-degree } K \ a$
define B **where** $B = (\lambda i. a \wedge i) \text{ ‘ } \{..<d\}$
have finB : *finite* B **by** (*simp add*: B -def)
have aL : $a \in L$
unfolding L -def **by** (*rule* *Subfield.eval-img-self*[*OF base.Subfield-axioms*])
have powL : $a \wedge i \in L$ **for** i
using aL **by** (*rule* *ext.power-closed*)
have BL : $B \subseteq L$ **using** powL **by** (*auto simp*: B -def)
have spanL : $L \subseteq \text{vs.span } B$

proof

fix y **assume** yL : $y \in L$
obtain c **where** cK : $\bigwedge i. i < d \implies c \ i \in K$
and yc : $y = (\sum i < d. c \ i * a \wedge i)$
using *Subfield.power-basis-span*[*OF base.Subfield-axioms alg*]
 yL **by** (*auto simp*: L -def d -def)
have sum-span :
 $(\sum i < d. c \ i * a \wedge i) \in \text{vs.span } ((\lambda i. a \wedge i) \text{ ‘ } \{..<d\})$
by (*rule* *sum-scale-in-span*) (*auto simp*: $cK \ \text{powL}$)
then show $y \in \text{vs.span } B$ **using** yc **by** (*simp add*: B -def)

qed

have mod-span : $\text{vs.mod.spanning } B$
by (*rule* *vs.mod.spanningI*[*OF BL*]) (*use spanL in blast*)
have spanB : $\text{vs.spanning } B$
by (*rule* *iffD2*[*OF vs.spanning-iff-mod-spanning*[*OF finB BL mod-span*])
have power-inj : *inj-on* $(\lambda i. a \wedge i) \text{ ‘ } \{..<d\}$
proof (*rule* *inj-onI*)

fix $i \ j$ **assume** i : $i \in \{..<d\}$ **and** j : $j \in \{..<d\}$
and eq : $a \wedge i = a \wedge j$
show $i = j$
proof (*rule* *ccontr*)
assume neq : $i \neq j$
define c **where** $c = (\lambda k. \text{if } k = i \text{ then } (1 :: 'a) \text{ else if } k = j \text{ then } - \ 1 \text{ else } 0)$
have neg-one : $-(1 :: 'a) \in K$
by (*rule* *base.uminus-closed*) (*rule* *base.one-closed*)
have cK : $\bigwedge k. k < d \implies c \ k \in K$
proof –

```

fix k assume kd: k < d
have h1: k = i ∨ k ≠ i by blast
then show c k ∈ K
proof (elim disjE)
  assume h1: k = i
  have c k = (1 :: 'a)
    unfolding c-def using h1 by simp
  then show ?thesis using base.one-closed by simp
next
  assume h1: k ≠ i
  have h2: k = j ∨ k ≠ j by blast
  then show ?thesis
  proof (elim disjE)
    assume h2: k = j
    have c k = -(1 :: 'a)
      unfolding c-def using h1 h2 by simp
    then show ?thesis using neg-one by simp
  next
    assume h2: k ≠ j
    have c k = (0 :: 'a)
      unfolding c-def using h1 h2 by simp
    then show ?thesis using base.zero-closed by simp
  qed
qed
qed
have sum0: (∑ k < d. c k * a ^ k) = 0
proof -
  have point:
    c k * a ^ k = (if k = i then a ^ k else if k = j then -(a ^ k) else 0) for k
    unfolding c-def
    by (simp split: if-split)
  have sum-eq:
    (∑ k < d. c k * a ^ k) =
    (∑ k < d. if k = i then a ^ k else if k = j then -(a ^ k) else 0)
    by (simp add: point)
  also have ... = a ^ i - a ^ j
    using i j neq by (simp add: sum.If-cases)
  also have ... = 0 using eq by simp
  finally show ?thesis .
qed
have cK': ⋀k. k < ext-degree K a ⟹ c k ∈ K
  using cK by (simp add: d-def)
have sum0': (∑ k < ext-degree K a. c k * a ^ k) = 0
  using sum0 by (simp add: d-def)
have all0: ∀ k < d. c k = 0
  unfolding d-def
  by (rule Subfield.power-basis-indep[OF base.Subfield-axioms alg cK' sum0'])
have ci0: c i = 0 using all0 i by blast
have ci1: c i = (1 :: 'a) unfolding c-def using neq by simp
show False using ci0 ci1 by simp

```

```

    qed
  qed
  have indB: vs.lin-indep B
    unfolding vs.lin-indep-def
  proof (intro conjI)
    show finite B by (rule finB)
    show  $B \subseteq L$  by (rule BL)
    show  $\forall c \in B \rightarrow_E K. \text{vs.lincomb } c \ B = 0 \longrightarrow$ 
       $(\forall v \in B. c \ v = 0)$ 
    proof (rule ballI, rule impI)
      fix c assume cB:  $c \in B \rightarrow_E K$  and zero:  $\text{vs.lincomb } c \ B = 0$ 
      have cK:  $\bigwedge v. v \in B \implies c \ v \in K$  using cB by auto
      have sum0:  $(\sum v \in B. c \ v * v) = 0$ 
        using vs.lincomb-eq-sum[OF finB BL cK] zero by simp
      have indexed:  $(\sum i < d. c \ (a \wedge i) * a \wedge i) = 0$ 
      proof -
        have reindexed:
           $(\sum v \in B. c \ v * v) = (\sum i < d. c \ (a \wedge i) * a \wedge i)$ 
        unfolding B-def
        by (subst sum.reindex[OF power-inj]) (simp add: o-def)
        then show ?thesis using sum0 by simp
      qed
      have cK':  $\bigwedge i. i < \text{ext-degree } K \ a \implies c \ (a \wedge i) \in K$ 
        using cK by (auto simp: B-def d-def)
      have indexed':  $(\sum i < \text{ext-degree } K \ a. c \ (a \wedge i) * a \wedge i) = 0$ 
        using indexed by (simp add: d-def)
      have all0:  $\forall i < d. c \ (a \wedge i) = 0$ 
        unfolding d-def
        by (rule Subfield.power-basis-indep[OF base.Subfield-axioms alg cK' indexed'])
      show  $\forall v \in B. c \ v = 0$ 
        using all0 by (auto simp: B-def)
    qed
  qed
  have basisB: vs.basis B by (rule vs.basisI[OF spanB indB])
  have cardB: card B = d
    unfolding B-def by (simp add: card-image[OF power-inj])
  show ?thesis
    using vs.dimension-eq-any-field[OF basisB] cardB by (simp add: d-def)
  qed
end

```

76.5 The tower law, for a two-step simple tower

For a two-step tower $K \subseteq K(a) \subseteq K(a)(b)$ the products $a^i b^j$ (with i below $\text{ext-degree } K \ a$ and j below $\text{ext-degree } (\text{eval-img } K \ a) \ b$) form a K -basis of $K(a)(b)$: the degrees *multiply*. We prove the two basis properties directly from the power bases of the two simple steps, so no dimension-uniqueness (Steinitz) lemma is needed.

Spanning. Every element of $K(a)(b)$ is a K -combination of the products $a^i b^j$. Expand along the power basis of $K(a)(b)$ over $K(a)$, then expand each $K(a)$ -coordinate along the power basis of $K(a)$ over K .

lemma *tower-power-basis-span:*

```

assumes  $K$ : Subfield  $K$ 
  and  $alg\text{-}a$ : algebraic-over  $K$   $a$ 
  and  $alg\text{-}b$ : algebraic-over (eval-imag  $K$   $a$ )  $b$ 
  and  $y$ :  $y \in eval\text{-}img$  (eval-imag  $K$   $a$ )  $b$ 
shows  $\exists c. (\forall i < ext\text{-}degree\ K\ a. \forall j < ext\text{-}degree\ (eval\text{-}img\ K\ a)\ b. c\ i\ j \in K) \wedge$ 
   $y = (\sum j < ext\text{-}degree\ (eval\text{-}img\ K\ a)\ b. \sum i < ext\text{-}degree\ K\ a. c\ i\ j * (a^i * b^j))$ 
proof -
  define  $E$  where  $E = eval\text{-}img\ K\ a$ 
  have  $subE$ : Subfield  $E$ 
    unfolding  $E\text{-}def$  by (rule Subfield.subfield-eval-imag[OF  $K\ alg\text{-}a$ ])
  have  $algbE$ : algebraic-over  $E$   $b$ 
    using  $alg\text{-}b$  by (simp add:  $E\text{-}def$ )
  have  $yE$ :  $y \in eval\text{-}img\ E\ b$  using  $y$  by (simp add:  $E\text{-}def$ )
  — Expand  $y$  along the power basis of  $E(b)$  over  $E$ .
  obtain  $c$  where  $cE$ :  $\bigwedge j. j < ext\text{-}degree\ E\ b \implies c\ j \in E$ 
    and  $yc$ :  $y = (\sum j < ext\text{-}degree\ E\ b. c\ j * b^j)$ 
    using Subfield.power-basis-span[OF  $subE\ algbE\ yE$ ] by auto
  — Each  $E$ -coordinate expands along the power basis of  $E$  over  $K$ .
  have  $Hall$ :  $\forall j \in \{..<ext\text{-}degree\ E\ b\}.$ 
     $\exists cc. (\forall i < ext\text{-}degree\ K\ a. cc\ i \in K) \wedge c\ j = (\sum i < ext\text{-}degree\ K\ a. cc\ i * a^i)$ 
  proof
    fix  $j$  assume  $j \in \{..<ext\text{-}degree\ E\ b\}$ 
    then have  $c\ j \in eval\text{-}img\ K\ a$  using  $cE$  by (simp add:  $E\text{-}def$ )
    then show  $\exists cc. (\forall i < ext\text{-}degree\ K\ a. cc\ i \in K) \wedge c\ j = (\sum i < ext\text{-}degree\ K\ a.$ 
     $cc\ i * a^i)$ 
    by (rule Subfield.power-basis-span[OF  $K\ alg\text{-}a$ ])
  qed
  then obtain  $CC$ 
    where  $CC$ :  $\forall j \in \{..<ext\text{-}degree\ E\ b\}.$ 
     $(\forall i < ext\text{-}degree\ K\ a. CC\ j\ i \in K) \wedge c\ j = (\sum i < ext\text{-}degree\ K\ a. CC\ j\ i * a^i)$ 
    by metis
  — Substitute and regroup into a double sum over the products  $a^i b^j$ .
  have  $y = (\sum j < ext\text{-}degree\ E\ b. c\ j * b^j)$  by (rule  $yc$ )
  also have  $\dots = (\sum j < ext\text{-}degree\ E\ b. (\sum i < ext\text{-}degree\ K\ a. CC\ j\ i * a^i) * b^j)$ 
  by (rule sum.cong[OF refl]) (use  $CC$  in auto)
  also have  $\dots = (\sum j < ext\text{-}degree\ E\ b. \sum i < ext\text{-}degree\ K\ a. CC\ j\ i * (a^i * b^j))$ 
  by (auto simp: sum-distrib-right mult.assoc)
  finally have  $yeq$ :  $y = (\sum j < ext\text{-}degree\ E\ b. \sum i < ext\text{-}degree\ K\ a. CC\ j\ i * (a^i * b^j))$ 
  show ?thesis

```

using *CC* by (force simp: *E-def* yeq)
qed

Independence. A vanishing K -combination of the products $a^i b^j$ is trivial. Factor b^j out of the inner sum: the coefficients of the b -powers are E -elements, so they vanish by independence over E ; each is itself a K -combination of the a -powers, so its coefficients vanish by independence over K .

lemma *tower-power-basis-indep*:

assumes K : Subfield K
 and *alg-a*: algebraic-over K a
 and *alg-b*: algebraic-over (eval-*img* K a) b
 and cK : $\bigwedge i j. i < \text{ext-degree } K \ a \implies j < \text{ext-degree } (\text{eval-} \text{img } K \ a) \ b \implies c \ i \ j \in K$
 and zero: $(\sum j < \text{ext-degree } (\text{eval-} \text{img } K \ a) \ b. \sum i < \text{ext-degree } K \ a. c \ i \ j * (a^i * b^j)) = 0$
 shows $\forall i < \text{ext-degree } K \ a. \forall j < \text{ext-degree } (\text{eval-} \text{img } K \ a) \ b. c \ i \ j = 0$
proof —
 define E where $E = \text{eval-} \text{img } K \ a$
 have *subE*: Subfield E **unfolding** *E-def* **by** (rule Subfield.subfield-eval-*img*[*OF* K *alg-a*])
 have *algbE*: algebraic-over E b **using** *alg-b* **by** (simp add: *E-def*)
 have *aE*: $a \in E$ **unfolding** *E-def* **by** (rule Subfield.eval-*img*-self[*OF* K])
 have cK' : $\bigwedge i j. i < \text{ext-degree } K \ a \implies j < \text{ext-degree } E \ b \implies c \ i \ j \in K$
 using cK **by** (simp add: *E-def*)
 define e where $e = (\lambda j. \sum i < \text{ext-degree } K \ a. c \ i \ j * a^i)$
 — Each $e \ j$ is an element of E (a K -combination of powers of a).
 have *eE*: $e \ j \in E$ **if** $j < \text{ext-degree } E \ b$ **for** j
unfolding *e-def*
proof (rule Subfield.sum-closed[*OF* *subE*])
 fix i **assume** $i \in \{.. < \text{ext-degree } K \ a\}$
then have $i < \text{ext-degree } K \ a$ **by** *simp*
 have $c \ i \ j \in K$ **using** cK' [*OF* $i \ j$].
then have $c \ i \ j \in E$ **unfolding** *E-def*
by (simp add: K Subfield.eval-*img*-base)
 with *aE subE* **show** $c \ i \ j * a^i \in E$
by (simp add: Subfield.mult-closed Subfield.power-closed)
qed
 — Rewrite the double sum by factoring b^j out of each inner sum.
 have $(\sum j < \text{ext-degree } E \ b. e \ j * b^j)$
 = $(\sum j < \text{ext-degree } E \ b. \sum i < \text{ext-degree } K \ a. c \ i \ j * (a^i * b^j))$
by (simp add: *e-def* mult.assoc sum-distrib-right)
then have *esum0*: $(\sum j < \text{ext-degree } E \ b. e \ j * b^j) = 0$
using zero **by** (simp add: *E-def*)
 — Independence over E kills every $e \ j$.
 have *ej0*: $\forall j < \text{ext-degree } E \ b. e \ j = 0$
using Subfield.power-basis-indep[*OF* *subE* *algbE* *eE* *esum0*].
 — Independence over K then kills every coefficient.
show ?thesis
proof (intro allI *impI*)

```

    fix i j assume i: i < ext-degree K a and j: j < ext-degree (eval-img K a) b
    then have e: e j = 0 using ej0 by (simp add: E-def)
    have mem:  $\bigwedge i. i < \text{ext-degree } K a \implies c\ i\ j \in K$ 
      using cK j by blast
    have  $\forall i < \text{ext-degree } K a. c\ i\ j = 0$ 
      using Subfield.power-basis-indep[OF K alg-a mem] e by (simp add: e-def)
    then show c i j = 0 using i by simp
  qed
qed

end

```

77 Finite field extensions and their degree

```

theory Finite-Extension
  imports Extension-Degree
begin

```

A finite extension is a finitely spanned field tower in the native set-based representation. The underlying vector space already supplies the basis and dimension machinery; this theory gives that machinery a field-theoretic name and keeps the extension degree independent of a choice of basis.

```

locale finite-subfield-tower = subfield-tower +
  assumes finite-basis:  $\exists B. \text{vs.basis } B$ 
begin

```

```

definition extension-degree :: nat
  where extension-degree  $\equiv \text{vs.dimension}$ 

```

```

lemma extension-degree-eq-dimension:
  extension-degree = vs.dimension
  by (simp add: extension-degree-def)

```

```

lemma extension-degree-eq-card-basis:
  assumes vs.basis B
  shows extension-degree = card B
  using vs.dimension-eq-any-field[OF assms]
  by (simp add: extension-degree-def)

```

```

lemma finite-basis-exists:  $\exists B. \text{vs.basis } B$ 
  by (rule finite-basis)

```

The degree is positive because the ambient field contains distinct zero and one. This proof does not require the carrier of the extension to be finite: finite-dimensionality is exactly the basis premise of the locale.

```

lemma extension-degree-pos: extension-degree > 0
proof -
  obtain B where B: vs.basis B

```

```

    using finite-basis by blast
  show ?thesis
proof (rule ccontr)
  assume  $\neg \text{extension-degree} > 0$ 
  then have degree0:  $\text{extension-degree} = 0$  by simp
  have cardB0:  $\text{card } B = 0$ 
    using extension-degree-eq-card-basis[OF B] degree0 by simp
  have Bempty-or-infinite:  $B = \{\} \vee \text{infinite } B$ 
    using cardB0 by (simp add: card-eq-0-iff)
  have Bempty:  $B = \{\}$ 
    using Bempty-or-infinite B by (simp add: vs.basis-def)
  obtain c where  $c: c \in \{\} \rightarrow_E K$  and one-eq:  $1 = \text{vs.lincomb } c \ \{\}$ 
    using vs.basis-spanning[OF B, unfolded Bempty] ext.one-closed
    unfolding vs.spanning-def by blast
  then show False by simp
qed
qed

```

A simple algebraic extension is finite, and its vector-space dimension is the degree of the minimal polynomial. The substantive basis argument lives in the extension-degree theory; this corollary is the stable finite-extension-facing entry point.

```

lemma extension-degree-eq-ext-degree:
  assumes alg: algebraic-over K a and L-def:  $L = \text{eval-img } K \ a$ 
  shows  $\text{extension-degree} = \text{ext-degree } K \ a$ 
  using simple-extension-dimension-eq-ext-degree[OF alg L-def]
  by (simp add: extension-degree-def)

```

```

lemma finite-extension-cardinality:
  assumes finite L
  shows  $\text{card } L = \text{card } K \wedge \text{extension-degree}$ 
  by (simp add: asms extension-degree-eq-dimension finite-subfield-tower-cardinality)

```

A finite-dimensional extension of a finite carrier field again has finite carrier. A basis identifies the extension bijectively with the finitely supported coordinate functions on that basis; when both the basis and coefficient carrier are finite, so is that function space.

```

lemma finite-carrier-of-finite-base:
  assumes finK: finite K
  shows finite L
proof -
  obtain B where basis: vs.basis B
    using finite-basis-exists by blast
  have fin-coords: finite (B  $\rightarrow_E$  K)
    using basis finK unfolding vs.basis-def by (meson finite-PiE)
  have image:  $(\lambda c. \text{vs.lincomb } c \ B) \ ` (B \rightarrow_E K) = L$ 
    using basis by (simp add: vs.basis-def bij-betw-def)
  show ?thesis
    using finite-imageI[OF fin-coords, of  $\lambda c. \text{vs.lincomb } c \ B$ ] image by simp

```

qed

Every vector-space basis is also a field-generating set. One inclusion is minimality of *generate-field*; the other expands an arbitrary vector in its basis coordinates.

```

lemma generate-field-basis:
  assumes basis: vs.basis B
  shows  $L = \text{generate-field } (K \cup B)$ 
proof –
  have spanning: vs.spanning B by (rule vs.basis-spanning[OF basis])
  have finB: finite B
    using spanning unfolding vs.spanning-def by blast
  have BL:  $B \subseteq L$ 
    using spanning unfolding vs.spanning-def by blast
  show ?thesis
proof
  show  $\text{generate-field } (K \cup B) \subseteq L$ 
    by (rule generate-field-least[OF ext.Subfield-axioms]) (use base-subset BL in auto)
  show  $L \subseteq \text{generate-field } (K \cup B)$ 
proof
  fix x assume xL:  $x \in L$ 
  obtain c where  $c: c \in B \rightarrow_E K \ x = \text{vs.lincomb } c \ B$ 
    using vs.basis-spanning[OF basis] xL unfolding vs.spanning-def by blast
  have cK:  $\bigwedge v. v \in B \implies c \ v \in K$  using c(1) by auto
  have lincomb-sum:  $\text{vs.lincomb } c \ B = (\sum_{v \in B.} c \ v * v)$ 
    by (rule vs.lincomb-eq-sum[OF finB BL cK])
  interpret G: Subfield generate-field  $(K \cup B)$  by (rule subfield-generate-field)
  have sumG:  $(\sum_{v \in B.} c \ v * v) \in \text{generate-field } (K \cup B)$ 
    by (simp add: G.mult-closed G.sum-closed cK generate-field.generate-field-base)
  show  $x \in \text{generate-field } (K \cup B)$ 
    using c(2) lincomb-sum sumG by simp
  qed
qed
qed
end

```

```

context subfield-tower
begin

```

Finite spanning is the representation-independent introduction rule for the finite-extension locale. It is deliberately phrased using the existing span API, so callers need not manufacture a basis merely to establish finite-dimensionality.

```

lemma finite-subfield-tower-of-finite-spanning:
  assumes finB: finite B and BL:  $B \subseteq L$  and span:  $L \subseteq \text{vs.span } B$ 
  shows finite-subfield-tower K L
proof –
  obtain C where  $C: C \subseteq B$  and basisC: vs.basis C
    using subfield-tower-basis-exists-of-finite-spanning[OF finB BL span] by blast

```

```

show ?thesis
  using basisC finite-subfield-tower-axioms.intro finite-subfield-tower-def subfield-tower-axioms
  by blast
qed

lemma finite-subfield-tower-iff-finite-spanning:
  finite-subfield-tower K L  $\longleftrightarrow$ 
  ( $\exists B. \text{finite } B \wedge B \subseteq L \wedge L \subseteq \text{vs.span } B$ )
proof
  assume T: finite-subfield-tower K L
  interpret T: finite-subfield-tower K L by (rule T)
  obtain B where B: vs.basis B using T.finite-basis by blast
  have finB: finite B and BL: B  $\subseteq$  L
    using B by (auto simp: vs.basis-def)
  have L  $\subseteq$  vs.span B
    using B BL finB vs.basis-spanning vs.mod.spanning-def vs.spanning-iff-mod-spanning
by blast
  then show  $\exists B. \text{finite } B \wedge B \subseteq L \wedge L \subseteq \text{vs.span } B$ 
    using finB BL by blast
next
  assume  $\exists B. \text{finite } B \wedge B \subseteq L \wedge L \subseteq \text{vs.span } B$ 
  then show finite-subfield-tower K L
    using finite-subfield-tower-of-finite-spanning by force
qed

end

```

77.1 Finite intermediate extensions

A subfield between the endpoints of a finite extension is finite-dimensional over the base. It is a vector subspace of the ambient finite-dimensional extension, so the subspace basis theorem supplies a finite basis.

```

theorem finite-subfield-tower-intermediate-left:
  fixes F E K :: 'a :: field set
  assumes T: finite-subfield-tower F K
    and sfE: Subfield E and FE: F  $\subseteq$  E and EK: E  $\subseteq$  K
  shows finite-subfield-tower F E
proof –
  interpret T: finite-subfield-tower F K by (rule T)
  interpret FE: subfield-tower F E
    by (simp add: FE T.base.Subfield-axioms sfE subfield-tower.intro subfield-tower-axioms.intro)
  have E-submodule: T.vs.mod.submodule E
    by (simp add: EK FE.vs.scale-closed T.vs.mod.submoduleI)
  interpret S: vector-subspace F (+) (*) 0 1 (+) 0 K (*) E
    by (simp add: E-submodule T.vs.Vector-Space-axioms vector-subspace-axioms-def
      vector-subspace-def)
  obtain B where B: T.vs.basis B using T.finite-basis by blast
  obtain A where basis-FE: FE.vs.basis A
    using S.subspace-basis-exists[OF B] by blast

```

```

show ?thesis
  unfolding finite-subfield-tower-def
  using FE.subfield-tower-axioms basis-FE finite-subfield-tower-axioms.intro
  by blast
qed

```

The upper part of the same tower is finite-dimensional as well. An ambient basis over the smaller base still spans after scalars are enlarged to the intermediate field.

```

theorem finite-subfield-tower-intermediate-right:
  fixes F E K :: 'a :: field set
  assumes T: finite-subfield-tower F K
    and sfE: Subfield E and FE: F ⊆ E and EK: E ⊆ K
  shows finite-subfield-tower E K
proof -
  interpret T: finite-subfield-tower F K by (rule T)
  interpret EK: subfield-tower E K
    using EK T.ext.Subfield-axioms sfE subfield-tower.intro subfield-tower-axioms-def
  by blast
  obtain B where B: T.vs.basis B using T.finite-basis by blast
  have finB: finite B and BK: B ⊆ K
    using B by (auto simp: T.vs.basis-def)
  have K-span: K ⊆ EK.vs.span B
  proof
    fix x assume xK: x ∈ K
    obtain c where c: c ∈ B →E F x = T.vs.lincomb c B
      using T.vs.basis-spanning[OF B] xK unfolding T.vs.spanning-def by blast
    have xsum: x = (∑ v∈B. c v * v)
      using c(2) T.vs.lincomb-eq-sum[OF finB BK] c by auto
    have (∑ v∈B. c v * id v) ∈ EK.vs.span (id ` B)
      using BK EK.sum-scale-in-span[of B id c] c FE by fastforce
    then show x ∈ EK.vs.span B using xsum by simp
  qed
  show ?thesis by (rule EK.finite-subfield-tower-of-finite-spanning[OF finB BK K-span])
qed
end

```

78 Finite-dimensional extensions and transitivity of algebraicity

```

theory Algebraic-Transitivity
  imports Finite-Extension
begin

```

With the vector-space bridge of *Extension-Vector-Space* in place, the Steinitz cardinality bound *Vector-Space.independent-le-span* yields the classical chain

- a finite-dimensional extension is *algebraic* (every element satisfies a polynomial: its powers cannot all be independent);
- *transitivity of algebraicity* — if a is algebraic over K and b is algebraic over the simple extension $K(a)$, then b is algebraic over K .

This is the last field-theoretic prerequisite of the quintic finish: it upgrades “a radical is algebraic over the immediate base” to “a radical is algebraic over the ground field”, so the normal closure can be taken as the splitting field of a single ground-field polynomial.

notation *plus* (**infixl** $\langle + \rangle$ 65)
notation *minus* (**infixl** $\langle - \rangle$ 65)
unbundle *uminus-syntax*

78.1 A finite-dimensional extension is algebraic

If the extension L of K is spanned over K by a finite set B , then every element of L is algebraic over K . For $y \in L$ the $\text{card } B + 1$ powers y^i (with $i \leq \text{card } B$) all lie in the span of B , so by the Steinitz bound they are linearly dependent: some nontrivial K -combination vanishes, i.e. a nonzero polynomial over K kills y .

lemma (**in** *subfield-tower*) *findim-imp-algebraic*:

assumes *finB*: *finite B* **and** *BL*: $B \subseteq L$ **and** *span*: $L \subseteq \text{vs.span } B$ **and** *y*: $y \in L$

shows *algebraic-over K y*

proof (*cases inj-on* $(\lambda i. y \wedge i) \{.. \text{card } B\}$)

case *False*

— Two distinct powers of y coincide: the nonzero polynomial $X^i - X^j$ (over the prime field, so over K) kills y .

then obtain $i \ j$ **where** ij : $i \in \{.. \text{card } B\} \ j \in \{.. \text{card } B\} \ i \neq j \ y \wedge i = y \wedge j$

by (*auto simp: inj-on-def*)

define p **where** $p = (\text{monom } 1 \ i - \text{monom } 1 \ j :: 'a \ \text{poly})$

have $p\text{-over}$: $p \in \text{poly-over } K$

unfolding $p\text{-def}$ **by** (*intro base.poly-over-diff base.poly-over-monom base.one-closed*)

have $p\text{-root}$: $\text{poly } p \ y = 0$

unfolding $p\text{-def}$ **using** $ij(4)$ **by** (*simp add: poly-monom*)

have $p\text{-nz}$: $p \neq 0$

unfolding $p\text{-def}$ **by** (*meson eq-iff-diff-eq-0 monom-eq-iff' one-neq-zero* $\langle i \neq j \rangle$)

show *?thesis* **unfolding** *algebraic-over-def* **using** $p\text{-over } p\text{-nz } p\text{-root}$ **by** *blast*

next

case *True*

define n **where** $n \equiv \text{card } B$

define P **where** $P \equiv (\lambda i. y \wedge i) \{..n\}$

have $\text{inj}n$: *inj-on* $(\lambda i. y \wedge i) \{..n\}$ **using** *True* **by** (*simp add: n-def*)

have $yp\text{-}L$: $\bigwedge i. y \wedge i \in L$ **using** y **by** (*intro ext.power-closed*)

have PL : $P \subseteq L$ **using** $yp\text{-}L$ **by** (*auto simp: P-def*)

have $\text{fin}P$: *finite P* **by** (*simp add: P-def*)

have $P\text{span}$: $P \subseteq \text{vs.span } B$ **using** $PL \ \text{span}$ **by** *blast*

— P has $n + 1$ elements but sits in $vs.span\ B$ with $card\ B = n$, so by the Steinitz bound it cannot be independent.

```

have cardP:  $card\ P = n + 1$  using injn by (simp add: P-def card-image)
have  $\neg vs.lin-indep\ P$ 
  using BL Pspan cardP finB n-def vs.independent-le-span by fastforce
— The dependence gives a nonzero  $K$ -combination of powers that vanishes.
then obtain c where cPiE:  $c \in P \rightarrow_E K$  and lc0:  $vs.lincomb\ c\ P = 0$ 
  and nz:  $\neg (\forall v \in P. c\ v = 0)$ 
  using finP PL by (auto simp: vs.lin-indep-def)
have cK:  $\bigwedge v. v \in P \implies c\ v \in K$  using cPiE by auto
have sum0:  $(\sum v \in P. c\ v * v) = 0$  using vs.lincomb-eq-sum[OF finP PL cK] lc0
by simp

```

— Re-index through the (injective) power map to a genuine polynomial sum.

```

define d where  $d \equiv (\lambda i. c\ (y \wedge i))$ 
have  $(\sum v \in P. c\ v * v) = (\sum i \leq n. c\ (y \wedge i) * y \wedge i)$ 
  unfolding P-def by (subst sum.reindex[OF injn]) (simp add: o-def)
then have sumd0:  $(\sum i \leq n. d\ i * y \wedge i) = 0$  using sum0 by (simp add: d-def)
define p where  $p \equiv (\sum i \leq n. monom\ (d\ i)\ i :: 'a\ poly)$ 
have dK:  $\bigwedge i. i \leq n \implies d\ i \in K$  using cK by (auto simp: d-def P-def)
have p-over:  $p \in poly-over\ K$ 
  unfolding p-def
  by (intro base.poly-over-sum base.poly-over-monom) (simp add: dK)
have p-root:  $poly\ p\ y = 0$ 
  unfolding p-def by (simp add: poly-sum poly-monom sumd0)
have p-nz:  $p \neq 0$ 
proof —
  from nz obtain v where  $v: v \in P\ c\ v \neq 0$  by blast
  then obtain i where  $i: i \leq n\ v = y \wedge i$  by (auto simp: P-def)
  have coeff p i = d i
    unfolding p-def using i by (simp add: coeff-sum coeff-monom)
  then show ?thesis
    using d-def i(2) v(2) by fastforce
qed
show ?thesis unfolding algebraic-over-def using p-over p-nz p-root by blast
qed

```

Finite-dimensionality makes every target element algebraic over the base. This bridge belongs with the finite-dimensional algebraicity theorem, so clients need not import primitive-element constructions merely to use it.

lemma (in finite-subfield-tower) finite-extension-algebraic:

```

  assumes aL:  $a \in L$ 
  shows algebraic-over  $K\ a$ 
proof —
  obtain B where  $B: vs.basis\ B$  using finite-basis-exists by blast
  have finB: finite  $B$  using B unfolding vs.basis-def by (rule conjunct1)
  have BL:  $B \subseteq L$ 
    using B vs.basis-def by blast
  have mod-span:  $vs.mod.spanning\ B$ 
    using vs.basis-spanning[OF B] by (simp add: vs.spanning-iff-mod-spanning[OF

```

```

finB BL])
  have spanL:  $L \subseteq \text{vs.span } B$ 
  using mod-span vs.mod.spanning-def by force
  show ?thesis using findim-imp-algebraic[OF finB BL spanL aL] .
qed

```

78.2 Transitivity of algebraicity

Transitivity. If a is algebraic over K and b is algebraic over the simple extension $\text{eval-img } K \ a$, then b is algebraic over K .

The two-step tower $K \subseteq K(a) \subseteq K(a)(b)$ is spanned over K by the finite set of products $a^i * b^j$ (*tower-power-basis-span*); so $K(a)(b)$ is a finite-dimensional extension of K , and b , being one of its elements, is algebraic over K by *subfield-tower.findim-imp-algebraic*.

```

theorem algebraic-over-transitive:
  assumes K: Subfield K
    and alg-a: algebraic-over K a
    and alg-b: algebraic-over (eval-img K a) b
  shows algebraic-over K b
proof -
  define E where  $E \equiv \text{eval-img } K \ a$ 
  define L where  $L \equiv \text{eval-img } E \ b$ 
  have subE: Subfield E unfolding E-def by (rule Subfield.subfield-eval-img[OF K alg-a])
  have algbE: algebraic-over E b using alg-b by (simp add: E-def)
  have subL: Subfield L unfolding L-def by (rule Subfield.subfield-eval-img[OF subE algbE])
  —  $K \subseteq L$ :  $K$  sits inside  $E = K(a)$  inside  $L = E(b)$ .
  interpret KL: subfield-tower K L
    using subfield-tower.intro[OF K subL]
    by (metis E-def K L-def subE Subfield.eval-img-base subfield-tower-axioms.intro subsetI)
  — The finite product set  $a^i * b^j$  spans  $L$  over  $K$ .
  define da where  $da \equiv \text{ext-degree } K \ a$ 
  define db where  $db \equiv \text{ext-degree } E \ b$ 
  define B where  $B \equiv (\lambda(j,i). a^i * b^j) \text{ ' } (\{..<db\} \times \{..<da\})$ 
  have finB: finite B by (simp add: B-def)
  have aE:  $a \in E$  unfolding E-def by (rule Subfield.eval-img-self[OF K])
  obtain aL:  $a \in L$  and bL:  $b \in L$ 
    using L-def aE subE Subfield.eval-img-base Subfield.eval-img-self by blast
  have prodL:  $\bigwedge i j. a^i * b^j \in L$ 
    using aL bL by (intro KL.ext.mult-closed KL.ext.power-closed)
  have BL:  $B \subseteq L$  using prodL by (auto simp: B-def)
  have spanL:  $L \subseteq \text{KL.vs.span } B$ 
proof
  fix y assume  $y \in L$ 
  then
  obtain c where  $cK: \bigwedge i j. i < da \implies j < db \implies c \ i \ j \in K$ 

```

and $y_{eq}: y = (\sum j < db. \sum i < da. c\ i\ j * (a \wedge i * b \wedge j))$
using *tower-power-basis-span*[*OF K alg-a alg-b*]
using *E-def L-def da-def db-def* **by** *blast*
 — Flatten the double sum over the index rectangle, then push it into the span of B .
then have $y_{eq}': y = (\sum ji \in \{..<db\} \times \{..<da\}. (\lambda(j,i). c\ i\ j)\ ji * (\lambda(j,i). a \wedge i * b \wedge j)\ ji)$
by (*force simp: sum.cartesian-product intro: sum.cong*)
then show $y \in KL.vs.span\ B$
unfolding $y_{eq}'\ B-def$
by (*intro KL.sum-scale-in-span*) (*auto simp: prodL cK*)
qed
show *algebraic-over K b* **using** *KL.findim-imp-algebraic*[*OF finB BL spanL bL*] .
qed

78.3 Algebraicity along a radical/simple tower

Every element of a simple algebraic extension *eval-img K a* is itself algebraic over K : the extension is spanned over K by the finite power basis a^i (*Subfield.power-basis-span*), so *subfield-tower.findim-imp-algebraic* applies.

theorem *simple-ext-algebraic*:
assumes K : *Subfield K* **and** $alg-a$: *algebraic-over K a* **and** y : $y \in eval-img\ K\ a$
shows *algebraic-over K y*
proof —
define L **where** $L \equiv eval-img\ K\ a$
define d **where** $d \equiv ext-degree\ K\ a$
have $subL$: *Subfield L* **unfolding** $L-def$ **by** (*rule Subfield.subfield-eval-img*[*OF K alg-a*])
have KL : $K \subseteq L$ **unfolding** $L-def$ **using** *Subfield.eval-img-base*[*OF K*] **by** *blast*
interpret KL : *subfield-tower K L* **by** (*rule subfield-tower.intro*[*OF K subL*]) (*unfold-locales, rule KL*)
 — The power basis $\{a^i \mid i < d\}$ spans L over K .
define B **where** $B = (\lambda i. a \wedge i) \text{ ' } \{..<d\}$
have $finB$: *finite B* **by** (*simp add: B-def*)
have aL : $a \in L$ **unfolding** $L-def$ **by** (*rule Subfield.eval-img-self*[*OF K*])
have $powL$: $\bigwedge i. a \wedge i \in L$ **using** aL **by** (*intro KL.ext.power-closed*)
have $spanL$: $L \subseteq KL.vs.span\ B$
proof
fix z **assume** $z \in L$
then obtain c **where** cK : $\bigwedge i. i < d \implies c\ i \in K$ **and** zc : $z = (\sum i < d. c\ i * a \wedge i)$
i) **using** *Subfield.power-basis-span*[*OF K alg-a*] **by** (*auto simp: L-def d-def*)
then show $z \in KL.vs.span\ B$
unfolding $B-def$
by (*metis* (*full-types*) *KL.sum-scale-in-span finite-lessThan lessThan-iff powL*)
qed
have BL : $B \subseteq L$ **using** $powL$ **by** (*auto simp: B-def*)
show *?thesis* **using** *KL.findim-imp-algebraic*[*OF finB BL spanL*] y **by** (*simp add: L-def*)

qed

Two successive simple algebraic extensions are algebraic over the original base. This interface is useful when proving closure of the algebraic elements under field operations: the operation is first placed in the two-step generated field, and this lemma supplies its algebraicity over the base.

```

lemma algebraic-over-two-step:
  fixes  $F :: 'a :: \text{field set}$  and  $a\ b\ y :: 'a$ 
  assumes  $\text{sfF}: \text{Subfield } F$ 
    and  $\text{alg-a}: \text{algebraic-over } F\ a$ 
    and  $\text{alg-b}: \text{algebraic-over } F\ b$ 
    and  $y: y \in \text{eval-img } (\text{eval-img } F\ a)\ b$ 
  shows  $\text{algebraic-over } F\ y$ 
proof –
  have  $\text{sfE}: \text{Subfield } (\text{eval-img } F\ a)$ 
    by (rule  $\text{Subfield.subfield-eval-img}[OF\ \text{sfF}\ \text{alg-a}]$ )
  have  $\text{FE}: F \subseteq \text{eval-img } F\ a$ 
    using  $\text{Subfield.eval-img-base}[OF\ \text{sfF}]$  by blast
  have  $\text{alg-bE}: \text{algebraic-over } (\text{eval-img } F\ a)\ b$ 
    by (rule  $\text{algebraic-over-mono}[OF\ \text{alg-b}\ \text{FE}]$ )
  have  $\text{alg-yE}: \text{algebraic-over } (\text{eval-img } F\ a)\ y$ 
    by (rule  $\text{simple-ext-algebraic}[OF\ \text{sfE}\ \text{alg-bE}\ y]$ )
  show ?thesis
    by (rule  $\text{algebraic-over-transitive}[OF\ \text{sfF}\ \text{alg-a}\ \text{alg-yE}]$ )
qed

```

definition $\text{algebraic-elements} :: 'a :: \text{field set} \Rightarrow 'a\ \text{set}$ **where**
 $\text{algebraic-elements } F = \{a. \text{algebraic-over } F\ a\}$

Algebraic elements over a subfield form a subfield. Addition and multiplication are handled through *algebraic-over-two-step*; negation and inversion already lie in the corresponding simple extension.

```

lemma subfield-algebraic-elements:
  fixes  $F :: 'a :: \text{field set}$ 
  assumes  $\text{sfF}: \text{Subfield } F$ 
  shows  $\text{Subfield } (\text{algebraic-elements } F)$ 
proof
  show  $0 \in \text{algebraic-elements } F\ 1 \in \text{algebraic-elements } F$ 
    using  $\text{Subfield.algebraic-over-self}\ \text{Subfield-def}\ \text{algebraic-elements-def}\ \text{sfF}$  by blast+
  next
  fix  $a\ b$ 
  assume  $a: a \in \text{algebraic-elements } F$  and  $b: b \in \text{algebraic-elements } F$ 
  have  $\text{alg-a}: \text{algebraic-over } F\ a$ 
    using  $a$  by (simp add:  $\text{algebraic-elements-def}$ )
  have  $\text{sfE}: \text{Subfield } (\text{eval-img } F\ a)$ 
    by (rule  $\text{Subfield.subfield-eval-img}[OF\ \text{sfF}\ \text{alg-a}]$ )
  have  $\text{aE}: a \in \text{eval-img } F\ a$ 
    by (rule  $\text{Subfield.eval-img-self}[OF\ \text{sfF}]$ )
  have  $\text{negE}: -a \in \text{eval-img } F\ a$ 

```

```

    using sfE aE by (rule Subfield.uminus-closed)
  show  $-a \in \text{algebraic-elements } F$ 
    using simple-ext-algebraic[OF sfF alg-a negE]
    by (simp add: algebraic-elements-def)
  have invE:  $\text{inverse } a \in \text{eval-img } F \ a$ 
    using sfE aE by (rule Subfield.inverse-closed)
  show  $\text{inverse } a \in \text{algebraic-elements } F$ 
    using simple-ext-algebraic[OF sfF alg-a invE]
    by (simp add: algebraic-elements-def)
  have aL:  $a \in \text{eval-img } (\text{eval-img } F \ a) \ b$ 
    using Subfield.eval-img-base[OF sfE aE] .
  have bL:  $b \in \text{eval-img } (\text{eval-img } F \ a) \ b$ 
    by (rule Subfield.eval-img-self[OF sfE])
  have sumL:  $a + b \in \text{eval-img } (\text{eval-img } F \ a) \ b$ 
    by (rule Subfield.eval-img-add[OF sfE aL bL])
  have alg-b:  $\text{algebraic-over } F \ b$ 
    using b by (simp add: algebraic-elements-def)
  show  $a + b \in \text{algebraic-elements } F$ 
    using algebraic-over-two-step[OF sfF alg-a alg-b sumL]
    by (simp add: algebraic-elements-def)
  have multL:  $a * b \in \text{eval-img } (\text{eval-img } F \ a) \ b$ 
    by (rule Subfield.eval-img-mult[OF sfE aL bL])
  show  $a * b \in \text{algebraic-elements } F$ 
    using algebraic-over-two-step[OF sfF alg-a alg-b multL]
    by (simp add: algebraic-elements-def)
qed

```

Generation by algebraic elements preserves algebraicity. The generated field is contained in the subfield of all elements algebraic over the base, so no representation of generated elements or basis choice is exposed to clients.

```

lemma generated-algebraic:
  fixes F A :: 'a :: field set
  assumes sfF: Subfield F
    and algA:  $\bigwedge a. a \in A \implies \text{algebraic-over } F \ a$ 
    and x:  $x \in \text{generate-field } (F \cup A)$ 
  shows  $\text{algebraic-over } F \ x$ 
proof -
  have FA:  $F \cup A \subseteq \text{algebraic-elements } F$ 
    by (simp add: Subfield.algebraic-over-self algA algebraic-elements-def sfF subsetI)
  have gen:  $\text{generate-field } (F \cup A) \subseteq \text{algebraic-elements } F$ 
    by (rule generate-field-least[OF subfield-algebraic-elements[OF sfF] FA])
  show ?thesis
    using x algebraic-elements-def gen by blast
qed
end

```

79 Generic properties of algebraic field extensions

```
theory Extension-Properties
imports Extension-Degree
begin
```

These are the representation-independent extension predicates used by the Galois development. A field extension is represented by nested set-based subfields of one ambient field, as in *Subfield*; no complex-specific subfield locale or algebraic-closure theorem is needed for the definitions.

A splitting field is presented by a nonzero polynomial over the base and the subfield generated by all of its roots. The normality predicate uses the same root-set presentation for every irreducible polynomial over the base, while separability is stated through minimal polynomials.

```
definition poly-root-set :: 'a :: field poly  $\Rightarrow$  'a set where
  poly-root-set p  $\equiv$  {z. poly p z = 0}
```

```
definition splitting-field ::
  'a :: field set  $\Rightarrow$  'a poly  $\Rightarrow$  'a set  $\Rightarrow$  bool where
  splitting-field F p K  $\equiv$ 
    p  $\in$  poly-over F  $\wedge$  p  $\neq$  0  $\wedge$  K = generate-field (F  $\cup$  poly-root-set p)
```

```
definition algebraic-extension :: 'a :: field set  $\Rightarrow$  'a set  $\Rightarrow$  bool where
  algebraic-extension K F  $\equiv$  ( $\forall$  a  $\in$  K. algebraic-over F a)
```

```
definition normal-extension :: 'a :: field set  $\Rightarrow$  'a set  $\Rightarrow$  bool where
  normal-extension K F  $\equiv$ 
    ( $\forall$  p a. p  $\in$  poly-over F  $\longrightarrow$  irreducible-over F p  $\longrightarrow$  a  $\in$  K  $\longrightarrow$ 
      poly p a = 0  $\longrightarrow$  poly-root-set p  $\subseteq$  K)
```

```
definition separable-extension :: 'a :: field set  $\Rightarrow$  'a set  $\Rightarrow$  bool where
  separable-extension K F  $\equiv$ 
    ( $\forall$  a  $\in$  K. algebraic-over F a  $\longrightarrow$  rsquarefree (minpoly F a))
```

```
lemma poly-root-set-iff [simp]:
  z  $\in$  poly-root-set p  $\longleftrightarrow$  poly p z = 0
by (simp add: poly-root-set-def)
```

```
lemma finite-poly-root-set:
  p  $\neq$  0  $\implies$  finite (poly-root-set p)
unfolding poly-root-set-def by (rule poly-roots-finite)
```

```
lemma splitting-fieldD:
  splitting-field F p K  $\implies$ 
    p  $\in$  poly-over F  $\wedge$  p  $\neq$  0  $\wedge$  K = generate-field (F  $\cup$  poly-root-set p)
by (simp add: splitting-field-def)
```

```

lemma
  assumes splitting-field  $F$   $p$   $K$ 
  shows splitting-field-subfield: Subfield  $K$ 
    and splitting-field-base-subset:  $F \subseteq K$ 
    and splitting-field-roots-subset: poly-root-set  $p \subseteq K$ 
  using assms splitting-fieldD by blast+

lemma algebraic-extensionI:
  assumes  $\bigwedge a. a \in K \implies \text{algebraic-over } F a$ 
  shows algebraic-extension  $K$   $F$ 
  using assms by (simp add: algebraic-extension-def)

lemma algebraic-extensionD:
  assumes algebraic-extension  $K$   $F$  and  $a \in K$ 
  shows algebraic-over  $F$   $a$ 
  using assms by (simp add: algebraic-extension-def)

lemma normal-extensionI:
  assumes
     $\bigwedge p a. [p \in \text{poly-over } F; \text{irreducible-over } F p; a \in K; \text{poly } p a = 0] \implies \text{poly-root-set}$ 
     $p \subseteq K$ 
  shows normal-extension  $K$   $F$ 
  using assms by (simp add: normal-extension-def)

lemma normal-extensionD:
  assumes normal-extension  $K$   $F$ 
    and  $p \in \text{poly-over } F$  irreducible-over  $F$   $p$   $a \in K$  poly  $p$   $a = 0$ 
  shows poly-root-set  $p \subseteq K$ 
  using assms by (auto simp: normal-extension-def)

lemma separable-extensionI:
  assumes  $\bigwedge a. a \in K \implies \text{algebraic-over } F a \implies \text{rsquarefree } (\text{minpoly } F a)$ 
  shows separable-extension  $K$   $F$ 
  using assms by (simp add: separable-extension-def)

lemma separable-extensionD:
  assumes separable-extension  $K$   $F$  and  $a \in K$  and algebraic-over  $F$   $a$ 
  shows rsquarefree (minpoly  $F$   $a$ )
  using assms by (auto simp: separable-extension-def)

A divisor of a root-squarefree polynomial is root-squarefree. This is the base-
change bridge used below: a minimal polynomial over a larger subfield divides
the minimal polynomial over the smaller one, so separability descends to the
larger base.

lemma rsquarefree-dvd:
  fixes  $p$   $q :: 'a :: \text{idom poly}$ 
  assumes dvd:  $p \text{ dvd } q$  and sq: rsquarefree  $q$ 
  shows rsquarefree  $p$ 
proof –

```

```

have q0: q ≠ 0 using sq by (simp add: rsquarefree-def)
have order-le: order x p ≤ order x q for x
  by (rule dvd-imp-order-le[OF q0 dvd])
have p ≠ 0 using dvd q0 by auto
with order-le sq show ?thesis
  unfolding rsquarefree-def
  by (metis One-nat-def bot-nat-0.extremum-unique le-SucE)
qed

```

Over an algebraically closed field, root-squarefreeness turns the multiset of polynomial roots into an ordinary set of the same cardinality.

```

lemma card-roots-eq-degree-alg-closed:
  fixes p :: 'a :: alg-closed-field poly
  assumes p0: p ≠ 0 and rsf: rsquarefree p
  shows card {x. poly p x = 0} = degree p
proof -
  obtain A where sizeA: size A = degree p
    and pA: p = smult (lead-coeff p) (∏ x∈#A. [:-x, 1:])
    using alg-closed-imp-factorization[OF p0] by blast
  have rootsA: roots p = A
  proof -
    have roots-prod: roots (∏ x∈#A. [:-x, 1:]) = (∑ x∈#A. roots [:-x, 1:])
    proof (induction A)
      case (add x A)
      have factor0: [:-x, 1:] ≠ (0 :: 'a poly) by simp
      have prod0: (∏ y∈#A. [:-y, 1:]) ≠ (0 :: 'a poly)
        by auto
      show ?case using add.IH roots-mult[OF factor0 prod0] by simp
    qed auto
    have roots p = roots (∏ x∈#A. [:-x, 1:])
      by (metis p0 pA roots-smult smult-0-left)
    then show ?thesis using roots-prod by simp
  qed
  have count-le: count (roots p) x ≤ 1 for x
    using rsf
  by (metis One-nat-def count-roots diff-Suc-Suc diff-is-0-eq rsquarefree-def zero-diff)
  have eq-mset: roots p = mset-set (set-mset (roots p))
    using rsf
  by (intro multiset-eqI) (metis count-mset-set' count-roots finite-set-mset not-in-iff
rsquarefree-def)
  have roots-set: set-mset (roots p) = {x. poly p x = 0}
    by (rule set-count-roots[OF p0])
  have card {x. poly p x = 0} = card (set-mset (roots p))
    using roots-set by simp
  also have ... = size (roots p)
    by (metis size-mset-set eq-mset)
  finally show ?thesis using rootsA sizeA by simp
qed

```

A root of the defining polynomial of a splitting field is algebraic over its base.


```

lemma splitting-field-root-algebraic:
  assumes split: splitting-field  $F$   $p$   $K$ 
    and  $r$ :  $r \in \text{poly-root-set } p$ 
  shows algebraic-over  $F$   $r$ 
  using algebraic-over-def assms(1) poly-root-set-iff  $r$  splitting-fieldD by blast

end

```

80 The tower law for finite field extensions

```

theory Finite-Extension-Tower
  imports Finite-Extension
begin

```

A pair of finite field extensions $K \subseteq L \subseteq M$ carries three native vector-space interpretations. This theory proves that a basis of L over K and a basis of M over L multiply to a basis of M over K . The resulting degree formula is independent of the chosen bases and is the reusable tower-law interface for later Galois work.

```

locale finite-subfield-tower-chain =
   $KL$ : finite-subfield-tower  $K$   $L$  +  $LM$ : finite-subfield-tower  $L$   $M$ 
  for  $K$   $L$   $M$  :: ' $a$  :: field set
begin

```

```

lemma subfield-tower-KM: subfield-tower  $K$   $M$ 
proof (rule subfield-tower.intro)
  show Subfield  $K$  by (rule KL.base.Subfield-axioms)
  show Subfield  $M$  by (rule LM.ext.Subfield-axioms)
  show subfield-tower-axioms  $K$   $M$ 
    by unfold-locales (rule subset-trans[OF KL.base-subset LM.base-subset])
qed

```

```

sublocale  $KM$ : subfield-tower  $K$   $M$ 
  by (rule subfield-tower-KM)

```

```

definition product-map :: ' $a \times 'a \Rightarrow 'a$ 
  where product-map  $z = \text{snd } z * \text{fst } z$ 

```

```

definition product-basis :: ' $a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$ 
  where product-basis  $B$   $C = \{b * c \mid b \in B \wedge c \in C\}$ 

```

```

lemma product-basis-image:
  product-basis  $B$   $C = \text{product-map } ' (C \times B)$ 
proof –
  have product-basis  $B$   $C = \{b * c \mid b \in B \wedge c \in C\}$ 
    by (simp add: product-basis-def)
  also have  $\dots = \text{product-map } ' (C \times B)$ 
proof (rule equalityI)
  show  $\{b * c \mid b \in B \wedge c \in C\} \subseteq \text{product-map } ' (C \times B)$ 

```

```

proof
  fix x
  assume x:  $x \in \{b * c \mid b \in B \wedge c \in C\}$ 
  then obtain b c where bB:  $b \in B$  and cC:  $c \in C$  and x:  $x = b * c$ 
    by blast
  have pair:  $(c, b) \in C \times B$  using cC bB by simp
  have img:  $\text{product-map } (c, b) \in \text{product-map } ' (C \times B)$ 
    by (rule imageI[OF pair])
  have bc:  $b * c \in \text{product-map } ' (C \times B)$ 
    using img by (simp add: product-map-def mult.commute)
  show  $x \in \text{product-map } ' (C \times B)$  using x bc by simp
qed
next
show  $\text{product-map } ' (C \times B) \subseteq \{b * c \mid b \in B \wedge c \in C\}$ 
proof
  fix x
  assume x:  $x \in \text{product-map } ' (C \times B)$ 
  then obtain z where z:  $z \in C \times B$  and x:  $x = \text{product-map } z$  by blast
  obtain c b where z:  $z = (c, b)$  and cC:  $c \in C$  and bB:  $b \in B$ 
    using z by (cases z) auto
  show  $x \in \{b * c \mid b \in B \wedge c \in C\}$ 
    using x z bB cC by (auto simp add: product-map-def mult.commute)
qed
qed
finally show ?thesis .
qed

lemma KL-basis-subset:
  assumes KL.vs.basis B
  shows  $B \subseteq L$ 
  using assms by (simp add: KL.vs.basis-def)

lemma LM-basis-subset:
  assumes LM.vs.basis C
  shows  $C \subseteq M$ 
  using assms by (simp add: LM.vs.basis-def)

lemma KM-product-map-inj:
  assumes B: KL.vs.basis B and C: LM.vs.basis C
  shows inj-on product-map  $(C \times B)$ 
proof (rule inj-onI)
  fix x y
  assume x:  $x \in C \times B$  and y:  $y \in C \times B$ 
  and eq:  $\text{product-map } x = \text{product-map } y$ 
  obtain c b where x':  $x = (c, b)$  and cC:  $c \in C$  and bB:  $b \in B$ 
    using x by (cases x) auto
  obtain c' b' where y':  $y = (c', b')$  and c'C:  $c' \in C$  and b'B:  $b' \in B$ 
    using y by (cases y) auto
  have BL:  $B \subseteq L$  by (rule KL-basis-subset[OF B])
  have bL:  $b \in L$  and b'L:  $b' \in L$  using BL bB b'B by blast+

```

```

have b0: b ≠ 0
proof
  assume b0: b = 0
  have bnot: b ∉ KL.vs.span (B - {b})
    using KL.vs.lin-indep-not-in-span[OF KL.vs.basis-lin-indep[OF B] bB] .
  have b ∈ KL.vs.span (B - {b}) using b0 by (simp add: KL.vs.span-zero)
  then show False using bnot by contradiction
qed
have c0: c ≠ 0
proof
  assume c0: c = 0
  have cnot: c ∉ LM.vs.span (C - {c})
    using LM.vs.lin-indep-not-in-span[OF LM.vs.basis-lin-indep[OF C] cC] .
  have c ∈ LM.vs.span (C - {c}) using c0 by (simp add: LM.vs.span-zero)
  then show False using cnot by contradiction
qed
have eq': b * c = b' * c'
  using eq x' y' by (simp add: product-map-def)
have c-eq: c = c'
proof (rule ccontr)
  assume cne: c ≠ c'
  have ibL: inverse b ∈ L
    using bL by (rule Subfield.inverse-closed[OF KL.ext.Subfield-axioms])
  have rL: inverse b * b' ∈ L
    using ibL b'L by (rule Subfield.mult-closed[OF KL.ext.Subfield-axioms])
  have eqc: c = (inverse b * b') * c'
proof -
  have c = inverse b * (b * c) using b0 by simp
  also have ... = inverse b * (b' * c') using eq' by simp
  also have ... = (inverse b * b') * c' by (simp add: mult.assoc)
  finally show ?thesis .
qed
have c'other: c' ∈ C - {c} using c'C cne by blast
have CsubM: C - {c} ⊆ M
  using LM-basis-subset[OF C] by blast
have c'span: c' ∈ LM.vs.span (C - {c})
  by (rule LM.vs.span-incl[OF CsubM c'other])
have rspan: (inverse b * b') * c' ∈ LM.vs.span (C - {c})
  by (rule LM.vs.span-scale[OF CsubM rL c'span])
have cspan: c ∈ LM.vs.span (C - {c}) using eqc rspan by simp
have cnot: c ∉ LM.vs.span (C - {c})
  using LM.vs.lin-indep-not-in-span[OF LM.vs.basis-lin-indep[OF C] cC] .
  then show False using cspan by contradiction
qed
have b-eq: b = b'
proof -
  have eqc': b * c = b' * c using eq' c-eq by simp
  have b = (b * c) * inverse c using c0 by (simp add: mult.assoc)
  also have ... = (b' * c) * inverse c using eqc' by simp
  also have ... = b' using c0 by (simp add: mult.assoc)

```

```

    finally show ?thesis .
qed
show  $x = y$  using  $x' y' b\text{-eq } c\text{-eq}$  by simp
qed

lemma KM-product-basis-card:
  assumes  $B: KL.\text{vs.basis } B$  and  $C: LM.\text{vs.basis } C$ 
  shows  $\text{card } (\text{product-basis } B \ C) = \text{card } B * \text{card } C$ 
  unfolding product-basis-image
  by (simp add: card-image[OF KM-product-map-inj[OF  $B \ C$ ]] card-cartesian-product
    mult.commute)

lemma KM-product-basis-subset:
  assumes  $B: KL.\text{vs.basis } B$  and  $C: LM.\text{vs.basis } C$ 
  shows  $\text{product-basis } B \ C \subseteq M$ 
proof
  fix  $x$  assume  $x \in \text{product-basis } B \ C$ 
  then obtain  $z$  where  $zP: z \in C \times B$  and  $xz: x = \text{product-map } z$ 
    unfolding product-basis-image by blast
  obtain  $c \ b$  where  $z: z = (c, b)$  and  $cC: c \in C$  and  $bB: b \in B$ 
    using  $zP$  by (cases  $z$ ) auto
  have  $x: x = b * c$  using  $xz \ z$  by (simp add: product-map-def)
  have  $cM: c \in M$  using LM-basis-subset[OF  $C$ ]  $cC$  by blast
  have  $bM: b \in M$ 
    using KL-basis-subset[OF  $B$ ]  $bB$  LM.ext.mult-closed LM.base-subset by blast
  show  $x \in M$  using  $x \ bM \ cM$  by (auto intro: LM.ext.mult-closed)
qed

lemma KM-product-basis-spanning:
  assumes  $B: KL.\text{vs.basis } B$  and  $C: LM.\text{vs.basis } C$ 
  shows  $M \subseteq KM.\text{vs.span } (\text{product-basis } B \ C)$ 
proof
  fix  $x$  assume  $xM: x \in M$ 
  have  $Cfin: \text{finite } C$  and  $CM: C \subseteq M$ 
    using  $C$  LM-basis-subset[OF  $C$ ] by (auto simp: LM.vs.basis-def)
  have  $Bfin: \text{finite } B$  and  $BL: B \subseteq L$ 
    using  $B$  KL-basis-subset[OF  $B$ ] by (auto simp: KL.vs.basis-def)
  obtain  $d$  where  $dC: d \in C \rightarrow_E L$  and  $xd: x = LM.\text{vs.lincomb } d \ C$ 
    using LM.vs.basis-spanning[OF  $C$ ]  $xM$  by (auto simp: LM.vs.spanning-def)
  define inner-sum ::
    ' $a$  set  $\Rightarrow$  (' $a \Rightarrow 'a \Rightarrow 'a$ )  $\Rightarrow$  ' $a \Rightarrow 'a$ 
    where inner-sum  $B \ e \ c = (\sum b \in B. e \ c \ b * (b * c))$  for  $B \ e \ c$ 
  define coefficient-sum ::
    ' $a$  set  $\Rightarrow$  (' $a \Rightarrow 'a \Rightarrow 'a$ )  $\Rightarrow$  ' $a \Rightarrow 'a$ 
    where coefficient-sum  $B \ f \ c = (\sum b \in B. f \ c \ b * b)$  for  $B \ f \ c$ 
  define outer-sum :: ' $a$  set  $\Rightarrow$  (' $a \Rightarrow 'a$ )  $\Rightarrow$  ' $a$ 
    where outer-sum  $C \ h = (\sum c \in C. h \ c)$  for  $C \ h$ 
  define scale-function :: (' $a \Rightarrow 'a$ )  $\Rightarrow$  (' $a \Rightarrow 'a$ )
    where scale-function  $d \ c = d \ c * c$  for  $d \ c$ 
  define product-sum :: (' $a \times 'a$ ) set  $\Rightarrow$  (' $a \times 'a \Rightarrow 'a$ )  $\Rightarrow$  ' $a$ 

```

```

    where product-sum  $S\ p = (\sum z \in S. p\ z)$  for  $S\ p$ 
  define product-term ::
    ('a  $\Rightarrow$  'a  $\Rightarrow$  'a)  $\Rightarrow$  'a  $\times$  'a  $\Rightarrow$  'a
    where product-term  $e\ z = e\ (fst\ z)\ (snd\ z) * product-map\ z$  for  $e\ z$ 
  define coeff :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a
    where coeff = ( $\lambda c. SOME\ f. f \in B \rightarrow_E K \wedge d\ c = KL.vs.lincomb\ f\ B$ )
  define h :: 'a  $\Rightarrow$  'a
    where h = inner-sum  $B\ coeff$ 
  define p :: 'a  $\times$  'a  $\Rightarrow$  'a
    where p = product-term coeff
  have coeff-spec-point:
    coeff  $c \in B \rightarrow_E K \wedge$ 
     $d\ c = KL.vs.lincomb\ (coeff\ c)\ B$ 
  if  $cC$ :  $c \in C$  for  $c$ 
  proof -
    have dcL:  $d\ c \in L$  using  $dC\ cC$  by (auto simp: PiE-def)
    have spans:
       $\forall v \in L. \exists f \in B \rightarrow_E K.$ 
       $v = KL.vs.lincomb\ f\ B$ 
    using  $KL.vs.basis-spanning[OF\ B]$ 
    by (simp add:  $KL.vs.spanning-def$ )
    have some-exists:
       $\exists f. f \in B \rightarrow_E K \wedge d\ c = KL.vs.lincomb\ f\ B$ 
    using spans dcL by blast
    show ?thesis
      unfolding coeff-def
      using some-exists by (rule someI-ex)
  qed
  have coeff-spec:
     $\forall c \in C. coeff\ c \in B \rightarrow_E K \wedge$ 
     $d\ c = KL.vs.lincomb\ (coeff\ c)\ B$ 
  using coeff-spec-point by blast
  have d-expands:  $d\ c = coefficient-sum\ B\ coeff\ c$ 
  if  $cC$ :  $c \in C$  for  $c$ 
  proof -
    have coeff-c: coeff  $c \in B \rightarrow_E K$ 
    using coeff-spec  $cC$  by blast
    have coeffK:  $\bigwedge b. b \in B \implies coeff\ c\ b \in K$ 
    using coeff-c by blast
    have lc:
       $KL.vs.lincomb\ (coeff\ c)\ B =$ 
       $(\sum b \in B. coeff\ c\ b * b)$ 
    using  $KL.vs.lincomb-eq-sum[OF\ Bfin\ BL\ coeffK]$  .
    show ?thesis
      using coeff-spec  $cC\ lc$  by (simp add: coefficient-sum-def)
  qed
  have xsum:  $x = (\sum c \in C. d\ c * c)$ 
  using  $xd\ LM.vs.lincomb-eq-sum[OF\ Cfin\ CM]\ dC$ 
  by (auto simp: PiE-def)
  have xdouble:  $x = outer-sum\ C\ h$ 

```

```

proof –
  have x-outer:  $x = \text{outer-sum } C \text{ (scale-function } d)$ 
    using xsum by (simp add: outer-sum-def scale-function-def)
  have h-outer:  $\text{outer-sum } C \text{ (scale-function } d) = \text{outer-sum } C \text{ } h$ 
    unfolding outer-sum-def
    by (rule sum.cong[OF refl])
    (simp add: h-def inner-sum-def coefficient-sum-def
      sum-distrib-right d-expands scale-function-def mult.assoc)
  show ?thesis by (rule trans[OF x-outer h-outer])
qed
have pair-span:
  ( $\sum z \in C \times B. p \ z$ )
   $\in KM.vs.span \text{ (product-basis } B \ C)$ 
proof –
  have pair-span':
    ( $\sum z \in C \times B.$ 
       $\text{coeff (fst } z) \text{ (snd } z) * \text{product-map } z$ )
     $\in KM.vs.span \text{ (image product-map } (C \times B))$ 
  proof (rule KM.sum-scale-in-span)
    show finite  $(C \times B)$  by (simp add: Bfin Cfin)
    show  $\text{product-map } z \in M$  if  $z: z \in C \times B$  for  $z$ 
    proof –
      have zP:  $\text{product-map } z \in \text{product-basis } B \ C$ 
        using  $z$  by (simp add: product-basis-image)
      show ?thesis
        using KM-product-basis-subset[OF B C] zP by blast
    qed
    show  $\text{coeff (fst } z) \text{ (snd } z) \in K$ 
      if  $z: z \in C \times B$  for  $z$ 
    proof –
      have cC:  $\text{fst } z \in C$  using  $z$  by (cases z) auto
      have bB:  $\text{snd } z \in B$  using  $z$  by (cases z) auto
      have coeff-c:  $\text{coeff (fst } z) \in B \rightarrow_E K$ 
        using coeff-spec cC by blast
      show ?thesis using coeff-c bB by (auto simp: PiE-def)
    qed
  show ?thesis
    using pair-span' by (simp add: p-def product-term-def product-basis-image)
qed
have pair-eq:  $\text{outer-sum } C \text{ } h = \text{product-sum } (C \times B) \text{ } p$ 
  unfolding outer-sum-def product-sum-def h-def p-def inner-sum-def product-term-def
  by (simp add: sum.cartesian-product product-map-def case-prod-unfold)
show  $x \in KM.vs.span \text{ (product-basis } B \ C)$ 
  using xdouble pair-eq pair-span by (simp add: product-sum-def)
qed

lemma KM-product-basis-condition:
assumes B: KL.vs.basis B and C: LM.vs.basis C
shows  $\forall f \in \text{product-basis } B \ C \rightarrow_E K.$ 

```

```

     $KM.vs.lincomb\ f\ (product-basis\ B\ C) = 0 \longrightarrow$ 
     $(\forall v \in product-basis\ B\ C. f\ v = 0)$ 
proof (rule ballI, rule impI)
  fix  $f$  assume  $fP: f \in product-basis\ B\ C \rightarrow_E K$ 
    and  $fzero: KM.vs.lincomb\ f\ (product-basis\ B\ C) = 0$ 
  define  $outer-sum :: 'a\ set \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a$ 
    where  $outer-sum\ C\ h = (\sum c \in C. h\ c)$  for  $C\ h$ 
  define  $scale-function :: ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a)$ 
    where  $scale-function\ d\ c = d\ c * c$  for  $d\ c$ 
  define  $pair-scale ::$ 
     $('a \Rightarrow 'a) \Rightarrow 'a \times 'a \Rightarrow 'a$ 
    where  $pair-scale\ f\ z = f\ (product-map\ z) * product-map\ z$  for  $f\ z$ 
  define  $fiber-function ::$ 
     $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ 
    where  $fiber-function\ f\ c\ b = f\ (b * c)$  for  $f\ c\ b$ 
  define  $fiber-sum ::$ 
     $('a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow 'a$ 
    where  $fiber-sum\ f\ B\ c = (\sum b \in B. f\ (b * c) * b)$  for  $f\ B\ c$ 
  define  $product-sum :: ('a \times 'a)\ set \Rightarrow ('a \times 'a \Rightarrow 'a) \Rightarrow 'a$ 
    where  $product-sum\ S\ p = (\sum z \in S. p\ z)$  for  $S\ p$ 
  have  $Bfin: finite\ B$  and  $BL: B \subseteq L$ 
    using  $B\ KL-basis-subset[OF\ B]$  by (auto simp:  $KL.vs.basis-def$ )
  have  $Cfin: finite\ C$  and  $CM: C \subseteq M$ 
    using  $C\ LM-basis-subset[OF\ C]$  by (auto simp:  $LM.vs.basis-def$ )
  have  $Pfin: finite\ (product-basis\ B\ C)$  by (simp add:  $product-basis-image\ Bfin\ Cfin$ )
  have  $PM: product-basis\ B\ C \subseteq M$  by (rule  $KM-product-basis-subset[OF\ B\ C]$ )
  have  $fK: f\ v \in K$  if  $v: v \in product-basis\ B\ C$  for  $v$ 
    using  $fP\ v$  by blast
  have  $sumzero:$ 
     $outer-sum\ (product-basis\ B\ C)\ (scale-function\ f) = 0$ 
  proof –
    have  $fK': \bigwedge v. v \in product-basis\ B\ C \implies f\ v \in K$ 
      using  $fP$  by blast
    have  $lincomb-sum:$ 
       $KM.vs.lincomb\ f\ (product-basis\ B\ C) =$ 
       $outer-sum\ (product-basis\ B\ C)\ (scale-function\ f)$ 
      using  $KM.vs.lincomb-eq-sum[OF\ Pfin\ PM\ fK']$ 
      by (simp add:  $outer-sum-def\ scale-function-def$ )
    show ?thesis using  $lincomb-sum\ fzero$  by simp
  qed
  have  $inj: inj-on\ product-map\ (C \times B)$ 
    by (rule  $KM-product-map-inj[OF\ B\ C]$ )
  have  $pairzero: product-sum\ (C \times B)\ (pair-scale\ f) = 0$ 
  proof –
    have  $pair-sum-eq:$ 
       $product-sum\ (C \times B)\ (pair-scale\ f) =$ 
       $outer-sum\ (product-basis\ B\ C)\ (scale-function\ f)$ 
      unfolding  $outer-sum-def\ scale-function-def\ product-sum-def\ pair-scale-def$ 
       $product-basis-image$ 
      by (subst  $sum.reindex[OF\ inj]$ ) (simp add:  $o-def$ )

```

```

  show ?thesis using pair-sum-eq sumzero by simp
qed
define e :: 'a ⇒ 'a
  where e = fiber-sum f B
have subL: Subfield L by (rule KL.ext.Subfield-axioms)
have eL: e c ∈ L if cC: c ∈ C for c
  unfolding e-def fiber-sum-def
proof (rule Subfield.sum-closed[OF subL])
  fix b assume bB: b ∈ B
  have bL: b ∈ L using BL bB by blast
  have bcP: b * c ∈ product-basis B C
  proof -
    have pair: (c, b) ∈ C × B by (simp add: cC bB)
    have img: product-map (c, b) ∈ image product-map (C × B)
      by (rule imageI[OF pair])
    have eq: product-map (c, b) = b * c
      by (simp add: product-map-def)
    show ?thesis unfolding product-basis-image using img eq by simp
  qed
  then have f (b * c) ∈ K by (rule fK)
  then have fL: f (b * c) ∈ L
    using KL.base-subset by blast
  then show f (b * c) * b ∈ L
    using bL by (rule Subfield.mult-closed[OF subL])
qed
have nested:
  (∑ z ∈ C × B. pair-scale f z) =
  (∑ c ∈ C. ∑ b ∈ B. pair-scale f (c, b))
  by (simp add: sum.cartesian-product)
have pair-terms:
  (∑ c ∈ C. ∑ b ∈ B. pair-scale f (c, b)) =
  (∑ c ∈ C. ∑ b ∈ B. (f (b * c) * b) * c)
  by (simp add: pair-scale-def product-map-def mult.assoc)
have distribute:
  (∑ c ∈ C. ∑ b ∈ B. (f (b * c) * b) * c) =
  (∑ c ∈ C. (∑ b ∈ B. f (b * c) * b) * c)
  by (simp add: sum-distrib-right)
have pair-outer-unfolded:
  (∑ z ∈ C × B. pair-scale f z) =
  (∑ c ∈ C. (∑ b ∈ B. f (b * c) * b) * c)
  by (rule trans[OF nested], rule trans[OF pair-terms distribute])
have pair-outer:
  product-sum (C × B) (pair-scale f) =
  outer-sum C (scale-function e)
  using pair-outer-unfolded
  by (simp only: product-sum-def outer-sum-def scale-function-def
    e-def fiber-sum-def)
have esumzero: outer-sum C (scale-function e) = 0
  using pairzero pair-outer by simp
have eC: restrict e C ∈ C →E L

```



```

    using eL by (auto simp: PiE-def)
  have eL':  $\bigwedge v. v \in C \implies \text{restrict } e \ C \ v \in L$ 
    using eL by (auto simp: PiE-def)
  have elincomb:  $LM.vs.lincomb (\text{restrict } e \ C) \ C = 0$ 
proof -
  have lincomb-sum:
     $LM.vs.lincomb (\text{restrict } e \ C) \ C =$ 
     $outer-sum \ C \ (\text{scale-function } e)$ 
    using  $LM.vs.lincomb-eq-sum[\text{where } c=\text{restrict } e \ C, \text{ OF } Cfin \ CM \ eL]$ 
    by (simp add: outer-sum-def scale-function-def restrict-apply')
  show ?thesis using lincomb-sum esumzero by simp
qed
have fzero-pair:  $f \ (b * c) = 0$  if  $bB: b \in B$  and  $cC: c \in C$  for  $b \ c$ 
proof -
  have e-restrict-zero:  $\text{restrict } e \ C \ c = 0$ 
proof -
  have ecoeffs:  $LM.vs.mod.coeffs-on (\text{restrict } e \ C) \ C$ 
    using eC by (auto intro!:  $LM.vs.mod.coeffs-onI$ )
  have eind:  $LM.vs.mod.lin-indep \ C$ 
    by (rule iffD1[ $OF \ LM.vs.lin-indep-iff-mod-lin-indep[OF \ Cfin \ CM]$ 
     $LM.vs.basis-lin-indep[OF \ C]$ ])
  have ezero':  $LM.vs.mod.lincomb (\text{restrict } e \ C) \ C = 0$ 
    by (rule elincomb)
  show  $\text{restrict } e \ C \ c = 0$ 
    by (rule  $LM.vs.mod.lin-indepD[OF \ eind \ Cfin \ subset-refl \ ecoeffs \ ezero' \ cC]$ )
qed
have ezero-c:  $e \ c = 0$ 
  using e-restrict-zero by (simp add: restrict-apply' cC)
define g :: ' $a \Rightarrow 'a$ ' where  $g = \text{fiber-function } f \ c$ 
have gK:  $g \ u \in K$  if  $uB: u \in B$  for  $u$ 
proof -
  have pair:  $(c, u) \in C \times B$  by (simp add: cC uB)
  have img:  $\text{product-map } (c, u) \in \text{image product-map } (C \times B)$ 
    by (rule imageI[ $OF \ pair$ ])
  have eq:  $\text{product-map } (c, u) = u * c$ 
    by (simp add: product-map-def)
  have ucP:  $u * c \in \text{product-basis } B \ C$ 
    unfolding product-basis-image using img eq by simp
  show  $g \ u \in K$ 
    unfolding g-def fiber-function-def
    by (rule fK[ $OF \ ucP$ ])
qed
have gK':  $\bigwedge u. u \in B \implies \text{restrict } g \ B \ u \in K$ 
  using gK by (auto simp: restrict-apply')
have glincomb:  $KL.vs.lincomb (\text{restrict } g \ B) \ B = 0$ 
proof -
  have lincomb-outer:
     $KL.vs.lincomb (\text{restrict } g \ B) \ B =$ 
     $outer-sum \ B \ (\text{scale-function } g)$ 
    using  $KL.vs.lincomb-eq-sum[\text{where } c=\text{restrict } g \ B, \text{ OF } Bfin \ BL \ gK]$ 

```

```

    by (simp add: outer-sum-def scale-function-def restrict-apply')
  have outer-e: outer-sum B (scale-function g) = e c
    unfolding outer-sum-def scale-function-def g-def
      e-def fiber-sum-def fiber-function-def
    by simp
  show KL.vs.lincomb (restrict g B) B = 0
    using lincomb-outer outer-e ezero-c by simp
qed
have g-ind: KL.vs.mod.lin-indep B
  by (rule iffD1[OF KL.vs.lin-indep-iff-mod-lin-indep[OF Bfin BL]
    KL.vs.basis-lin-indep[OF B]])
have g-coeffs: KL.vs.mod.coeffs-on (restrict g B) B
  using gK' by (auto intro!: KL.vs.mod.coeffs-onI)
have restrict g B b = 0
  by (rule KL.vs.mod.lin-indepD[OF g-ind Bfin subset-refl g-coeffs glincomb bB])
then show ?thesis by (simp add: restrict-apply' bB g-def fiber-function-def)
qed
show  $\forall v \in \text{product-basis } B \ C. f v = 0$ 
proof
  fix v assume vP:  $v \in \text{product-basis } B \ C$ 
  then obtain b c where bB:  $b \in B$  and cC:  $c \in C$  and v-eq:  $v = b * c$ 
    unfolding product-basis-def by blast
  show  $f v = 0$  using fzero-pair[OF bB cC] v-eq by simp
qed
qed

```

lemma *KM-product-basis-independent:*
assumes B : $KL.vs.basis \ B$ **and** C : $LM.vs.basis \ C$
shows $KM.vs.lin-indep \ (\text{product-basis } B \ C)$
proof –
 have Bfin: *finite* B **and** BL: $B \subseteq L$
 using $B \ KL.basis-subset[OF \ B]$ by (auto simp: $KL.vs.basis-def$)
 have Cfin: *finite* C **and** CM: $C \subseteq M$
 using $C \ LM.basis-subset[OF \ C]$ by (auto simp: $LM.vs.basis-def$)
 have Pfin: *finite* ($\text{product-basis } B \ C$) by (simp add: $\text{product-basis-image } Bfin \ Cfin$)
 have PM: $\text{product-basis } B \ C \subseteq M$ by (rule $KM-product-basis-subset[OF \ B \ C]$)
 show $KM.vs.lin-indep \ (\text{product-basis } B \ C)$
 unfolding $KM.vs.lin-indep-def$
 using Pfin PM $KM-product-basis-condition[OF \ B \ C]$ by blast
qed

theorem *KM-product-basis:*
assumes B : $KL.vs.basis \ B$ **and** C : $LM.vs.basis \ C$
shows $KM.vs.basis \ (\text{product-basis } B \ C)$
proof (rule $KM.vs.basisI$)
 have Bfin: *finite* B **using** B **by** (auto simp: $KL.vs.basis-def$)
 have Cfin: *finite* C **using** C **by** (auto simp: $LM.vs.basis-def$)
 have Pfin: *finite* ($\text{product-basis } B \ C$)
 by (simp add: $\text{product-basis-image } Bfin \ Cfin$)
 have PM: $\text{product-basis } B \ C \subseteq M$

```

    by (rule KM-product-basis-subset[OF B C])
  have mod-spanning: KM.vs.mod.spanning (product-basis B C)
proof (rule KM.vs.mod.spanningI[OF PM])
  fix v assume vM:  $v \in M$ 
  show  $v \in \text{KM.vs.mod.span}$  (product-basis B C)
    using KM-product-basis-spanning[OF B C] vM by blast
qed
show KM.vs.spanning (product-basis B C)
  using KM.vs.spanning-iff-mod-spanning[OF Pfin PM] mod-spanning by blast
show KM.vs.lin-indep (product-basis B C)
  by (rule KM-product-basis-independent[OF B C])
qed

sublocale KMF: finite-subfield-tower K M
proof -
  obtain B where B: KL.vs.basis B using KL.finite-basis-exists by blast
  obtain C where C: LM.vs.basis C using LM.finite-basis-exists by blast
  have P: KM.vs.basis (product-basis B C) by (rule KM-product-basis[OF B C])
  show finite-subfield-tower K M
proof (rule finite-subfield-tower.intro)
  show subfield-tower K M by (rule subfield-tower-KM)
  show finite-subfield-tower-axioms K M
    by (rule finite-subfield-tower-axioms.intro) (rule exI[of - product-basis B C], rule
P)
qed
qed

theorem extension-degree-tower-law:
  assumes B: KL.vs.basis B and C: LM.vs.basis C
  shows KMF.extension-degree = KL.extension-degree * LM.extension-degree
proof -
  have P: KM.vs.basis (product-basis B C) by (rule KM-product-basis[OF B C])
  have cardP:  $\text{card}(\text{product-basis } B \ C) = \text{card } B * \text{card } C$ 
    by (rule KM-product-basis-card[OF B C])
  have cardB:  $\text{card } B = \text{KL.extension-degree}$ 
    using KL.extension-degree-eq-card-basis[OF B] by simp
  have cardC:  $\text{card } C = \text{LM.extension-degree}$ 
    using LM.extension-degree-eq-card-basis[OF C] by simp
  have cardP':  $\text{card}(\text{product-basis } B \ C) = \text{KL.extension-degree} * \text{LM.extension-degree}$ 
    using cardP cardB cardC by simp
  show ?thesis
    using KMF.extension-degree-eq-card-basis[OF P] cardP' by simp
qed

theorem extension-degree-tower-law-exists:
  KMF.extension-degree = KL.extension-degree * LM.extension-degree
proof -
  obtain B where B: KL.vs.basis B using KL.finite-basis-exists by blast
  obtain C where C: LM.vs.basis C using LM.finite-basis-exists by blast
  show ?thesis by (rule extension-degree-tower-law[OF B C])

```

qed

end

end

81 Extensions generated by finitely many algebraic elements

```
theory Finite-Generated-Extension
  imports Finite-Extension-Tower
begin
```

Adjoining one algebraic element to a subfield gives a finite extension — the power basis $1, a, \dots, a^{d-1}$ spans it — and finite extensions compose. Adjoining finitely many algebraic elements one at a time therefore stays finite. This is HOL-Algebra's *finite-extension*, and it is what turns a polynomial with algebraic coefficients into a polynomial over a finite extension of the base field, so that a dimension count can be applied to it.

The pieces were all present: *power-basis-span* supplies the spanning set, *subfield-tower.finite-subfield-tower-of-finite-spanning* converts it into finite-dimensionality, and the chain locale composes two finite extensions. Only the assembly was missing.

81.1 A subfield is a finite extension of itself

```
lemma finite-subfield-tower-self:
  assumes sfK: Subfield K
  shows finite-subfield-tower K K
proof -
  interpret K: Subfield K by (rule sfK)
  interpret T: subfield-tower K K by (rule subfield-tower-refl [OF sfK])
  have span1: K  $\subseteq$  T.vs.span {1}
  proof
    fix x assume x: x  $\in$  K
    have ( $\sum y \in \{1 :: 'a\}. x * y$ )  $\in$  T.vs.span (( $\lambda y. y$ ) ` {1})
      by (rule T.sum-scale-in-span) (auto simp: x)
    then show x  $\in$  T.vs.span {1} by simp
  qed
  show ?thesis
    by (rule T.finite-subfield-tower-of-finite-spanning [OF - - span1]) auto
qed
```

81.2 A simple algebraic extension is finite

The spanning set is the power basis, and the coefficients supplied by *power-basis-span* are fed to *subfield-tower.sum-scale-in-span* unchanged.

```

lemma finite-subfield-tower-simple:
  assumes sfK: Subfield K and alg: algebraic-over K a
  shows finite-subfield-tower K (eval-img K a)
proof -
  interpret K: Subfield K by (rule sfK)
  have sfE: Subfield (eval-img K a) by (rule K.subfield-eval-img [OF alg])
  interpret E: Subfield eval-img K a by (rule sfE)
  interpret T: subfield-tower K eval-img K a
    by (rule subfield-tower.intro [OF sfK sfE]) (unfold-locales, use K.eval-img-base in blast)
  define d where d = ext-degree K a
  define B where B = ( $\lambda i. a \wedge i$ ) ‘ {.. $d$ }
  have finB: finite B by (simp add: B-def)
  have powE:  $a \wedge i \in \text{eval-img } K a$  for i
    by (rule E.power-closed [OF K.eval-img-self])
  then have BL:  $B \subseteq \text{eval-img } K a$  by (auto simp: B-def)
  have eval-img K a  $\subseteq T.\text{vs.span } B$ 
proof
  fix y assume y:  $y \in \text{eval-img } K a$ 
  obtain c where cK:  $\bigwedge i. i < d \implies c i \in K$  and yc:  $y = (\sum i < d. c i * a \wedge i)$ 
    using K.power-basis-span [OF alg y] by (auto simp: d-def)
  have ( $\sum i \in \{.. $d\}. c i * a \wedge i$ )  $\in T.\text{vs.span } ((\lambda i. a \wedge i) \text{ ‘ } \{.. $d\})$ 
    by (rule T.sum-scale-in-span) (auto simp: powE cK)
  then show  $y \in T.\text{vs.span } B$  by (simp add: yc B-def)
qed
then show ?thesis
  by (rule T.finite-subfield-tower-of-finite-spanning [OF finB BL])
qed$$ 
```

81.3 Finite extensions compose

```

lemma finite-subfield-tower-trans:
  assumes KL: finite-subfield-tower K L and LM: finite-subfield-tower L M
  shows finite-subfield-tower K M
proof -
  interpret C: finite-subfield-tower-chain K L M
    by (rule finite-subfield-tower-chain.intro [OF KL LM])
  show ?thesis
proof (rule finite-subfield-tower.intro)
  show subfield-tower K M by (rule C.subfield-tower-KM)
  show finite-subfield-tower-axioms K M
    by (rule finite-subfield-tower-axioms.intro) (use C.KMF.finite-basis in blast)
qed
qed

```

81.4 Finitely many algebraic elements generate a finite extension

Adjoin the elements one at a time. Each is algebraic over the base and hence, by *algebraic-over-mono*, over every intermediate field, so the simple-extension result applies at each step and the composition keeps the whole tower finite.

theorem *exists-finite-subfield-tower*:

```

assumes sfK: Subfield K and finS: finite S
and algS:  $\bigwedge a. a \in S \implies \text{algebraic-over } K \ a$ 
shows  $\exists L. \text{finite-subfield-tower } K \ L \wedge K \subseteq L \wedge S \subseteq L$ 
using finS algS
proof (induct S rule: finite-induct)
  case empty
  show ?case
    using finite-subfield-tower-self [OF sfK] by blast
next
  case (insert a S)
  then obtain L where L: finite-subfield-tower K L and KL:  $K \subseteq L$  and SL:  $S \subseteq L$ 
    by auto
  interpret L: finite-subfield-tower K L by (rule L)
  have sfL: Subfield L by (rule L.ext.Subfield-axioms)
  have algL: algebraic-over L a
    by (rule algebraic-over-mono [OF insert.prems [of a] KL]) simp
  have finite-subfield-tower L (eval-img L a)
    by (rule finite-subfield-tower-simple [OF sfL algL])
  with L have tower: finite-subfield-tower K (eval-img L a)
    by (rule finite-subfield-tower-trans)
  have  $L \subseteq \text{eval-img } L \ a$  using L.ext.eval-img-base by blast
  then have  $K \subseteq \text{eval-img } L \ a$  and  $\text{insert } a \ S \subseteq \text{eval-img } L \ a$ 
    using KL SL L.ext.eval-img-self by auto
  with tower show ?case by blast
qed

```

81.5 Coordinates over a finite extension

A finite extension carries a finite spanning set together with a coordinate function writing each of its elements as an explicit combination over the base. Stated inside the locale, where the two fields are parameters: a client whose base field is a described set — an image, say — would otherwise send the automatic methods hunting through that description.

lemma (*in finite-subfield-tower*) *exists-coordinates*:

```

obtains B c where finite B and  $B \subseteq L$ 
and  $\bigwedge x. x \in L \implies (\forall b \in B. \ c \ x \ b \in K)$ 
and  $\bigwedge x. x \in L \implies x = (\sum b \in B. \ c \ x \ b * b)$ 
proof –
obtain B where B: vs.basis B using finite-basis by blast
have sp: vs.spanning B using B by (rule vs.basis-spanning)
have finB: finite B and BL:  $B \subseteq L$  using sp by (auto simp: vs.spanning-def)

```

```

have ex:  $\exists f. (\forall b \in B. f \ b \in K) \wedge x = (\sum b \in B. f \ b * b)$  if  $x \in L$  for  $x$ 
proof –
  have  $\exists c \in B \rightarrow_E K. x = \text{vs.lincomb } c \ B$ 
    by (rule bspec [OF sp [unfolded vs.spanning-def, THEN conjunct2, THEN conjunct2] that])
  then obtain  $f$  where  $f: f \in B \rightarrow_E K$  and  $xf: x = \text{vs.lincomb } f \ B$  by blast
  have  $\text{vs.lincomb } f \ B = (\sum b \in B. f \ b * b)$ 
    by (rule vs-lincomb-eq-sum [OF finB BL]) (use  $f$  in auto)
  with  $xf \ f$  show ?thesis by (auto simp: PiE-iff)
qed
define  $c$  where  $c \ x = (\text{SOME } f. (\forall b \in B. f \ b \in K) \wedge x = (\sum b \in B. f \ b * b))$  for  $x$ 
have  $(\forall b \in B. c \ x \ b \in K) \wedge x = (\sum b \in B. c \ x \ b * b)$  if  $x \in L$  for  $x$ 
proof –
  have  $\exists f. (\forall b \in B. f \ b \in K) \wedge x = (\sum b \in B. f \ b * b)$  by (rule ex [OF that])
  then show ?thesis unfolding c-def by (rule someI-ex)
qed
then show thesis using finB BL by (blast intro: that)
qed
end

```

82 Homogeneous linear systems with more unknowns than equations

```

theory Linear-System
imports Subfield
begin

```

A homogeneous linear system with strictly more unknowns than equations has a nontrivial solution. This is the pigeonhole principle of linear algebra, and it is what powers *Artin's theorem* on the fixed field of a finite group of automorphisms: there one produces a nonzero solution of the system $\sum_j \sigma(x_j) \ c_j = 0$, one equation per automorphism σ , from an assumed excess of independent elements x_j .

We index equations by a finite set I and unknowns by a finite set J , rather than by initial segments $\{..<m\}$ and $\{..<n\}$. This matters: the proof eliminates one unknown at each step, and with index *sets* that is just $J - \{k\}$, whereas with initial segments every step would need a reindexing bijection.

Everything happens inside a subfield K : the coefficients lie in K and so does the solution, since Gaussian elimination only ever adds, multiplies and divides.

```

context Subfield
begin

```

The proof is Gaussian elimination, by induction on the set of equations.

With no equations at all, any nonzero vector will do — and there is an unknown to put it in, since $\text{card } \{\} < \text{card } J$ forces J nonempty.

For the inductive step, single out one equation. If all its coefficients vanish it is no constraint, and the induction hypothesis applied to the remaining equations already gives a solution. Otherwise some coefficient $A\ i_0\ k$ is nonzero, and we use that equation to *eliminate* the unknown k : subtracting a suitable multiple of equation i_0 from each remaining equation produces a smaller system in the unknowns $J - \{k\}$, to which the induction hypothesis applies because $\text{card } I < \text{card } J - 1$. Solving that and then choosing $c\ k$ to satisfy equation i_0 solves the whole system. Nontriviality survives because the recursively obtained solution is already nonzero *somewhere in* $J - \{k\}$, and we do not touch those entries.

theorem *underdetermined-solution*:

```

fixes  $A :: 'i \Rightarrow 'j \Rightarrow 'a$ 
assumes finite I and finite J and card I < card J
and  $\bigwedge i\ j. \llbracket i \in I; j \in J \rrbracket \Longrightarrow A\ i\ j \in K$ 
shows  $\exists c. (\forall j. c\ j \in K) \wedge (\exists j \in J. c\ j \neq 0) \wedge (\forall i \in I. (\sum j \in J. A\ i\ j * c\ j) = 0)$ 
using assms
proof (induction I arbitrary: J A rule: finite-induct)
  case empty
    — No equations: put a 1 in any single unknown.
    then obtain  $j_0$  where  $j_0: j_0 \in J$  by (metis card.empty gr-implies-not-zero card-gt-0-iff equals0I)
    — The constant-one vector: every entry lies in  $K$ , and it is nonzero at  $j_0$ . (A one-hot vector would do as well, but the if makes the nontriviality step fiddly.) The witness is supplied explicitly by exI[of - c] rather than left to an automatic method, so that nothing has to be guessed for the function  $c$ .
    define  $c :: 'j \Rightarrow 'a$  where  $c = (\lambda\_. 1)$ 
    show ?case
    proof (intro exI[of - c] conjI)
      show  $\forall j. c\ j \in K$  by (simp add: c-def)
      show  $\exists j \in J. c\ j \neq 0$  using  $j_0$  by (intro bexI[of - j_0]) (simp-all add: c-def)
      show  $\forall i \in \{\}. (\sum j \in J. A\ i\ j * c\ j) = 0$  by simp
    qed
  next
    case (insert i_0 I)
    then have  $fJ: \text{finite } J$  and card: card (insert i_0 I) < card J
    and  $\text{ent}: \bigwedge i\ j. \llbracket i \in \text{insert } i_0\ I; j \in J \rrbracket \Longrightarrow A\ i\ j \in K$ 
    by auto
    have cardI: card I < card J using card insert.hyps by (simp add: card-insert-if)
    show ?case
    proof (cases  $\forall j \in J. A\ i_0\ j = 0$ )
      case True
        — The singled-out equation is vacuous, so the rest of the system already decides the matter.
        The induction generalises over  $A$ , so the hypothesis must be told which matrix to use: where  $A = A$ . Without it, unifying the closure premise leaves the matrix arguments as unknowns — the hypothesis comes back mentioning  $A\ (?i\ i\ j)\ (?j\ i\ j)$  — and any automatic method then searches for those two index functions instead of simply stripping the existential. That search does not terminate.
        have entI:  $\bigwedge i\ j. \llbracket i \in I; j \in J \rrbracket \Longrightarrow A\ i\ j \in K$  using ent by auto

```


obtain c **where** $c: \forall j. c\ j \in K \exists j \in J. c\ j \neq 0$
 $\forall i \in I. (\sum j \in J. A\ i\ j * c\ j) = 0$
using *insert.IH*[**where** $A = A$, *OF fJ cardI entI*] **by** *blast*
have $(\sum j \in J. A\ i_0\ j * c\ j) = 0$ **using** *True* **by** *simp*
with c **show** *?thesis* **by** (*intro exI[of - c] conjI*) *blast+*
next
case *False*
then obtain k **where** $k: k \in J$ **and** $a: A\ i_0\ k \neq 0$ **by** *blast*
define a **where** $a = A\ i_0\ k$
have $aK: a \in K$ **and** $anz: a \neq 0$ **using** *ent k a* **by** (*auto simp: a-def*)
— Eliminate the unknown k from every remaining equation.
define B **where** $B = (\lambda i\ j. A\ i\ j - A\ i\ k * A\ i_0\ j / a)$
have $BK: \bigwedge i\ j. \llbracket i \in I; j \in J - \{k\} \rrbracket \implies B\ i\ j \in K$
unfolding B -*def* **using** *ent k aK* **by** (*auto intro!: diff-closed mult-closed divide-closed*)
have $cardJk: card\ I < card\ (J - \{k\})$
using *card k fJ insert.hyps* **by** (*simp add: card-insert-if card-Diff-singleton*)
obtain c' **where** $c'K: \forall j. c'\ j \in K$
and $c'nz: \exists j \in J - \{k\}. c'\ j \neq 0$
and $c'eq: \forall i \in I. (\sum j \in J - \{k\}. B\ i\ j * c'\ j) = 0$
— Same instantiation caveat as in the vacuous case: pin both the matrix and the index set.
using *insert.IH*[**where** $A = B$ **and** $J = J - \{k\}$, *OF finite-Diff*[*OF fJ*] *cardJk BK*] **by** *blast*
— Now choose the eliminated unknown so as to satisfy the singled-out equation.
define S **where** $S = (\sum j \in J - \{k\}. A\ i_0\ j * c'\ j)$
define c **where** $c = (\lambda j. \text{if } j = k \text{ then } -S / a \text{ else } c'\ j)$
have $SK: S \in K$ **unfolding** S -*def* **using** *ent c'K* **by** (*auto intro!: sum-closed mult-closed*)
have $cK: \bigwedge j. c\ j \in K$
unfolding c -*def* **using** $SK\ aK\ c'K$ **by** (*auto intro!: divide-closed uminus-closed*)
have $ck: c\ k = -S / a$ **by** (*simp add: c-def*)
have $cj: \bigwedge j. j \in J - \{k\} \implies c\ j = c'\ j$ **by** (*simp add: c-def*)
— Split every sum over J at the eliminated unknown.
have *split*: $(\sum j \in J. f\ j * c\ j) = f\ k * c\ k + (\sum j \in J - \{k\}. f\ j * c'\ j)$ **for** f
proof —
have $(\sum j \in J. f\ j * c\ j) = f\ k * c\ k + (\sum j \in J - \{k\}. f\ j * c\ j)$
using $k\ fJ$ **by** (*simp add: sum.remove*)
also have $(\sum j \in J - \{k\}. f\ j * c\ j) = (\sum j \in J - \{k\}. f\ j * c'\ j)$
by (*rule sum.cong*) (*auto simp: cj*)
finally show *?thesis* .
qed
have $eq0: (\sum j \in J. A\ i_0\ j * c\ j) = 0$
using *split*[*of A i_0*] *anz* **by** (*simp add: ck a-def[symmetric] S-def[symmetric]*)
have $eqI: (\sum j \in J. A\ i\ j * c\ j) = 0$ **if** $i: i \in I$ **for** i
proof —
— The eliminated equation says exactly that the tail sum equals $A\ i\ k / a * S$.
have $(\sum j \in J - \{k\}. A\ i\ j * c'\ j) - A\ i\ k / a * S = 0$
using $c'eq\ i$ **unfolding** B -*def* S -*def*
by (*simp add: sum-subtractf field-simps sum-distrib-left flip: sum-divide-distrib*)

```

then have tail:  $(\sum j \in J - \{k\}. A\ i\ j * c'\ j) = A\ i\ k / a * S$  by simp
have  $(\sum j \in J. A\ i\ j * c\ j) = A\ i\ k * (-\ S / a) + A\ i\ k / a * S$ 
  using split[of A i] by (simp add: ck tail)
also have  $\dots = 0$  by (simp add: field-simps)
finally show ?thesis .
qed
show ?thesis
proof (intro exI conjI)
  show  $\forall j. c\ j \in K$  using cK by blast
  show  $\exists j \in J. c\ j \neq 0$  using c'nz cj by force
  show  $\forall i \in \text{insert } i_0\ I. (\sum j \in J. A\ i\ j * c\ j) = 0$  using eq0 eqI by blast
qed
qed
qed

end

end

```

83 The Algebraic Closure of a Field

```

theory Closure-Algebraic
imports Closure-Extension Poly-Typeclass Algebraic-Transitivity Extension-Properties
  Finite-Generated-Extension Linear-System
  HOL-Computational-Algebra.Polynomial-Factorial HOL-Computational-Algebra.Field-as-Ring
begin

```

Closure-Extension produces, for any field, an extension inside the indexed polynomial ring in which every polynomial over the base field splits. That extension is not yet an algebraic closure: it is far too big, being generated by one indeterminate for every polynomial and root index, and it is not algebraic over the base. The closure is cut out of it as the set of elements algebraic over the base. Rather than redo the theory of algebraicity over a subfield for locale fields, we transport the extension onto a type. The development's field-extension and Galois theory already works with an ambient type-class field and subfields as carrier sets, and once the extension is a type all of it applies — in particular *subfield-algebraic-elements*, which is the statement that the algebraic elements form a subfield.

83.1 An extension of the whole type in which every polynomial splits

```

context poly-TC
begin

```

A definite choice among the extensions supplied by *exists-extension-with-roots*, so that the type constructed from it below depends on nothing but $'a$.

definition *wide-extension*
where *wide-extension* $\equiv$
 $(\text{SOME } M. \text{is-extension } M$
 $\quad \wedge (\forall P \in \text{poly-carrier.}$
 $\quad \quad \text{Field.splits (ecarrier } M) (\text{eadd } M) (\text{emult } M) \text{ipoly-zero ipoly-one}$
 $\quad \quad (\text{lift-poly } P)))$

lemma *wide-extension-spec*:
 $\text{is-extension } \text{wide-extension}$
 $\wedge (\forall P \in \text{poly-carrier.}$
 $\quad \text{Field.splits (ecarrier } \text{wide-extension}) (\text{eadd } \text{wide-extension}) (\text{emult } \text{wide-extension})$
 $\quad \quad \text{ipoly-zero ipoly-one (lift-poly } P))$
unfolding *wide-extension-def*
by (rule *someI-ex*) (metis *exists-extension-with-roots*)

lemmas *is-extension-wide-extension* = *wide-extension-spec* [THEN *conjunct1*]

lemma *wide-extension-splits*:
 $P \in \text{poly-carrier}$
 $\implies \text{Field.splits (ecarrier } \text{wide-extension}) (\text{eadd } \text{wide-extension}) (\text{emult } \text{wide-extension})$
 $\quad \quad \text{ipoly-zero ipoly-one (lift-poly } P)$
using *wide-extension-spec* **by** *blast*

The field structure of the chosen extension, named once for the transport below.

lemma *field-wide-extension*:
 $\text{Field (ecarrier } \text{wide-extension}) (\text{eadd } \text{wide-extension}) (\text{emult } \text{wide-extension})$
 $\quad \quad \text{ipoly-zero ipoly-one}$
by (rule *is-extension-field* [OF *is-extension-wide-extension*])

lemma *wide-extension-nonempty*: $\text{ipoly-zero} \in \text{ecarrier } \text{wide-extension}$
proof –
interpret *W*: $\text{Field ecarrier } \text{wide-extension eadd } \text{wide-extension emult } \text{wide-extension}$
 $\quad \quad \text{ipoly-zero ipoly-one}$
by (rule *field-wide-extension*)
show ?thesis **by** (rule *W.additive.unit-closed*)
qed

How the base field sits inside the extension. These are restatements of the extension conditions at the chosen extension: only facts proved *in this locale* are available once it is left, and the type constructed below can only be spoken of outside it.

lemma *wide-const-mem*: $\text{ipoly-const } a \in \text{ecarrier } \text{wide-extension}$
by (simp add: *is-extension-const-mem* [OF *is-extension-wide-extension*])

lemma *wide-const-add*:
 $\text{eadd } \text{wide-extension} (\text{ipoly-const } a) (\text{ipoly-const } b) = \text{ipoly-const } (a + b)$
by (simp add: *is-extension-const-add* [OF *is-extension-wide-extension*])

lemma *wide-const-mult*:

```

emult wide-extension (ipoly-const a) (ipoly-const b) = ipoly-const (a * b)
by (simp add: is-extension-const-mult [OF is-extension-wide-extension])

```

```

lemmas wide-const-inj = ipoly-const-inj-on
lemmas wide-lift-poly-apply = lift-poly-apply
lemmas wide-const-zero = ipoly-const-zero
lemmas wide-const-one = ipoly-const-one

```

```

end

```

83.2 The extension as a type

The extension is a field, but a locale one, on a carrier set inside the indexed polynomial ring. Carving that carrier out as a type and transporting the operations onto it makes it a type-class field, which is what the rest of the development can use.

```

typedef (overloaded) 'a :: field wide = ecarrier (poly-TC.wide-extension :: 'a iext)
using wide-extension-nonempty by blast

```

```

setup-lifting type-definition-wide

```

Names for the transported structure. The locale's constants must be spelled out here because *poly-TC* fixes no parameters, so its abbreviations do not reach outside it; each takes only the parameters occurring in its definition.

```

abbreviation wcarrier :: 'a :: field ipoly set
where wcarrier ≡ ecarrier (poly-TC.wide-extension :: 'a iext)

```

```

abbreviation wadd :: 'a :: field ipoly ⇒ 'a ipoly ⇒ 'a ipoly
where wadd ≡ eadd (poly-TC.wide-extension :: 'a iext)

```

```

abbreviation wmult :: 'a :: field ipoly ⇒ 'a ipoly ⇒ 'a ipoly
where wmult ≡ emult (poly-TC.wide-extension :: 'a iext)

```

```

abbreviation wzero :: 'a :: field ipoly
where wzero ≡ Ring.ipoly-zero 0

```

```

abbreviation wone :: 'a :: field ipoly
where wone ≡ Ring.ipoly-one 0 1

```

```

abbreviation wneg :: 'a :: field ipoly ⇒ 'a ipoly
where wneg ≡ Monoid.inverse wcarrier wadd wzero

```

```

abbreviation winv :: 'a :: field ipoly ⇒ 'a ipoly
where winv ≡ Monoid.inverse wcarrier wmult wone

```

```

interpretation W: Field wcarrier wadd wmult wzero wone
by (rule field-wide-extension)

```

instantiation *wide* :: (*field*) *field*
begin

lift-definition *zero-wide* :: '*a wide* **is** *wzero* :: '*a ipoly*
by (*rule W.additive.unit-closed*)

lift-definition *one-wide* :: '*a wide* **is** *wone* :: '*a ipoly*
by (*rule W.multiplicative.unit-closed*)

lift-definition *plus-wide* :: '*a wide* $\Rightarrow$ '*a wide* $\Rightarrow$ '*a wide* **is** *wadd* :: '*a ipoly* $\Rightarrow$ -
by (*rule W.additive.composition-closed*)

lift-definition *times-wide* :: '*a wide* $\Rightarrow$ '*a wide* $\Rightarrow$ '*a wide* **is** *wmult* :: '*a ipoly* $\Rightarrow$ -
by (*rule W.multiplicative.composition-closed*)

lift-definition *uminus-wide* :: '*a wide* $\Rightarrow$ '*a wide* **is** *wneg* :: '*a ipoly* $\Rightarrow$ -
by *simp*

lift-definition *minus-wide* :: '*a wide* $\Rightarrow$ '*a wide* $\Rightarrow$ '*a wide*
is $\lambda x y. wadd\ x\ (wneg\ y) :: '*a ipoly*
by *simp*$

HOL's field class requires *inverse* $0 = 0$, where the locale leaves the inverse of *wzero* unspecified; the two are reconciled by the case split.

lift-definition *inverse-wide* :: '*a wide* $\Rightarrow$ '*a wide*
is $\lambda x. \text{if } x = wzero \text{ then } wzero \text{ else } winv\ x :: '*a ipoly*
by (*auto intro: W.multiplicative.invertible-inverse-closed W.field-inverse*)$

lift-definition *divide-wide* :: '*a wide* $\Rightarrow$ '*a wide* $\Rightarrow$ '*a wide*
is $\lambda x y. \text{if } y = wzero \text{ then } wzero \text{ else } wmult\ x\ (winv\ y) :: '*a ipoly*
by (*auto intro: W.multiplicative.invertible-inverse-closed W.field-inverse*)$

instance

proof (*standard, goal-cases*)

case 1 show ?case by transfer (*rule W.multiplicative.associative*)

next

case 2 show ?case by transfer (*rule W.multiplicative.commutative*)

next

case 3 show ?case by transfer (*rule W.multiplicative.left-unit*)

next

case 4 show ?case by transfer (*rule W.additive.associative*)

next

case 5 show ?case by transfer (*rule W.additive.commutative*)

next

case 6 show ?case by transfer (*rule W.additive.left-unit*)

next

case 7 show ?case by transfer *simp*

next

case 8 show ?case by transfer *simp*

```

next
  case 9 show ?case by transfer (rule W.distributive)
next
  case 10 show ?case by transfer (rule not-sym [OF W.nontrivial])
next
  case 11 then show ?case
    by transfer (auto intro: W.field-inverse W.multiplicative.invertible-left-inverse)
next
  case 12 then show ?case by transfer auto
next
  case 13 then show ?case by transfer auto
qed

end

```

A field is a Euclidean ring in a trivial way. Spelling that out for *'a wide* — as HOL-Algebra's *Algebraic-Closure-Type* does for its own closure type — is what puts *'a wide poly* in *factorial-semiring*, and so makes HOL's factorisation machinery, in particular the primality of a monic linear polynomial, available over the extension.

```

instantiation wide :: (field)
  {unique-euclidean-ring, normalization-euclidean-semiring, normalization-semidom-multiplicative}
begin

```

```

definition [simp]: normalize-wide = (normalize-field :: 'a wide  $\Rightarrow$  -)
definition [simp]: unit-factor-wide = (unit-factor-field :: 'a wide  $\Rightarrow$  -)
definition [simp]: modulo-wide = (mod-field :: 'a wide  $\Rightarrow$  -)
definition [simp]: euclidean-size-wide = (euclidean-size-field :: 'a wide  $\Rightarrow$  -)
definition [simp]: division-segment (x :: 'a wide) = 1

```

```

instance
  by standard (simp-all add: dvd-field-iff field-split-simps split: if-splits)

```

```

end

```

```

instantiation wide :: (field) euclidean-ring-gcd
begin

```

```

definition gcd-wide :: 'a wide  $\Rightarrow$  'a wide  $\Rightarrow$  'a wide
  where gcd-wide = Euclidean-Algorithm.gcd
definition lcm-wide :: 'a wide  $\Rightarrow$  'a wide  $\Rightarrow$  'a wide
  where lcm-wide = Euclidean-Algorithm.lcm
definition Gcd-wide :: 'a wide set  $\Rightarrow$  'a wide
  where Gcd-wide = Euclidean-Algorithm.Gcd
definition Lcm-wide :: 'a wide set  $\Rightarrow$  'a wide
  where Lcm-wide = Euclidean-Algorithm.Lcm

```

```

instance
  by standard (simp-all add: gcd-wide-def lcm-wide-def Gcd-wide-def Lcm-wide-def)

```

end

instance *wide* :: (field) semiring-gcd-mult-normalize ..

The four instances above are exactly *field-gcd*, which is the sort under which HOL states the primality of a monic linear polynomial.

instance *wide* :: (field) field-gcd ..

83.3 The base field inside the extension

The base field sits in the extension as the constants. This is the embedding along which the algebraic closure will be described: a subfield of *'a wide* isomorphic to *'a*.

lift-definition *to-wide* :: 'a :: field $\Rightarrow$ 'a wide **is** Ring.ipoly-const 0 :: 'a $\Rightarrow$ 'a ipoly
by (rule wide-const-mem)

lemma *to-wide-zero* [simp]: *to-wide* 0 = 0
by transfer (rule wide-const-zero)

lemma *to-wide-one* [simp]: *to-wide* 1 = 1
by transfer (rule wide-const-one)

lemma *to-wide-add* [simp]: *to-wide* (x + y) = *to-wide* x + *to-wide* y
by transfer (rule wide-const-add [symmetric])

lemma *to-wide-mult* [simp]: *to-wide* (x * y) = *to-wide* x * *to-wide* y
by transfer (rule wide-const-mult [symmetric])

lemma *inj-to-wide*: inj (*to-wide* :: 'a :: field $\Rightarrow$ 'a wide)
by (metis UNIV-I injI inj-on-eq-iff *to-wide.rep-eq* wide-const-inj)

lemma *to-wide-eq-iff* [simp]: *to-wide* x = *to-wide* y $\longleftrightarrow$ x = y
using *inj-to-wide* by (auto dest: injD)

Being a ring homomorphism between fields, the embedding automatically respects negation and inversion: each is pinned by the equation it solves.

lemma *to-wide-uminus* [simp]: *to-wide* (− x) = − *to-wide* x
using *to-wide-add* [of − x x] *eq-neg-iff-add-eq-0* by auto

lemma *to-wide-inverse* [simp]: *to-wide* (inverse x) = inverse (*to-wide* x)
using *to-wide-mult* [of x inverse x] *inverse-unique right-inverse*[of x] by fastforce

lemma *to-wide-diff* [simp]: *to-wide* (x − y) = *to-wide* x − *to-wide* y
using *to-wide-add* [of x − y] by simp

theorem *subfield-range-to-wide*: Subfield (range (*to-wide* :: 'a :: field $\Rightarrow$ 'a wide))
proof

```

show (0 :: 'a wide) ∈ range to-wide using rangeI [of to-wide 0] by simp
show (1 :: 'a wide) ∈ range to-wide using rangeI [of to-wide 1] by simp
qed (auto simp flip: to-wide-add to-wide-uminus to-wide-mult to-wide-inverse)

```

The inverse of the embedding, so that a polynomial over the base subfield can be read back as a polynomial over 'a.

```

definition of-wide :: 'a :: field wide ⇒ 'a
where of-wide = inv to-wide

```

```

lemma of-wide-to-wide [simp]: of-wide (to-wide x) = x
by (simp add: of-wide-def inj-to-wide)

```

```

lemma to-wide-of-wide [simp]: x ∈ range to-wide ⇒ to-wide (of-wide x) = x
by (auto simp: of-wide-def f-inv-into-f)

```

```

lemma of-wide-zero [simp]: of-wide 0 = 0
by (metis of-wide-to-wide to-wide-zero)

```

83.4 Every polynomial over the base subfield splits

Abs-wide is a field isomorphism from the carrier onto the type, so *splits-hom* carries the splitting of a base polynomial across. Each equation below is read off from the corresponding *rep-eq* of the transported operation.

```

lemma Abs-wide-zero [simp]: Abs-wide wzero = 0
using Rep-wide-inverse [of 0] by (simp add: zero-wide.rep-eq)

```

```

lemma Abs-wide-one [simp]: Abs-wide wone = 1
using Rep-wide-inverse [of 1] by (simp add: one-wide.rep-eq)

```

```

lemma Abs-wide-add:
assumes x ∈ wcarrier and y ∈ wcarrier
shows Abs-wide (wadd x y) = Abs-wide x + Abs-wide y
using Rep-wide-inverse [of Abs-wide x + Abs-wide y]
by (simp add: plus-wide.rep-eq Abs-wide-inverse assms)

```

```

lemma Abs-wide-mult:
assumes x ∈ wcarrier and y ∈ wcarrier
shows Abs-wide (wmult x y) = Abs-wide x * Abs-wide y
using Rep-wide-inverse [of Abs-wide x * Abs-wide y]
by (simp add: times-wide.rep-eq Abs-wide-inverse assms)

```

```

lemma Abs-wide-const [simp]: Abs-wide (Ring.ipoly-const 0 a) = to-wide a
by (simp add: to-wide.abs-eq)

```

A polynomial whose coefficients lie in the base subfield, read back as a polynomial over the base field itself.

```

definition wide-pull :: 'a :: field wide poly ⇒ nat ⇒ 'a
where wide-pull p ≡ (λk. of-wide (Polynomial.coeff p k))

```



```

lemma wide-pull-carrier: wide-pull  $p \in \text{Ring.poly-carrier}$  ( $\text{UNIV} :: 'a :: \text{field set}$ ) 0
proof –
  have  $\{k. \text{wide-pull } p \ k \neq 0\} \subseteq \{k. \text{Polynomial.coeff } p \ k \neq 0\}$ 
    by (auto simp: wide-pull-def)
  then have finite  $\{k. \text{wide-pull } p \ k \neq (0 :: 'a)\}$ 
    by (rule finite-subset) simp
  then show ?thesis by (simp add: poly-carrier-iff)
qed

```

```

lemma to-wide-wide-pull:
  assumes  $p: p \in \text{poly-over}$  ( $\text{range } (\text{to-wide} :: 'a :: \text{field} \Rightarrow 'a \text{ wide})$ )
  shows  $\text{to-wide } (\text{wide-pull } p \ k) = \text{Polynomial.coeff } p \ k$ 
  by (simp add: Subfield.poly-over-coeff  $p$  subfield-range-to-wide wide-pull-def)

```

```

theorem wide-splits:
  fixes  $p :: 'a :: \text{field wide poly}$ 
  assumes  $p: p \in \text{poly-over}$  ( $\text{range to-wide}$ )
  obtains  $c$  as where  $p = [:c:] * (\prod a \leftarrow \text{as}. [: \text{uminus } a, 1:])$ 
proof –
  have  $\text{RA}: \text{Ring } w\text{carrier} \text{ wadd } w\text{mult}$  ( $w\text{zero} :: 'a \text{ ipoly}$ )  $w\text{one}$ 
    using  $\text{W.Ring-axioms}$  by simp
  have  $\text{RB}: \text{Ring } (\text{UNIV} :: 'a \text{ wide set})$   $(+)$   $(*)$  0 1
    using  $\text{field-TC.Ring-axioms}$  by simp
  have lift:  $\text{Ring.lift-poly } 0$  ( $\text{wide-pull } p$ )  $k \in w\text{carrier}$  for  $k$ 
    by (simp add: wide-lift-poly-apply wide-const-mem)
  have  $\text{spB}: \text{Field.splits } (\text{UNIV} :: 'a \text{ wide set})$   $(+)$   $(*)$  0 1
    ( $\lambda k. \text{Abs-wide } (\text{Ring.lift-poly } 0$  ( $\text{wide-pull } p$ )  $k$ ))
  proof (intro splits-hom [OF RA RB])
    show  $\bigwedge y \ y'. [\![ y \in w\text{carrier}; y' \in w\text{carrier} ]\!] \Longrightarrow \text{Abs-wide } (w\text{add } y \ y') = \text{Abs-wide}$ 
 $y + \text{Abs-wide } y'$ 
      by (rule Abs-wide-add)
    show  $\bigwedge y \ y'. [\![ y \in w\text{carrier}; y' \in w\text{carrier} ]\!] \Longrightarrow \text{Abs-wide } (w\text{mult } y \ y') = \text{Abs-wide}$ 
 $y * \text{Abs-wide } y'$ 
      by (rule Abs-wide-mult)
    show  $\text{Field } w\text{carrier} \text{ wadd } w\text{mult}$  ( $w\text{zero} :: 'a \text{ ipoly}$ )  $w\text{one}$ 
      by (rule field-wide-extension)
    show  $\text{Field } (\text{UNIV} :: 'a \text{ wide set})$   $(+)$   $(*)$  0 1
      by (rule field-TC.Field-axioms)
    show  $\text{Field.splits } w\text{carrier} \text{ wadd } w\text{mult}$   $w\text{zero} \ w\text{one}$  ( $\text{Ring.lift-poly } 0$  ( $\text{wide-pull } p$ ))
      by (rule wide-extension-splits [OF wide-pull-carrier])
  qed auto
  have ( $\lambda k. \text{Abs-wide } (\text{Ring.lift-poly } 0$  ( $\text{wide-pull } p$ )  $k$ )) =  $\text{Polynomial.coeff } p$ 
    by (simp add: wide-lift-poly-apply to-wide-wide-pull [OF  $p$ ])
  with  $\text{spB}$  have  $\text{Field.splits } (\text{UNIV} :: 'a \text{ wide set})$   $(+)$   $(*)$  0 1 ( $\text{Polynomial.coeff } p$ )
    by simp
  then show thesis
    using splits-coeff-iff that by blast
qed

```

83.5 The algebraic closure

The splitting extension is far larger than an algebraic closure — it has an indeterminate for every polynomial and root index, and is not algebraic over the base. Cutting it down to the elements algebraic over the base gives the closure. Nothing has to be proved about that set here: *subfield-algebraic-elements* already says it is a subfield, which is exactly what transporting the extension onto a type bought.

definition *alg-closure* :: 'a :: field wide set
where *alg-closure* = *algebraic-elements* (*range to-wide*)

theorem *subfield-alg-closure*: *Subfield* (*alg-closure* :: 'a :: field wide set)
unfolding *alg-closure-def* **by** (*rule* *subfield-algebraic-elements* [*OF* *subfield-range-to-wide*])

lemma *alg-closure-iff*: $x \in \text{alg-closure} \longleftrightarrow \text{algebraic-over } (\text{range to-wide}) \ x$
by (*simp* *add*: *alg-closure-def* *algebraic-elements-def*)

theorem *base-subset-alg-closure*: $\text{range } (\text{to-wide} :: 'a :: \text{field} \Rightarrow 'a \text{ wide}) \subseteq \text{alg-closure}$
using *Subfield.algebraic-over-self* [*OF* *subfield-range-to-wide*] *alg-closure-iff* **by** *blast*

theorem *algebraic-extension-alg-closure*:
algebraic-extension (*alg-closure* :: 'a :: field wide set) (*range to-wide*)
by (*simp* *add*: *algebraic-extension-def* *alg-closure-iff*)

Two facts about a product of linear factors: it is monic, and its roots are exactly the elements of the list.

lemma *lead-coeff-linear-prod*: $\text{lead-coeff } (\prod a \leftarrow as. [: \text{uminus } a, 1:]) = (1 :: 'b :: \text{idom})$

proof (*induct* *as*)

case *Nil*

show *?case* **by** *simp*

next

case (*Cons* *a as*)

have $\text{lead-coeff } ([: \text{uminus } a, 1:] * (\prod b \leftarrow as. [: \text{uminus } b, 1:])))$

$= \text{lead-coeff } [: \text{uminus } a, 1:] * \text{lead-coeff } (\prod b \leftarrow as. [: \text{uminus } b, 1:])))$

by (*rule* *lead-coeff-mult*)

with *Cons* **show** *?case* **by** *simp*

qed

lemma *poly-linear-prod-eq-0-iff*:
 $\text{poly } (\prod a \leftarrow as. [: \text{uminus } a, 1:]) \ z = (0 :: 'b :: \text{idom}) \longleftrightarrow z \in \text{set } as$
by (*induct* *as*) *auto*

A root of a nonzero polynomial over the base is algebraic over it, so lies in the closure — that polynomial is its own witness.

lemma *alg-closure-root*:
fixes *p* :: 'a :: field wide poly
assumes *p* ∈ *poly-over* (*range to-wide*) **and** *p* ≠ 0 **and** *poly* *p* *z* = 0

shows $z \in \text{alg-closure}$
using *assms* **by** (*auto simp: alg-closure-iff algebraic-over-def*)

Every nonzero polynomial over the base splits, with all of its roots and its leading coefficient inside the closure. Because *splits* carries an explicit list of roots, the descent from the splitting extension to the closure is immediate: the very same product works, every factor being a root of a base polynomial and hence algebraic.

theorem *alg-closure-splits*:
fixes $p :: 'a :: \text{field wide poly}$
assumes $p \in \text{poly-over (range to-wide)}$ **and** $\text{nz: } p \neq 0$
obtains c **as** **where** $c \in \text{alg-closure}$ **and** $\text{set as} \subseteq \text{alg-closure}$
and $p = [:c:] * (\prod a \leftarrow \text{as. } [:uminus\ a,\ 1:])$
proof –
obtain c **as** **where** $\text{peq: } p = [:c:] * (\prod a \leftarrow \text{as. } [:uminus\ a,\ 1:])$
using *wide-splits [OF p]* **by** *blast*
have $c = \text{lead-coeff } p$
by (*simp add: peq lead-coeff-mult lead-coeff-linear-prod*)
then have $c \in \text{range to-wide}$
using *Subfield.lead-coeff-closed [OF subfield-range-to-wide p]* **by** *simp*
then have $cA: c \in \text{alg-closure}$
using *base-subset-alg-closure* **by** *blast*
have $a \in \text{alg-closure}$ **if** $a \in \text{set as}$ **for** a
using *that alg-closure-root [OF p nz]* **by** (*simp add: peq poly-linear-prod-eq-0-iff*)
then have $\text{asA: set as} \subseteq \text{alg-closure}$ **by** *blast*
show *thesis* **by** (*rule that [OF cA asA peq]*)
qed

corollary *poly-root-set-subset-alg-closure*:
fixes $p :: 'a :: \text{field wide poly}$
assumes $p \in \text{poly-over (range to-wide)}$ **and** $p \neq 0$
shows $\text{poly-root-set } p \subseteq \text{alg-closure}$
using *alg-closure-root [OF assms]* **by** (*auto simp: poly-root-set-def*)

The existence statement, in the shape of HOL-Algebra's *exists-closure*: every field has an algebraic extension in which each of its polynomials splits.

theorem *exists-closure*:
obtains $L :: 'a :: \text{field wide set}$
where *Subfield L* **and** $\text{range to-wide} \subseteq L$ **and** *algebraic-extension L (range to-wide)*
and $\bigwedge p. [p \in \text{poly-over (range to-wide)}; p \neq 0] \implies \exists c \text{ as. } c \in L \wedge \text{set as} \subseteq L \wedge p = [:c:] * (\prod a \leftarrow \text{as. } [:uminus\ a,\ 1:])$
by (*metis alg-closure-splits algebraic-extension-alg-closure base-subset-alg-closure subfield-alg-closure*)

83.6 The closure is algebraically closed

A divisor of a split polynomial of positive degree has a root among its factors. Peeling one factor at a time: either it is a root of the divisor, or the divisor

is coprime to it (linear monic polynomials are prime) and the factor can be cancelled.

```

lemma dvd-split-imp-root:
  fixes  $q :: 'b :: \text{field-gcd poly}$ 
  assumes  $q \text{ dvd } [: \gamma:] * (\prod a \leftarrow \text{as}. [: \text{uminus } a, 1:])$  and  $\gamma \neq 0$  and  $0 < \text{degree } q$ 
  shows  $\exists a \in \text{set as}. \text{poly } q \ a = 0$ 
  using assms
proof (induct as arbitrary:  $\gamma$ )
  case Nil
  then have  $[: \gamma:] \neq 0$  by simp
  with Nil have  $\text{degree } q \leq \text{degree } [: \gamma:]$ 
    by (metis dvd-imp-degree-le list.map(1) mult-cancel-left1 prod-list.Nil)
  with Nil.premis(3) show ?case by simp
next
  case (Cons a as)
  then have  $\text{dvd1}: q \text{ dvd } [: \text{uminus } a, 1:] * ([: \gamma:] * (\prod b \leftarrow \text{as}. [: \text{uminus } b, 1:]))$ 
    by simp
  show ?case
  proof (cases poly q a = 0)
    case True
    then show ?thesis by auto
  next
    case False
    then have  $\text{nd}: \neg [: \text{uminus } a, 1:] \text{ dvd } q$ 
      by (simp add: poly-eq-0-iff-dvd)
    have pe: prime-elem  $[: \text{uminus } a, 1 :: 'b:]$ 
      by (rule prime-elem-linear-poly) simp-all
    have copq: coprime  $q [: \text{uminus } a, 1 :: 'b:]$ 
      using coprime-commute prime-elem-imp-coprime [OF pe nd] by blast
    have  $q \text{ dvd } [: \gamma:] * (\prod b \leftarrow \text{as}. [: \text{uminus } b, 1:])$ 
      using dvd1 coprime-dvd-mult-right-iff [OF copq] by blast
    then show ?thesis
      using Cons by auto
  qed
qed

```

Every polynomial over the closure of positive degree has a root there, so the closure is algebraically closed.

This is the one step of the construction that needs a dimension count, and it is HOL-Algebra's rupture argument in elementary form. Normalise the polynomial to be monic and let L be a finite extension of the base containing its coefficients, which exists because those coefficients are algebraic (*exists-finite-subfield-tower*). Divide X^i by the polynomial for $i \leq n * \text{card } B$, where B is a basis of L : each remainder has degree below n and coefficients in L , so it is described by $n * \text{card } B$ scalars from the base field, while there are $n * \text{card } B + 1$ of them. *Subfield.underdetermined-solution* therefore supplies a nontrivial vanishing combination, whose coefficients assemble into a nonzero polynomial *over the base* that the given polynomial divides. That one splits in the closure, and a divisor

of a split polynomial has a root.

Where HOL-Algebra forms the quotient $A[X]/(P)$ and compares its dimension with that of $K[X]$, the linear system does the same counting directly, so no quotient (which would change the type) is ever constructed.

theorem *alg-closure-alg-closed*:

fixes $p :: 'a :: \text{field wide poly}$
assumes $p: p \in \text{poly-over alg-closure}$ **and** $dp: 0 < \text{degree } p$
obtains z **where** $z \in \text{alg-closure}$ **and** $\text{poly } p \ z = 0$

proof —

interpret K : *Subfield range (to-wide :: 'a $\Rightarrow$ 'a wide)* **by** (rule *subfield-range-to-wide*)
interpret A : *Subfield alg-closure :: 'a wide set* **by** (rule *subfield-alg-closure*)

— Normalise to a monic polynomial with the same roots.

define q **where** $q \equiv \text{smult (inverse (lead-coeff } p)) } p$
have $pnz: p \neq 0$ **using** dp **by** *auto*
then have $lc: \text{lead-coeff } p \neq 0$ **by** *simp*
have $lcA: \text{lead-coeff } p \in \text{alg-closure}$ **by** (rule *A.lead-coeff-closed* [OF p])
have $qA: q \in \text{poly-over alg-closure}$
unfolding $q\text{-def}$ **using** p lcA **by** (auto intro: *A.poly-over-smult A.inverse-closed*)
have $dq: \text{degree } q = \text{degree } p$ **by** (simp add: $q\text{-def } lc$)
have $\text{monic: lead-coeff } q = 1$ **by** (simp add: $q\text{-def } lc$ pnz)
have $\text{roots: poly } q \ z = 0 \longleftrightarrow \text{poly } p \ z = 0$ **for** z
using lc **by** (auto simp: $q\text{-def}$)
define n **where** $n \equiv \text{degree } q$
have $nps: 0 < n$ **using** dp dq **by** (simp add: $n\text{-def}$)

— A finite extension of the base containing the coefficients.

define S **where** $S \equiv (\lambda i. \text{coeff } q \ i) \ ' \{..n\}$
have $\text{finS: finite } S$ **by** (simp add: $S\text{-def}$)
have $SA: S \subseteq \text{alg-closure}$ **using** qA **by** (auto simp: $S\text{-def } A.\text{poly-over-coeff}$)
then have $\text{algS: algebraic-over (range to-wide) } x$ **if** $x \in S$ **for** x
using that **by** (auto simp: alg-closure-iff)
obtain L **where** $L: \text{finite-subfield-tower (range (to-wide :: 'a $\Rightarrow$ 'a wide)) } L$
and $KL: \text{range to-wide} \subseteq L$ **and** $SL: S \subseteq L$
using $\text{exists-finite-subfield-tower [OF subfield-range-to-wide finS algS]}$ **by** *blast*
interpret T : *finite-subfield-tower range (to-wide :: 'a $\Rightarrow$ 'a wide) } L* **by** (rule L)
have $\text{sfL: Subfield } L$ **by** (rule $T.\text{ext.Subfield-axioms}$)
have $qL: q \in \text{poly-over } L$
proof (rule *Subfield.poly-overI* [OF sfL])
fix i
show $\text{coeff } q \ i \in L$
proof (cases $i \leq n$)
case *True* **then show** $?thesis$ **using** SL **by** (auto simp: $S\text{-def}$)
next
case *False*
then show $?thesis$ **using** *Subfield.zero-closed* [OF sfL] **by** (simp add: coeff-eq-0 $n\text{-def}$)
qed
qed

— Coordinates with respect to a basis of L over the base.

obtain B *coord* **where** $\text{fin}B$: *finite* B **and** BL : $B \subseteq L$
and $\text{coord}K$: $\bigwedge x. x \in L \implies (\forall b \in B. \text{coord } x \ b \in \text{range } \text{to-wide})$
and $\text{coord}S$: $\bigwedge x. x \in L \implies x = (\sum b \in B. \text{coord } x \ b * b)$
using $T.\text{exists-coordinates}$ **by** blast

— Divide each power of the indeterminate by q .

define N **where** $N \equiv n * \text{card } B$
have $\exists s \ r. s \in \text{poly-over } L \wedge r \in \text{poly-over } L \wedge \text{monom } 1 \ i = s * q + r \wedge \text{degree } r < n$ **for** i
using $T.\text{ext.one-closed } T.\text{ext.poly-over-divmod-exists } T.\text{ext.poly-over-monom monic}$
 $n\text{-def } n\text{pos } qL$
by presburger
then obtain $s \ r$ **where** sr : $\bigwedge i. s \ i \in \text{poly-over } L \wedge r \ i \in \text{poly-over } L$
 $\wedge \text{monom } 1 \ i = s \ i * q + r \ i \wedge \text{degree } (r \ i) < n$
by metis
have rL : $r \ i \in \text{poly-over } L$ **and** $rdeg$: $\text{degree } (r \ i) < n$
and divid : $\text{monom } (1 :: 'a \text{ wide}) \ i = s \ i * q + r \ i$ **for** i
using sr **by** blast+
have $\text{rcoeff}L$: $\text{coeff } (r \ i) \ j \in L$ **for** $i \ j$
using rL **by** $(\text{simp add: Subfield.poly-over-coeff [OF sfL]})$

— More unknowns than equations, so a nontrivial vanishing combination exists.

define I **where** $I \equiv \{..<n\} \times B$
define Amat **where** $\text{Amat} \equiv (\lambda(j, b) \ i. \text{coord } (\text{coeff } (r \ i) \ j) \ b)$
have $\text{fin}I$: *finite* I **by** $(\text{simp add: } I\text{-def } \text{fin}B)$
have $\text{card}I$: $\text{card } I = N$ **by** $(\text{simp add: } I\text{-def } N\text{-def } \text{card-cartesian-product})$
have $\text{card } I < \text{card } \{..N\}$ **by** $(\text{simp add: } \text{card}I)$
moreover have $\text{Amat } e \ i \in \text{range } \text{to-wide}$ **if** $e \in I$ **for** $e \ i$
using $\text{that } \text{coord}K \ [OF \ \text{rcoeff}L]$ **by** $(\text{auto simp: } \text{Amat-def } I\text{-def})$
ultimately obtain c **where** $cKall$: $\forall j. c \ j \in \text{range } (\text{to-wide} :: 'a \Rightarrow 'a \text{ wide})$
and cnz : $\exists j \in \{..N\}. c \ j \neq 0$
and sysall : $\forall e \in I. (\sum i \leq N. \text{Amat } e \ i * c \ i) = 0$
using $K.\text{underdetermined-solution} \ [OF \ \text{fin}I, \text{ of } \{..N\} \ \text{Amat}]$ **by** blast
have cK : $c \ i \in \text{range } (\text{to-wide} :: 'a \Rightarrow 'a \text{ wide})$ **for** i **using** $cKall$ **by** blast
have sys : $(\sum i \leq N. \text{Amat } e \ i * c \ i) = 0$ **if** $e \in I$ **for** e **using** sysall that **by** blast

— The combination of remainders vanishes identically.

define R **where** $R = (\sum i \leq N. \text{smult } (c \ i) \ (r \ i))$
have $Rdeg$: $\text{degree } R < n$
using $rdeg \ R\text{-def}$ **by** $(\text{simp add: } \text{degree-sum-less } n\text{pos})$
have $Rcoeff$: $\text{coeff } R \ j = 0$ **if** $j: j < n$ **for** j
proof —
have $\text{coeff } R \ j = (\sum i \leq N. c \ i * (\sum b \in B. \text{coord } (\text{coeff } (r \ i) \ j) \ b * b))$
using $\text{coord}S \ [OF \ \text{rcoeff}L]$ **by** $(\text{simp add: } R\text{-def } \text{coeff-sum})$
also have $\dots = (\sum i \leq N. \sum b \in B. c \ i * (\text{coord } (\text{coeff } (r \ i) \ j) \ b * b))$
by $(\text{simp add: } \text{sum-distrib-left})$
also have $\dots = (\sum b \in B. \sum i \leq N. c \ i * (\text{coord } (\text{coeff } (r \ i) \ j) \ b * b))$
by $(\text{rule } \text{sum.swap})$

```

also have ... = ( $\sum b \in B. (\sum i \leq N. \text{Amat } (j, b) \ i * c \ i) * b$ )
proof (rule sum.cong [OF refl])
  fix b assume  $b \in B$ 
  have ( $\sum i \leq N. c \ i * (\text{coord } (\text{coeff } (r \ i) \ j) \ b * b)$ ) = ( $\sum i \leq N. \text{Amat } (j, b) \ i * c \ i * b$ )
    by (simp add: Amat-def ac-simps)
  also have ... = ( $\sum i \leq N. \text{Amat } (j, b) \ i * c \ i) * b$ 
    by (simp add: sum-distrib-right)
  finally show ( $\sum i \leq N. c \ i * (\text{coord } (\text{coeff } (r \ i) \ j) \ b * b)$ )
    = ( $\sum i \leq N. \text{Amat } (j, b) \ i * c \ i) * b$  .
qed
also have ... = 0
  using sys j by (intro sum.neutral) (simp add: I-def)
finally show ?thesis .
qed
have  $R = 0$ 
  using Rcoeff Rdeg by fastforce

```

— Hence q divides a nonzero polynomial over the base field.

```

define  $Q$  where  $Q \equiv (\sum i \leq N. \text{monom } (c \ i) \ i)$ 
have  $QK$ :  $Q \in \text{poly-over } (\text{range } (\text{to-wide} :: 'a \Rightarrow 'a \ \text{wide}))$ 
  unfolding  $Q\text{-def}$  using  $cK$  by (auto intro!: K.poly-over-sum K.poly-over-monom)
have  $\text{coeff} Q$ :  $\text{coeff } Q \ i = c \ i$  if  $i \leq N$  for  $i$ 
  using that by (simp add: Q-def coeff-sum)
have  $Qnz$ :  $Q \neq 0$  using cnz coeff Q by force
have  $Q\text{split}$ :  $Q = (\sum i \leq N. \text{smult } (c \ i) \ (s \ i)) * q + R$ 
proof —
  have  $\text{monom } (c \ i) \ i = \text{smult } (c \ i) \ (s \ i) * q + \text{smult } (c \ i) \ (r \ i)$  for  $i$ 
proof —
  have  $\text{monom } (c \ i) \ i = \text{smult } (c \ i) \ (\text{monom } 1 \ i)$  by (simp add: smult-monom)
  also have ... =  $\text{smult } (c \ i) \ (s \ i) * q + \text{smult } (c \ i) \ (r \ i)$ 
    by (simp add: smult-monom divid [of i] smult-add-right mult-smult-left)
  finally show ?thesis .
qed
then have  $Q = (\sum i \leq N. \text{smult } (c \ i) \ (s \ i) * q + \text{smult } (c \ i) \ (r \ i))$ 
  by (simp add: Q-def)
also have ... = ( $\sum i \leq N. \text{smult } (c \ i) \ (s \ i)) * q + R$ 
  by (simp add: R-def sum.distrib sum-distrib-right)
finally show ?thesis .
qed
with  $\langle R = 0 \rangle$  have  $q \text{ dvd } Q$  by auto

```

— That polynomial splits in the closure, so q has a root there.

```

obtain  $\gamma$  as where  $\gamma: \gamma \in \text{alg-closure}$  and  $as: \text{set } as \subseteq \text{alg-closure}$ 
  and  $Qeq: Q = [:\gamma:] * (\prod a \leftarrow as. [:uminus \ a, \ 1:])$ 
  using alg-closure-splits [OF  $QK \ Qnz$ ] by blast
have  $\gamma \neq 0$  using  $Qnz \ Qeq$  by auto
then obtain  $z$  where  $z: z \in \text{set } as$  and  $\text{poly } q \ z = 0$ 
  using dvd-split-imp-root [OF  $q \text{ dvd } [unfolded \ Qeq]$ ] npos by (auto simp: n-def)
then show thesis

```

```

    using as roots by (blast intro: that)
qed

end

```

84 Module homomorphisms

```

theory Module-Homomorphism
  imports Module-Basics
begin

```

A *module homomorphism* (over a fixed ring R) is a map η between R -modules that is additive and commutes with scaling. This mirrors the group and ring homomorphism locales of theories *Group-Theory* and *Ring-Theory*, with the extra axiom linking the two scaling operations. We develop the kernel, image, and the injectivity criterion $Ker = \{0\}$.

The import of *FiniteProduct* (through *Module*) re-merges HOL's $+/-$ concrete syntax alongside the ring locale's, which is harmless in term positions (type inference disambiguates) but ambiguous in a locale-header instantiation argument. We therefore remove the HOL syntax again, as *Ring-Theory* does, so that within this theory $+$ unambiguously denotes the ring's addition.

```

no-notation plus (infixl <+> 65)
no-notation minus (infixl <-> 65)
unbundle no uminus-syntax

```

A module homomorphism fixes a *single* scalar ring R and two R -modules — the source M and the target M' — sharing that ring. The ring parameters $+ \cdot \mathbf{0} \mathbf{1}$ appear once in the *for* clause and are threaded verbatim into both module interpretations, so no token is bound twice.

```

locale module-homomorphism =
  source: Module R (+) (·)  $\mathbf{0} \mathbf{1} (\oplus) \mathbf{0}_M M (\odot) +$ 
  target: Module R (+) (·)  $\mathbf{0} \mathbf{1} (\oplus_2) \mathbf{0}_2 M' (\odot_2) +$ 
  map  $\eta M M'$ 
  for  $R$  and addition (infixl <+> 65) and multiplication (infixl <·> 70) and zero
  (<0>) and unit (<1>)
  and  $M$  and madd (infixl <⊕> 65) and mzero (<0MM' and madd' (infixl <⊕2> 65) and mzero' (<022>
  75)
  and  $\eta +$ 
  assumes hom-add:  $\llbracket u \in M; v \in M \rrbracket \implies \eta (u \oplus v) = \eta u \oplus_2 \eta v$ 
  and hom-scale:  $\llbracket a \in R; v \in M \rrbracket \implies \eta (a \odot v) = a \odot_2 \eta v$ 
begin

```

84.1 Basic properties

```

lemma hom-closed:  $v \in M \implies \eta v \in M'$ 
  using graph by (auto simp: PiE-iff)

```



```

lemma hom-zero:  $\eta \mathbf{0}_M = \mathbf{0}_2$ 
proof -
  have  $\eta \mathbf{0}_M = \eta (\mathbf{0} \odot \mathbf{0}_M)$  using source.scale-zero-scalar by simp
  also have  $\dots = \mathbf{0}_2$ 
    by (simp add: hom-scale target.scale-zero-scalar)
  finally show ?thesis .
qed

lemma hom-neg:
  assumes  $v: v \in M$  shows  $\eta (\text{source.madd.inverse } v) = \text{target.madd.inverse } (\eta v)$ 
proof -
  define  $n$  where  $n = \text{source.additive.inverse } \mathbf{1}$ 
  — The module negation is scaling by the ring's additive inverse of  $\mathbf{1}$ .
  have sneg:  $\text{source.madd.inverse } v = n \odot v$ 
    unfolding n-def using v by (simp add: source.scale-neg-scalar source.scale-one)
  have tneg:  $\text{target.madd.inverse } (\eta v) = n \odot_2 \eta v$ 
    unfolding n-def using hom-closed[OF v] by (simp add: target.scale-neg-scalar target.scale-one)
  show ?thesis
    by (simp add: hom-scale n-def sneg tneg v)
qed

```

A module homomorphism commutes with finite linear combinations. The target sum remains indexed by the source vectors; reindexing it over their images additionally requires injectivity on the chosen index set.

```

lemma hom-lincomb:
  assumes finA: finite A and AM:  $A \subseteq M$ 
  and c:  $\bigwedge v. v \in A \implies c v \in R$ 
  shows  $\eta (\text{source.lincomb } c A) =$ 
     $\text{target.madd.fincomp } (\lambda v. c v \odot_2 \eta v) A$ 
  using finA AM c
proof (induction A rule: finite-induct)
  case empty
  then show ?case unfolding source.lincomb-def by (simp add: hom-zero)
next
  case (insert v A)
  have vM:  $v \in M$  and AM:  $A \subseteq M$  and cv:  $c v \in R$ 
    using insert.prem by auto
  have cA:  $\bigwedge u. u \in A \implies c u \in R$  using insert.prem by auto
  have source-terms:  $(\lambda u. c u \odot u) \in A \rightarrow M$ 
    using AM cA by (auto intro!: source.scale-closed)
  have target-terms:  $(\lambda u. c u \odot_2 \eta u) \in A \rightarrow M'$ 
    using AM cA
    by (auto intro!: target.scale-closed hom-closed)
  have source.lincomb c (insert v A) =  $(c v \odot v) \oplus \text{source.lincomb } c A$ 
    unfolding source.lincomb-def
    using insert.hyps source-terms cv vM by (simp add: source.scale-closed)
  then have  $\eta (\text{source.lincomb } c (\text{insert } v A)) =$ 

```

```

     $\eta (c \ v \odot \ v) \oplus_2 \eta (source.lincomb \ c \ A)$ 
  using cv vM AM cA
  by (simp add: hom-add source.scale-closed source.lincomb-closed)
  also have  $\eta (c \ v \odot \ v) = c \ v \odot_2 \eta \ v$  using cv vM by (rule hom-scale)
  also have  $\eta (source.lincomb \ c \ A) =$ 
    target.madd.fincomp ( $\lambda u. \ c \ u \odot_2 \eta \ u$ ) A
  using insert.IH[OF AM cA] .
  also have  $(c \ v \odot_2 \eta \ v) \oplus_2$ 
    target.madd.fincomp ( $\lambda u. \ c \ u \odot_2 \eta \ u$ ) A =
    target.madd.fincomp ( $\lambda u. \ c \ u \odot_2 \eta \ u$ ) (insert v A)
  using insert.hyps target-terms cv vM
  by (auto intro!: target.madd.fincomp-insert[symmetric] target.scale-closed hom-closed)
  finally show ?case .
qed

```

84.2 Composition and inverses

Composition is restricted to the source carrier, preserving the extensional-map convention used throughout the set-based algebra hierarchy.

```

theorem compose-module-homomorphism:
  assumes second-hom: module-homomorphism R (+) (·) 0 1
     $M' (\oplus_2) \mathbf{0}_2 (\odot_2) N \text{ nadd } nzero \text{ nscale } \vartheta$ 
  shows module-homomorphism R (+) (·) 0 1
     $M (\oplus) \mathbf{0}_M (\odot) N \text{ nadd } nzero \text{ nscale } (compose \ M \ \vartheta \ \eta)$ 
proof -
  interpret second: module-homomorphism R (+) (·) 0 1
     $M' (\oplus_2) \mathbf{0}_2 (\odot_2) N \text{ nadd } nzero \text{ nscale } \vartheta$ 
  by (rule second-hom)
  show ?thesis
proof (intro module-homomorphism.intro map.intro module-homomorphism-axioms.intro)
  show Module R (+) (·) 0 1 ( $\oplus$ )  $\mathbf{0}_M \ M (\odot)$ 
  by (rule source.Module-axioms)
  show Module R (+) (·) 0 1 nadd nzero N nscale
  by (rule second.target.Module-axioms)
  show compose M \vartheta \eta \in M \rightarrow_E N
  using graph second.graph by (auto simp: PiE-iff compose-def)
next
  fix u v assume uv: u \in M v \in M
  then show compose M \vartheta \eta (u \oplus v) =
    nadd (compose M \vartheta \eta u) (compose M \vartheta \eta v)
  by (simp add: compose-def hom-add second.hom-add hom-closed)
next
  fix a v assume av: a \in R v \in M
  then show compose M \vartheta \eta (a \odot v) =
    nscale a (compose M \vartheta \eta v)
  by (simp add: compose-def hom-scale second.hom-scale hom-closed source.scale-closed)
qed
qed

```

The inverse of a bijective homomorphism is again restricted to its source carrier. This is essential: a raw *inv-into* does not satisfy the repository's extensional-map convention away from the carrier.

```

theorem inverse-module-homomorphism:
  assumes bij: bij-betw  $\eta$   $M$   $M'$ 
  shows module-homomorphism  $R$   $(+)$   $(\cdot)$  0 1
     $M' (\oplus_2) \mathbf{0}_2 (\odot_2)$ 
     $M (\oplus) \mathbf{0}_M (\odot) (\text{restrict } (\text{inv-into } M \ \eta) \ M')$ 
proof –
  interpret iso: bijective-map  $\eta$   $M$   $M'$ 
    by unfold-locales (rule bij)
  show ?thesis
proof (intro module-homomorphism.intro map.intro module-homomorphism-axioms.intro)
  show Module  $R$   $(+)$   $(\cdot)$  0 1  $(\oplus_2) \mathbf{0}_2 \ M' (\odot_2)$ 
    by (rule target.Module-axioms)
  show Module  $R$   $(+)$   $(\cdot)$  0 1  $(\oplus) \mathbf{0}_M \ M (\odot)$ 
    by (rule source.Module-axioms)
  show restrict  $(\text{inv-into } M \ \eta) \ M' \in M' \rightarrow_E M$ 
    by (rule iso.inverse.graph)
next
  fix  $x \ y$  assume  $xy$ :  $x \in M' \ y \in M'$ 
  then obtain  $u \ v$  where  $uv$ :  $u \in M \ v \in M \ \eta \ u = x \ \eta \ v = y$ 
    using bij-betw-imp-surj-on[OF bij] by blast
  then show restrict  $(\text{inv-into } M \ \eta) \ M' (x \oplus_2 y) =$ 
    restrict  $(\text{inv-into } M \ \eta) \ M' x \oplus$  restrict  $(\text{inv-into } M \ \eta) \ M' y$ 
    using  $xy$  bij by (auto simp: inv-into-f-eq hom-add)
next
  fix  $a \ x$  assume  $ax$ :  $a \in R \ x \in M'$ 
  then obtain  $v$  where  $v$ :  $v \in M \ \eta \ v = x$ 
    using bij-betw-imp-surj-on[OF bij] by blast
  have  $av$ :  $a \odot v \in M$  using  $ax(1) \ v(1)$  by (rule source.scale-closed)
  have image:  $\eta (a \odot v) = a \odot_2 x$ 
    using  $ax \ v$  by (simp add: hom-scale)
  have inverse-image:  $\text{inv-into } M \ \eta (a \odot_2 x) = a \odot v$ 
    using  $bij \ av \ \text{image}$  by (metis bij-betw-inv-into-left)
  have inverse-x:  $\text{inv-into } M \ \eta \ x = v$ 
    using  $bij \ v$  by (metis bij-betw-inv-into-left)
  have target-ax:  $a \odot_2 x \in M'$ 
    using  $ax$  by (rule target.scale-closed)
  show restrict  $(\text{inv-into } M \ \eta) \ M' (a \odot_2 x) =$ 
     $a \odot$  restrict  $(\text{inv-into } M \ \eta) \ M' x$ 
    using  $ax$  inverse-image inverse-x target-ax by simp
qed
qed

lemma inverse-module-homomorphism-left:
  assumes bij: bij-betw  $\eta$   $M$   $M'$  and  $x$ :  $x \in M$ 
  shows restrict  $(\text{inv-into } M \ \eta) \ M' (\eta \ x) = x$ 
  using  $bij \ x$  hom-closed by (simp add: bij-betw-inv-into-left)

```

lemma *inverse-module-homomorphism-right*:
assumes *bij*: *bij-betw* η M M' **and** $y: y \in M'$
shows η (*restrict* (*inv-into* M η) M' y) = y
using *bij y* **by** (*simp add: bij-betw-inv-into-right*)

corollary *inverse-module-isomorphism*:
assumes *bij*: *bij-betw* η M M'
shows *module-homomorphism* R $(+)$ $(\cdot)$ $\mathbf{0}$ $\mathbf{1}$
 $M' (\oplus_2) \mathbf{0}_2 (\odot_2)$
 $M (\oplus) \mathbf{0}_M (\odot) (\text{restrict } (\text{inv-into } M \ \eta) \ M')$
 $\wedge$ *bij-betw* (*restrict* (*inv-into* M η) M') $M' M$
using *bij inverse-module-homomorphism* **by** (*simp add: bij-betw-inv-into*)

84.3 The kernel

definition *Ker* :: 'b set
where *Ker* = $\{v \in M. \eta \ v = \mathbf{0}_2\}$

lemma *Ker-subset*: $Ker \subseteq M$
by (*auto simp: Ker-def*)

lemma *Ker-mem*: $v \in Ker \implies v \in M$
by (*simp add: Ker-def*)

lemma *Ker-image*: $v \in Ker \implies \eta \ v = \mathbf{0}_2$
by (*simp add: Ker-def*)

lemma *Ker-memI*: $\llbracket \eta \ v = \mathbf{0}_2; v \in M \rrbracket \implies v \in Ker$
by (*simp add: Ker-def*)

The kernel is a submodule of the source.

theorem *Ker-submodule*: *source.submodule* *Ker*

proof (*intro source.submoduleI Ker-subset*)

show $\mathbf{0}_M \in Ker$ **by** (*simp add: Ker-def hom-zero*)

next

fix $u \ v$ **assume** $u: u \in Ker$ **and** $v: v \in Ker$

have $uM: u \in M$ **and** $vM: v \in M$ **using** $u \ v$ **by** (*auto simp: Ker-def*)

show $u \oplus v \in Ker$

by (*simp add: Ker-image Ker-memI hom-add u uM v vM*)

next

show $\bigwedge a \ v. \llbracket a \in R; v \in Ker \rrbracket \implies a \odot v \in Ker$

using *Ker-def hom-scale source.scale-closed target.scale-zero-elem* **by** *auto*

qed

84.4 The image

The image of η is a submodule of the target.

theorem *image-submodule*: *target.submodule* ($\eta \text{ `` } M$)

```

proof (rule target.submoduleI)
  show  $\bigwedge u v. \llbracket u \in \eta \text{ ' } M; v \in \eta \text{ ' } M \rrbracket \implies u \oplus_2 v \in \eta \text{ ' } M$ 
    by (smt (verit, ccfv-threshold) hom-add image-iff source.madd-closed)
  show  $\bigwedge a v. \llbracket a \in R; v \in \eta \text{ ' } M \rrbracket \implies a \odot_2 v \in \eta \text{ ' } M$ 
    by (metis hom-scale imageE image-subset-iff source.scale-closed subset-refl)
qed (use hom-closed hom-zero in blast)+

```

84.5 Injectivity criterion

A module homomorphism is injective iff its kernel is trivial.

theorem *injective-iff-kernel-trivial*:

$\text{inj-on } \eta \text{ } M \longleftrightarrow \text{Ker} = \{\mathbf{0}_M\}$

proof

assume *inj*: $\text{inj-on } \eta \text{ } M$

show $\text{Ker} = \{\mathbf{0}_M\}$

using *hom-zero inj inj-on-contrad* **by** (fastforce simp: *Ker-def hom-zero*)

next

assume *ker*: $\text{Ker} = \{\mathbf{0}_M\}$

show $\text{inj-on } \eta \text{ } M$

proof (rule *inj-onI*)

fix *x y* **assume** *xy*: $x \in M \text{ } y \in M$ **and** *eq*: $\eta \text{ } x = \eta \text{ } y$

have *diff*: $x \oplus \text{source.madd.inverse } y \in M$ **using** *xy* **by** *auto*

have $\eta \text{ } (x \oplus \text{source.madd.inverse } y) = \eta \text{ } x \oplus_2 \eta \text{ } (\text{source.madd.inverse } y)$

using *xy* **by** (simp *add*: *hom-add*)

also have $\dots = \mathbf{0}_2$

using *xy eq* **by** (simp *add*: *hom-neg*)

finally have *z*: $x \oplus \text{source.madd.inverse } y = \mathbf{0}_M$

using *Ker-memI diff ker* **by** *blast*

— Cancel: $x \oplus (-y) = \mathbf{0}_M$ forces $x = y$.

have $\text{source.madd.inverse } y \oplus y = \mathbf{0}_M$

using *xy(2)* **by** (simp *add*: *source.madd.invertible-left-inverse*)

then show $x = y$

using *source.madd.inverse-unique xy z* **by** *blast*

qed

qed

end

context *Module*

begin

The carrier-restricted identity is the identity morphism in the extensional-map representation.

theorem *identity-module-homomorphism*:

module-homomorphism $R \text{ } (+) \text{ } (\cdot) \text{ } \mathbf{0} \text{ } \mathbf{1}$

$M \text{ } (\oplus) \text{ } \mathbf{0}_M \text{ } (\odot) \text{ } M \text{ } (\oplus) \text{ } \mathbf{0}_M \text{ } (\odot) \text{ } (\text{identity } M)$

proof (*intro module-homomorphism.intro map.intro module-homomorphism-axioms.intro*)

show *Module* $R \text{ } (+) \text{ } (\cdot) \text{ } \mathbf{0} \text{ } \mathbf{1} \text{ } (\oplus) \text{ } \mathbf{0}_M \text{ } M \text{ } (\odot)$

by (rule *Module-axioms*)

```

show Module  $R$   $(+)$   $(\cdot)$   $\mathbf{0}$   $\mathbf{1}$   $(\oplus)$   $\mathbf{0}_M$   $M$   $(\odot)$ 
  by (rule Module-axioms)
show identity  $M \in M \rightarrow_E M$  by simp
next
  fix  $u\ v$  assume  $u \in M\ v \in M$ 
  then show identity  $M$   $(u \oplus v) = \text{identity } M\ u \oplus \text{identity } M\ v$ 
    by (simp add: madd-closed)
next
  fix  $a\ v$  assume  $a \in R\ v \in M$ 
  then show identity  $M$   $(a \odot v) = a \odot \text{identity } M\ v$ 
    by (simp add: scale-closed)
qed

end

end

```

85 Quotient modules and the first isomorphism theorem

```

theory Quotient-Module
  imports Module-Homomorphism
begin

```

Given a submodule N of an R -module M , the quotient set M/N inherits a module structure with scaling $a \cdot \text{Class}(v) = \text{Class}(a \odot v)$. The construction reuses the group-quotient machinery of *Group-Theory* for the additive structure (any subgroup of an abelian group is normal, so cosets already form an abelian group under coset addition) and only requires the induced scaling to be shown well-defined.

On this base we prove the natural projection to be a module homomorphism with kernel N , and derive the first isomorphism theorem: for a module homomorphism η , the induced map on $M/\text{Ker } \eta$ is an isomorphism onto the image $\eta \text{ `` } M$.

85.1 The quotient module of a submodule

Work inside the *Module* locale, with a submodule fixed.

```

locale submodule-in-module = Module +
  fixes  $N :: 'b\ \text{set}$ 
  assumes  $N\text{-submodule: submodule } N$ 
begin

```

The submodule is an additive subgroup of the module. Since the additive group is abelian, it is automatically normal.

```

lemma  $N\text{-subgroup: Subgroup } N\ M\ (\oplus)\ \mathbf{0}_M$ 
proof (rule madd.subgroupI)

```

```

show  $N \subseteq M$  using submodule-subset[OF N-submodule] .
show  $\mathbf{0}_M \in N$  using submodule-zero[OF N-submodule] .
next
  fix  $g\ h$  assume  $g \in N$  and  $h \in N$ 
  then show  $g \oplus h \in N$  using submodule-add[OF N-submodule] by blast
next
  fix  $g$  assume  $gN: g \in N$ 
  then have  $g \in M$  using submodule-subset[OF N-submodule] by blast
  then show madd.invertible  $g$  by simp
  show madd.inverse  $g \in N$  using submodule-neg[OF N-submodule  $gN$ ] .
qed

```

```

sublocale madd-sub: subgroup-of-abelian-group  $N\ M\ (\oplus)\ \mathbf{0}_M$ 
proof –
  interpret  $S$ : Subgroup  $N\ M\ (\oplus)\ \mathbf{0}_M$  by (rule N-subgroup)
  show subgroup-of-abelian-group  $N\ M\ (\oplus)\ \mathbf{0}_M$  ..
qed

```

Notation: the quotient carrier, the coset class, and the induced coset addition.

```

abbreviation Qcarrier :: 'b set set ( $\langle M' / N \rangle$ )
  where  $M / N \equiv$  madd-sub.Factor-Group

```

```

abbreviation Qclass :: 'b  $\Rightarrow$  'b set
  where Qclass  $v \equiv$  madd-sub.Class  $v$ 

```

```

notation madd-sub.quotient-composition (infixl  $\langle [\oplus] \rangle$  65)

```

An explicit constant for the coset addition, so it can be referred to by name from outside the locale (in **sublocale**-free consumers).

```

abbreviation qadd :: 'b set  $\Rightarrow$  'b set  $\Rightarrow$  'b set
  where qadd  $\equiv$  madd-sub.quotient-composition

```

The additive coset structure is already a (commutative) group, inherited from *subgroup-of-abelian-group*.

```

lemma qmodule-additive: Abelian-Group  $(M / N)\ ([\oplus])\ (Qclass\ \mathbf{0}_M)$ 
  by (rule madd-sub.quotient-abelian)

```

85.1.1 The induced scaling is well-defined

Two module elements lie in the same coset iff their difference lies in N .

```

lemma Class-eq-iff-diff-in-N:
  assumes  $u: u \in M$  and  $v: v \in M$ 
  shows Qclass  $u = Qclass\ v \iff u \oplus madd.inverse\ v \in N$ 
proof –
  have raw: Qclass  $u = Qclass\ v \iff madd.inverse\ u \oplus v \in N$ 
    using  $u\ v$  madd-sub.Congruence-def madd-sub.Class-equivalence by auto
  — In an abelian group  $N$  is closed under negation, and negation exchanges the two forms.

```

```

have iff: madd.inverse u  $\oplus$  v  $\in$  N  $\longleftrightarrow$  u  $\oplus$  madd.inverse v  $\in$  N
proof
  assume madd.inverse u  $\oplus$  v  $\in$  N
  then have madd.inverse (madd.inverse u  $\oplus$  v)  $\in$  N
    using N-submodule submodule-neg by blast
  also have madd.inverse (madd.inverse u  $\oplus$  v) = madd.inverse v  $\oplus$  u
    using u v by (simp add: madd.inverse-composition-commute)
  also have ... = u  $\oplus$  madd.inverse v using u v by (simp add: madd.commutative)
  finally show u  $\oplus$  madd.inverse v  $\in$  N .
next
  assume u  $\oplus$  madd.inverse v  $\in$  N
  then have madd.inverse (u  $\oplus$  madd.inverse v)  $\in$  N
    using N-submodule submodule-neg by blast
  also have madd.inverse (u  $\oplus$  madd.inverse v) = v  $\oplus$  madd.inverse u
    using u v by (simp add: madd.inverse-composition-commute)
  also have ... = madd.inverse u  $\oplus$  v using u v by (simp add: madd.commutative)
  finally show madd.inverse u  $\oplus$  v  $\in$  N .
qed
from raw iff show ?thesis by simp
qed

```

The scaling is well-defined on cosets: if $u - v \in N$ (in the module additive sense) then $a \odot u - a \odot v \in N$, since $a \odot (u \oplus \text{madd.inverse } v)$ equals $(a \odot u) \oplus (a \odot \text{madd.inverse } v) = (a \odot u) \oplus \text{madd.inverse } (a \odot v)$ and lies in N by scaling-closedness of the submodule.

lemma *scale-respects-Class*:

```

assumes a: a  $\in$  R and uv: Qclass u = Qclass v and u: u  $\in$  M and v: v  $\in$  M
shows Qclass (a  $\odot$  u) = Qclass (a  $\odot$  v)
proof -
  from uv u v have diff: u  $\oplus$  madd.inverse v  $\in$  N
    using Class-eq-iff-diff-in-N by blast
  have a  $\odot$  (u  $\oplus$  madd.inverse v) = (a  $\odot$  u)  $\oplus$  (a  $\odot$  madd.inverse v)
    using a u v by (simp add: scale-distrib-madd)
  also have a  $\odot$  madd.inverse v = madd.inverse (a  $\odot$  v)
    proof -
      — madd.inverse v = (- 1)  $\odot$  v in any module; then use scale-scale and scale-neg-scalar
      on the scalar side.
      have inv-as-scale: madd.inverse v = (- 1)  $\odot$  v
        using v by (simp add: scale-neg-scalar scale-one)
      have a  $\odot$  madd.inverse v = a  $\odot$  ((- 1)  $\odot$  v) by (simp add: inv-as-scale)
      also have ... = (a  $\cdot$  (- 1))  $\odot$  v using a v by (simp add: scale-scale)
      also have a  $\cdot$  (- 1) = - a using a by (simp add: right-minus)
      also have (- a)  $\odot$  v = madd.inverse (a  $\odot$  v) using a v by (rule scale-neg-scalar)
      finally show ?thesis .
    qed
  finally have expand: a  $\odot$  (u  $\oplus$  madd.inverse v) = (a  $\odot$  u)  $\oplus$  madd.inverse (a  $\odot$ 
  v) .
  have scaled-diff-in-N: a  $\odot$  (u  $\oplus$  madd.inverse v)  $\in$  N
    by (rule submodule-scale[OF N-submodule a diff])

```



```

from scaled-diff-in-N have  $(a \odot u) \oplus \text{madd.inverse } (a \odot v) \in N$  by (simp add:
expand)
moreover have  $a \odot u \in M$  using  $a \ u$  by (rule scale-closed)
moreover have  $a \odot v \in M$  using  $a \ v$  by (rule scale-closed)
ultimately show ?thesis by (simp add: Class-eq-iff-diff-in-N)
qed

```

85.1.2 The quotient scaling operation and the quotient module

Choose any representative and scale it; well-definedness (*scale-respects-Class*) makes the choice immaterial.

```

definition qscale :: ' $a \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}$ ' (infixr  $\langle [\odot] \rangle$  75)
where  $a \ [\odot] \ X = (\text{THE } Y. \exists v \in M. X = \text{Qclass } v \wedge Y = \text{Qclass } (a \odot v))$ 

```

```

lemma qscale-Class:
assumes  $a: a \in R$  and  $v: v \in M$ 
shows  $a \ [\odot] \ \text{Qclass } v = \text{Qclass } (a \odot v)$ 
proof –
have  $(\text{THE } Y. \exists u \in M. \text{Qclass } v = \text{Qclass } u \wedge Y = \text{Qclass } (a \odot u)) = \text{Qclass } (a \odot v)$ 
proof (rule the-equality)
show  $\exists u \in M. \text{Qclass } v = \text{Qclass } u \wedge \text{Qclass } (a \odot v) = \text{Qclass } (a \odot u)$ 
using  $v$  by blast
next
fix  $Y$  assume  $\exists u \in M. \text{Qclass } v = \text{Qclass } u \wedge Y = \text{Qclass } (a \odot u)$ 
then obtain  $u$  where  $u: u \in M \ \text{Qclass } v = \text{Qclass } u \ Y = \text{Qclass } (a \odot u)$  by
blast
have  $\text{Qclass } (a \odot v) = \text{Qclass } (a \odot u)$ 
by (rule scale-respects-Class[OF a u(2) v u(1)])
then show  $Y = \text{Qclass } (a \odot v)$  using  $u(3)$  by simp
qed
then show ?thesis unfolding qscale-def by simp
qed

```

```

lemma qscale-closed [intro, simp]:
assumes  $a: a \in R$  and  $X: X \in M/N$  shows  $a \ [\odot] \ X \in M/N$ 
proof –
from  $X$  obtain  $v$  where  $v: v \in M \ X = \text{Qclass } v$  using madd-sub.representant-exists
by auto
have  $avM: a \odot v \in M$  using  $a \ v(1)$  by (rule scale-closed)
then have  $\text{Qclass } (a \odot v) \in M/N$  by (rule madd-sub.Class-in-Partition)
then show ?thesis using  $a \ v$  by (simp add: qscale-Class)
qed

```

The six module axioms on the quotient, packaged as a *Module-axioms* fact. This is the pinch point: proving *Module-axioms* separately, and then composing with the ambient *Ring-axioms* via *Module.intro*, avoids the fragile mid-interpretation goal-order handshake we would face by opening a bare **sublocale** on *Module*.

lemma *quotient-module-axioms*:

```

Module-axioms R (+) (·) 1 ([⊕]) (Qclass 0M) (M/N) ([⊙])
proof (rule Module-axioms.intro)
  show Abelian-Group (M/N) ([⊕]) (Qclass 0M) by (rule qmodule-additive)
next
  fix a X assume a: a ∈ R and X: X ∈ M/N show a [⊙] X ∈ M/N
    using a X by (rule qscale-closed)
next
  fix a X Y assume a: a ∈ R and X: X ∈ M/N and Y: Y ∈ M/N
  from X obtain u where u: u ∈ M X = Qclass u using madd-sub.representant-exists
by auto
  from Y obtain v where v: v ∈ M Y = Qclass v using madd-sub.representant-exists
by auto
  have a [⊙] (X [⊕] Y) = a [⊙] Qclass (u ⊕ v)
    using u v by (simp add: madd-sub.Class-commutes-with-composition)
  also have ... = Qclass (a ⊙ (u ⊕ v))
    using a u v by (simp add: qscale-Class)
  also have ... = Qclass ((a ⊙ u) ⊕ (a ⊙ v))
    using a u v by (simp add: scale-distrib-madd)
  also have ... = Qclass (a ⊙ u) [⊕] Qclass (a ⊙ v)
    using a u v by (simp add: madd-sub.Class-commutes-with-composition scale-closed)
  also have ... = (a [⊙] X) [⊕] (a [⊙] Y)
    using a u v by (simp add: qscale-Class)
  finally show a [⊙] (X [⊕] Y) = (a [⊙] X) [⊕] (a [⊙] Y) .
next
  fix a b X assume a: a ∈ R and b: b ∈ R and X: X ∈ M/N
  from X obtain v where v: v ∈ M X = Qclass v using madd-sub.representant-exists
by auto
  have (a + b) [⊙] X = Qclass ((a + b) ⊙ v) using a b v by (simp add: qscale-Class)
  also have ... = Qclass ((a ⊙ v) ⊕ (b ⊙ v)) using a b v by (simp add: scale-distrib-add)
  also have ... = Qclass (a ⊙ v) [⊕] Qclass (b ⊙ v)
    using a b v by (simp add: madd-sub.Class-commutes-with-composition scale-closed)
  also have ... = (a [⊙] X) [⊕] (b [⊙] X) using a b v by (simp add: qscale-Class)
  finally show (a + b) [⊙] X = (a [⊙] X) [⊕] (b [⊙] X) .
next
  fix a b X assume a: a ∈ R and b: b ∈ R and X: X ∈ M/N
  from X obtain v where v: v ∈ M X = Qclass v using madd-sub.representant-exists
by auto
  have (a · b) [⊙] X = Qclass ((a · b) ⊙ v) using a b v by (simp add: qscale-Class)
  also have ... = Qclass (a ⊙ (b ⊙ v)) using a b v by (simp add: scale-scale)
  also have ... = a [⊙] Qclass (b ⊙ v) using a b v by (simp add: qscale-Class
scale-closed)
  also have ... = a [⊙] (b [⊙] X) using b v by (simp add: qscale-Class)
  finally show (a · b) [⊙] X = a [⊙] (b [⊙] X) .
next
  fix X assume X: X ∈ M/N
  from X obtain v where v: v ∈ M X = Qclass v using madd-sub.representant-exists
by auto
  show 1 [⊙] X = X using v by (simp add: qscale-Class scale-one)
qed

```

sublocale *quotient*: $Module\ R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}\ ([\oplus])\ Qclass\ \mathbf{0}_M\ M/N\ ([\odot])$
by (*rule* $Module.intro[OF\ Ring-axioms\ quotient-module-axioms]$)

85.1.3 The natural projection

The natural projection $v \mapsto Qclass\ v$ extensional on M .

definition $nat-proj :: 'b \Rightarrow 'b\ set$
where $nat-proj = (\lambda v \in M. Qclass\ v)$

lemma $nat-proj-apply\ [simp]: v \in M \implies nat-proj\ v = Qclass\ v$
unfolding $nat-proj-def$ **by** $simp$

lemma $nat-proj-undefined: v \notin M \implies nat-proj\ v = undefined$
unfolding $nat-proj-def$ **by** $simp$

The natural projection is a module homomorphism with kernel N .

sublocale $nat-proj$: *module-homomorphism*
 $R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}$
 $M\ (\oplus)\ \mathbf{0}_M\ (\odot)$
 $M/N\ ([\oplus])\ Qclass\ \mathbf{0}_M\ ([\odot])$
 $nat-proj$

proof

show $nat-proj \in M \rightarrow_E M/N$

unfolding $nat-proj-def$ **by** (*rule* $PiE-I$) (*auto* $simp$: *extensional-def*)

next

fix $u\ v$ **assume** $u \in M\ v \in M$

then show $nat-proj\ (u \oplus v) = nat-proj\ u\ [\oplus]\ nat-proj\ v$
by (*simp* add : *madd-sub.Class-commutes-with-composition*)

next

fix $a\ v$ **assume** $a \in R\ v \in M$

then show $nat-proj\ (a \odot v) = a\ [\odot]\ nat-proj\ v$
by (*simp* add : *qscale-Class scale-closed*)

qed

The natural projection maps onto the entire quotient carrier.

theorem $nat-proj-image: nat-proj\ ` M = M/N$

proof

show $nat-proj\ ` M \subseteq M/N$ **using** $nat-proj.hom-closed$ **by** $blast$

next

show $M/N \subseteq nat-proj\ ` M$

proof

fix X **assume** $X: X \in M/N$

then obtain v **where** $v: v \in M\ X = Qclass\ v$

using $madd-sub.representant-exists$ **by** $blast$

then have $X = nat-proj\ v$ **by** $simp$

then show $X \in nat-proj\ ` M$ **using** $v(1)$ **by** $blast$

qed

qed

The kernel of the natural projection is exactly N .

```

theorem nat-proj-Ker: nat-proj.Ker = N
proof
  show nat-proj.Ker  $\subseteq$  N
  proof
    fix v assume vK: v  $\in$  nat-proj.Ker
    then have vM: v  $\in$  M using nat-proj.Ker-mem by simp
    from vK have nat-proj v = Qclass 0_M using nat-proj.Ker-image by simp
    with vM have Q: Qclass v = Qclass 0_M by simp
    from Q vM have v  $\oplus$  madd.inverse 0_M  $\in$  N
      using Class-eq-iff-diff-in-N by simp
    then show v  $\in$  N using vM by (simp add: madd.inverse-unit)
  qed
next
  show N  $\subseteq$  nat-proj.Ker
  proof
    fix v assume vN: v  $\in$  N
    then have vM: v  $\in$  M using N-submodule submodule-subset by blast
    have v  $\oplus$  madd.inverse 0_M  $\in$  N using vN vM by (simp add: madd.inverse-unit)
    then have Qclass v = Qclass 0_M using vM by (simp add: Class-eq-iff-diff-in-N)
    then have nat-proj v = Qclass 0_M using vM by simp
    then show v  $\in$  nat-proj.Ker using vM nat-proj.Ker-memI by simp
  qed
qed
end
end

```

86 The first isomorphism theorem for modules

```

theory Module-First-Iso
  imports Quotient-Module
begin

```

Given a module homomorphism $\eta \in M \rightarrow M'$, its kernel is a submodule of M , and the induced map on the quotient $M/\text{Ker } \eta$ onto $\eta^{-1} M$ is a module isomorphism.

The natural way to organise this development inside *module-homomorphism* is to prove once and for all that the kernel gives rise to a *submodule-in-module* instance, and then use the operations $K.Qcarrier$, $K.Qclass$, $K.qadd$, $K.qscale$ supplied by that instance in the theorems below. We do this via **interpretation** K : *submodule-in-module* ... *Ker* at the head of the **context** block, discharged by the lemma *Ker-submodule-in-module* below.

```

context module-homomorphism
begin

```

86.1 The kernel-quotient and the induced map

The kernel is a submodule (*Ker-submodule*), so the *submodule-in-module* locale holds on it. This lemma is the witness fed to the **interpretation** below.

lemma *Ker-submodule-in-module*:

submodule-in-module R $(+)$ $(\cdot)$ $\mathbf{0}$ $\mathbf{1}$ $(\oplus)$ $\mathbf{0}_M$ M $(\odot)$ Ker

proof

show *source.submodule* Ker **by** (*rule Ker-submodule*)

qed

Install *submodule-in-module* on the kernel with prefix K . The theorems below then use $K.Qcarrier$, $K.Qclass$, $K.qadd$, $K.qscale$, and other facts derived from the locale ($K.Class\text{-}eq\text{-}iff\text{-}diff\text{-}in\text{-}N$, $K.qscale\text{-}Class$, $K.quotient.Module\text{-}axioms$, $K.madd\text{-}sub.representant\text{-}exists$, $K.madd\text{-}sub.Class\text{-}commutes\text{-}with\text{-}composition$).

interpretation K : *submodule-in-module*

R $(+)$ $(\cdot)$ $\mathbf{0}$ $\mathbf{1}$ $(\oplus)$ $\mathbf{0}_M$ M $(\odot)$ Ker

by (*rule Ker-submodule-in-module*)

The induced map on cosets. Made extensional on $K.Qcarrier$ so that it fits the *map* locale in the *module-homomorphism* obligation.

definition *ind* :: ' b set $\Rightarrow$ ' c

where *ind* = ($\lambda X \in K.Qcarrier. THE y. \exists v \in M. X = K.Qclass v \wedge y = \eta v$)

lemma *ind-Class*:

assumes $v: v \in M$ **shows** *ind* ($K.Qclass v$) = ηv

proof –

have ($THE y. \exists u \in M. K.Qclass v = K.Qclass u \wedge y = \eta u$) = ηv

proof (*rule the-equality*)

show $\exists u \in M. K.Qclass v = K.Qclass u \wedge \eta v = \eta u$ **using** v **by** *blast*

next

fix y **assume** $\exists u \in M. K.Qclass v = K.Qclass u \wedge y = \eta u$

then obtain u **where** $u: u \in M$ $K.Qclass v = K.Qclass u$ $y = \eta u$ **by** *blast*

have $v \oplus source.madd.inverse u \in Ker$

using $u(2)$ v $u(1)$ **by** (*subst K.Class-eq-iff-diff-in-N[symmetric]*) *simp-all*

then have $\eta (v \oplus source.madd.inverse u) = \mathbf{0}_2$

using *Ker-image* **by** *blast*

moreover have $iuM: source.madd.inverse u \in M$ **using** $u(1)$ **by** *simp*

ultimately have $\eta v \oplus_2 \eta (source.madd.inverse u) = \mathbf{0}_2$

using v $u(1)$ **by** (*simp add: hom-add*)

then have *eq*: $\eta v \oplus_2 target.madd.inverse (\eta u) = \mathbf{0}_2$

using $u(1)$ **by** (*simp add: hom-neg*)

have $\eta u \oplus_2 target.madd.inverse (\eta u) = \mathbf{0}_2$

using *hom-closed[OF u(1)]* **by** (*simp add: target.madd.invertible-right-inverse*)

with *eq* **have** $\eta v \oplus_2 target.madd.inverse (\eta u) = \eta u \oplus_2 target.madd.inverse (\eta$

$u)$

by *simp*

then have $\eta v = \eta u$

using *hom-closed[OF v]* *hom-closed[OF u(1)]* *target.madd.invertible-right-cancel*

by (*metis target.madd.invertible target.madd.invertible-inverse-closed*)

```

    then show  $y = \eta \ v$  using  $u(\beta)$  by simp
  qed
  moreover have  $K.Qclass \ v \in K.Qcarrier$  using  $v$  by simp
  ultimately show ?thesis unfolding ind-def by simp
qed

```

```

lemma ind-closed:
  assumes  $X \in K.Qcarrier$  shows  $ind \ X \in \eta \ ' \ M$ 
proof -
  from assms obtain  $v$  where  $v \in M \ X = K.Qclass \ v$ 
  using  $K.madd-sub.representant-exists$  by auto
  then show ?thesis by (simp add: ind-Class)
qed

```

86.2 The image $\eta \ ' \ M$ is a submodule of the target

```

lemma image-abelian-group: Abelian-Group  $(\eta \ ' \ M) \ (\oplus_2) \ 0_2$ 
proof -
  have sub: Subgroup  $(\eta \ ' \ M) \ M' \ (\oplus_2) \ 0_2$ 
  proof (rule target.madd.subgroupI)
    show  $\eta \ ' \ M \subseteq M'$  using hom-closed by auto
    show  $0_2 \in \eta \ ' \ M$  using hom-zero source.mzero-closed by (metis image-eqI)
  next
    fix  $u \ v$  assume  $u \in \eta \ ' \ M \ v \in \eta \ ' \ M$ 
    then obtain  $a \ b$  where  $ab$ :  $a \in M \ b \in M \ u = \eta \ a \ v = \eta \ b$  by auto
    then have  $u \oplus_2 \ v = \eta \ (a \oplus \ b)$  by (simp add: hom-add)
    then show  $u \oplus_2 \ v \in \eta \ ' \ M$  using ab by auto
  next
    fix  $u$  assume  $u \in \eta \ ' \ M$  then obtain  $a$  where  $a$ :  $a \in M \ u = \eta \ a$  by auto
    then have  $u \in M'$  using hom-closed by auto
    then show target.madd.invertible  $u$  by simp
    have target.madd.inverse  $u = \eta \ (source.madd.inverse \ a)$ 
      using  $a$  by (simp add: hom-neg source.madd.invertible)
    then show target.madd.inverse  $u \in \eta \ ' \ M$  using  $a$  by auto
  qed
  interpret Sub: Subgroup  $\eta \ ' \ M \ M' \ (\oplus_2) \ 0_2$  by (rule sub)
  show ?thesis
  proof (rule Abelian-Group.intro)
    show Group  $(\eta \ ' \ M) \ (\oplus_2) \ 0_2$  by (rule Sub.sub.Group-axioms)
    show commutative-monoid  $(\eta \ ' \ M) \ (\oplus_2) \ 0_2$ 
      by unfold-locale (auto simp: target.madd.commutative)
  qed
qed

```

```

lemma image-module-axioms:
  Module-axioms  $R \ (+) \ (\cdot) \ 1 \ (\oplus_2) \ 0_2 \ (\eta \ ' \ M) \ (\odot_2)$ 
proof (rule Module-axioms.intro)
  show Abelian-Group  $(\eta \ ' \ M) \ (\oplus_2) \ 0_2$  by (rule image-abelian-group)
next
  fix  $a \ v$  assume  $a$ :  $a \in R$  and  $v$ :  $v \in \eta \ ' \ M$ 

```

```

then obtain x where x: x ∈ M v = η x by auto
have a ⊙2 v = η (a ⊙ x) using a x by (simp add: hom-scale)
then show a ⊙2 v ∈ η ' M using a x by (auto intro: source.scale-closed)
next
fix a u v assume a: a ∈ R and u ∈ η ' M and v ∈ η ' M
then obtain x y where x ∈ M y ∈ M u = η x v = η y by auto
with a show a ⊙2 (u ⊕2 v) = (a ⊙2 u) ⊕2 (a ⊙2 v)
  by (simp add: target.scale-distrib-madd hom-closed)
next
fix a b v assume a: a ∈ R and b: b ∈ R and v: v ∈ η ' M
then obtain x where x ∈ M v = η x by auto
with a b show (a + b) ⊙2 v = (a ⊙2 v) ⊕2 (b ⊙2 v)
  by (simp add: target.scale-distrib-add hom-closed)
next
fix a b v assume a: a ∈ R and b: b ∈ R and v: v ∈ η ' M
then obtain x where x ∈ M v = η x by auto
with a b show (a · b) ⊙2 v = a ⊙2 b ⊙2 v
  by (simp add: target.scale-scale hom-closed)
next
fix v assume v ∈ η ' M
then obtain x where x ∈ M v = η x by auto
then show 1 ⊙2 v = v by (simp add: target.scale-one hom-closed)
qed

```

lemma *image-is-module*: *Module R (+) (·) 0 1 (⊕₂) 0₂ (η ' M) (⊙₂)*
 by (rule *Module.intro*[*OF source.Ring-axioms image-module-axioms*])

86.3 The induced map is a module homomorphism, injective, surjective

The quotient module on the kernel is a module (specialisation of the quotient's own *Module-axioms* to the kernel-as-submodule).

lemma *quotient-is-module*:
Module R (+) (·) 0 1 K.qadd (K.Qclass 0_M) K.Qcarrier K.qscale
 by (rule *K.quotient.Module-axioms*)

theorem *ind-module-homomorphism*:
module-homomorphism R (+) (·) 0 1
K.Qcarrier K.qadd (K.Qclass 0_M) K.qscale
(η ' M) (⊕₂) 0₂ (⊙₂) ind
proof (*intro module-homomorphism.intro map.intro module-homomorphism-axioms.intro*)
 show *Module R (+) (·) 0 1 K.qadd (K.Qclass 0_M) K.Qcarrier K.qscale*
 by (rule *quotient-is-module*)
 show *Module R (+) (·) 0 1 (⊕₂) 0₂ (η ' M) (⊙₂)*
 by (rule *image-is-module*)
 show *ind ∈ K.Qcarrier →_E η ' M*
 using *ind-closed* by (auto simp: *ind-def PiE-iff extensional-def*)
next
fix U V assume U ∈ K.Qcarrier and V ∈ K.Qcarrier

then obtain $u \ v$ **where** $uv: u \in M \ v \in M$
 $U = K.Qclass \ u \ V = K.Qclass \ v$
using $K.madd-sub.representant-exists$ **by** (*metis (mono-tags, lifting)*)
have $ind \ (K.qadd \ U \ V) = ind \ (K.Qclass \ (u \oplus \ v))$
using uv **by** (*simp add: K.madd-sub.Class-commutes-with-composition*)
also have $\dots = \eta \ (u \oplus \ v)$ **using** uv **by** (*simp add: ind-Class*)
also have $\dots = \eta \ u \oplus_2 \eta \ v$ **using** uv **by** (*simp add: hom-add*)
also have $\dots = (ind \ U) \oplus_2 \ (ind \ V)$ **using** uv **by** (*simp add: ind-Class*)
finally show $ind \ (K.qadd \ U \ V) = (ind \ U) \oplus_2 \ (ind \ V)$.
next
fix $a \ V$ **assume** $a: a \in R$ **and** $V \in K.Qcarrier$
then obtain v **where** $v: v \in M \ V = K.Qclass \ v$
using $K.madd-sub.representant-exists$ **by** (*metis (mono-tags, lifting)*)
have $ind \ (K.qscale \ a \ V) = ind \ (K.Qclass \ (a \odot \ v))$
using $a \ v$ **by** (*simp add: K.qscale-Class*)
also have $\dots = \eta \ (a \odot \ v)$ **using** $a \ v$ **by** (*simp add: ind-Class source.scale-closed*)
also have $\dots = a \odot_2 \eta \ v$ **using** $a \ v$ **by** (*simp add: hom-scale*)
also have $\dots = a \odot_2 \ ind \ V$ **using** v **by** (*simp add: ind-Class*)
finally show $ind \ (K.qscale \ a \ V) = a \odot_2 \ ind \ V$.
qed

theorem *ind-inj: inj-on ind K.Qcarrier*
proof (*rule inj-onI*)
fix $U \ V$ **assume** $U: U \in K.Qcarrier$ **and** $V: V \in K.Qcarrier$ **and** $eq: ind \ U = ind \ V$
from U **obtain** u **where** $u: u \in M \ U = K.Qclass \ u$
using $K.madd-sub.representant-exists$ **by** (*metis (mono-tags, lifting)*)
from V **obtain** v **where** $v: v \in M \ V = K.Qclass \ v$
using $K.madd-sub.representant-exists$ **by** (*metis (mono-tags, lifting)*)
from $eq \ u \ v$ **have** $h: \eta \ u = \eta \ v$ **by** (*simp add: ind-Class*)
have $\eta \ (u \oplus \ source.madd.inverse \ v) = \eta \ u \oplus_2 \eta \ (source.madd.inverse \ v)$
using $u \ v$ **by** (*simp add: hom-add*)
also have $\dots = \eta \ v \oplus_2 \ target.madd.inverse \ (\eta \ v)$ **using** $u \ v \ h$ **by** (*simp add: hom-neg*)
also have $\dots = \mathbf{0}_2$
using *hom-closed[OF v(1)]* **by** (*simp add: target.madd.invertible-right-inverse*)
finally have $\eta \ (u \oplus \ source.madd.inverse \ v) = \mathbf{0}_2$.
moreover have $u \oplus \ source.madd.inverse \ v \in M$ **using** $u \ v$ **by** *simp*
ultimately have $u \oplus \ source.madd.inverse \ v \in Ker$ **by** (*rule Ker-memI*)
then have $K.Qclass \ u = K.Qclass \ v$
using $u \ v$ **by** (*simp add: K.Class-eq-iff-diff-in-N*)
then show $U = V$ **using** $u \ v$ **by** *simp*
qed

theorem *ind-surj: ind ‘ K.Qcarrier = η ‘ M*
proof
show $ind \ ‘ K.Qcarrier \subseteq \eta \ ‘ M$ **using** *ind-closed* **by** *auto*
show $\eta \ ‘ M \subseteq ind \ ‘ K.Qcarrier$


```

proof
  fix  $y$  assume  $y \in \eta \text{ ' } M$ 
  then obtain  $v$  where  $v: v \in M \ y = \eta \ v$  by auto
  then have  $y = \text{ind } (K.Qclass \ v)$  by (simp add: ind-Class)
  moreover have  $K.Qclass \ v \in K.Qcarrier$  using  $v$  by simp
  ultimately show  $y \in \text{ind ' } K.Qcarrier$  by auto
qed
qed

```

The first isomorphism theorem: the induced map *ind* is a bijective module homomorphism from $M/\text{Ker } \eta$ onto $\eta \text{ ' } M$.

```

theorem first-isomorphism:
  module-homomorphism  $R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}$ 
     $K.Qcarrier \ K.qadd \ (K.Qclass \ \mathbf{0}_M) \ K.qscale$ 
     $(\eta \text{ ' } M) \ (\oplus_2) \ \mathbf{0}_2 \ (\odot_2) \ \text{ind}$ 
     $\wedge \text{bij-betw } \text{ind } K.Qcarrier \ (\eta \text{ ' } M)$ 
  using ind-module-homomorphism ind-inj ind-surj by (auto simp: bij-betw-def)

```

end

end

87 The second and third isomorphism theorems for modules

The third isomorphism theorem for modules: for submodules $K \subseteq N \subseteq M$, $(M/K)/(N/K) \cong M/N$; and the second: for submodules N and K , $(N \oplus_S K)/K \cong N/(N \cap K)$, where $\oplus_S$ is the submodule sum defined in *Module*.

Performance. This theory loads in a fraction of a second. Earlier drafts of this comment claimed that the two proof-local **interpret...**: *submodule-in-module* ... calls per theorem were prohibitively expensive, and that this was why the theory was kept out of *ROOT*. That was a misdiagnosis: the long build times seen at the time were machine contention, not locale instantiation cost. There is nothing here to optimise.

```

theory Module-Iso-Theorems
  imports Module-First-Iso
begin

```

Building on the first isomorphism theorem in *Module-First-Iso*, we derive the third: given submodules $K \subseteq N \subseteq M$, the natural map $v + K \mapsto v + N$ is a surjective module homomorphism from M/K onto M/N with kernel N/K ; by the first iso, $(M/K) / (N/K) \cong M/N$.

Note on the two interprets. Each theorem below interprets *submodule-in-module* twice inside its own proof, once for K and once for N . This is cheap and needs no refactoring. A previous version of this note asserted that the alternative — two **interpretations** at the head of the enclosing **context** *Module* block —

triggers a non-terminating locale roundup. That claim was never soundly established (it shares an origin with the contention misdiagnosis above) and should be re-measured before being repeated or acted upon.

context *Module*
begin

The natural map $v + K \mapsto v + N$ from M/K to M/N . Made extensional on M/K by construction so that the *map* obligation of *module-homomorphism* is directly discharged.

definition *proj-KN* :: '*b set* $\Rightarrow$ '*b set* $\Rightarrow$ '*b set* $\Rightarrow$ '*b set*
where *proj-KN* $K\ N =$
 $(\lambda A \in \text{submodule-in-module.Qcarrier } (\oplus) \ \mathbf{0}_M\ M\ K.$
 $\text{THE } B. \exists v \in M. A = \text{submodule-in-module.Qclass } (\oplus) \ \mathbf{0}_M\ M\ K\ v$
 $\wedge B = \text{submodule-in-module.Qclass } (\oplus) \ \mathbf{0}_M\ M\ N\ v)$

Well-definedness of the projection: two representatives in the same K -coset lie in the same N -coset, because $K \subseteq N$. This is the workhorse fact for everything below, so we prove it once here rather than inside each theorem.

lemma *proj-KN-Class*:

assumes *N-sub*: *submodule* N **and** *K-sub*: *submodule* K **and** *K-in-N*: $K \subseteq N$ **and**
v: $v \in M$

shows *proj-KN* $K\ N$ (*submodule-in-module.Qclass* $(\oplus) \ \mathbf{0}_M\ M\ K\ v$)
 $= \text{submodule-in-module.Qclass } (\oplus) \ \mathbf{0}_M\ M\ N\ v$

proof –

interpret *K*: *submodule-in-module* $R\ (+)\ (\cdot) \ \mathbf{0} \ \mathbf{1} \ (\oplus) \ \mathbf{0}_M\ M\ (\odot) \ K$
using *K-sub* **by** *unfold-locales*

interpret *N*: *submodule-in-module* $R\ (+)\ (\cdot) \ \mathbf{0} \ \mathbf{1} \ (\oplus) \ \mathbf{0}_M\ M\ (\odot) \ N$
using *N-sub* **by** *unfold-locales*

have (*THE* $B. \exists u \in M. K.\text{Qclass } v = K.\text{Qclass } u \wedge B = N.\text{Qclass } u$) $= N.\text{Qclass } v$

proof (*rule the-equality*)

show $\exists u \in M. K.\text{Qclass } v = K.\text{Qclass } u \wedge N.\text{Qclass } v = N.\text{Qclass } u$ **using** *v* **by**

blast

next

fix B **assume** $\exists u \in M. K.\text{Qclass } v = K.\text{Qclass } u \wedge B = N.\text{Qclass } u$

then obtain u **where** $u: u \in M\ K.\text{Qclass } v = K.\text{Qclass } u\ B = N.\text{Qclass } u$ **by**

blast

from $u(2)\ v\ u(1)$ **have** $v \oplus \text{madd.inverse } u \in K$

by (*simp add: K.Class-eq-iff-diff-in-N*)

with *K-in-N* **have** $v \oplus \text{madd.inverse } u \in N$ **by** *blast*

then have $N.\text{Qclass } v = N.\text{Qclass } u$ **using** $v\ u(1)$ **by** (*simp add: N.Class-eq-iff-diff-in-N*)

with $u(3)$ **show** $B = N.\text{Qclass } v$ **by** *simp*

qed

moreover have $K.\text{Qclass } v \in K.\text{Qcarrier}$ **using** *v* **by** *simp*

ultimately show *?thesis* **unfolding** *proj-KN-def* **by** *simp*

qed

The main theorem: for submodules K, N of M with $K \subseteq N$, the map *proj-KN*

$K \cdot N$ is a surjective module homomorphism from M/K onto M/N whose kernel is $K \cdot Qclass \text{ ` } N$ (which is N/K viewed inside M/K).

theorem *third-isomorphism-setup:*

assumes $N\text{-sub}$: submodule N

and $K\text{-sub}$: submodule K

and $K\text{-in-}N$: $K \subseteq N$

shows module-homomorphism $R (+) (\cdot) \mathbf{0} \mathbf{1}$

(submodule-in-module.Qcarrier $(\oplus) \mathbf{0}_M M K$)

(submodule-in-module.qadd $(\oplus) \mathbf{0}_M M K$)

(submodule-in-module.Qclass $(\oplus) \mathbf{0}_M M K \mathbf{0}_M$)

(submodule-in-module.qscale $(\oplus) \mathbf{0}_M M (\odot) K$)

(submodule-in-module.Qcarrier $(\oplus) \mathbf{0}_M M N$)

(submodule-in-module.qadd $(\oplus) \mathbf{0}_M M N$)

(submodule-in-module.Qclass $(\oplus) \mathbf{0}_M M N \mathbf{0}_M$)

(submodule-in-module.qscale $(\oplus) \mathbf{0}_M M (\odot) N$)

(proj-KN $K N$)

(is module-homomorphism - - - - ?QK ?qaddK ?zK ?qscaleK ?QN ?qaddN

?zN ?qscaleN -)

proof –

interpret K : submodule-in-module $R (+) (\cdot) \mathbf{0} \mathbf{1} (\oplus) \mathbf{0}_M M (\odot) K$

using $K\text{-sub}$ **by** *unfold-locales*

interpret N : submodule-in-module $R (+) (\cdot) \mathbf{0} \mathbf{1} (\oplus) \mathbf{0}_M M (\odot) N$

using $N\text{-sub}$ **by** *unfold-locales*

note $\text{proj-Class} = \text{proj-KN-Class}[OF\ N\text{-sub}\ K\text{-sub}\ K\text{-in-}N]$

have proj-closed : $\text{proj-KN}\ K\ N \in K.Qcarrier \rightarrow_E N.Qcarrier$

proof (rule *PiE-I*)

fix A **assume** $A \in K.Qcarrier$

then obtain v **where** $v: v \in M\ A = K.Qclass\ v$ **using** $K.madd\text{-sub.representant-exists}$

by *auto*

then have $\text{proj-KN}\ K\ N\ A = N.Qclass\ v$ **using** proj-Class **by** *simp*

moreover have $N.Qclass\ v \in N.Qcarrier$ **using** v **by** *simp*

ultimately show $\text{proj-KN}\ K\ N\ A \in N.Qcarrier$ **by** *simp*

qed (*auto simp: proj-KN-def*)

show *?thesis*

proof (*intro module-homomorphism.intro map.intro module-homomorphism-axioms.intro*)

show $\text{Module}\ R (+) (\cdot) \mathbf{0} \mathbf{1} ?qaddK ?zK ?QK ?qscaleK$ **by** (rule $K.\text{quotient.Module-axioms}$)

show $\text{Module}\ R (+) (\cdot) \mathbf{0} \mathbf{1} ?qaddN ?zN ?QN ?qscaleN$ **by** (rule $N.\text{quotient.Module-axioms}$)

show $\text{proj-KN}\ K\ N \in ?QK \rightarrow_E ?QN$ **by** (rule proj-closed)

next

fix $A\ B$ **assume** $A \in ?QK$ **and** $B \in ?QK$

then obtain $a\ b$ **where** $ab: a \in M\ b \in M\ A = K.Qclass\ a\ B = K.Qclass\ b$

using $K.madd\text{-sub.representant-exists}$ **by** (*metis (mono-tags, lifting)*)

have $\text{proj-KN}\ K\ N\ (K.qadd\ A\ B) = \text{proj-KN}\ K\ N\ (K.Qclass\ (a \oplus b))$

using ab **by** (*simp add: K.madd-sub.Class-commutes-with-composition*)

also have $\dots = N.Qclass\ (a \oplus b)$ **using** ab **by** (*simp add: proj-Class*)

also have $\dots = N.qadd\ (N.Qclass\ a)\ (N.Qclass\ b)$

```

    using ab by (simp add: N.madd-sub.Class-commutes-with-composition)
    also have ... = N.qadd (proj-KN K N A) (proj-KN K N B) using ab by (simp
add: proj-Class)
    finally show proj-KN K N (K.qadd A B) = N.qadd (proj-KN K N A) (proj-KN
K N B) .
next
  fix a A assume a: a ∈ R and A ∈ ?QK
  then obtain v where v: v ∈ M A = K.Qclass v using K.madd-sub.representant-exists
  by (metis (mono-tags, lifting))
  have proj-KN K N (K.qscale a A) = proj-KN K N (K.Qclass (a ⊙ v))
  using a v by (simp add: K.qscale-Class)
  also have ... = N.Qclass (a ⊙ v) using a v by (simp add: proj-Class scale-closed)
  also have ... = N.qscale a (N.Qclass v) using a v by (simp add: N.qscale-Class)
  also have ... = N.qscale a (proj-KN K N A) using v by (simp add: proj-Class)
  finally show proj-KN K N (K.qscale a A) = N.qscale a (proj-KN K N A) .
qed
qed

theorem third-isomorphism-surj:
  assumes N-sub: submodule N
  and K-sub: submodule K
  and K-in-N: K ⊆ N
  shows proj-KN K N ' (submodule-in-module.Qcarrier (⊕) 0_M M K)
    = submodule-in-module.Qcarrier (⊕) 0_M M N
proof -
  interpret K: submodule-in-module R (+) (·) 0 1 (⊕) 0_M M (⊙) K
  using K-sub by unfold-locales
  interpret N: submodule-in-module R (+) (·) 0 1 (⊕) 0_M M (⊙) N
  using N-sub by unfold-locales
  note proj-Class = proj-KN-Class[OF N-sub K-sub K-in-N]
  show ?thesis
  proof
    show proj-KN K N ' K.Qcarrier ⊆ N.Qcarrier
    proof
      fix Y assume Y ∈ proj-KN K N ' K.Qcarrier
      then obtain A where A: A ∈ K.Qcarrier Y = proj-KN K N A by auto
      from A(1) obtain v where v: v ∈ M A = K.Qclass v using K.madd-sub.representant-exists
      by auto
      with A(2) proj-Class[OF v(1)] have Y = N.Qclass v by simp
      then show Y ∈ N.Qcarrier using v(1) by simp
    qed
    show N.Qcarrier ⊆ proj-KN K N ' K.Qcarrier
    proof
      fix Y assume Y ∈ N.Qcarrier
      then obtain v where v: v ∈ M Y = N.Qclass v using N.madd-sub.representant-exists
      by auto
      then have Y = proj-KN K N (K.Qclass v) using proj-Class by simp
      moreover have K.Qclass v ∈ K.Qcarrier using v(1) by simp
      ultimately show Y ∈ proj-KN K N ' K.Qcarrier by auto
    qed
  qed

```

qed
qed

87.1 The kernel of the projection is N/K

The kernel of $\text{proj-KN } K \ N$ consists of exactly those K -cosets whose representatives lie in N — that is, of N/K viewed as a subset of M/K . Note the kernel is computed with respect to the zero of M/N , namely $N.Qclass \ \mathbf{0}_M$.

theorem *phi-Ker*:

assumes $N\text{-sub}$: *submodule* N

and $K\text{-sub}$: *submodule* K

and $K\text{-in-}N$: $K \subseteq N$

shows $\{A \in \text{submodule-in-module}.Qcarrier \ (\oplus) \ \mathbf{0}_M \ M \ K.$
 $\text{proj-KN } K \ N \ A = \text{submodule-in-module}.Qclass \ (\oplus) \ \mathbf{0}_M \ M \ N \ \mathbf{0}_M\}$
 $= \text{submodule-in-module}.Qclass \ (\oplus) \ \mathbf{0}_M \ M \ K \ ' \ N$

proof —

interpret K : *submodule-in-module* $R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \ (\oplus) \ \mathbf{0}_M \ M \ (\odot) \ K$
using $K\text{-sub}$ **by** *unfold-locales*

interpret N : *submodule-in-module* $R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \ (\oplus) \ \mathbf{0}_M \ M \ (\odot) \ N$
using $N\text{-sub}$ **by** *unfold-locales*

note $\text{proj-Class} = \text{proj-KN-Class}[OF \ N\text{-sub} \ K\text{-sub} \ K\text{-in-}N]$

have $N\text{-in-}M$: $N \subseteq M$ **using** *submodule-subset[OF N-sub]* .

show *?thesis*

proof

show $\{A \in K.Qcarrier. \text{proj-KN } K \ N \ A = N.Qclass \ \mathbf{0}_M\} \subseteq K.Qclass \ ' \ N$

proof

fix A **assume** A : $A \in \{A \in K.Qcarrier. \text{proj-KN } K \ N \ A = N.Qclass \ \mathbf{0}_M\}$

then obtain v **where** v : $v \in M \ A = K.Qclass \ v$

using $K.madd\text{-sub}.representant\text{-exists}$ **by** *auto*

from A **have** $\text{proj-KN } K \ N \ A = N.Qclass \ \mathbf{0}_M$ **by** *simp*

with v proj-Class **have** $N.Qclass \ v = N.Qclass \ \mathbf{0}_M$ **by** *simp*

— A coset equals the zero coset exactly when its representative lies in the submodule — this is *submodule-in-module.Class-eq-iff-diff-in-N* with $\mathbf{0}_M$, whose inverse is itself.

then have $v \oplus madd.inverse \ \mathbf{0}_M \in N$

using $v(1)$ **by** (*simp add: N.Class-eq-iff-diff-in-N*)

then have $v \in N$ **using** $v(1)$ **by** (*simp add: madd.inverse-unit*)

with $v(2)$ **show** $A \in K.Qclass \ ' \ N$ **by** *blast*

qed

show $K.Qclass \ ' \ N \subseteq \{A \in K.Qcarrier. \text{proj-KN } K \ N \ A = N.Qclass \ \mathbf{0}_M\}$

proof

fix A **assume** $A \in K.Qclass \ ' \ N$

then obtain v **where** v : $v \in N \ A = K.Qclass \ v$ **by** *blast*

then have vM : $v \in M$ **using** $N\text{-in-}M$ **by** *blast*

have $A \in K.Qcarrier$ **using** $v(2)$ vM **by** *simp*

moreover have $\text{proj-KN } K \ N \ A = N.Qclass \ \mathbf{0}_M$

proof —

have $v \oplus madd.inverse \ \mathbf{0}_M \in N$ **using** $v(1)$ vM **by** (*simp add: madd.inverse-unit*)

then have $N.Qclass \ v = N.Qclass \ \mathbf{0}_M$ **using** vM **by** (*simp add: N.Class-eq-iff-diff-in-N*)

```

    with v(2) vM proj-Class show ?thesis by simp
  qed
  ultimately show A ∈ {A ∈ K.Qcarrier. proj-KN K N A = N.Qclass 0_M} by
simp
  qed
  qed
  qed

```

87.2 The third isomorphism theorem

Assembling the pieces: $\text{proj-KN } K \ N$ is a module homomorphism (*third-isomorphism-setup*) from M/K onto M/N (*third-isomorphism-surj*) with kernel N/K (*phi-Ker*). Applying the first isomorphism theorem to it yields $(M/K)/(N/K) \cong M/N$.

We state the conclusion as the *existence* of an isomorphism rather than by naming the induced map explicitly. Naming it would mean writing out *module-homomorphism.ind* applied to the iterated-quotient operations, which is a long and fragile parameter list; existence is the standard textbook phrasing and is what consumers actually use.

theorem *third-isomorphism:*

assumes *N-sub: submodule N*

and *K-sub: submodule K*

and *K-in-N: K ⊆ N*

shows $\exists \varphi. \text{ module-homomorphism } R \ (+) \ (\cdot) \ 0 \ 1$

```

(submodule-in-module.Qcarrier
  (submodule-in-module.qadd (⊕) 0_M M K)
  (submodule-in-module.Qclass (⊕) 0_M M K 0_M)
  (submodule-in-module.Qcarrier (⊕) 0_M M K)
  (submodule-in-module.Qclass (⊕) 0_M M K ' N))
(submodule-in-module.qadd
  (submodule-in-module.qadd (⊕) 0_M M K)
  (submodule-in-module.Qclass (⊕) 0_M M K 0_M)
  (submodule-in-module.Qcarrier (⊕) 0_M M K)
  (submodule-in-module.Qclass (⊕) 0_M M K ' N))
(submodule-in-module.Qclass
  (submodule-in-module.qadd (⊕) 0_M M K)
  (submodule-in-module.Qclass (⊕) 0_M M K 0_M)
  (submodule-in-module.Qcarrier (⊕) 0_M M K)
  (submodule-in-module.Qclass (⊕) 0_M M K ' N)
  (submodule-in-module.Qclass (⊕) 0_M M K 0_M))
(submodule-in-module.qscale
  (submodule-in-module.qadd (⊕) 0_M M K)
  (submodule-in-module.Qclass (⊕) 0_M M K 0_M)
  (submodule-in-module.Qcarrier (⊕) 0_M M K)
  (submodule-in-module.qscale (⊕) 0_M M (⊙) K)
  (submodule-in-module.Qclass (⊕) 0_M M K ' N))
(submodule-in-module.Qcarrier (⊕) 0_M M N)
(submodule-in-module.qadd (⊕) 0_M M N)
(submodule-in-module.Qclass (⊕) 0_M M N 0_M)

```

$$\begin{aligned}
& (submodule-in-module.qscale \ (\oplus) \ \mathbf{0}_M \ M \ (\odot) \ N) \\
& \quad \varphi \\
& \wedge \text{bij-betw } \varphi \\
& \quad (submodule-in-module.Qcarrier \\
& \quad \quad (submodule-in-module.qadd \ (\oplus) \ \mathbf{0}_M \ M \ K) \\
& \quad \quad (submodule-in-module.Qclass \ (\oplus) \ \mathbf{0}_M \ M \ K \ \mathbf{0}_M) \\
& \quad \quad (submodule-in-module.Qcarrier \ (\oplus) \ \mathbf{0}_M \ M \ K) \\
& \quad \quad (submodule-in-module.Qclass \ (\oplus) \ \mathbf{0}_M \ M \ K \ 'N)) \\
& \quad (submodule-in-module.Qcarrier \ (\oplus) \ \mathbf{0}_M \ M \ N)
\end{aligned}$$

proof —

— Interpret the projection as a module homomorphism; its own first isomorphism theorem then supplies the induced map, once we rewrite its kernel to N/K and its image to M/N .

```

interpret P: module-homomorphism R (+) (·) 0 1
  submodule-in-module.Qcarrier (⊕) 0_M M K
  submodule-in-module.qadd (⊕) 0_M M K
  submodule-in-module.Qclass (⊕) 0_M M K 0_M
  submodule-in-module.qscale (⊕) 0_M M (⊙) K
  submodule-in-module.Qcarrier (⊕) 0_M M N
  submodule-in-module.qadd (⊕) 0_M M N
  submodule-in-module.Qclass (⊕) 0_M M N 0_M
  submodule-in-module.qscale (⊕) 0_M M (⊙) N
  proj-KN K N
by (rule third-isomorphism-setup[OF N-sub K-sub K-in-N])
have Ker-eq: P.Ker = submodule-in-module.Qclass (⊕) 0_M M K ' N
unfolding P.Ker-def using phi-Ker[OF N-sub K-sub K-in-N] by simp
have image-eq: proj-KN K N ' submodule-in-module.Qcarrier (⊕) 0_M M K
  = submodule-in-module.Qcarrier (⊕) 0_M M N
by (rule third-isomorphism-surj[OF N-sub K-sub K-in-N])
show ?thesis
using P.first-isomorphism unfolding Ker-eq image-eq by blast
qed

```

87.3 The second isomorphism theorem

For submodules N and K , the second isomorphism theorem states $(N \oplus_S K)/K \cong N/(N \cap K)$.

The proof restricts the natural projection $M \rightarrow M/K$ to N . A submodule is itself a module, so the restriction is a module homomorphism from N to M/K ; its image is $(N \oplus_S K)/K$ and its kernel is $N \cap K$. The first isomorphism theorem applied to it then gives the result.

As with the third isomorphism theorem, the conclusion is stated as the *existence* of an isomorphism rather than by naming the induced map, whose argument list would otherwise have to be written out in full.

A submodule, viewed as a module in its own right.

lemma *submodule-is-module*:

assumes N : submodule N **shows** Module $R \ (+) \ (\cdot) \ 0 \ 1 \ (\oplus) \ \mathbf{0}_M \ N \ (\odot)$

```

proof (rule Module.intro[OF Ring-axioms], rule Module-axioms.intro)
  show Abelian-Group  $N \oplus \mathbf{0}_M$ 
  proof (rule Abelian-Group.intro)
    interpret S: Subgroup  $N \oplus \mathbf{0}_M$ 
    proof (rule madd.subgroupI)
      show  $N \subseteq M$  using submodule-subset[OF N] .
      show  $\mathbf{0}_M \in N$  using submodule-zero[OF N] .
      show  $\bigwedge g h. \llbracket g \in N; h \in N \rrbracket \implies g \oplus h \in N$  using submodule-add[OF N] by
        blast
      show  $\bigwedge g. g \in N \implies \text{madd.invertible } g$ 
        using submodule-subset[OF N] by auto
      show  $\bigwedge g. g \in N \implies \text{madd.inverse } g \in N$  using submodule-neg[OF N] by blast
    qed
    show Group  $N \oplus \mathbf{0}_M$  by (rule S.sub.Group-axioms)
    show commutative-monoid  $N \oplus \mathbf{0}_M$ 
      using submodule-subset[OF N] by unfold-locals (auto simp: madd.commutative)
    qed
  next
    show  $\bigwedge a v. \llbracket a \in R; v \in N \rrbracket \implies a \odot v \in N$  using submodule-scale[OF N] by
      blast
  next
    fix  $a u v$  assume  $a \in R u \in N v \in N$ 
    then show  $a \odot (u \oplus v) = (a \odot u) \oplus (a \odot v)$ 
      using submodule-subset[OF N] by (auto intro: scale-distrib-madd)
  next
    fix  $a b v$  assume  $a \in R b \in R v \in N$ 
    then show  $(a + b) \odot v = (a \odot v) \oplus (b \odot v)$ 
      using submodule-subset[OF N] by (auto intro: scale-distrib-add)
  next
    fix  $a b v$  assume  $a \in R b \in R v \in N$ 
    then show  $(a \cdot b) \odot v = a \odot (b \odot v)$ 
      using submodule-subset[OF N] by (auto intro: scale-scale)
  next
    fix  $v$  assume  $v \in N$ 
    then show  $\mathbf{1} \odot v = v$  using submodule-subset[OF N] by (auto intro: scale-one)
  qed

```

The natural projection $M \rightarrow M/K$ restricted to N .

definition *proj-res* :: 'b set $\Rightarrow$ 'b set $\Rightarrow$ 'b $\Rightarrow$ 'b set
where *proj-res* $N K = (\lambda v \in N. \text{submodule-in-module.Qclass } (\oplus) \mathbf{0}_M M K v)$

lemma *proj-res-apply* [*simp*]:
 $v \in N \implies \text{proj-res } N K v = \text{submodule-in-module.Qclass } (\oplus) \mathbf{0}_M M K v$
unfolding *proj-res-def* **by** *simp*

The restricted projection is a module homomorphism from N to M/K .

lemma *proj-res-hom*:
assumes N : *submodule* N **and** K : *submodule* K
shows *module-homomorphism* $R (+) (\cdot) \mathbf{0} \mathbf{1}$


```

      N (⊕) 0M (⊙)
      (submodule-in-module.Qcarrier (⊕) 0M M K)
      (submodule-in-module.qadd (⊕) 0M M K)
      (submodule-in-module.Qclass (⊕) 0M M K 0M)
      (submodule-in-module.qscale (⊕) 0M M (⊙) K)
      (proj-res N K)
proof (intro module-homomorphism.intro map.intro module-homomorphism-axioms.intro)
interpret K: submodule-in-module R (+) (·) 0 1 (⊕) 0M M (⊙) K
  using K by unfold-locales
have NM: N ⊆ M using submodule-subset[OF N] .
show Module R (+) (·) 0 1 (⊕) 0M N (⊙) by (rule submodule-is-module[OF N])
show Module R (+) (·) 0 1 K.qadd (K.Qclass 0M) K.Qcarrier K.qscale
  by (rule K.quotient.Module-axioms)
show proj-res N K ∈ N →E K.Qcarrier
  unfolding proj-res-def using NM by (rule-tac PiE-I) (auto simp: extensional-def)
next
interpret K: submodule-in-module R (+) (·) 0 1 (⊕) 0M M (⊙) K
  using K by unfold-locales
fix u v assume uv: u ∈ N v ∈ N
then have uvM: u ∈ M v ∈ M using submodule-subset[OF N] by auto
have u ⊕ v ∈ N using submodule-add[OF N uv(1) uv(2)] .
then show proj-res N K (u ⊕ v) = K.qadd (proj-res N K u) (proj-res N K v)
  using uv uvM by (simp add: K.madd-sub.Class-commutes-with-composition)
next
interpret K: submodule-in-module R (+) (·) 0 1 (⊕) 0M M (⊙) K
  using K by unfold-locales
fix a v assume av: a ∈ R v ∈ N
then have vM: v ∈ M using submodule-subset[OF N] by blast
have a ⊙ v ∈ N using submodule-scale[OF N av(1) av(2)] .
then show proj-res N K (a ⊙ v) = K.qscale a (proj-res N K v)
  using av vM by (simp add: K.qscale-Class)
qed

Its kernel is  $N \cap K$ : a coset  $v + K$  with  $v \in N$  is zero exactly when  $v \in K$ .

lemma proj-res-Ker:
  assumes N: submodule N and K: submodule K
  shows {v ∈ N. proj-res N K v = submodule-in-module.Qclass (⊕) 0M M K 0M} =
    N ∩ K
proof -
interpret K: submodule-in-module R (+) (·) 0 1 (⊕) 0M M (⊙) K
  using K by unfold-locales
have NM: N ⊆ M using submodule-subset[OF N] .
show ?thesis
proof
  show {v ∈ N. proj-res N K v = K.Qclass 0M} ⊆ N ∩ K
  proof
    fix v assume v: v ∈ {v ∈ N. proj-res N K v = K.Qclass 0M}
    — Extract the two conjuncts explicitly; auto here also unfolds proj-res and times
    out.
    from v have vN: v ∈ N by blast

```

```

from v have proj-res N K v = K.Qclass 0_M by blast
then have eq: K.Qclass v = K.Qclass 0_M using vN by simp
from vN have vM: v ∈ M using NM by blast
have v ⊕ madd.inverse 0_M ∈ K
  by (rule K.Class-eq-iff-diff-in-N[OF vM mzero-closed, THEN iffD1, OF eq])
then have v ∈ K using vM by (simp add: madd.inverse-unit)
with vN show v ∈ N ∩ K by blast
qed
show N ∩ K ⊆ {v ∈ N. proj-res N K v = K.Qclass 0_M}
proof
  fix v assume v: v ∈ N ∩ K
  then have vN: v ∈ N and vK: v ∈ K by auto
  from vN have vM: v ∈ M using NM by blast
  have v ⊕ madd.inverse 0_M ∈ K using vK vM by (simp add: madd.inverse-unit)
  then have K.Qclass v = K.Qclass 0_M
    by (rule K.Class-eq-iff-diff-in-N[OF vM mzero-closed, THEN iffD2])
  with vN show v ∈ {v ∈ N. proj-res N K v = K.Qclass 0_M} by simp
qed
qed
qed

```

Its image is $(N \oplus_S K)/K$: the K -cosets of elements of N are exactly the K -cosets of elements of $N \oplus_S K$, since adding an element of K to a representative does not change its coset.

lemma *proj-res-image*:

assumes N : submodule N **and** K : submodule K

shows $\text{proj-res } N \text{ } K \text{ } N$

$= \text{submodule-in-module.Qclass } (\oplus) \text{ } 0_M \text{ } M \text{ } K \text{ } (N \oplus_S K)$

proof –

interpret K : submodule-in-module $R \text{ } (+) \text{ } (\cdot) \text{ } 0 \text{ } 1 \text{ } (\oplus) \text{ } 0_M \text{ } M \text{ } (\odot) \text{ } K$

using K **by** *unfold-locales*

have NM : $N \subseteq M$ **using** *submodule-subset[OF N]* .

have KM : $K \subseteq M$ **using** *submodule-subset[OF K]* .

show *?thesis*

proof

show $\text{proj-res } N \text{ } K \text{ } N \subseteq K.Qclass \text{ } (N \oplus_S K)$

proof

fix X **assume** $X \in \text{proj-res } N \text{ } K \text{ } N$

then obtain v **where** $v: v \in N \text{ } X = \text{proj-res } N \text{ } K \text{ } v$ **by** *blast*

then have $X = K.Qclass \text{ } v$ **by** *simp*

moreover have $v \in N \oplus_S K$ **using** *submodule-sum-incl-left[OF N K]* $v(1)$ **by**

blast

ultimately show $X \in K.Qclass \text{ } (N \oplus_S K)$ **by** *blast*

qed

show $K.Qclass \text{ } (N \oplus_S K) \subseteq \text{proj-res } N \text{ } K \text{ } N$

proof

fix X **assume** $X \in K.Qclass \text{ } (N \oplus_S K)$

then obtain x **where** $x: x \in N \oplus_S K \text{ } X = K.Qclass \text{ } x$ **by** *blast*

from $x(1)$ **obtain** $u \text{ } w$ **where** $uw: u \in N \text{ } w \in K \text{ } x = u \oplus w$

```

    by (rule submodule-sum-memE)
  have uM:  $u \in M$  using uw(1) NM by blast
  have wM:  $w \in M$  using uw(2) KM by blast
  — The representative may be taken in  $N$ : the  $K$ -part is absorbed.
  have  $x \oplus \text{madd.inverse } u = (w \oplus u) \oplus \text{madd.inverse } u$ 
    using uw(3) uM wM by (simp add: madd.commutative)
  also have  $\dots = w \oplus (u \oplus \text{madd.inverse } u)$ 
    using uM wM by (simp add: madd.associative)
  also have  $\dots = w$  using uM wM by simp
  finally have  $x \oplus \text{madd.inverse } u = w$  .
  then have  $x \oplus \text{madd.inverse } u \in K$  using uw(2) by simp
  moreover have  $xM$ :  $x \in M$  using uw(3) uM wM by simp
  ultimately have  $K.Qclass\ x = K.Qclass\ u$ 
    using xM uM by (simp add: K.Class-eq-iff-diff-in-N)
  with x(2) have  $X = \text{proj-res } N\ K\ u$  using uw(1) by simp
  then show  $X \in \text{proj-res } N\ K\ 'N$  using uw(1) by blast
qed
qed
qed

```

If N complements K , the restricted natural projection identifies N with the whole quotient M/K . The two complement conditions have separate roles: trivial intersection gives injectivity through the kernel calculation, while the full submodule sum gives surjectivity through the image calculation.

theorem *complement-quotient-isomorphism:*

```

assumes N: submodule N and K: submodule K
  and disjoint:  $N \cap K = \{0_M\}$ 
  and full:  $N \oplus_S K = M$ 

```

shows *module-homomorphism* $R\ (+)\ (\cdot)\ 0\ 1$

```

   $N\ (\oplus)\ 0_M\ (\odot)$ 
  (submodule-in-module.Qcarrier  $(\oplus)\ 0_M\ M\ K$ )
  (submodule-in-module.qadd  $(\oplus)\ 0_M\ M\ K$ )
  (submodule-in-module.Qclass  $(\oplus)\ 0_M\ M\ K\ 0_M$ )
  (submodule-in-module.qscale  $(\oplus)\ 0_M\ M\ (\odot)\ K$ )
  (proj-res  $N\ K$ )
   $\wedge$  bij-betw (proj-res  $N\ K$ )  $N$ 
  (submodule-in-module.Qcarrier  $(\oplus)\ 0_M\ M\ K$ )

```

proof —

```

interpret K: submodule-in-module R (+) (·) 0 1
   $(\oplus)\ 0_M\ M\ (\odot)\ K$ 

```

```

using K by unfold-locales

```

```

interpret P: module-homomorphism R (+) (·) 0 1

```

```

   $N\ (\oplus)\ 0_M\ (\odot)$ 
   $K.Qcarrier\ K.qadd\ K.Qclass\ 0_M\ K.qscale\ \text{proj-res } N\ K$ 
  by (rule proj-res-hom[OF N K])

```

```

have kernel:  $P.Ker = N \cap K$ 

```

```

  unfolding P.Ker-def using proj-res-Ker[OF N K] by simp

```

```

have injective: inj-on (proj-res  $N\ K$ )  $N$ 

```

```

using P.injective-iff-kernel-trivial kernel disjoint by simp

have class-image:  $K.Qclass \text{ ` } M = K.Qcarrier$ 
proof -
  have  $K.Qclass \text{ ` } M = K.nat\text{-}proj \text{ ` } M$ 
  proof (rule image-cong)
    show  $M = M$  by simp
  next
    fix  $x$  assume  $x: x \in M$ 
    show  $K.Qclass \ x = K.nat\text{-}proj \ x$ 
    by (rule sym, rule K.nat-proj-apply[OF x])
  qed
  also have  $\dots = K.Qcarrier$  by (rule K.nat-proj-image)
  finally show ?thesis .
qed

have surjective:  $proj\text{-}res \ N \ K \text{ ` } N = K.Qcarrier$ 
using proj-res-image[OF N K] full class-image by simp

show ?thesis
using P.module-homomorphism-axioms injective surjective
by (simp add: bij-betw-def)
qed

```

The second isomorphism theorem.

theorem *second-isomorphism*:

assumes N : submodule N and K : submodule K

shows $\exists \varphi. module\text{-}homomorphism \ R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}$

$(submodule\text{-}in\text{-}module.Qcarrier \ (\oplus) \ \mathbf{0}_M \ N \ (N \cap K))$
 $(submodule\text{-}in\text{-}module.qadd \ (\oplus) \ \mathbf{0}_M \ N \ (N \cap K))$
 $(submodule\text{-}in\text{-}module.Qclass \ (\oplus) \ \mathbf{0}_M \ N \ (N \cap K) \ \mathbf{0}_M)$
 $(submodule\text{-}in\text{-}module.qscale \ (\oplus) \ \mathbf{0}_M \ N \ (\odot) \ (N \cap K))$
 $(submodule\text{-}in\text{-}module.Qclass \ (\oplus) \ \mathbf{0}_M \ M \ K \text{ ` } (N \oplus_S K))$
 $(submodule\text{-}in\text{-}module.qadd \ (\oplus) \ \mathbf{0}_M \ M \ K)$
 $(submodule\text{-}in\text{-}module.Qclass \ (\oplus) \ \mathbf{0}_M \ M \ K \ \mathbf{0}_M)$
 $(submodule\text{-}in\text{-}module.qscale \ (\oplus) \ \mathbf{0}_M \ M \ (\odot) \ K)$

φ

$\wedge \text{bij-betw } \varphi$

$(submodule\text{-}in\text{-}module.Qcarrier \ (\oplus) \ \mathbf{0}_M \ N \ (N \cap K))$
 $(submodule\text{-}in\text{-}module.Qclass \ (\oplus) \ \mathbf{0}_M \ M \ K \text{ ` } (N \oplus_S K))$

proof —

— Interpret the restricted projection as a homomorphism out of N , then quote its own first isomorphism theorem, rewriting the kernel to $N \cap K$ and the image to $(N \oplus_S K)/K$.

— Established BEFORE the interpretation below: once P is interpreted, its *source* module supplies a second *submodule-sum* and the $\oplus_S$ notation becomes ambiguous.

have *image-eq*: $proj\text{-}res \ N \ K \text{ ` } N$

$= submodule\text{-}in\text{-}module.Qclass \ (\oplus) \ \mathbf{0}_M \ M \ K \text{ ` } (N \oplus_S K)$

by (rule *proj-res-image[OF N K]*)

interpret P : module-homomorphism $R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}$

$N \ (\oplus) \ \mathbf{0}_M \ (\odot)$

```

    submodule-in-module.Qcarrier ( $\oplus$ )  $\mathbf{0}_M$   $M$   $K$ 
    submodule-in-module.qadd ( $\oplus$ )  $\mathbf{0}_M$   $M$   $K$ 
    submodule-in-module.Qclass ( $\oplus$ )  $\mathbf{0}_M$   $M$   $K$   $\mathbf{0}_M$ 
    submodule-in-module.qscale ( $\oplus$ )  $\mathbf{0}_M$   $M$  ( $\odot$ )  $K$ 
    proj-res  $N$   $K$ 
  by (rule proj-res-hom[ $OF$   $N$   $K$ ])
have Ker-eq:  $P.Ker = N \cap K$ 
  unfolding  $P.Ker-def$  using proj-res-Ker[ $OF$   $N$   $K$ ] by simp
show ?thesis
  using  $P.first-isomorphism$  unfolding Ker-eq image-eq by blast
qed

end
end

```

88 Complementary submodules and projections

```

theory Module-Complements
imports Module-Iso-Theorems
begin

```

An idempotent module endomorphism is a projection onto its image. The image and kernel therefore recover the two internal summands without any finite-dimensional assumption or choice of bases.

```

locale idempotent-module-endomorphism =
  module-homomorphism  $R$  ( $+$ ) ( $\cdot$ )  $\mathbf{0}$   $\mathbf{1}$ 
   $M$  ( $\oplus$ )  $\mathbf{0}_M$  ( $\odot$ )
   $M$  ( $\oplus$ )  $\mathbf{0}_M$  ( $\odot$ )  $p$ 
  for  $R$  and addition (infixl  $\langle + \rangle$  65) and multiplication (infixl  $\langle \cdot \rangle$  70)
  and zero ( $\langle \mathbf{0} \rangle$ ) and unit ( $\langle \mathbf{1} \rangle$ )
  and  $M$  and madd (infixl  $\langle \oplus \rangle$  65) and mzero ( $\langle \mathbf{0}_M \rangle$ )
  and scale (infixr  $\langle \odot \rangle$  75) and  $p +$ 
  assumes idempotent:  $\bigwedge x. x \in M \implies p (p x) = p x$ 
begin

```

```

theorem image-kernel-complementary:

```

```

   $p \text{ ' } M \cap Ker = \{\mathbf{0}_M\}$ 

```

```

   $p \text{ ' } M \oplus_S Ker = M$ 

```

```

proof -

```

```

  have image-sub: source.submodule ( $p \text{ ' } M$ )

```

```

    using image-submodule .

```

```

  show disjoint:  $p \text{ ' } M \cap Ker = \{\mathbf{0}_M\}$ 

```

```

  proof (rule set-eqI, rule iffI)

```

```

    fix  $x$  assume  $x: x \in p \text{ ' } M \cap Ker$ 

```

```

    then obtain  $y$  where  $y: y \in M \ x = p y$  by auto

```

```

    have  $x = p x$  using  $y$  idempotent[ $OF$   $y(1)$ ] by simp

```

```

    also have  $\dots = \mathbf{0}_M$  using  $x$  by (simp add: Ker-image)

```

```

    finally show  $x \in \{\mathbf{0}_M\}$  by simp

```

```

  next

```

```

fix x assume x: x ∈ {0M}
have xeq: x = 0M using x by simp
have zero-image: 0M ∈ p ‘ M
proof -
  have p 0M ∈ p ‘ M
  by (rule imageI[OF source.mzero-closed])
  then show ?thesis using hom-zero by simp
qed
have zero-kernel: 0M ∈ Ker
  by (rule source.submodule-zero[OF Ker-submodule])
have image-x: x ∈ p ‘ M using xeq zero-image by simp
have kernel-x: x ∈ Ker using xeq zero-kernel by simp
show x ∈ p ‘ M ∩ Ker by (rule IntI[OF image-x kernel-x])
qed
show p ‘ M ⊕S Ker = M
proof
  have sum-sub: source.submodule (p ‘ M ⊕S Ker)
  by (rule source.submodule-sum-submodule[OF image-sub Ker-submodule])
  show p ‘ M ⊕S Ker ⊆ M
  by (rule source.submodule-subset[OF sum-sub])
  show M ⊆ p ‘ M ⊕S Ker
  proof
    fix x assume x: x ∈ M
    have px: p x ∈ M by (rule hom-closed[OF x])
    have remainder: x ⊕ source.madd.inverse (p x) ∈ Ker
    proof (rule Ker-memI)
      show p (x ⊕ source.madd.inverse (p x)) = 0M
      using x px idempotent[OF x]
      by (simp add: hom-add hom-neg)
      show x ⊕ source.madd.inverse (p x) ∈ M using x px by simp
    qed
    have image: p x ∈ p ‘ M by (rule imageI[OF x])
    have decomposition:
      x = p x ⊕ (x ⊕ source.madd.inverse (p x))
      using x px by (simp add: source.madd.ac)
    have p x ⊕ (x ⊕ source.madd.inverse (p x))
      ∈ p ‘ M ⊕S Ker
      by (rule source.submodule-sum-memI[OF image remainder])
    show x ∈ p ‘ M ⊕S Ker
    using decomposition ⟨p x ⊕ (x ⊕ source.madd.inverse (p x))
      ∈ p ‘ M ⊕S Ker⟩ by simp
  qed
qed
qed
end

context Module
begin

```

The projection onto N along K is obtained canonically from the quotient: first map an element of M to its class in M/K , then use the inverse of the restricted quotient isomorphism from N . Both maps are restricted to their carriers, so the resulting projection follows the extensional-map convention of the module hierarchy.

definition *complement-projection* :: 'b set $\Rightarrow$ 'b set $\Rightarrow$ 'b $\Rightarrow$ 'b
where *complement-projection* $N K =$
compose M
 (*restrict* (*inv-into* N (*proj-res* $N K$))
 (*submodule-in-module.Qcarrier* ($\oplus$) $\mathbf{0}_M$ $M K$))
 (*submodule-in-module.nat-proj* ($\oplus$) $\mathbf{0}_M$ $M K$)

The construction is a module homomorphism whenever the two submodules are complementary. This packages the quotient projection, the complement-quotient isomorphism, and its carrier-restricted inverse.

theorem *complement-projection-hom*:
assumes N : submodule N **and** K : submodule K
and disjoint: $N \cap K = \{\mathbf{0}_M\}$
and full: $N \oplus_S K = M$
shows *module-homomorphism* $R (+) (\cdot) \mathbf{0} \mathbf{1}$
 $M (\oplus) \mathbf{0}_M (\odot) N (\oplus) \mathbf{0}_M (\odot)$
 (*complement-projection* $N K$)
proof –
interpret K : *submodule-in-module* $R (+) (\cdot) \mathbf{0} \mathbf{1}$
 ($\oplus$) $\mathbf{0}_M$ $M (\odot) K$
using K **by** *unfold-locales*
interpret P : *module-homomorphism* $R (+) (\cdot) \mathbf{0} \mathbf{1}$
 $N (\oplus) \mathbf{0}_M (\odot)$
 $K.Qcarrier K.qadd K.Qclass \mathbf{0}_M K.qscale$ *proj-res* $N K$
by (*rule* *proj-res-hom*[*OF* $N K$])
have *bij*: *bij-betw* (*proj-res* $N K$) $N K.Qcarrier$
using *complement-quotient-isomorphism*[*OF* $N K$ *disjoint full*] **by** *blast*
have *inverse-hom*: *module-homomorphism* $R (+) (\cdot) \mathbf{0} \mathbf{1}$
 $K.Qcarrier K.qadd (K.Qclass \mathbf{0}_M) K.qscale$
 $N (\oplus) \mathbf{0}_M (\odot)$
 (*restrict* (*inv-into* N (*proj-res* $N K$)) $K.Qcarrier$)
by (*rule* $P.inverse-module-homomorphism$ [*OF* *bij*])
show *?thesis*
unfolding *complement-projection-def*
by (*rule* $K.nat-proj.compose-module-homomorphism$ [*OF* *inverse-hom*])
qed

The projection fixes the chosen summand.

lemma *complement-projection-apply-left*:
assumes N : submodule N **and** K : submodule K
and disjoint: $N \cap K = \{\mathbf{0}_M\}$
and full: $N \oplus_S K = M$
and x : $x \in N$

```

shows complement-projection  $N\ K\ x = x$ 
proof –
interpret  $K$ : submodule-in-module  $R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}$ 
   $(\oplus)\ \mathbf{0}_M\ M\ (\odot)\ K$ 
using  $K$  by unfold-locales
interpret  $P$ : module-homomorphism  $R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}$ 
   $N\ (\oplus)\ \mathbf{0}_M\ (\odot)$ 
   $K.Qcarrier\ K.qadd\ K.Qclass\ \mathbf{0}_M\ K.qscale\ proj-res\ N\ K$ 
by (rule proj-res-hom[OF  $N\ K$ ])
have  $bij$ :  $bij-betw\ (proj-res\ N\ K)\ N\ K.Qcarrier$ 
using complement-quotient-isomorphism[OF  $N\ K\ disjoint\ full$ ] by blast
have  $xM$ :  $x \in M$  using  $x$  submodule-subset[OF  $N$ ] by blast
show ?thesis
unfolding complement-projection-def compose-def
using  $P.inverse-module-homomorphism-left$ [OF  $bij\ x$ ]  $x\ xM$  by simp
qed

```

The complementary summand is annihilated.

```

lemma complement-projection-apply-right:
assumes  $N$ : submodule  $N$  and  $K$ : submodule  $K$ 
and  $disjoint$ :  $N \cap K = \{\mathbf{0}_M\}$ 
and  $full$ :  $N \oplus_S K = M$ 
and  $x$ :  $x \in K$ 
shows complement-projection  $N\ K\ x = \mathbf{0}_M$ 
proof –
interpret  $K$ : submodule-in-module  $R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}$ 
   $(\oplus)\ \mathbf{0}_M\ M\ (\odot)\ K$ 
using  $K$  by unfold-locales
interpret  $P$ : module-homomorphism  $R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}$ 
   $N\ (\oplus)\ \mathbf{0}_M\ (\odot)$ 
   $K.Qcarrier\ K.qadd\ K.Qclass\ \mathbf{0}_M\ K.qscale\ proj-res\ N\ K$ 
by (rule proj-res-hom[OF  $N\ K$ ])
have  $bij$ :  $bij-betw\ (proj-res\ N\ K)\ N\ K.Qcarrier$ 
using complement-quotient-isomorphism[OF  $N\ K\ disjoint\ full$ ] by blast
interpret  $I$ : module-homomorphism  $R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}$ 
   $K.Qcarrier\ K.qadd\ K.Qclass\ \mathbf{0}_M\ K.qscale$ 
   $N\ (\oplus)\ \mathbf{0}_M\ (\odot)$ 
   $restrict\ (inv-into\ N\ (proj-res\ N\ K))\ K.Qcarrier$ 
by (rule  $P.inverse-module-homomorphism$ [OF  $bij$ ])
have  $xM$ :  $x \in M$  using  $x$  submodule-subset[OF  $K$ ] by blast
have  $xKer$ :  $x \in K.nat-proj.Ker$  using  $x\ K.nat-proj-Ker$  by simp
have  $quotient-zero$ :  $K.nat-proj\ x = K.Qclass\ \mathbf{0}_M$ 
by (rule  $K.nat-proj.Ker-image$ [OF  $xKer$ ])
show ?thesis
unfolding complement-projection-def compose-def
using  $xM\ quotient-zero\ I.hom-zero$  by simp
qed

```

On an explicitly decomposed element, the projection selects its left component.

lemma complement-projection-apply-sum:


```

assumes  $N$ : submodule  $N$  and  $K$ : submodule  $K$ 
and  $disjoint$ :  $N \cap K = \{0_M\}$ 
and  $full$ :  $N \oplus_S K = M$ 
and  $u$ :  $u \in N$  and  $v$ :  $v \in K$ 
shows  $complement\text{-}projection\ N\ K\ (u \oplus v) = u$ 
proof –
interpret  $P$ : module-homomorphism  $R\ (+)\ (\cdot)\ 0\ 1$ 
 $M\ (\oplus)\ 0_M\ (\odot)$ 
 $N\ (\oplus)\ 0_M\ (\odot)$   $complement\text{-}projection\ N\ K$ 
by (rule  $complement\text{-}projection\text{-}hom[OF\ N\ K\ disjoint\ full]$ )
have  $uM$ :  $u \in M$  using  $u$   $submodule\text{-}subset[OF\ N]$  by blast
have  $vM$ :  $v \in M$  using  $v$   $submodule\text{-}subset[OF\ K]$  by blast
have  $complement\text{-}projection\ N\ K\ (u \oplus v) =$ 
 $complement\text{-}projection\ N\ K\ u \oplus complement\text{-}projection\ N\ K\ v$ 
by (rule  $P.hom\text{-}add[OF\ uM\ vM]$ )
also have  $\dots = u \oplus 0_M$ 
using  $complement\text{-}projection\text{-}apply\text{-}left[OF\ N\ K\ disjoint\ full\ u]$ 
 $complement\text{-}projection\text{-}apply\text{-}right[OF\ N\ K\ disjoint\ full\ v]$ 
by simp
also have  $\dots = u$  using  $uM$  by simp
finally show  $?thesis$  .
qed

```

Every element of a full, disjoint submodule sum has exactly one ordered pair of components. Pairing the witnesses makes the uniqueness quantifier directly usable by clients.

```

theorem  $complementary\text{-}decomposition\text{-}exists\text{-}unique$ :
assumes  $N$ : submodule  $N$  and  $K$ : submodule  $K$ 
and  $disjoint$ :  $N \cap K = \{0_M\}$ 
and  $full$ :  $N \oplus_S K = M$ 
and  $x$ :  $x \in M$ 
shows  $\exists! uv. fst\ uv \in N \wedge snd\ uv \in K \wedge x = fst\ uv \oplus snd\ uv$ 
proof –
have  $xsum$ :  $x \in N \oplus_S K$  using  $x$   $full$  by simp
then obtain  $u\ v$  where  $u$ :  $u \in N$  and  $v$ :  $v \in K$  and  $xeq$ :  $x = u \oplus v$ 
by (rule  $submodule\text{-}sum\text{-}memE$ )
show  $?thesis$ 
proof (rule  $ex1I[of\ -\ (u, v)]$ )
show  $fst\ (u, v) \in N \wedge snd\ (u, v) \in K \wedge x = fst\ (u, v) \oplus snd\ (u, v)$ 
using  $u\ v\ xeq$  by simp
next
fix  $uv$  assume  $uv$ :  $fst\ uv \in N \wedge snd\ uv \in K \wedge$ 
 $x = fst\ uv \oplus snd\ uv$ 
have  $components$ :  $u = fst\ uv \wedge v = snd\ uv$ 
by (rule  $submodule\text{-}sum\text{-}decomposition\text{-}unique[OF\ N\ K\ disjoint\ u\ v]$ )
 $(use\ uv\ xeq\ in\ auto)$ 
then show  $uv = (u, v)$  by (cases  $uv$ )  $auto$ 
qed
qed

```

The image is precisely the summand onto which the map projects.

theorem *complement-projection-image:*

assumes N : submodule N **and** K : submodule K

and *disjoint*: $N \cap K = \{0_M\}$

and *full*: $N \oplus_S K = M$

shows *complement-projection* $N K \text{ ' } M = N$

proof –

interpret P : module-homomorphism $R (+) (\cdot) 0 1$

$M (\oplus) 0_M (\odot)$

$N (\oplus) 0_M (\odot)$ *complement-projection* $N K$

by (*rule complement-projection-hom*[*OF* $N K$ *disjoint full*])

show *?thesis*

proof

show *complement-projection* $N K \text{ ' } M \subseteq N$ **using** P .*hom-closed* **by** *blast*

show $N \subseteq$ *complement-projection* $N K \text{ ' } M$

proof

fix x **assume** $x: x \in N$

have $xM: x \in M$ **using** x *submodule-subset*[*OF* N] **by** *blast*

have *complement-projection* $N K x = x$

by (*rule complement-projection-apply-left*[*OF* $N K$ *disjoint full* x])

moreover **have** *complement-projection* $N K x \in$ *complement-projection* $N K \text{ ' }$

M

by (*rule imageI*[*OF* xM])

ultimately show $x \in$ *complement-projection* $N K \text{ ' } M$ **by** *simp*

qed

qed

qed

The kernel is precisely the complementary summand.

theorem *complement-projection-Ker:*

assumes N : submodule N **and** K : submodule K

and *disjoint*: $N \cap K = \{0_M\}$

and *full*: $N \oplus_S K = M$

shows $\{x \in M. \text{complement-projection } N K x = 0_M\} = K$

proof –

have *decompose*: $\bigwedge x. x \in M \implies$

$\exists u \in N. \exists v \in K. x = u \oplus v$

proof –

fix x **assume** $x: x \in M$

have $x \in N \oplus_S K$ **using** x *full* **by** *simp*

then show $\exists u \in N. \exists v \in K. x = u \oplus v$

by (*rule submodule-sum-memE*) *blast*

qed

interpret P : module-homomorphism $R (+) (\cdot) 0 1$

$M (\oplus) 0_M (\odot)$

$N (\oplus) 0_M (\odot)$ *complement-projection* $N K$

by (*rule complement-projection-hom*[*OF* $N K$ *disjoint full*])

have *kernel*: $P.Ker = K$

proof

```

show  $P.Ker \subseteq K$ 
proof
  fix  $x$  assume  $xKer: x \in P.Ker$ 
  then have  $xM: x \in M$  and  $px: complement-projection\ N\ K\ x = \mathbf{0}_M$ 
    using  $P.Ker-mem\ P.Ker-image$  by auto
  obtain  $u\ v$  where  $u: u \in N$  and  $v: v \in K$  and  $xeq: x = u \oplus v$ 
    using  $decompose[OF\ xM]$  by blast
  have  $complement-projection\ N\ K\ x = u$ 
    using  $xeq\ complement-projection-apply-sum[OF\ N\ K\ disjoint\ full\ u\ v]$  by simp
  then have  $u = \mathbf{0}_M$  using  $px$  by simp
  moreover have  $vM: v \in M$  using  $v\ submodule-subset[OF\ K]$  by blast
  ultimately have  $x = v$  using  $xeq$  by (simp add:  $madd.left-unit$ )
  then show  $x \in K$  using  $v$  by simp
qed
show  $K \subseteq P.Ker$ 
proof
  fix  $x$  assume  $x: x \in K$ 
  have  $xM: x \in M$  using  $x\ submodule-subset[OF\ K]$  by blast
  have  $complement-projection\ N\ K\ x = \mathbf{0}_M$ 
    by (rule  $complement-projection-apply-right[OF\ N\ K\ disjoint\ full\ x]$ )
  then show  $x \in P.Ker$  by (rule  $P.Ker-memI[OF\ -\ xM]$ )
qed
qed
show ?thesis using kernel by (simp add:  $P.Ker-def$ )
qed

```

Applying a complementary projection twice has no further effect.

```

lemma complement-projection-idempotent:
  assumes  $N: submodule\ N$  and  $K: submodule\ K$ 
    and  $disjoint: N \cap K = \{\mathbf{0}_M\}$ 
    and  $full: N \oplus_S K = M$ 
    and  $x: x \in M$ 
  shows  $complement-projection\ N\ K\ (complement-projection\ N\ K\ x) =$ 
     $complement-projection\ N\ K\ x$ 
proof -
  interpret  $P: module-homomorphism\ R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}$ 
     $M\ (\oplus)\ \mathbf{0}_M\ (\odot)$ 
     $N\ (\oplus)\ \mathbf{0}_M\ (\odot)\ complement-projection\ N\ K$ 
  by (rule  $complement-projection-hom[OF\ N\ K\ disjoint\ full]$ )
  have  $complement-projection\ N\ K\ x \in N$  by (rule  $P.hom-closed[OF\ x]$ )
  then show ?thesis by (rule  $complement-projection-apply-left[OF\ N\ K\ disjoint\ full]$ )
qed

```

The two complementary projections recover the original element. This also cross-checks the orientation of the construction: the first map selects the N -component and the swapped map selects the K -component.

```

theorem complement-projections-sum:
  assumes  $N: submodule\ N$  and  $K: submodule\ K$ 
    and  $disjoint: N \cap K = \{\mathbf{0}_M\}$ 

```

```

    and full:  $N \oplus_S K = M$ 
    and x:  $x \in M$ 
    shows complement-projection  $N K x \oplus$  complement-projection  $K N x = x$ 
  proof -
    have disjoint':  $K \cap N = \{0_M\}$  using disjoint by (simp add: Int-commute)
    have full':  $K \oplus_S N = M$ 
      using full submodule-sum-commute[OF K N] by simp
    have xsum:  $x \in N \oplus_S K$  using x full by simp
    then obtain u v where u:  $u \in N$  and v:  $v \in K$  and xeq:  $x = u \oplus v$ 
      by (rule submodule-sum-memE)
    have first: complement-projection  $N K x = u$ 
      using xeq complement-projection-apply-sum[OF N K disjoint full u v] by simp
    have uM:  $u \in M$  using u submodule-subset[OF N] by blast
    have vM:  $v \in M$  using v submodule-subset[OF K] by blast
    have xeq':  $x = v \oplus u$  using xeq uM vM by (simp add: madd.commutative)
    have second: complement-projection  $K N x = v$ 
      using xeq' complement-projection-apply-sum[OF K N disjoint' full' v u] by simp
    show ?thesis using first second xeq by simp
  qed

end

```

end

89 Exact sequences of modules

```

theory Module-Exact-Sequence
  imports Module-Complements
begin

```

```

locale composable-module-homomorphisms =
  first: module-homomorphism  $R (+) (\cdot) 0 1$ 
     $M (\oplus) 0_M (\odot) M' (\oplus_2) 0_2 (\odot_2) \eta +$ 
  second: module-homomorphism  $R (+) (\cdot) 0 1$ 
     $M' (\oplus_2) 0_2 (\odot_2) M'' (\oplus_3) 0_3 (\odot_3) \vartheta$ 
  for  $R$  and addition (infixl  $\langle + \rangle$  65) and multiplication (infixl  $\langle \cdot \rangle$  70)
  and zero ( $\langle 0 \rangle$ ) and unit ( $\langle 1 \rangle$ )
  and  $M$  and madd (infixl  $\langle \oplus \rangle$  65) and mzero ( $\langle 0_M \rangle$ )
  and scale (infixr  $\langle \odot \rangle$  75)
  and  $M'$  and madd' (infixl  $\langle \oplus_2 \rangle$  65) and mzero' ( $\langle 0_2 \rangle$ )
  and scale' (infixr  $\langle \odot_2 \rangle$  75)
  and  $M''$  and madd'' (infixl  $\langle \oplus_3 \rangle$  65) and mzero'' ( $\langle 0_3 \rangle$ )
  and scale'' (infixr  $\langle \odot_3 \rangle$  75)
  and  $\eta$  and  $\vartheta$ 
begin

```

```

definition exact :: bool
  where exact  $\longleftrightarrow \eta \text{ ' } M = \text{second.Ker}$ 

```

definition *short-exact* :: bool
where *short-exact* $\longleftrightarrow$ *inj-on* η *M* $\wedge$ *exact* $\wedge$ $\vartheta \text{ ' } M' = M''$

definition *right-split* :: bool
where *right-split* $\longleftrightarrow$
 $(\exists \sigma. \text{ module-homomorphism } R (+) (\cdot) \mathbf{0} \mathbf{1}$
 $M'' (\oplus_3) \mathbf{0}_3 (\odot_3) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \sigma \wedge$
 $(\forall x \in M''. \vartheta (\sigma x) = x))$

definition *left-split* :: bool
where *left-split* $\longleftrightarrow$
 $(\exists \varrho. \text{ module-homomorphism } R (+) (\cdot) \mathbf{0} \mathbf{1}$
 $M' (\oplus_2) \mathbf{0}_2 (\odot_2) M (\oplus) \mathbf{0}_M (\odot) \varrho \wedge$
 $(\forall x \in M. \varrho (\eta x) = x))$

lemma *exact-iff*: *exact* $\longleftrightarrow$ $\eta \text{ ' } M = \text{second.Ker}$
by (*rule exact-def*)

lemma *short-exact-iff*:
short-exact $\longleftrightarrow$ *inj-on* η *M* $\wedge$ $\eta \text{ ' } M = \text{second.Ker} \wedge \vartheta \text{ ' } M' = M''$
by (*simp add: short-exact-def exact-def*)

lemma *exact-imp-composition-zero*:
assumes *ex*: *exact* **and** *x*: $x \in M$
shows $\vartheta (\eta x) = \mathbf{0}_3$
proof –
have $\eta x \in \eta \text{ ' } M$ **using** *x* **by** (*rule imageI*)
then have $\eta x \in \text{second.Ker}$ **using** *ex* **by** (*simp add: exact-def*)
then show ?thesis **by** (*rule second.Ker-image*)
qed

lemma *right-split-imp-surjective*:
assumes *split*: *right-split*
shows $\vartheta \text{ ' } M' = M''$
proof
show $\vartheta \text{ ' } M' \subseteq M''$ **using** *second.hom-closed* **by** *blast*
show $M'' \subseteq \vartheta \text{ ' } M'$
proof
fix *x* **assume** *x*: $x \in M''$
obtain σ **where** *hom*: *module-homomorphism* $R (+) (\cdot) \mathbf{0} \mathbf{1}$
 $M'' (\oplus_3) \mathbf{0}_3 (\odot_3) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \sigma$
and *sec*: $\forall x \in M''. \vartheta (\sigma x) = x$
using *split* **unfolding** *right-split-def* **by** *blast*
interpret *S*: *module-homomorphism* $R (+) (\cdot) \mathbf{0} \mathbf{1}$
 $M'' (\oplus_3) \mathbf{0}_3 (\odot_3) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \sigma$
by (*rule hom*)
have *sx*: $\sigma x \in M'$ **by** (*rule S.hom-closed[OF x]*)
have *eq*: $\vartheta (\sigma x) = x$ **using** *sec x* **by** *blast*
have $\vartheta (\sigma x) \in \vartheta \text{ ' } M'$ **by** (*rule imageI[OF sx]*)

```

    then show  $x \in \vartheta \text{ ' } M'$  using eq by simp
  qed
qed

```

The image of an explicit right section is complementary to the kernel of the split map. The composite $\sigma \circ \vartheta$ is the corresponding idempotent projection on the middle module.

theorem *right-section-image-kernel-complementary*:

```

assumes section-hom: module-homomorphism  $R (+) (\cdot) \mathbf{0} \mathbf{1}$ 
       $M'' (\oplus_3) \mathbf{0}_3 (\odot_3) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \sigma$ 
and sec-eq:  $\bigwedge x. x \in M'' \implies \vartheta (\sigma x) = x$ 
shows  $\sigma \text{ ' } M'' \cap \text{second.Ker} = \{\mathbf{0}_2\}$ 
and  $\text{Module.submodule-sum } (\oplus_2) (\sigma \text{ ' } M'') \text{second.Ker} = M'$ 

```

proof –

```

interpret S: module-homomorphism  $R (+) (\cdot) \mathbf{0} \mathbf{1}$ 
       $M'' (\oplus_3) \mathbf{0}_3 (\odot_3)$ 
       $M' (\oplus_2) \mathbf{0}_2 (\odot_2) \sigma$ 
by (rule section-hom)
define p where  $p = \text{compose } M' \sigma \vartheta$ 
have p-hom: module-homomorphism  $R (+) (\cdot) \mathbf{0} \mathbf{1}$ 
       $M' (\oplus_2) \mathbf{0}_2 (\odot_2) M' (\oplus_2) \mathbf{0}_2 (\odot_2) p$ 
unfolding p-def by (rule second.compose-module-homomorphism[OF section-hom])
interpret P: idempotent-module-endomorphism  $R (+) (\cdot) \mathbf{0} \mathbf{1}$ 
       $M' (\oplus_2) \mathbf{0}_2 (\odot_2) p$ 
proof (rule idempotent-module-endomorphism.intro[OF p-hom])
show idempotent-module-endomorphism-axioms  $M' p$ 
proof
  fix x assume  $x: x \in M'$ 
  have tx:  $\vartheta x \in M''$  by (rule second.hom-closed[OF x])
  have str:  $\sigma (\vartheta x) \in M'$  by (rule S.hom-closed[OF tx])
  show  $p (p x) = p x$ 
    unfolding p-def compose-def
    using  $x \text{ tx str sec-eq}$ [OF tx] by simp

```

qed

qed

have *image-p*: $p \text{ ' } M' = \sigma \text{ ' } M''$

proof (*rule set-eqI*, *rule iffI*)

fix *y* **assume** $y: y \in p \text{ ' } M'$

then obtain *x* **where** $x: x \in M' y = p x$ **by** *blast*

have *tx*: $\vartheta x \in M''$ **by** (*rule second.hom-closed*[*OF x*(1)])

have *value-eq*: $y = \sigma (\vartheta x)$ **using** *x* **unfolding** *p-def* *compose-def* **by** *simp*

have $\sigma (\vartheta x) \in \sigma \text{ ' } M''$ **by** (*rule imageI*[*OF tx*])

then show $y \in \sigma \text{ ' } M''$ **using** *value-eq* **by** *simp*

next

fix *y* **assume** $y: y \in \sigma \text{ ' } M''$

then obtain *z* **where** $z: z \in M'' y = \sigma z$ **by** *blast*

have *sz*: $\sigma z \in M'$ **by** (*rule S.hom-closed*[*OF z*(1)])

have *value-eq*: $p (\sigma z) = \sigma z$

unfolding *p-def* *compose-def* **using** $z \text{ sz sec-eq}$ [*OF z*(1)] **by** *simp*

```

have p (σ z) ∈ p ' M' by (rule imageI[OF sz])
then show y ∈ p ' M' using z value-eq by simp
qed
have kernel-p: P.Ker = second.Ker
proof (rule set-eqI, rule iffI)
  fix x assume x: x ∈ P.Ker
  have xM: x ∈ M' by (rule P.Ker-mem[OF x])
  have tx: ∅ x ∈ M'' by (rule second.hom-closed[OF xM])
  have theta-p: ∅ (p x) = ∅ x
    unfolding p-def compose-def using xM tx sec-eq[OF tx] by simp
  have ∅ (p x) = 03
    using P.Ker-image[OF x] second.hom-zero by simp
  then have ∅ x = 03 using theta-p by simp
  then show x ∈ second.Ker by (rule second.Ker-memI[OF - xM])
next
fix x assume x: x ∈ second.Ker
have xM: x ∈ M' by (rule second.Ker-mem[OF x])
have p x = 02
  unfolding p-def compose-def
  using xM second.Ker-image[OF x] S.hom-zero by simp
then show x ∈ P.Ker by (rule P.Ker-memI[OF - xM])
qed
show σ ' M'' ∩ second.Ker = {02}
  using P.image-kernel-complementary(1) image-p kernel-p by simp
show Module.submodule-sum (⊕2) (σ ' M'') second.Ker = M'
  using P.image-kernel-complementary(2) image-p kernel-p by simp
qed

```

theorem *right-split-obtains-image-kernel-complement*:

assumes *split*: *right-split*

obtains σ **where**

```

module-homomorphism R (+) (·) 0 1
M'' (⊕3) 03 (⊙3) M' (⊕2) 02 (⊙2) σ
∧ x. x ∈ M'' ⇒ ∅ (σ x) = x
σ ' M'' ∩ second.Ker = {02}
Module.submodule-sum (⊕2) (σ ' M'') second.Ker = M'

```

proof –

obtain σ **where** *hom*: *module-homomorphism* R (+) (·) 0 1

$M'' (\oplus_3) 0_3 (\odot_3) M' (\oplus_2) 0_2 (\odot_2) \sigma$

and *sec*: $\forall x \in M''. \emptyset (\sigma x) = x$

using *split* **unfolding** *right-split-def* **by** *blast*

have *sec-eq*: $\bigwedge x. x \in M'' \Rightarrow \emptyset (\sigma x) = x$ **using** *sec* **by** *blast*

have *complementary*:

$\sigma ' M'' \cap \text{second.Ker} = \{0_2\}$

$\text{Module.submodule-sum } (\oplus_2) (\sigma ' M'') \text{ second.Ker} = M'$

proof –

show $\sigma ' M'' \cap \text{second.Ker} = \{0_2\}$

by (rule *right-section-image-kernel-complementary*(1)[OF *hom sec-eq*])

show $\text{Module.submodule-sum } (\oplus_2) (\sigma ' M'') \text{ second.Ker} = M'$

by (rule *right-section-image-kernel-complementary*(2)[OF *hom sec-eq*])

qed
 show thesis by (rule that[OF hom sec-eq complementary])
 qed

theorem short-exact-right-split-obtains-decomposition:

assumes short: short-exact **and** split: right-split

obtains σ **where**

module-homomorphism $R (+) (\cdot) \mathbf{0} \mathbf{1}$

$M'' (\oplus_3) \mathbf{0}_3 (\odot_3) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \sigma$

$\bigwedge x. x \in M'' \implies \vartheta (\sigma x) = x$

$\sigma ' M'' \cap \eta ' M = \{\mathbf{0}_2\}$

Module.submodule-sum $(\oplus_2) (\sigma ' M'') (\eta ' M) = M'$

proof –

obtain σ **where** hom: module-homomorphism $R (+) (\cdot) \mathbf{0} \mathbf{1}$

$M'' (\oplus_3) \mathbf{0}_3 (\odot_3) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \sigma$

and sec-eq: $\bigwedge x. x \in M'' \implies \vartheta (\sigma x) = x$

and disjoint: $\sigma ' M'' \cap \text{second.Ker} = \{\mathbf{0}_2\}$

and full: Module.submodule-sum $(\oplus_2) (\sigma ' M'') \text{second.Ker} = M'$

using right-split-obtains-image-kernel-complement[OF split] **by** blast

have exact: $\eta ' M = \text{second.Ker}$ **using** short **by** (simp add: short-exact-iff)

show thesis

by (rule that[OF hom sec-eq]) (use disjoint full exact in simp-all)

qed

A right section yields a retraction of the injection. The projection onto the image of η along the section image is followed by the inverse of η onto its image.

theorem short-exact-right-split-imp-left-split:

assumes short: short-exact **and** split: right-split

shows left-split

proof –

obtain σ **where** section-hom: module-homomorphism $R (+) (\cdot) \mathbf{0} \mathbf{1}$

$M'' (\oplus_3) \mathbf{0}_3 (\odot_3) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \sigma$

and sec-eq: $\bigwedge x. x \in M'' \implies \vartheta (\sigma x) = x$

and disjoint: $\sigma ' M'' \cap \text{second.Ker} = \{\mathbf{0}_2\}$

and full: Module.submodule-sum $(\oplus_2) (\sigma ' M'') \text{second.Ker} = M'$

using right-split-obtains-image-kernel-complement[OF split] **by** blast

have exact: $\eta ' M = \text{second.Ker}$ **using** short **by** (simp add: short-exact-iff)

have eta-sub: first.target.submodule $(\eta ' M)$ **using** first.image-submodule .

interpret S : module-homomorphism $R (+) (\cdot) \mathbf{0} \mathbf{1}$

$M'' (\oplus_3) \mathbf{0}_3 (\odot_3)$

$M' (\oplus_2) \mathbf{0}_2 (\odot_2) \sigma$

by (rule section-hom)

have sigma-sub: first.target.submodule $(\sigma ' M'')$ **using** $S.\text{image-submodule}$.

have eta-sub': Module.submodule $R (\oplus_2) \mathbf{0}_2 M' (\odot_2) (\eta ' M)$

using eta-sub .

have sigma-sub': Module.submodule $R (\oplus_2) \mathbf{0}_2 M' (\odot_2) (\sigma ' M'')$

using sigma-sub .

have disjoint': $\eta ' M \cap \sigma ' M'' = \{\mathbf{0}_2\}$

using disjoint exact **by** (simp add: Int-commute)


```

have full': Module.submodule-sum ( $\oplus_2$ ) ( $\eta \text{ ' } M$ ) ( $\sigma \text{ ' } M''$ ) =  $M'$ 
  using full exact first.target.submodule-sum-commute[OF eta-sub sigma-sub] by
simp
interpret Kq: submodule-in-module R (+) ( $\cdot$ ) 0 1
  ( $\oplus_2$ ) 02 M' ( $\odot_2$ ) ( $\sigma \text{ ' } M''$ )
  using sigma-sub' by unfold-locales
interpret Q: module-homomorphism R (+) ( $\cdot$ ) 0 1
  ( $\eta \text{ ' } M$ ) ( $\oplus_2$ ) 02 ( $\odot_2$ )
  Kq.Qcarrier Kq.qadd Kq.Qclass 02 Kq.qscale
  first.target.proj-res ( $\eta \text{ ' } M$ ) ( $\sigma \text{ ' } M''$ )
  by (rule first.target.proj-res-hom[OF eta-sub' sigma-sub'])
have projection-bij:
  bij-betw (first.target.proj-res ( $\eta \text{ ' } M$ ) ( $\sigma \text{ ' } M''$ ))
    ( $\eta \text{ ' } M$ ) Kq.Qcarrier
  using first.target.complement-quotient-isomorphism
    [OF eta-sub' sigma-sub' disjoint' full'] by blast
have inverse-projection:
  module-homomorphism R (+) ( $\cdot$ ) 0 1
  Kq.Qcarrier Kq.qadd (Kq.Qclass 02) Kq.qscale
  ( $\eta \text{ ' } M$ ) ( $\oplus_2$ ) 02 ( $\odot_2$ )
  (restrict (inv-into ( $\eta \text{ ' } M$ ) (first.target.proj-res ( $\eta \text{ ' } M$ ) ( $\sigma \text{ ' } M''$ )))) Kq.Qcarrier)
  by (rule Q.inverse-module-homomorphism[OF projection-bij])
interpret P: module-homomorphism R (+) ( $\cdot$ ) 0 1
  M' ( $\oplus_2$ ) 02 ( $\odot_2$ )
  ( $\eta \text{ ' } M$ ) ( $\oplus_2$ ) 02 ( $\odot_2$ )
  first.target.complement-projection ( $\eta \text{ ' } M$ ) ( $\sigma \text{ ' } M''$ )
  by (rule first.target.complement-projection-hom
    [OF eta-sub' sigma-sub' disjoint' full'])
have eta-image-hom:
  module-homomorphism R (+) ( $\cdot$ ) 0 1
  M ( $\oplus$ ) 0M ( $\odot$ )
  ( $\eta \text{ ' } M$ ) ( $\oplus_2$ ) 02 ( $\odot_2$ )  $\eta$ 
proof (intro module-homomorphism.intro map.intro module-homomorphism-axioms.intro)
  show Module R (+) ( $\cdot$ ) 0 1 ( $\oplus$ ) 0M M ( $\odot$ )
    by (rule first.source.Module-axioms)
  show Module R (+) ( $\cdot$ ) 0 1
    ( $\oplus_2$ ) 02 ( $\eta \text{ ' } M$ ) ( $\odot_2$ )
    by (rule first.image-is-module)
  show  $\eta \in M \rightarrow_E (\eta \text{ ' } M)$ 
    using first.hom-closed by (auto simp: PiE-iff extensional-def)
next
fix u v assume uv:  $u \in M \ v \in M$ 
then show  $\eta (u \oplus v) = \eta u \oplus_2 \eta v$ 
  by (rule first.hom-add)
next
fix a v assume av:  $a \in R \ v \in M$ 
then show  $\eta (a \odot v) = a \odot_2 \eta v$ 
  by (rule first.hom-scale)
qed
interpret I: module-homomorphism R (+) ( $\cdot$ ) 0 1

```

```

    M (⊕) 0M (⊙)
    (η ' M) (⊕2) 02 (⊙2) η
  by (rule eta-image-hom)
have inj-eta: inj-on η M using short by (simp add: short-exact-def)
have eta-bij: bij-betw η M (η ' M)
  using inj-eta by (simp add: bij-betw-def)
have J-hom: module-homomorphism R (+) (·) 0 1
  (η ' M) (⊕2) 02 (⊙2)
  M (⊕) 0M (⊙)
  (restrict (inv-into M η) (η ' M))
  by (rule I.inverse-module-homomorphism[OF eta-bij])
define ϱ where
  ϱ = compose M' (restrict (inv-into M η) (η ' M))
  (first.target.complement-projection (η ' M) (σ ' M''))
have rho-hom:
  module-homomorphism R (+) (·) 0 1
  M' (⊕2) 02 (⊙2)
  M (⊕) 0M (⊙) ϱ
  unfolding ϱ-def
  by (rule P.compose-module-homomorphism[OF J-hom])
have retract: ⋀x. x ∈ M ⟹ ϱ (η x) = x
proof -
  fix x assume x: x ∈ M
  have eta-x: η x ∈ η ' M by (rule imageI[OF x])
  have eta-xM': η x ∈ M' by (rule first.hom-closed[OF x])
  have projection: first.target.complement-projection (η ' M) (σ ' M'') (η x) = η x
    by (rule first.target.complement-projection-apply-left
      [OF eta-sub sigma-sub disjoint' full' eta-x])
  have inverse: restrict (inv-into M η) (η ' M) (η x) = x
    by (rule I.inverse-module-homomorphism-left[OF eta-bij x])
  show ϱ (η x) = x
    unfolding ϱ-def compose-def
    using projection inverse eta-x eta-xM' by (simp add: restrict-apply')
qed
show ?thesis unfolding left-split-def using rho-hom retract by blast
qed

```

A retraction of the injection makes its image a direct summand. Exactness then identifies the complementary summand with the quotient, so the second map has a carrier-restricted inverse that supplies a right section.

theorem *short-exact-left-split-imp-right-split:*
assumes *short: short-exact* **and** *split: left-split*
shows *right-split*

proof –
obtain ϱ **where** rho-hom: module-homomorphism R (+) (·) 0 1
 M' (⊕₂) 0₂ (⊙₂) M (⊕) 0_M (⊙) ϱ
and retract: ⋀x. x ∈ M ⟹ ϱ (η x) = x
using *split* **unfolding** *left-split-def* **by** *blast*
interpret Rho: module-homomorphism R (+) (·) 0 1

```

     $M' (\oplus_2) \mathbf{0}_2 (\odot_2)$ 
     $M (\oplus) \mathbf{0}_M (\odot) \varrho$ 
    by (rule rho-hom)
  have eta-hom: module-homomorphism  $R (+) (\cdot) \mathbf{0} \mathbf{1}$ 
     $M (\oplus) \mathbf{0}_M (\odot)$ 
     $M' (\oplus_2) \mathbf{0}_2 (\odot_2) \eta$ 
    by unfold-locales
  define p where p = compose  $M' \eta \varrho$ 
  have p-hom: module-homomorphism  $R (+) (\cdot) \mathbf{0} \mathbf{1}$ 
     $M' (\oplus_2) \mathbf{0}_2 (\odot_2)$ 
     $M' (\oplus_2) \mathbf{0}_2 (\odot_2) p$ 
    unfolding p-def
    by (rule Rho.compose-module-homomorphism[OF eta-hom])
  interpret P: idempotent-module-endomorphism  $R (+) (\cdot) \mathbf{0} \mathbf{1}$ 
     $M' (\oplus_2) \mathbf{0}_2 (\odot_2) p$ 
  proof (rule idempotent-module-endomorphism.intro[OF p-hom])
    show idempotent-module-endomorphism-axioms  $M' p$ 
  proof
    fix x assume  $xM'$ :  $x \in M'$ 
    have rx:  $\varrho x \in M$  by (rule Rho.hom-closed[OF  $xM'$ ])
    have etarx:  $\eta (\varrho x) \in M'$  by (rule first.hom-closed[OF rx])
    show  $p (p x) = p x$ 
      unfolding p-def compose-def
      using  $xM'$  rx etarx retract[OF rx] by simp
  qed
qed
  have inj-eta: inj-on  $\eta M$  using short by (simp add: short-exact-def)
  have image-p:  $p \text{ ' } M' = \eta \text{ ' } M$ 
  proof (rule set-eqI, rule iffI)
    fix y assume  $y$ :  $y \in p \text{ ' } M'$ 
    then obtain x where  $x$ :  $x \in M' y = p x$  by blast
    have rx:  $\varrho x \in M$  by (rule Rho.hom-closed[OF  $x(1)$ ])
    have value-eq:  $y = \eta (\varrho x)$ 
      using x unfolding p-def compose-def by simp
    then show  $y \in \eta \text{ ' } M$  using rx by blast
  next
    fix y assume  $y$ :  $y \in \eta \text{ ' } M$ 
    then obtain x where  $x$ :  $x \in M y = \eta x$  by blast
    have etax:  $\eta x \in M'$  by (rule first.hom-closed[OF  $x(1)$ ])
    have value-eq:  $p (\eta x) = \eta x$ 
      unfolding p-def compose-def using x retract[OF  $x(1)$ ] by simp
    have image-value:  $p (\eta x) \in p \text{ ' } M'$  by (rule imageI[OF etax])
    then show  $y \in p \text{ ' } M'$  using value-eq  $x(2)$  by simp
  qed
  have Pker-sub: first.target.submodule  $P.Ker$  using  $P.Ker$ -submodule .
  have eta-sub: first.target.submodule  $(\eta \text{ ' } M)$  using first.image-submodule .
  have disjoint-P:  $P.Ker \cap (\eta \text{ ' } M) = \{\mathbf{0}_2\}$ 
    using  $P$ .image-kernel-complementary(1) image-p by (simp add: Int-commute)
  have full-eta-P: Module.submodule-sum  $(\oplus_2) (\eta \text{ ' } M) P.Ker = M'$ 
    using  $P$ .image-kernel-complementary(2) image-p by simp

```

```

have full-P-eta: Module.submodule-sum ( $\oplus_2$ ) P.Ker ( $\eta \text{ ' } M$ ) =  $M'$ 
using full-eta-P first.target.submodule-sum-commute[OF Pker-sub eta-sub] by simp
have exact:  $\eta \text{ ' } M = \text{second.Ker}$  using short by (simp add: short-exact-iff)
have theta-restrict-hom:
  module-homomorphism R (+) ( $\cdot$ ) 0 1
  P.Ker ( $\oplus_2$ ) 02 ( $\odot_2$ )
  M'' ( $\oplus_3$ ) 03 ( $\odot_3$ )
  (restrict  $\vartheta$  P.Ker)
proof (intro module-homomorphism.intro map.intro module-homomorphism-axioms.intro)
  show Module R (+) ( $\cdot$ ) 0 1
    ( $\oplus_2$ ) 02 P.Ker ( $\odot_2$ )
    by (rule first.target.submodule-is-module[OF Pker-sub])
  show Module R (+) ( $\cdot$ ) 0 1
    ( $\oplus_3$ ) 03 M'' ( $\odot_3$ )
    by (rule second.target.Module-axioms)
  show restrict  $\vartheta$  P.Ker  $\in$  P.Ker  $\rightarrow_E$  M''
    using first.target.submodule-subset[OF Pker-sub] second.hom-closed
    by (auto simp: PiE-iff extensional-def)
next
fix u v assume uv:  $u \in P.Ker$   $v \in P.Ker$ 
have uM':  $u \in M'$  and vM':  $v \in M'$ 
  using uv first.target.submodule-subset[OF Pker-sub] by auto
have uvP:  $u \oplus_2 v \in P.Ker$ 
  by (rule first.target.submodule-add[OF Pker-sub uv(1) uv(2)])
show restrict  $\vartheta$  P.Ker ( $u \oplus_2 v$ ) =
  restrict  $\vartheta$  P.Ker  $u \oplus_3$  restrict  $\vartheta$  P.Ker  $v$ 
  using uv uvP uM' vM' by (simp add: restrict-apply' second.hom-add)
next
fix a v assume av:  $a \in R$   $v \in P.Ker$ 
have vM':  $v \in M'$  using av(2) first.target.submodule-subset[OF Pker-sub] by blast
have avP:  $a \odot_2 v \in P.Ker$ 
  by (rule first.target.submodule-scale[OF Pker-sub av(1) av(2)])
show restrict  $\vartheta$  P.Ker ( $a \odot_2 v$ ) =
  a  $\odot_3$  restrict  $\vartheta$  P.Ker  $v$ 
  using av avP vM' by (simp add: restrict-apply' second.hom-scale)
qed
interpret T: module-homomorphism R (+) ( $\cdot$ ) 0 1
  P.Ker ( $\oplus_2$ ) 02 ( $\odot_2$ )
  M'' ( $\oplus_3$ ) 03 ( $\odot_3$ ) restrict  $\vartheta$  P.Ker
  by (rule theta-restrict-hom)
have T-kernel: T.Ker = {02}
proof (rule set-eqI, rule iffI)
  fix x assume x:  $x \in T.Ker$ 
  have xP:  $x \in P.Ker$  by (rule T.Ker-mem[OF x])
  have xM':  $x \in M'$  using xP first.target.submodule-subset[OF Pker-sub] by blast
  have theta-x:  $\vartheta x = 0_3$ 
    using T.Ker-image[OF x] xP by (simp add: restrict-apply')
  have xeta:  $x \in \eta \text{ ' } M$ 
  proof -
    have xsecond:  $x \in \text{second.Ker}$  by (rule second.Ker-memI[OF theta-x xM'])

```

```

    then show ?thesis using exact by simp
  qed
  have  $x \in P.Ker \cap (\eta \text{ ' } M)$  by (rule IntI[OF xP xeta])
  then show  $x \in \{0_2\}$  using disjoint-P by simp
next
  fix x assume  $x: x \in \{0_2\}$ 
  have zeroP:  $0_2 \in P.Ker$  by (rule first.target.submodule-zero[OF Pker-sub])
  have zeroT:  $restrict \vartheta P.Ker \ 0_2 = 0_3$  by (rule T.hom-zero)
  show  $x \in T.Ker$  using x zeroP zeroT by (simp add: T.Ker-def)
qed
have T-inj:  $inj\text{-}on (restrict \vartheta P.Ker) P.Ker$ 
  using T.injective-iff-kernel-trivial T-kernel by blast
have T-image:  $(restrict \vartheta P.Ker) \text{ ' } P.Ker = M''$ 
proof (rule set-eqI, rule iffI)
  fix z assume  $z: z \in (restrict \vartheta P.Ker) \text{ ' } P.Ker$ 
  then obtain u where  $u: u \in P.Ker \ z = restrict \vartheta P.Ker \ u$  by blast
  have uM':  $u \in M'$  using u(1) first.target.submodule-subset[OF Pker-sub] by blast
  show  $z \in M''$  using u(1) u(2) second.hom-closed[OF uM'] by (simp add: restrict-apply')
next
  fix z assume  $z: z \in M''$ 
  have theta-surj:  $\vartheta \text{ ' } M' = M''$  using short by (simp add: short-exact-iff)
  have zimage:  $z \in \vartheta \text{ ' } M'$  using z theta-surj by simp
  then obtain y where  $y: y \in M' \ z = \vartheta \ y$  by blast
  have ysum:  $y \in Module.submodule\text{-}sum (\oplus_2) P.Ker (\eta \text{ ' } M)$ 
  using y(1) full-P-eta by simp
  then obtain u v where  $uv: u \in P.Ker \ v \in \eta \text{ ' } M \ y = u \oplus_2 v$ 
  by (rule first.target.submodule-sum-memE)
  have uM':  $u \in M'$  using uv(1) first.target.submodule-subset[OF Pker-sub] by blast
blast
  have vM':  $v \in M'$  using uv(2) first.target.submodule-subset[OF eta-sub] by blast
  have theta-v:  $\vartheta \ v = 0_3$ 
  proof -
    have vker:  $v \in second.Ker$  using uv(2) exact by simp
    then show ?thesis by (rule second.Ker-image)
  qed
  have theta-u:  $\vartheta \ u = z$ 
  using y uv(3) uM' vM' theta-v by (simp add: second.hom-add)
  have image-u:  $(restrict \vartheta P.Ker) \ u \in$ 
     $(restrict \vartheta P.Ker) \text{ ' } P.Ker$  by (rule imageI[OF uv(1)])
  show  $z \in (restrict \vartheta P.Ker) \text{ ' } P.Ker$ 
  using image-u theta-u uv(1) by (simp add: restrict-apply')
qed
have T-bij:  $bij\text{-}betw (restrict \vartheta P.Ker) P.Ker M''$ 
  using T-inj T-image by (simp add: bij-betw-def)
have inverse-hom:
   $module\text{-}homomorphism \ R \ (+) \ (\cdot) \ 0 \ 1$ 
   $M'' (\oplus_3) 0_3 (\odot_3)$ 
   $P.Ker (\oplus_2) 0_2 (\odot_2)$ 
   $(restrict (inv\text{-}into P.Ker (restrict \vartheta P.Ker)) M'')$ 

```

```

    by (rule T.inverse-module-homomorphism[OF T-bij])
  interpret U: module-homomorphism R (+) (·) 0 1
    M'' (⊕3) 03 (⊙3)
    P.Ker (⊕2) 02 (⊙2)
    restrict (inv-into P.Ker (restrict ∅ P.Ker)) M''
  by (rule inverse-hom)
  have inclusion-hom:
    module-homomorphism R (+) (·) 0 1
    P.Ker (⊕2) 02 (⊙2)
    M' (⊕2) 02 (⊙2) (identity P.Ker)
  proof (intro module-homomorphism.intro map.intro module-homomorphism-axioms.intro)
    show Module R (+) (·) 0 1
      (⊕2) 02 P.Ker (⊙2)
    by (rule first.target.submodule-is-module[OF Pker-sub])
    show Module R (+) (·) 0 1
      (⊕2) 02 M' (⊙2)
    by (rule first.target.Module-axioms)
    show identity P.Ker ∈ P.Ker →E M'
      using first.target.submodule-subset[OF Pker-sub] by (auto simp: PiE-iff)
  next
    fix u v assume u: u ∈ P.Ker and v: v ∈ P.Ker
    have uv: u ⊕2 v ∈ P.Ker
      by (rule first.target.submodule-add[OF Pker-sub u v])
    then show identity P.Ker (u ⊕2 v) =
      identity P.Ker u ⊕2 identity P.Ker v
      using u v by simp
  next
    fix a v assume a: a ∈ R and v: v ∈ P.Ker
    have av: a ⊙2 v ∈ P.Ker
      by (rule first.target.submodule-scale[OF Pker-sub a v])
    then show identity P.Ker (a ⊙2 v) =
      a ⊙2 identity P.Ker v
      using v by simp
  qed
  define σ where
    σ = compose M'' (identity P.Ker)
      (restrict (inv-into P.Ker (restrict ∅ P.Ker)) M'')
  have section-hom:
    module-homomorphism R (+) (·) 0 1
    M'' (⊕3) 03 (⊙3)
    M' (⊕2) 02 (⊙2) σ
  unfolding σ-def
  by (rule U.compose-module-homomorphism[OF inclusion-hom])
  have section-eq: ⋀z. z ∈ M'' ⟹ ∅ (σ z) = z
  proof -
    fix z assume z: z ∈ M''
    have uz: restrict (inv-into P.Ker (restrict ∅ P.Ker)) M'' z ∈ P.Ker
      by (rule U.hom-closed[OF z])
    have inverse: restrict ∅ P.Ker
      (restrict (inv-into P.Ker (restrict ∅ P.Ker)) M'' z) = z
  
```

```

    by (rule T.inverse-module-homomorphism-right[OF T-bij z])
  have sigma-apply:  $\sigma z =$ 
    restrict (inv-into P.Ker (restrict  $\vartheta$  P.Ker))  $M'' z$ 
  unfolding  $\sigma$ -def compose-def
  using uz z by (simp add: restrict-apply')
  have inverse-value:  $\vartheta$ 
    (restrict (inv-into P.Ker (restrict  $\vartheta$  P.Ker))  $M'' z$ ) =
    restrict  $\vartheta$  P.Ker
    (restrict (inv-into P.Ker (restrict  $\vartheta$  P.Ker))  $M'' z$ )
  by (rule sym, rule restrict-apply'[OF uz])
  have theta-sigma:  $\vartheta (\sigma z) = \vartheta$ 
    (restrict (inv-into P.Ker (restrict  $\vartheta$  P.Ker))  $M'' z$ )
  by (simp only: sigma-apply)
  have theta-restrict:  $\vartheta$ 
    (restrict (inv-into P.Ker (restrict  $\vartheta$  P.Ker))  $M'' z$ ) =
    restrict  $\vartheta$  P.Ker
    (restrict (inv-into P.Ker (restrict  $\vartheta$  P.Ker))  $M'' z$ )
  by (rule inverse-value)
  have theta-intermediate:  $\vartheta (\sigma z) =$ 
    restrict  $\vartheta$  P.Ker
    (restrict (inv-into P.Ker (restrict  $\vartheta$  P.Ker))  $M'' z$ )
  by (rule trans[OF theta-sigma theta-restrict])
  show  $\vartheta (\sigma z) = z$ 
  by (rule trans[OF theta-intermediate inverse])
qed
show ?thesis unfolding right-split-def using section-hom section-eq by blast
qed

```

```

theorem short-exact-right-split-iff-left-split:
  assumes short: short-exact
  shows right-split  $\longleftrightarrow$  left-split
  using short-exact-right-split-imp-left-split[OF short]
    short-exact-left-split-imp-right-split[OF short] by blast

```

end

```

context Module
begin

```

The carrier-restricted identity on a submodule is its inclusion into the ambient module.

```

theorem submodule-inclusion-hom:
  assumes N: submodule N
  shows module-homomorphism R (+) ( $\cdot$ )  $\mathbf{0}$   $\mathbf{1}$ 
    N ( $\oplus$ )  $\mathbf{0}_M$  ( $\odot$ ) M ( $\oplus$ )  $\mathbf{0}_M$  ( $\odot$ ) (identity N)
proof (intro module-homomorphism.intro map.intro module-homomorphism-axioms.intro)
  show Module R (+) ( $\cdot$ )  $\mathbf{0}$   $\mathbf{1}$  ( $\oplus$ )  $\mathbf{0}_M$  N ( $\odot$ )
  by (rule submodule-is-module[OF N])
  show Module R (+) ( $\cdot$ )  $\mathbf{0}$   $\mathbf{1}$  ( $\oplus$ )  $\mathbf{0}_M$  M ( $\odot$ )

```

```

    by (rule Module-axioms)
  show identity  $N \in N \rightarrow_E M$ 
    using submodule-subset[OF N] by (auto simp: PiE-iff)
next
  fix  $u\ v$  assume  $u: u \in N$  and  $v: v \in N$ 
  have  $uv: u \oplus v \in N$  using N  $u\ v$  by (rule submodule-add)
  then show identity  $N (u \oplus v) = \text{identity } N\ u \oplus \text{identity } N\ v$ 
    using  $u\ v$  by simp
next
  fix  $a\ v$  assume  $a: a \in R$  and  $v: v \in N$ 
  have  $av: a \odot v \in N$  using N  $a\ v$  by (rule submodule-scale)
  then show identity  $N (a \odot v) = a \odot \text{identity } N\ v$ 
    using  $v$  by simp
qed

```

Complementary submodules give the canonical split short exact sequence $0 \longrightarrow K \longrightarrow M \longrightarrow N \longrightarrow 0$. The inclusion of N is a section of the complementary projection.

```

theorem complementary-submodules-split-short-exact:
  assumes N: submodule N and K: submodule K
    and disjoint:  $N \cap K = \{0_M\}$ 
    and full:  $N \oplus_S K = M$ 
  shows composable-module-homomorphisms.short-exact
     $K\ M\ N\ 0_M\ (\text{identity } K)\ (\text{complement-projection } N\ K)$ 
  ∧ composable-module-homomorphisms.right-split
     $R\ (+)\ (\cdot)\ 0\ 1$ 
     $M\ (\oplus)\ 0_M\ (\odot)\ N\ (\oplus)\ 0_M\ (\odot)$ 
    ( $\text{complement-projection } N\ K$ )
proof -
  interpret I: module-homomorphism  $R\ (+)\ (\cdot)\ 0\ 1$ 
     $K\ (\oplus)\ 0_M\ (\odot)\ M\ (\oplus)\ 0_M\ (\odot)\ \text{identity } K$ 
    by (rule submodule-inclusion-hom[OF K])
  interpret P: module-homomorphism  $R\ (+)\ (\cdot)\ 0\ 1$ 
     $M\ (\oplus)\ 0_M\ (\odot)\ N\ (\oplus)\ 0_M\ (\odot)\ \text{complement-projection } N\ K$ 
    by (rule complement-projection-hom[OF N K disjoint full])
  interpret E: composable-module-homomorphisms  $R\ (+)\ (\cdot)\ 0\ 1$ 
     $K\ (\oplus)\ 0_M\ (\odot)\ M\ (\oplus)\ 0_M\ (\odot)$ 
     $N\ (\oplus)\ 0_M\ (\odot)\ \text{identity } K\ \text{complement-projection } N\ K$ 
    by unfold-locales
  have image-I: identity  $K \text{ ' } K = K$  by auto
  have ker-P:  $P.\text{Ker} = K$ 
    using complement-projection-Ker[OF N K disjoint full]
    by (simp add: P.Ker-def)
  have image-P: complement-projection  $N\ K \text{ ' } M = N$ 
    by (rule complement-projection-image[OF N K disjoint full])
  have short: E.short-exact
    by (simp add: E.short-exact-def E.exact-def image-I ker-P image-P)
  have section-hom: module-homomorphism  $R\ (+)\ (\cdot)\ 0\ 1$ 
     $N\ (\oplus)\ 0_M\ (\odot)\ M\ (\oplus)\ 0_M\ (\odot)\ (\text{identity } N)$ 

```



```

    by (rule submodule-inclusion-hom[OF N])
  have sec:  $\forall x \in N. \text{complement-projection } N \ K \ (\text{identity } N \ x) = x$ 
    using complement-projection-apply-left[OF N K disjoint full] by simp
  have split:  $E.\text{right-split}$ 
    unfolding  $E.\text{right-split-def}$  using section-hom sec by blast
  show ?thesis using short split by simp
qed

end

end

```

90 The direct product and direct sum of a family of modules

```

theory Module-Family-Product
  imports Module-Homomorphism
begin

```

```

no-notation plus (infixl  $\langle + \rangle$  65)
no-notation minus (infixl  $\langle - \rangle$  65)
unbundle no uminus-syntax

```

The external direct product of a family of R -modules $\text{Mod } i$ ($i \in I$): the carrier is the extensional dependent function space $Pi_E \ I \ \text{Mod}$ with pointwise addition and scaling. As with the homomorphism locale, the scalar ring R is fixed *once* and shared by all the modules. We show the product is an R -module, the projections are module homomorphisms, and the direct *sum* (elements of finite support) is a submodule of the product.

90.1 The product locale

```

locale module-family =
  base: Ring R (+) ( $\cdot$ ) 0 1
  for R and addition (infixl  $\langle + \rangle$  65) and multiplication (infixl  $\langle \cdot \rangle$  70) and zero
  ( $\langle 0 \rangle$ ) and unit ( $\langle 1 \rangle$ ) +
  fixes I :: 'i set
  and Mod :: 'i  $\Rightarrow$  'b set
  and madd :: 'i  $\Rightarrow$  'b  $\Rightarrow$  'b  $\Rightarrow$  'b
  and mzero :: 'i  $\Rightarrow$  'b
  and scale :: 'i  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  'b
  assumes mod:  $\bigwedge i. i \in I \implies$ 
    Module R (+) ( $\cdot$ ) 0 1 (madd i) (mzero i) (Mod i) (scale i)
begin

lemma comp-module:  $i \in I \implies$  Module R (+) ( $\cdot$ ) 0 1 (madd i) (mzero i) (Mod i)
(scale i)
  by (rule mod)

```

90.2 Carrier and operations

definition $Pcarrier :: ('i \Rightarrow 'b) \text{ set}$
where $Pcarrier = (\Pi_E \ i \in I. \text{Mod } i)$

definition $Padd :: ('i \Rightarrow 'b) \Rightarrow ('i \Rightarrow 'b) \Rightarrow ('i \Rightarrow 'b) \text{ (infixl } \langle \oplus_\Pi \rangle \text{ 65)}$
where $x \oplus_\Pi y = (\lambda i \in I. \text{madd } i \ (x \ i) \ (y \ i))$

definition $Pzero :: 'i \Rightarrow 'b \text{ (}\langle \mathbf{0}_\Pi \rangle\text{)}$
where $\mathbf{0}_\Pi = (\lambda i \in I. \text{mzero } i)$

definition $Pscale :: 'a \Rightarrow ('i \Rightarrow 'b) \Rightarrow ('i \Rightarrow 'b) \text{ (infixr } \langle \odot_\Pi \rangle \text{ 75)}$
where $a \odot_\Pi x = (\lambda i \in I. \text{scale } i \ a \ (x \ i))$

lemma $Pcarrier\text{-mem}I$:
 $\llbracket \bigwedge i. i \in I \implies x \ i \in \text{Mod } i; x \in \text{extensional } I \rrbracket \implies x \in Pcarrier$
by $(\text{auto simp: } Pcarrier\text{-def } PiE\text{-iff})$

lemma $Pcarrier\text{-component}$: $\llbracket x \in Pcarrier; i \in I \rrbracket \implies x \ i \in \text{Mod } i$
by $(\text{auto simp: } Pcarrier\text{-def } PiE\text{-iff})$

lemma $Pcarrier\text{-extensional}$: $x \in Pcarrier \implies x \in \text{extensional } I$
by $(\text{auto simp: } Pcarrier\text{-def } PiE\text{-iff})$

lemma $Pcarrier\text{-eq}I$:
 $\llbracket x \in Pcarrier; y \in Pcarrier; \bigwedge i. i \in I \implies x \ i = y \ i \rrbracket \implies x = y$
by $(\text{metis } Pcarrier\text{-extensional extensionality}I)$

lemma $Padd\text{-apply}$: $i \in I \implies (x \oplus_\Pi y) \ i = \text{madd } i \ (x \ i) \ (y \ i)$
by $(\text{simp add: } Padd\text{-def})$

lemma $Pzero\text{-apply}$: $i \in I \implies \mathbf{0}_\Pi \ i = \text{mzero } i$
by $(\text{simp add: } Pzero\text{-def})$

lemma $Pscale\text{-apply}$: $i \in I \implies (a \odot_\Pi x) \ i = \text{scale } i \ a \ (x \ i)$
by $(\text{simp add: } Pscale\text{-def})$

lemma $Padd\text{-closed}$ $[intro, simp]$:
assumes $x \in Pcarrier \ y \in Pcarrier$ **shows** $x \oplus_\Pi y \in Pcarrier$
proof $(\text{rule } Pcarrier\text{-mem}I)$
fix i **assume** $i: i \in I$
interpret Mi : $\text{Module } R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \ \text{madd } i \ \text{mzero } i \ \text{Mod } i \ \text{scale } i$
using i **by** $(\text{rule } mod)$
show $(x \oplus_\Pi y) \ i \in \text{Mod } i$ **using** $assms \ i$ **by** $(\text{simp add: } Padd\text{-apply } Pcarrier\text{-component})$
qed $(\text{auto simp: } Padd\text{-def})$

lemma $Pzero\text{-closed}$ $[intro, simp]$: $\mathbf{0}_\Pi \in Pcarrier$
using $Pcarrier\text{-def } Pzero\text{-def } mod \ \text{Module.mzero-closed}$ **by** fastforce

lemma $Pscale\text{-closed}$ $[intro, simp]$:

assumes $a \in R$ and $x \in Pcarrier$ shows $a \odot_{\Pi} x \in Pcarrier$
 using $Pcarrier-def$ $Pscale-def$ mod $Module.scale-closed$ $assms$ by $fastforce$

90.3 The additive abelian group of the product

```

lemma Padd-abelian-group: Abelian-Group Pcarrier  $(\oplus_{\Pi}) \mathbf{0}_{\Pi}$ 
proof (rule Abelian-Group.intro)
  show  $0: Group\ Pcarrier\ (\oplus_{\Pi})\ \mathbf{0}_{\Pi}$ 
  proof (intro GroupI Pzero-closed Padd-closed)
    fix  $x\ y\ z$  assume  $xyz: x \in Pcarrier\ y \in Pcarrier\ z \in Pcarrier$ 
    show  $(x \oplus_{\Pi} y) \oplus_{\Pi} z = x \oplus_{\Pi} (y \oplus_{\Pi} z)$ 
    proof (rule Pcarrier-eqI)
      fix  $i$  assume  $i: i \in I$ 
      interpret  $Mi: Module\ R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}\ madd\ i\ mzero\ i\ Mod\ i\ scale\ i$ 
      using  $i$  by (rule mod)
      show  $((x \oplus_{\Pi} y) \oplus_{\Pi} z)\ i = (x \oplus_{\Pi} (y \oplus_{\Pi} z))\ i$ 
      using  $xyz\ i$  by (simp add: Padd-apply Pcarrier-component Mi.madd.associative)
    qed (use  $xyz$  in auto)
  next
    fix  $x$  assume  $x: x \in Pcarrier$ 
    show  $\mathbf{0}_{\Pi} \oplus_{\Pi} x = x$ 
    proof (rule Pcarrier-eqI)
      fix  $i$  assume  $i: i \in I$ 
      interpret  $Mi: Module\ R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}\ madd\ i\ mzero\ i\ Mod\ i\ scale\ i$ 
      using  $i$  by (rule mod)
      show  $(\mathbf{0}_{\Pi} \oplus_{\Pi} x)\ i = x\ i$ 
      using  $x\ i$  by (simp add: Padd-apply Pzero-apply Pcarrier-component)
    qed (use  $x$  in auto)
  next
    fix  $x$  assume  $x: x \in Pcarrier$ 
    show  $x \oplus_{\Pi} \mathbf{0}_{\Pi} = x$ 
    proof (rule Pcarrier-eqI)
      fix  $i$  assume  $i: i \in I$ 
      interpret  $Mi: Module\ R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}\ madd\ i\ mzero\ i\ Mod\ i\ scale\ i$ 
      using  $i$  by (rule mod)
      show  $(x \oplus_{\Pi} \mathbf{0}_{\Pi})\ i = x\ i$ 
      using  $x\ i$  by (simp add: Padd-apply Pzero-apply Pcarrier-component)
    qed (use  $x$  in auto)
  next
    fix  $x$  assume  $x: x \in Pcarrier$ 
    have  $\exists v. \forall i \in I. v\ i \in Mod\ i \wedge madd\ i\ (x\ i)\ (v\ i) = mzero\ i \wedge madd\ i\ (v\ i)\ (x\ i) = mzero\ i$ 
    proof (rule bchoice, intro ballI)
      fix  $i$  assume  $i: i \in I$ 
      interpret  $Mi: Module\ R\ (+)\ (\cdot)\ \mathbf{0}\ \mathbf{1}\ madd\ i\ mzero\ i\ Mod\ i\ scale\ i$ 
      using  $i$  by (rule mod)
      show  $\exists v. v \in Mod\ i \wedge madd\ i\ (x\ i)\ v = mzero\ i \wedge madd\ i\ v\ (x\ i) = mzero\ i$ 
      using  $Mi.madd.Monoid-axioms\ Monoid.invertible-inverse-closed\ Pcarrier-component$ 
    qed
  next
    fix  $x$ 
    by fastforce

```

```

qed
then obtain v where v:  $\bigwedge i. i \in I \implies v\ i \in \text{Mod } i$ 
       $\wedge \text{madd } i\ (x\ i)\ (v\ i) = \text{mzero } i \wedge \text{madd } i\ (v\ i)\ (x\ i) = \text{mzero } i$  by blast
show  $\exists y \in \text{Pcarrier}. x \oplus_{\Pi} y = \mathbf{0}_{\Pi} \wedge y \oplus_{\Pi} x = \mathbf{0}_{\Pi}$ 
proof
  define y where  $y = (\lambda i \in I. v\ i)$ 
  show  $yP: y \in \text{Pcarrier}$  by (rule Pcarrier-memI) (use v in <auto simp: y-def>)
  show  $x \oplus_{\Pi} y = \mathbf{0}_{\Pi} \wedge y \oplus_{\Pi} x = \mathbf{0}_{\Pi}$ 
    using Padd-def Pzero-def y-def v by auto
qed
qed
show commutative-monoid Pcarrier  $(\oplus_{\Pi})\ \mathbf{0}_{\Pi}$ 
proof (rule commutative-monoid.intro)
  show Monoid Pcarrier  $(\oplus_{\Pi})\ \mathbf{0}_{\Pi}$ 
  proof
    fix x y z assume xyz:  $x \in \text{Pcarrier } y \in \text{Pcarrier } z \in \text{Pcarrier}$ 
    show  $(x \oplus_{\Pi} y) \oplus_{\Pi} z = x \oplus_{\Pi} (y \oplus_{\Pi} z)$ 
      using 0 Group-def Monoid.associative xyz by fastforce
  next
    fix x assume x:  $x \in \text{Pcarrier}$ 
    show  $\mathbf{0}_{\Pi} \oplus_{\Pi} x = x$ 
      by (meson 0 Group-def Monoid.left-unit x)
    show  $x \oplus_{\Pi} \mathbf{0}_{\Pi} = x$ 
      by (meson Group-def Monoid.right-unit 0 x)
  qed auto
show commutative-monoid-axioms Pcarrier  $(\oplus_{\Pi})$ 
proof
  fix x y assume xy:  $x \in \text{Pcarrier } y \in \text{Pcarrier}$ 
  show  $x \oplus_{\Pi} y = y \oplus_{\Pi} x$ 
  proof (intro Pcarrier-eqI)
    fix i assume i:  $i \in I$ 
    interpret Mi: Module R (+) ( $\cdot$ )  $\mathbf{0}\ \mathbf{1}$  madd i mzero i Mod i scale i
    using i by (rule mod)
    show  $(x \oplus_{\Pi} y)\ i = (y \oplus_{\Pi} x)\ i$ 
    using xy i by (simp add: Padd-apply Pcarrier-component Mi.madd.commutative)
  qed (use xy in auto)
qed
qed
qed

```

90.4 The product is a module

```

theorem module-Pcarrier: Module R (+) ( $\cdot$ )  $\mathbf{0}\ \mathbf{1}\ (\oplus_{\Pi})\ \mathbf{0}_{\Pi}\ \text{Pcarrier}\ (\odot_{\Pi})$ 
proof (intro Module.intro Module-axioms.intro base.Ring-axioms Padd-abelian-group)
  fix a x assume a  $\in R\ x \in \text{Pcarrier}$  then show  $a \odot_{\Pi} x \in \text{Pcarrier}$  by (rule
Pscale-closed)
next
  fix a x y assume a:  $a \in R$  and xy:  $x \in \text{Pcarrier } y \in \text{Pcarrier}$ 
  show  $a \odot_{\Pi} (x \oplus_{\Pi} y) = (a \odot_{\Pi} x) \oplus_{\Pi} (a \odot_{\Pi} y)$ 
  proof (rule Pcarrier-eqI)

```

```

fix  $i$  assume  $i: i \in I$ 
interpret  $Mi$ : Module  $R$   $(+)$   $(\cdot)$  0 1 madd  $i$  mzero  $i$  Mod  $i$  scale  $i$ 
  using  $i$  by (rule mod)
  show  $(a \odot_{\Pi} (x \oplus_{\Pi} y)) i = ((a \odot_{\Pi} x) \oplus_{\Pi} (a \odot_{\Pi} y)) i$ 
    using  $a$   $xy$   $i$  by (simp add: Pscale-apply Padd-apply Pcarrier-component
Mi.scale-distrib-madd)
  qed (use a xy in auto)
next
  fix  $a$   $b$   $x$  assume  $ab: a \in R$   $b \in R$  and  $x: x \in Pcarrier$ 
  then show  $(a + b) \odot_{\Pi} x = (a \odot_{\Pi} x) \oplus_{\Pi} (b \odot_{\Pi} x)$ 
    using Pcarrier-component Pscale-apply mod
    by (fastforce intro!: Pcarrier-eqI simp: Padd-apply Module.scale-distrib-add) +
  show  $(a \cdot b) \odot_{\Pi} x = a \odot_{\Pi} (b \odot_{\Pi} x)$ 
    using Pcarrier-component Pscale-apply ab mod Module.scale-scale x
    by (fastforce intro!: Pcarrier-eqI)
  next
  fix  $x$  assume  $x \in Pcarrier$ 
  then show  $1 \odot_{\Pi} x = x$ 
    using Pcarrier-component Pscale-apply mod Module.scale-one
    by (fastforce intro!: Pcarrier-eqI)
qed

sublocale product: Module  $R$   $(+)$   $(\cdot)$  0 1  $(\oplus_{\Pi})$  Pzero Pcarrier  $(\odot_{\Pi})$ 
  by (rule module-Pcarrier)

```

90.5 Projections

definition $Pproj :: 'i \Rightarrow ('i \Rightarrow 'b) \Rightarrow 'b$
where $Pproj\ i = \text{restrict } (\lambda x. x\ i)\ Pcarrier$

lemma *Pproj-apply*: $x \in Pcarrier \implies Pproj\ i\ x = x\ i$
by (*simp add: Pproj-def*)

Each projection is additive and commutes with scaling (it is a module homomorphism onto the factor *Mod* i); we record the two defining equations.

lemma *Pproj-add*: $\llbracket i \in I; x \in Pcarrier; y \in Pcarrier \rrbracket \implies Pproj\ i\ (x \oplus_{\Pi} y) = madd\ i\ (Pproj\ i\ x)\ (Pproj\ i\ y)$
by (*simp add: Pproj-apply Padd-apply*)

lemma *Pproj-scale*: $\llbracket i \in I; a \in R; x \in Pcarrier \rrbracket \implies Pproj\ i\ (a \odot_{\Pi} x) = scale\ i\ a\ (Pproj\ i\ x)$
by (*simp add: Pproj-apply Pscale-apply*)

90.6 The direct sum

The *direct sum* consists of the product elements of *finite support*: those x equal to the zero element in all but finitely many factors. It is a submodule of the product.

definition $Psum :: ('i \Rightarrow 'b)\ set$

```

where  $Psum = \{x \in Pcarrier. \text{finite } \{i \in I. x\ i \neq mzero\ i\}\}$ 

lemma  $Psum\text{-subset}: Psum \subseteq Pcarrier$ 
by (auto simp: Psum-def)

lemma  $Psum\text{-memI}$ :
 $\llbracket x \in Pcarrier; \text{finite } \{i \in I. x\ i \neq mzero\ i\} \rrbracket \implies x \in Psum$ 
by (simp add: Psum-def)

theorem  $Psum\text{-submodule}: \text{product.submodule } Psum$ 
proof (rule product.submoduleI)
  show  $Psum \subseteq Pcarrier$  by (rule Psum-subset)
next
  show  $0_\Pi \in Psum$  using  $Pzero\text{-closed}$ 
  by (simp add: Psum-memI Pzero-def)
next
  fix  $x\ y$  assume  $x: x \in Psum$  and  $y: y \in Psum$ 
  have  $xP: x \in Pcarrier$  and  $yP: y \in Pcarrier$  using  $x\ y$  by (auto simp: Psum-def)
  — The support of a sum is contained in the union of the supports.
  have  $\bigwedge i. \llbracket i \in I; y\ i = mzero\ i; x\ i = mzero\ i \rrbracket \implies (x \oplus_\Pi y)\ i = mzero\ i$ 
  by (metis Padd-apply Pzero-apply product.madd.left-unit xP)
  then have  $\{i \in I. (x \oplus_\Pi y)\ i \neq mzero\ i\} \subseteq \{i \in I. x\ i \neq mzero\ i\} \cup \{i \in I. y\ i \neq mzero\ i\}$ 
  by auto
  moreover have  $\text{finite } (\{i \in I. x\ i \neq mzero\ i\} \cup \{i \in I. y\ i \neq mzero\ i\})$ 
  using  $x\ y$  by (auto simp: Psum-def)
  ultimately show  $x \oplus_\Pi y \in Psum$  using  $xP\ yP\ Psum\text{-memI}\ \text{finite-subset}$  by blast
next
  fix  $a\ x$  assume  $a: a \in R$  and  $x: x \in Psum$ 
  have  $xP: x \in Pcarrier$  using  $x$  by (simp add: Psum-def)
  — Scaling can only shrink the support.
  have  $\bigwedge i. \llbracket i \in I; x\ i = mzero\ i \rrbracket \implies (a \odot_\Pi x)\ i = mzero\ i$ 
  by (metis Pscale-apply Pzero-apply a product.scale-zero-elem)
  then have  $\{i \in I. (a \odot_\Pi x)\ i \neq mzero\ i\} \subseteq \{i \in I. x\ i \neq mzero\ i\}$ 
  by auto
  moreover have  $\text{finite } \{i \in I. x\ i \neq mzero\ i\}$  using  $x$  by (simp add: Psum-def)
  ultimately have  $\text{finite } \{i \in I. (a \odot_\Pi x)\ i \neq mzero\ i\}$  by (rule finite-subset)
  then show  $a \odot_\Pi x \in Psum$  using  $a\ xP$  by (auto intro: Psum-memI)
qed

end

end

```

91 The universal property of a free module

```

theory Free-Module-Universal
imports Free-Module Module-Homomorphism
begin

```

As in *Module-Homomorphism*: the import of *FiniteProduct* (through *Module*) re-merges HOL's $+/−$ concrete syntax alongside the ring locale's. That is harmless in term positions but ambiguous in the locale-header instantiation arguments below, so we remove the HOL syntax again.

no-notation *plus* (**infixl** $\langle + \rangle$ 65)
no-notation *minus* (**infixl** $\langle - \rangle$ 65)
unbundle *no uminus-syntax*

A free module is free in the sense of a universal property: a homomorphism out of it may be prescribed *arbitrarily* on a basis, and is then determined. Precisely: if B is a basis of the R -module M and f maps B into an R -module M' , then there is exactly one module homomorphism $M \rightarrow M'$ extending f .

This is what the uniqueness-of-coordinates results in *Free-Module* were for. Existence needs coordinates to be *readable* — every element has a representation over a finite subset of the basis — and well-definedness needs them to be *unique* in the strong sense of *lin-indep-lincomb-unique-gen*, across different index sets, since nothing fixes an index set in advance.

The locale below fixes the ring once in the *for* clause and shares it between the two modules, exactly as *module-homomorphism* does; writing the two module interpretations with separate ring parameters would make the $+$ token ambiguous.

locale *free-module-ext* =
 source: *Module* R $(+)$ $(\cdot)$ **0** **1** $(\oplus)$ $\mathbf{0}_M$ M $(\odot)$ $+$
 target: *Module* R $(+)$ $(\cdot)$ **0** **1** $(\oplus_2)$ $\mathbf{0}_2$ M' $(\odot_2)$
 for R **and** *addition* (**infixl** $\langle + \rangle$ 65) **and** *multiplication* (**infixl** $\langle \cdot \rangle$ 70) **and** *zero* $(\langle \mathbf{0} \rangle)$ **and** *unit* $(\langle \mathbf{1} \rangle)$
 and M **and** *madd* (**infixl** $\langle \oplus \rangle$ 65) **and** *mzero* $(\langle \mathbf{0}_M \rangle)$ **and** *scale* (**infixr** $\langle \odot \rangle$ 75)
 and M' **and** *madd'* (**infixl** $\langle \oplus_2 \rangle$ 65) **and** *mzero'* $(\langle \mathbf{0}_2 \rangle)$ **and** *scale'* (**infixr** $\langle \odot_2 \rangle$ 75)
 and B **and** f $+$
 assumes *basis*: *source.module-basis* B
 and *f-closed*: $v \in B \implies f\ v \in M'$
begin

The basis lies in the source module, and is independent and spanning.

lemma *B-subset*: $B \subseteq M$
 using *basis* *source.module-basis-spanning* **unfolding** *source.spanning-def* **by** *blast*

lemma *B-lin-indep*: *source.lin-indep* B
 by (*rule* *source.module-basis-lin-indep* [*OF* *basis*])

91.1 The twisted sum

The image of a linear combination must be $\bigoplus_{v \in A} c\ v \odot_2 f\ v$ — a sum in the *target* indexed by elements of the source. We give this “twisted” sum a name and prove for it the three facts the homomorphism obligations need: it is closed,

it is unchanged by padding the index set with zero coefficients, and it is additive and homogeneous in the coefficients.

definition $tsum :: ('b \Rightarrow 'a) \Rightarrow 'b \text{ set} \Rightarrow 'c$
where $tsum\ c\ A = target.madd.fincomp\ (\lambda v. c\ v \odot_2 f\ v)\ A$

lemma $tsum-Pi$:
assumes $A: A \subseteq B$ **and** $c: source.coeffs-on\ c\ A$
shows $(\lambda v. c\ v \odot_2 f\ v) \in A \rightarrow M'$
using $A\ c\ f\text{-closed}$ **by** $(auto\ intro!:\ target.scale-closed\ intro:\ source.coeffs-onD)$

lemma $tsum-closed\ [intro,\ simp]$:
assumes $A: A \subseteq B$ **and** $c: source.coeffs-on\ c\ A$
shows $tsum\ c\ A \in M'$
unfolding $tsum-def$ **using** $tsum-Pi[OF\ A\ c]$ **by** $(rule\ target.madd.fincomp-closed)$

lemma $tsum-empty\ [simp]$: $tsum\ c\ \{\} = \mathbf{0}_2$
by $(simp\ add:\ tsum-def)$

Only the values on the index set matter.

lemma $tsum-cong$:
assumes $A: A \subseteq B$ **and** $eq: \bigwedge v. v \in A \implies c\ v = d\ v$ **and** $d: source.coeffs-on\ d\ A$
shows $tsum\ c\ A = tsum\ d\ A$
unfolding $tsum-def$
proof $(rule\ target.madd.fincomp-cong')$
show $A = A ..$
show $(\lambda v. d\ v \odot_2 f\ v) \in A \rightarrow M'$ **by** $(rule\ tsum-Pi[OF\ A\ d])$
show $\bigwedge v. v \in A \implies c\ v \odot_2 f\ v = d\ v \odot_2 f\ v$ **using** eq **by** $simp$
qed

Padding the index set with zero coefficients does not change the sum: the extra terms are $\mathbf{0} \odot_2 f\ v = \mathbf{0}_2$.

lemma $tsum-mono-zero$:
assumes $finA'$: $finite\ A'$ **and** AA' : $A \subseteq A'$ **and** $A'B$: $A' \subseteq B$
and $zero$: $\bigwedge v. v \in A' \setminus A \implies c\ v = \mathbf{0}$ **and** $c: source.coeffs-on\ c\ A'$
shows $tsum\ c\ A = tsum\ c\ A'$
unfolding $tsum-def$
proof $(rule\ target.madd.fincomp-mono-neutral-cong-left)$
show $finite\ A'$ **and** $A \subseteq A'$ **using** $finA'\ AA'$ **by** $simp-all$
show $\bigwedge v. v \in A' \setminus A \implies c\ v \odot_2 f\ v = \mathbf{0}_2$
using $zero\ A'B\ f\text{-closed}$ **by** $(auto\ simp:\ target.scale-zero-scalar)$
show $\bigwedge v. v \in A \implies c\ v \odot_2 f\ v = c\ v \odot_2 f\ v$ **by** $simp$
show $(\lambda v. c\ v \odot_2 f\ v) \in A' \rightarrow M'$ **by** $(rule\ tsum-Pi[OF\ A'B\ c])$
qed

Additivity in the coefficients.

lemma $tsum-add-coeffs$:
assumes $A: A \subseteq B$ **and** $c: source.coeffs-on\ c\ A$ **and** $d: source.coeffs-on\ d\ A$
shows $tsum\ (\lambda v. c\ v + d\ v)\ A = tsum\ c\ A \oplus_2 tsum\ d\ A$


```

unfolding tsum-def
proof (rule trans)
  show target.madd.fincomp ( $\lambda v. (c \ v + d \ v) \odot_2 f \ v$ ) A
    = target.madd.fincomp ( $\lambda v. (c \ v \odot_2 f \ v) \oplus_2 (d \ v \odot_2 f \ v)$ ) A
  proof (rule target.madd.fincomp-cong')
    show A = A ..
    show ( $\lambda v. (c \ v \odot_2 f \ v) \oplus_2 (d \ v \odot_2 f \ v)$ )  $\in A \rightarrow M'$ 
      using tsum-Pi[OF A c] tsum-Pi[OF A d] by auto
    fix v assume v:  $v \in A$ 
    then have c v  $\in R$  and d v  $\in R$  and f v  $\in M'$ 
      using source.coeffs-onD[OF c] source.coeffs-onD[OF d] A f-closed by auto
    then show  $(c \ v + d \ v) \odot_2 f \ v = (c \ v \odot_2 f \ v) \oplus_2 (d \ v \odot_2 f \ v)$ 
      by (rule target.scale-distrib-add)
    qed
  show target.madd.fincomp ( $\lambda v. (c \ v \odot_2 f \ v) \oplus_2 (d \ v \odot_2 f \ v)$ ) A
    = target.madd.fincomp ( $\lambda v. c \ v \odot_2 f \ v$ ) A  $\oplus_2$  target.madd.fincomp ( $\lambda v. d \ v \odot_2 f \ v$ ) A
    using tsum-Pi[OF A c] tsum-Pi[OF A d] by (rule target.madd.fincomp-comp)
  qed

```

Homogeneity in the coefficients.

```

lemma tsum-scale-coeffs:
  assumes A:  $A \subseteq B$  and a:  $a \in R$  and c: source.coeffs-on c A
  shows tsum ( $\lambda v. a \cdot c \ v$ ) A =  $a \odot_2$  tsum c A
  unfolding tsum-def
proof (rule trans)
  show target.madd.fincomp ( $\lambda v. (a \cdot c \ v) \odot_2 f \ v$ ) A
    = target.madd.fincomp ( $\lambda v. a \odot_2 (c \ v \odot_2 f \ v)$ ) A
  proof (rule target.madd.fincomp-cong')
    show A = A ..
    show ( $\lambda v. a \odot_2 (c \ v \odot_2 f \ v)$ )  $\in A \rightarrow M'$ 
      using a tsum-Pi[OF A c] by (auto intro: target.scale-closed)
    fix v assume v:  $v \in A$ 
    then have cv:  $c \ v \in R$  and fv:  $f \ v \in M'$ 
      using source.coeffs-onD[OF c] A f-closed by auto
    show  $(a \cdot c \ v) \odot_2 f \ v = a \odot_2 (c \ v \odot_2 f \ v)$ 
      by (rule target.scale-scale[OF a cv fv])
    qed
  show target.madd.fincomp ( $\lambda v. a \odot_2 (c \ v \odot_2 f \ v)$ ) A
    =  $a \odot_2$  target.madd.fincomp ( $\lambda v. c \ v \odot_2 f \ v$ ) A
    using a tsum-Pi[OF A c] by (rule target.scale-fincomp[symmetric])
  qed

```

91.2 The extension is well defined

Two representations of the same element give the same twisted sum. This is where *lin-indep-lincomb-unique-gen* is used: the coefficient functions, extended by **0**, agree, and each twisted sum equals the one over the union of the two index sets.

lemma *tsum-well-defined*:

assumes A : *finite* $A \subseteq B$ **and** A' : *finite* $A' \subseteq B$
and c : *source.coeffs-on* $c \ A$ **and** d : *source.coeffs-on* $d \ A'$
and eq : *source.lincomb* $c \ A = \text{source.lincomb } d \ A'$
shows $tsum \ c \ A = tsum \ d \ A'$

proof –

define c' **where** $c' = (\lambda u. \text{if } u \in A \text{ then } c \ u \text{ else } \mathbf{0})$
define d' **where** $d' = (\lambda u. \text{if } u \in A' \text{ then } d \ u \text{ else } \mathbf{0})$
have $finU$: *finite* $(A \cup A')$ **using** $A(1) \ A'(1)$ **by** *simp*
have UB : $A \cup A' \subseteq B$ **using** $A(2) \ A'(2)$ **by** *blast*
have $c'R$: *source.coeffs-on* $c' \ (A \cup A')$
by (*rule* *source.coeffs-onI*) (*use* *source.coeffs-onD*[*OF* c] **in** $\langle \text{simp add: } c'\text{-def} \rangle$)
have $d'R$: *source.coeffs-on* $d' \ (A \cup A')$
by (*rule* *source.coeffs-onI*) (*use* *source.coeffs-onD*[*OF* d] **in** $\langle \text{simp add: } d'\text{-def} \rangle$)
– The extended coefficient functions agree, by uniqueness of coordinates.
have $agree$: $\bigwedge v. c' \ v = d' \ v$
unfolding $c'\text{-def } d'\text{-def}$
by (*rule* *source.lin-indep-lincomb-unique-gen*[*OF* $B\text{-lin-indep } A \ A' \ c \ d \ eq$])
– Each twisted sum is unchanged by padding to the union.
have $tsum \ c \ A = tsum \ c' \ A$
by (*rule* *tsum-cong*[*OF* $A(2)$]) (*use* c **in** $\langle \text{auto simp: } c'\text{-def } \text{source.coeffs-on-def} \rangle$)
also have $\dots = tsum \ c' \ (A \cup A')$
proof (*rule* *tsum-mono-zero*[*OF* $finU \ Un\text{-upper1 } UB$])
show $\bigwedge v. v \in A \cup A' \setminus A \implies c' \ v = \mathbf{0}$ **unfolding** $c'\text{-def}$ **by** *simp*
show *source.coeffs-on* $c' \ (A \cup A')$ **by** (*rule* $c'R$)
qed
also have $\dots = tsum \ d' \ (A \cup A')$
by (*rule* *tsum-cong*[*OF* UB]) (*use* $agree \ d'R$ **in** *auto*)
also have $\dots = tsum \ d' \ A'$
proof (*rule* *sym*, *rule* *tsum-mono-zero*[*OF* $finU \ Un\text{-upper2 } UB$])
show $\bigwedge v. v \in A \cup A' \setminus A' \implies d' \ v = \mathbf{0}$ **unfolding** $d'\text{-def}$ **by** *simp*
show *source.coeffs-on* $d' \ (A \cup A')$ **by** (*rule* $d'R$)
qed
also have $\dots = tsum \ d \ A'$
by (*rule* *tsum-cong*[*OF* $A'(2)$]) (*use* d **in** $\langle \text{auto simp: } d'\text{-def } \text{source.coeffs-on-def} \rangle$)
finally show *?thesis* .
qed

91.3 The extension

The extension of f to all of M : read off any representation over the basis and take the twisted sum. Made extensional on M so that it satisfies the *map* obligation of *module-homomorphism* directly.

definition *free-ext* :: $'b \Rightarrow 'c$
where *free-ext* =
 $(\lambda x \in M. \text{THE } y. \exists A \ c. \text{finite } A \wedge A \subseteq B \wedge \text{source.coeffs-on } c \ A$
 $\wedge x = \text{source.lincomb } c \ A \wedge y = tsum \ c \ A)$

On a representation, the extension is the twisted sum.

lemma *free-ext-repr*:

assumes A : *finite* A $A \subseteq B$ **and** c : *source.coeffs-on* c A **and** x : $x = \text{source.lincomb } c \ A$
 $c \ A$
shows *free-ext* $x = \text{tsum } c \ A$
proof –
have xM : $x \in M$
using $x \ A \ c$ *source.lincomb-closed*[*of* $A \ c$] *B-subset* *source.coeffs-onD* **by** *blast*
have (*THE* y . $\exists A' \ c'. \text{finite } A' \wedge A' \subseteq B \wedge \text{source.coeffs-on } c' \ A'$
 $\wedge x = \text{source.lincomb } c' \ A' \wedge y = \text{tsum } c' \ A') = \text{tsum } c \ A$
proof (*rule the-equality*)
show $\exists A' \ c'. \text{finite } A' \wedge A' \subseteq B \wedge \text{source.coeffs-on } c' \ A'$
 $\wedge x = \text{source.lincomb } c' \ A' \wedge \text{tsum } c \ A = \text{tsum } c' \ A'$
using $A \ c \ x$ **by** *blast*
next
fix y **assume** $\exists A' \ c'. \text{finite } A' \wedge A' \subseteq B \wedge \text{source.coeffs-on } c' \ A'$
 $\wedge x = \text{source.lincomb } c' \ A' \wedge y = \text{tsum } c' \ A'$
then obtain $A' \ c'$ **where** y : *finite* $A' \ A' \subseteq B$ *source.coeffs-on* $c' \ A'$
 $x = \text{source.lincomb } c' \ A' \ y = \text{tsum } c' \ A'$ **by** *blast*
have *source.lincomb* $c \ A = \text{source.lincomb } c' \ A'$ **using** $x \ y(4)$ **by** *simp*
then have $\text{tsum } c \ A = \text{tsum } c' \ A'$
by (*rule tsum-well-defined*[*OF* $A \ y(1,2) \ c \ y(3)$])
then show $y = \text{tsum } c \ A$ **using** $y(5)$ **by** *simp*
qed
then show *?thesis* **unfolding** *free-ext-def* **using** xM **by** *simp*
qed

The extension agrees with f on the basis: a basis element is its own singleton combination with coefficient **1**.

theorem *free-ext-extends*:

assumes v : $v \in B$ **shows** *free-ext* $v = f \ v$
proof –
have vM : $v \in M$ **using** $v \ B$ -*subset* **by** *blast*
have $c1$: *source.coeffs-on* $(\lambda-. \mathbf{1}) \ \{v\}$ **by** (*rule* *source.coeffs-onI*) *simp*
have *source.lincomb* $(\lambda-. \mathbf{1}) \ \{v\} = v$
using vM **by** (*simp add*: *source.scale-one*)
then have *free-ext* $v = \text{tsum } (\lambda-. \mathbf{1}) \ \{v\}$
using $v \ c1$ **by** (*auto intro*: *free-ext-repr*[**where** $A = \{v\}$])
also have $\dots = \mathbf{1} \odot_2 f \ v$
unfolding *tsum-def* **using** *f-closed*[*OF* v]
by (*simp add*: *target.scale-one*)
also have $\dots = f \ v$ **using** *f-closed*[*OF* v] **by** (*rule* *target.scale-one*)
finally show *?thesis* .
qed

Existence. The extension is a module homomorphism.

theorem *free-ext-hom*:

module-homomorphism $R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \ M \ (\oplus) \ \mathbf{0}_M \ (\odot) \ M' \ (\oplus_2) \ \mathbf{0}_2 \ (\odot_2)$ *free-ext*
proof (*intro module-homomorphism.intro map.intro module-homomorphism-axioms.intro*)
show *Module* $R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1} \ (\oplus) \ \mathbf{0}_M \ M \ (\odot)$ **by** (*rule* *source.Module-axioms*)

```

show Module  $R$   $(+)$   $(\cdot)$   $\mathbf{0}$   $\mathbf{1}$   $(\oplus_2)$   $\mathbf{0}_2$   $M'$   $(\odot_2)$  by (rule target.Module-axioms)
show free-ext  $\in M \rightarrow_E M'$ 
proof (rule PiE-I)
  fix  $x$  assume  $x: x \in M$ 
  obtain  $A$   $c$  where  $A: \text{finite } A \subseteq B$  source.coeffs-on  $c$   $A$   $x = \text{source.lincomb } c$   $A$ 
    by (rule source.module-basis-repr-exists[OF basis x])
  then show free-ext  $x \in M'$  using free-ext-repr[OF A] by simp
qed (simp add: free-ext-def)
next
  fix  $x$   $y$  assume  $x: x \in M$  and  $y: y \in M$ 
  obtain  $A$   $c$  where  $A: \text{finite } A \subseteq B$  source.coeffs-on  $c$   $A$   $x = \text{source.lincomb } c$   $A$ 
    by (rule source.module-basis-repr-exists[OF basis x])
  obtain  $A'$   $d$  where  $A': \text{finite } A' \subseteq B$  source.coeffs-on  $d$   $A'$   $y = \text{source.lincomb } d$   $A'$ 
    by (rule source.module-basis-repr-exists[OF basis y])
  — Pad both representations to the union, add coefficients there, and use additivity
  of the twisted sum.
  define  $c'$  where  $c' = (\lambda u. \text{if } u \in A \text{ then } c \ u \text{ else } \mathbf{0})$ 
  define  $d'$  where  $d' = (\lambda u. \text{if } u \in A' \text{ then } d \ u \text{ else } \mathbf{0})$ 
  have finU: finite  $(A \cup A')$  using  $A(1)$   $A'(1)$  by simp
  have  $UB: A \cup A' \subseteq B$  using  $A(2)$   $A'(2)$  by blast
  have  $UM: A \cup A' \subseteq M$  using  $UB$  B-subset by blast
  have  $c'R: \text{source.coeffs-on } c' (A \cup A')$ 
    by (rule source.coeffs-onI) (use source.coeffs-onD[OF A(3)] in <simp add: c'-def>)
  have  $d'R: \text{source.coeffs-on } d' (A \cup A')$ 
    by (rule source.coeffs-onI) (use source.coeffs-onD[OF A'(3)] in <simp add: d'-def>)
  — Source side: the padded combinations are unchanged, so their sum represents  $x$ 
   $\oplus y$ .
  have  $cU: \text{source.lincomb } c' (A \cup A') = x$ 
proof —
    have  $\text{source.lincomb } c' A = \text{source.lincomb } c' (A \cup A')$ 
      by (rule source.lincomb-mono-zero[OF finU Un-upper1 UM])
      (use c'R in <simp-all add: c'-def>)
    moreover have  $\text{source.lincomb } c' A = \text{source.lincomb } c$   $A$ 
      by (rule source.lincomb-cong)
      (use A(2) A(3) B-subset in <auto simp: c'-def>)
    ultimately show ?thesis using  $A(4)$  by simp
  qed
  have  $dU: \text{source.lincomb } d' (A \cup A') = y$ 
proof —
    have  $\text{source.lincomb } d' A' = \text{source.lincomb } d' (A \cup A')$ 
      by (rule source.lincomb-mono-zero[OF finU Un-upper2 UM])
      (use d'R in <simp-all add: d'-def>)
    moreover have  $\text{source.lincomb } d' A' = \text{source.lincomb } d$   $A'$ 
      by (rule source.lincomb-cong)
      (use A'(2) A'(3) B-subset in <auto simp: d'-def>)
    ultimately show ?thesis using  $A'(4)$  by simp
  qed
  have sumR: source.coeffs-on  $(\lambda u. c' \ u + d' \ u) (A \cup A')$ 
    by (rule source.coeffs-onI)

```

```

      (use source.coeffs-onD[OF c'R] source.coeffs-onD[OF d'R] in simp)
    have  $x \oplus y = \text{source.lincomb } (\lambda u. c' u + d' u) (A \cup A')$ 
      using source.lincomb-add-coeffs[OF UM c'R d'R] cU dU by simp
    then have  $\text{free-ext } (x \oplus y) = \text{tsum } (\lambda u. c' u + d' u) (A \cup A')$ 
      by (rule free-ext-repr[OF finU UB sumR])
    also have  $\dots = \text{tsum } c' (A \cup A') \oplus_2 \text{tsum } d' (A \cup A')$ 
      by (rule tsum-add-coeffs[OF UB c'R d'R])
    also have  $\text{tsum } c' (A \cup A') = \text{free-ext } x$ 
      using free-ext-repr[OF finU UB c'R cU[symmetric]] by simp
    also have  $\text{tsum } d' (A \cup A') = \text{free-ext } y$ 
      using free-ext-repr[OF finU UB d'R dU[symmetric]] by simp
    finally show  $\text{free-ext } (x \oplus y) = \text{free-ext } x \oplus_2 \text{free-ext } y$  .
  next
  fix a x assume a:  $a \in R$  and x:  $x \in M$ 
  obtain A c where A:  $\text{finite } A \ A \subseteq B$  source.coeffs-on c A  $x = \text{source.lincomb } c \ A$ 
    by (rule source.module-basis-repr-exists[OF basis x])
  have AM:  $A \subseteq M$  using A(2) B-subset by blast
  have scR: source.coeffs-on  $(\lambda u. a \cdot c \ u) \ A$ 
    by (rule source.coeffs-onI) (use a source.coeffs-onD[OF A(3)] in simp)
  — Scaling a combination scales its coefficients (the computation inside span-scale-closed,
  reused here).
  have  $a \odot x = \text{source.lincomb } (\lambda u. a \cdot c \ u) \ A$ 
  proof —
    have pi:  $(\lambda v. c \ v \odot v) \in A \rightarrow M$ 
      using AM source.coeffs-onD[OF A(3)] by (auto intro!: source.scale-closed)
    have  $a \odot \text{source.lincomb } c \ A$ 
      = source.madd.fincomp  $(\lambda v. a \odot (c \ v \odot v)) \ A$ 
      unfolding source.lincomb-def using a pi by (rule source.scale-fincomp)
    also have  $\dots = \text{source.madd.fincomp } (\lambda v. (a \cdot c \ v) \odot v) \ A$ 
    proof (rule source.madd.fincomp-cong')
      show  $A = A$  ..
      show  $(\lambda v. (a \cdot c \ v) \odot v) \in A \rightarrow M$ 
        using AM a source.coeffs-onD[OF A(3)] by (auto intro!: source.scale-closed)
      fix v assume v:  $v \in A$ 
      then have vM:  $v \in M$  and cv:  $c \ v \in R$ 
        using AM source.coeffs-onD[OF A(3)] by auto
      show  $a \odot (c \ v \odot v) = (a \cdot c \ v) \odot v$  by (rule source.scale-scale[OF a cv vM,
      symmetric])
    qed
  qed
  finally show ?thesis
    by (simp add: A(4) source.lincomb-def)
  qed
  then have  $\text{free-ext } (a \odot x) = \text{tsum } (\lambda u. a \cdot c \ u) \ A$ 
    by (rule free-ext-repr[OF A(1,2) scR])
  also have  $\dots = a \odot_2 \text{tsum } c \ A$  by (rule tsum-scale-coeffs[OF A(2) a A(3)])
  also have  $\text{tsum } c \ A = \text{free-ext } x$  using free-ext-repr[OF A] by simp
  finally show  $\text{free-ext } (a \odot x) = a \odot_2 \text{free-ext } x$  .
  qed

```

Uniqueness. Any homomorphism agreeing with f on the basis equals the exten-

sion: two homomorphisms agreeing on a spanning set agree everywhere, because each element is a finite combination of basis elements and both maps commute with the combination.

theorem *free-ext-unique*:

assumes η : module-homomorphism $R (+) (\cdot) \mathbf{0} \mathbf{1} M (\oplus) \mathbf{0}_M (\odot) M' (\oplus_2) \mathbf{0}_2 (\odot_2)$
 η
and *agree*: $\bigwedge v. v \in B \implies \eta v = f v$
and $x: x \in M$
shows $\eta x = \text{free-ext } x$
proof –
interpret H : module-homomorphism $R (+) (\cdot) \mathbf{0} \mathbf{1} M (\oplus) \mathbf{0}_M (\odot)$
 $M' (\oplus_2) \mathbf{0}_2 (\odot_2) \eta$
by (*rule* η)
obtain $A \ c$ **where** A : finite $A \ A \subseteq B$ *source.coeffs-on* $c \ A \ x = \text{source.lincomb } c \ A$
by (*rule* *source.module-basis-repr-exists*[*OF* *basis* x])
have AM : $A \subseteq M$ **using** $A(2)$ *B-subset* **by** *blast*
– A homomorphism carries a linear combination to the twisted sum of the images.
have $\eta (\text{source.lincomb } c \ A) = \text{target.madd.fincomp } (\lambda v. c \ v \odot_2 \eta \ v) \ A$
by (*rule* $H.\text{hom-lincomb}$ [*OF* $A(1) \ AM$]) (*use* $A(3)$ **in** $\langle \text{auto intro: source.coeffs-onD} \rangle$)
– On the basis the images agree with f , so that sum is exactly the twisted sum.
also have $\text{target.madd.fincomp } (\lambda v. c \ v \odot_2 \eta \ v) \ A = \text{tsum } c \ A$
unfolding *tsum-def*
proof (*rule* *target.madd.fincomp-cong*)
show $A = A ..$
show $(\lambda v. c \ v \odot_2 f \ v) \in A \rightarrow M'$ **by** (*rule* *tsum-Pi*[*OF* $A(2) \ A(3)$])
show $\bigwedge v. v \in A \implies c \ v \odot_2 \eta \ v = c \ v \odot_2 f \ v$ **using** *agree* $A(2)$ **by** *auto*
qed
finally show *?thesis* **using** $A(4)$ *free-ext-repr*[*OF* A] **by** *simp*
qed

The universal property. There is exactly one homomorphism extending f .

theorem *free-module-universal*:

$\exists ! \eta. \text{ module-homomorphism } R (+) (\cdot) \mathbf{0} \mathbf{1} M (\oplus) \mathbf{0}_M (\odot) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \eta$
 $\wedge (\forall v \in B. \eta v = f v) \wedge \eta \in M \rightarrow_E M'$
proof
show module-homomorphism $R (+) (\cdot) \mathbf{0} \mathbf{1} M (\oplus) \mathbf{0}_M (\odot) M' (\oplus_2) \mathbf{0}_2 (\odot_2)$ *free-ext*
 $\wedge (\forall v \in B. \text{free-ext } v = f v) \wedge \text{free-ext} \in M \rightarrow_E M'$
proof (*intro conjI ballI free-ext-hom*)
show $\bigwedge v. v \in B \implies \text{free-ext } v = f v$ **by** (*rule* *free-ext-extends*)
show $\text{free-ext} \in M \rightarrow_E M'$
using module-homomorphism. *axioms*(3)[*OF* *free-ext-hom*] *map.graph* **by** *blast*
qed
next
fix η
assume module-homomorphism $R (+) (\cdot) \mathbf{0} \mathbf{1} M (\oplus) \mathbf{0}_M (\odot) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \eta$
 $\wedge (\forall v \in B. \eta v = f v) \wedge \eta \in M \rightarrow_E M'$
then have H : module-homomorphism $R (+) (\cdot) \mathbf{0} \mathbf{1} M (\oplus) \mathbf{0}_M (\odot) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \eta$
and *ag*: $\bigwedge v. v \in B \implies \eta v = f v$ **and** *ext*: $\eta \in M \rightarrow_E M'$ **by** *auto*

```

show  $\eta = \text{free-ext}$ 
proof (rule extensionalityI)
  show  $\eta \in \text{extensional } M$  using ext by (simp add: PiE-iff)
  show  $\text{free-ext} \in \text{extensional } M$  unfolding free-ext-def by simp
  fix  $x$  assume  $x \in M$ 
  then show  $\eta x = \text{free-ext } x$ 
    using H ag free-ext-unique by blast
qed
qed

end

end

```

92 The rank–nullity theorem

```

theory Rank-Nullity
imports Steinitz Module-Homomorphism
begin

```

As in the module-homomorphism theory, suppress HOL arithmetic syntax while binding the scalar-ring operations in the locale header.

```

no-notation plus (infixl  $\langle + \rangle$  65)
no-notation minus (infixl  $\langle - \rangle$  65)
unbundle no uminus-syntax

```

A linear map is a module homomorphism between vector spaces over the same scalar field.

```

locale linear-map =
  source: Vector-Space  $R (+) (\cdot) \mathbf{0} \mathbf{1} (\oplus) \mathbf{0}_M M (\odot) +$ 
  target: Vector-Space  $R (+) (\cdot) \mathbf{0} \mathbf{1} (\oplus_2) \mathbf{0}_2 M' (\odot_2) +$ 
  hom: module-homomorphism
     $R (+) (\cdot) \mathbf{0} \mathbf{1}$ 
     $M (\oplus) \mathbf{0}_M (\odot)$ 
     $M' (\oplus_2) \mathbf{0}_2 (\odot_2) \eta$ 
  for  $R$  and addition (infixl  $\langle + \rangle$  65) and multiplication (infixl  $\langle \cdot \rangle$  70)
    and zero ( $\langle \mathbf{0} \rangle$ ) and unit ( $\langle \mathbf{1} \rangle$ )
    and  $M$  and madd (infixl  $\langle \oplus \rangle$  65) and mzero ( $\langle \mathbf{0}_M \rangle$ )
    and scale (infixr  $\langle \odot \rangle$  75)
    and  $M'$  and madd' (infixl  $\langle \oplus_2 \rangle$  65) and mzero' ( $\langle \mathbf{0}_2 \rangle$ )
    and scale' (infixr  $\langle \odot_2 \rangle$  75) and  $\eta$ 
begin

```

The kernel and image inherit vector-space structures from the source and target.

```

sublocale kernel: vector-subspace
   $R (+) (\cdot) \mathbf{0} \mathbf{1} (\oplus) \mathbf{0}_M M (\odot) \text{hom.Ker}$ 
proof
  show source.mod.submodule hom.Ker by (rule hom.Ker-submodule)

```

qed

sublocale *image*: *vector-subspace*

$R (+) (\cdot) \mathbf{0} \mathbf{1} (\oplus_2) \mathbf{0}_2 M' (\odot_2) \eta \text{ ' } M$

proof

show *target.mod.submodule* ($\eta \text{ ' } M$) **by** (*rule hom.image-submodule*)

qed

Reindex a target linear combination over the images of a finite set. Injectivity is exactly what makes the transported coefficient function unambiguous.

lemma *image-lincomb-reindex*:

assumes *finD*: *finite D* **and** *DM*: $D \subseteq M$ **and** *inj*: *inj-on* $\eta \ D$

and *c*: $\bigwedge v. v \in D \implies c \ v \in R$

defines $d \equiv \lambda y. c \ (inv\text{-}into \ D \ \eta \ y)$

shows *image.sub.lincomb* $d \ (\eta \text{ ' } D) =$

target.vadd.fincomp $(\lambda v. c \ v \ \odot_2 \ \eta \ v) \ D$

proof –

have *imageD*: $\eta \text{ ' } D \subseteq \eta \text{ ' } M$ **using** *DM* **by** *blast*

have *dR*: $\bigwedge y. y \in \eta \text{ ' } D \implies d \ y \in R$

using *inj c* **by** (*auto simp: d-def inv-into-f-f*)

have *terms*: $(\lambda y. d \ y \ \odot_2 \ y) \in \eta \text{ ' } D \rightarrow M'$

using *imageD dR* **by** (*auto intro!: target.scale-closed*)

have *image.sub.lincomb* $d \ (\eta \text{ ' } D) = \text{target.lincomb } d \ (\eta \text{ ' } D)$

by (*rule image.sub.lincomb-eq*) (*use finD imageD dR in auto*)

also **have** $\dots = \text{target.vadd.fincomp } (\lambda y. d \ y \ \odot_2 \ y) \ (\eta \text{ ' } D)$

by (*simp add: target.lincomb-def*)

also **have** $\dots = \text{target.vadd.fincomp } (\lambda v. d \ (\eta \ v) \ \odot_2 \ \eta \ v) \ D$

by (*rule target.vadd.fincomp-reindex[OF terms inj]*)

also **have** $\dots = \text{target.vadd.fincomp } (\lambda v. c \ v \ \odot_2 \ \eta \ v) \ D$

by (*rule target.vadd.fincomp-cong'*)

(*use DM inj c in 'auto simp: d-def inv-into-f-f intro!: target.scale-closed*

hom.hom-closed)

finally **show** *?thesis* .

qed

Once a kernel basis has been extended to a source basis, the map is injective on the complementary basis vectors. Equality of two images puts their difference in the kernel; the kernel basis would then express either vector in the span of the remaining source basis.

lemma *injective-on-basis-complement*:

assumes *A*: *kernel.sub.basis A* **and** *C*: *source.basis C* **and** *AC*: $A \subseteq C$

shows *inj-on* $\eta \ (C \setminus A)$

proof (*rule inj-onI*)

fix $x \ y$

assume $x: x \in C \setminus A$ **and** $y: y \in C \setminus A$ **and** *eq*: $\eta \ x = \eta \ y$

have *CM*: $C \subseteq M$ **using** *C* **by** (*simp add: source.basis-def*)

have *xM*: $x \in M$ **and** *yM*: $y \in M$ **using** $x \ y \ CM$ **by** *auto*

define z **where** $z = x \oplus \text{source.vadd.inverse } y$

have *zM*: $z \in M$ **using** *xM yM* **by** (*simp add: z-def*)


```

have  $\eta z = \eta x \oplus_2 \eta$  (source.vadd.inverse y)
  using xM yM by (simp add: z-def hom.hom-add)
also have  $\dots = \eta x \oplus_2$  target.vadd.inverse ( $\eta y$ )
  using yM by (simp add: hom.hom-neg)
also have  $\dots = 0_2$  using eq xM yM by (simp add: hom.hom-closed)
finally have zKer:  $z \in \text{hom.Ker}$  using zM by (rule hom.Ker-memI)
have AKer:  $A \subseteq \text{hom.Ker}$  using A by (simp add: kernel.sub.basis-def)
have z-sub-span:  $z \in \text{kernel.sub.span } A$ 
  using kernel.sub.basis-spanning[OF A] zKer by (rule kernel.sub.spanning-span-all)
have z-span:  $z \in \text{source.span } A$ 
  using z-sub-span kernel.sub-span-eq[OF AKer] by simp
show  $x = y$ 
proof (rule ccontr)
  assume xy:  $x \neq y$ 
  have AM:  $A \subseteq M$  using AKer hom.Ker-subset by blast
  have insM:  $\text{insert } y \ A \subseteq M$  using yM AM by simp
  have z-ins:  $z \in \text{source.span } (\text{insert } y \ A)$ 
    using z-span source.span-mono[of A insert y A] by blast
  have y-ins:  $y \in \text{source.span } (\text{insert } y \ A)$  by (rule source.span-incl[OF insM]) simp
  have  $z \oplus y \in \text{source.span } (\text{insert } y \ A)$ 
    by (rule source.span-vadd[OF insM z-ins y-ins])
  moreover have  $z \oplus y = x$  using xM yM by (simp add: z-def source.vadd.associative)
  ultimately have  $x \in \text{source.span } (\text{insert } y \ A)$  by simp
  moreover have  $\text{insert } y \ A \subseteq C \setminus \{x\}$  using AC x y xy by auto
  ultimately have  $x \in \text{source.span } (C \setminus \{x\})$  using source.span-mono by blast
  moreover have source.lin-indep C using C by (rule source.basis-lin-indep)
  ultimately show False using x by (simp add: source.lin-indep-not-in-span)
qed
qed

```

Split a linear combination over a basis into a chosen subset and its complement.

lemma source-lincomb-diff:

```

assumes finC: finite C and CM:  $C \subseteq M$  and AC:  $A \subseteq C$ 
  and c:  $\bigwedge v. v \in C \implies c \ v \in R$ 
shows source.lincomb c C =
  source.lincomb c A  $\oplus$  source.lincomb c (C \ A)

```

proof –

```

have finA: finite A and finD: finite (C \ A) using finC AC by (auto intro: fi-
nite-subset)
have C-eq:  $C = A \cup (C \setminus A)$  using AC by auto
have A-terms:  $(\lambda v. c \ v \odot v) \in A \rightarrow M$ 
  using AC CM c by (auto intro!: source.scale-closed)
have D-terms:  $(\lambda v. c \ v \odot v) \in C \setminus A \rightarrow M$ 
  using CM c by (auto intro!: source.scale-closed)
have source.lincomb c C = source.lincomb c (A  $\cup$  (C \ A))
  by (rule arg-cong[OF C-eq])
also have  $\dots = \text{source.lincomb } c \ A \oplus \text{source.lincomb } c \ (C \setminus A)$ 
  unfolding source.lincomb-def
  by (rule source.vadd.fincomp-Un-disjoint[OF finA finD])
  (use A-terms D-terms in auto)

```

finally show ?thesis .
qed

The images of the complementary basis vectors form a basis of the image.

lemma *image-basis-complement*:

assumes A : *kernel.sub.basis* A **and** C : *source.basis* C **and** AC : $A \subseteq C$

defines $D \equiv C \setminus A$

shows *image.sub.basis* $(\eta \restriction D)$

proof –

have $\text{fin}C$: *finite* C **and** CM : $C \subseteq M$ **using** C **by** (*auto simp: source.basis-def*)

have $\text{fin}D$: *finite* D **and** DM : $D \subseteq M$ **using** $\text{fin}C$ CM **by** (*auto simp: D-def*)

have $\text{fin}A$: *finite* A **by** (*rule finite-subset[OF AC finC]*)

have DC : $D \subseteq C$ **by** (*simp add: D-def*)

have $AKer$: $A \subseteq \text{hom.Ker}$ **using** A **by** (*simp add: kernel.sub.basis-def*)

have AM : $A \subseteq M$ **using** $AKer$ *hom.Ker-subset* **by** *blast*

have $\text{inj}D$: *inj-on* $\eta \restriction D$

using *injective-on-basis-complement[OF A C AC]* **by** (*simp add: D-def*)

have $\text{image}DM$: $\eta \restriction D \subseteq \eta \restriction M$ **using** DM **by** *blast*

have $\text{fin-image}D$: *finite* $(\eta \restriction D)$ **using** $\text{fin}D$ **by** *simp*

show ?thesis

proof (*rule image.sub.basisI*)

show *image.sub.spanning* $(\eta \restriction D)$

unfolding *image.sub.spanning-iff-mod-spanning[OF fin-imageD imageDM]*

proof (*rule image.sub.mod.spanningI[OF imageDM]*)

fix z **assume** $z \in \eta \restriction M$

then obtain x **where** xM : $x \in M$ **and** z : $z = \eta x$ **by** *blast*

obtain c **where** c : $c \in C \rightarrow_E R$ $x = \text{source.lincomb } c \ C$

using *source.basis-spanning[OF C]* xM **unfolding** *source.spanning-def* **by** *blast*

have cR : $\bigwedge v. v \in C \implies c \ v \in R$ **using** $c(1)$ **by** *auto*

have cA : $\bigwedge v. v \in A \implies c \ v \in R$ **using** AC cR **by** *blast*

have cD : $\bigwedge v. v \in D \implies c \ v \in R$ **using** cR **by** (*simp add: D-def*)

have split : $\text{source.lincomb } c \ C = \text{source.lincomb } c \ A \oplus \text{source.lincomb } c \ D$

using *source.lincomb-diff[OF finC CM AC cR]* **by** (*simp add: D-def*)

have $\text{sub}A\text{-closed}$: *kernel.sub.lincomb* $c \ A \in \text{hom.Ker}$

by (*rule kernel.sub.lincomb-closed[OF AKer cA]*)

have $\text{ambient}A$: $\text{source.lincomb } c \ A = \text{kernel.sub.lincomb } c \ A$

by (*rule kernel.sub.lincomb-eq[OF finA AKer cA, symmetric]*)

have $\text{lin}A\text{-Ker}$: $\text{source.lincomb } c \ A \in \text{hom.Ker}$ **using** $\text{sub}A\text{-closed}$ $\text{ambient}A$ **by**

simp

have $\text{lin}A\text{-}M$: $\text{source.lincomb } c \ A \in M$ **using** AM cA **by** (*rule source.lincomb-closed*)

have $\text{lin}D\text{-}M$: $\text{source.lincomb } c \ D \in M$ **using** DM cD **by** (*rule source.lincomb-closed*)

have map-split : $\eta \ x = \eta \ (\text{source.lincomb } c \ D)$

proof –

have $\eta \ x = \eta \ (\text{source.lincomb } c \ A \oplus \text{source.lincomb } c \ D)$ **using** $c(2)$ split **by**

simp

also have $\dots = \eta \ (\text{source.lincomb } c \ A) \oplus_2 \eta \ (\text{source.lincomb } c \ D)$

by (*rule hom.hom-add[OF linA-M linD-M]*)

also have $\eta \ (\text{source.lincomb } c \ A) = \mathbf{0}_2$ **using** $\text{lin}A\text{-Ker}$ **by** (*rule hom.Ker-image*)

also have $\mathbf{0}_2 \oplus_2 \eta \ (\text{source.lincomb } c \ D) = \eta \ (\text{source.lincomb } c \ D)$

using $\text{lin}D\text{-}M$ **by** (*simp add: hom.hom-closed*)

```

    finally show ?thesis .
qed
define d where d = (λy. c (inv-into D η y))
have dR:  $\bigwedge y. y \in \eta \text{ ' } D \implies d y \in R$ 
  using injD cD by (auto simp: d-def inv-into-f-f)
have image-sum: image.sub.lincomb d (η ' D) =
  target.vadd.fincomp (λv. c v ⊙2 η v) D
proof -
  have image.sub.lincomb (λy. c (inv-into D η y)) (η ' D) =
    target.vadd.fincomp (λv. c v ⊙2 η v) D
  by (rule image.lincomb-reindex[OF finD DM injD cD])
  then show ?thesis by (simp add: d-def)
qed
have map-sum: η (source.lincomb c D) =
  target.vadd.fincomp (λv. c v ⊙2 η v) D
  by (rule hom.hom.lincomb[OF finD DM cD])
have z-eq: z = image.sub.lincomb d (η ' D)
  using z map-split map-sum image-sum by simp
show z ∈ image.sub.span (η ' D)
proof (rule image.sub.mod.spanI[where c=d and A=η ' D])
  show finite (η ' D) by (rule fin-imageD)
  show η ' D ⊆ η ' D by simp
  show image.sub.mod.coeffs-on d (η ' D)
    using dR by (rule image.sub.mod.coeffs-onI)
  show z = image.sub.lincomb d (η ' D) by (rule z-eq)
qed
qed
next
show image.sub.lin-indep (η ' D)
  unfolding image.sub.lin-indep-def
proof (intro conjI ballI impI fin-imageD imageDM)
  fix d y
  assume d: d ∈ η ' D →E R
  and zero: image.sub.lincomb d (η ' D) = 02 and y: y ∈ η ' D
  define c where c = (λv. d (η v))
  have cR:  $\bigwedge v. v \in D \implies c v \in R$  using d by (auto simp: c-def)
  define e where e = (λz. c (inv-into D η z))
  have ed:  $\bigwedge z. z \in \eta \text{ ' } D \implies e z = d z$ 
    using injD by (auto simp: e-def c-def inv-into-f-f)
  have image-eq: image.sub.lincomb e (η ' D) = image.sub.lincomb d (η ' D)
  proof (rule image.sub.lincomb-cong)
    show η ' D ⊆ η ' M by (rule imageDM)
    show  $\bigwedge z. z \in \eta \text{ ' } D \implies d z \in R$  using d by auto
    show  $\bigwedge z. z \in \eta \text{ ' } D \implies e z = d z$  by (rule ed)
  qed
  have image-sum: image.sub.lincomb e (η ' D) =
    target.vadd.fincomp (λv. c v ⊙2 η v) D
  proof -
    have image.sub.lincomb (λz. c (inv-into D η z)) (η ' D) =
      target.vadd.fincomp (λv. c v ⊙2 η v) D

```

```

    by (rule image-lincomb-reindex[OF finD DM injD cR])
  then show ?thesis by (simp add: e-def)
qed
have map-zero:  $\eta$  (source.lincomb c D) =  $0_2$ 
proof -
  have  $\eta$  (source.lincomb c D) =
    target.vadd.fincomp ( $\lambda v. c \ v \odot_2 \eta \ v$ ) D
  by (rule hom.hom-lincomb[OF finD DM cR])
  also have ... = image.sub.lincomb e ( $\eta \text{ ` } D$ ) using image-sum by simp
  also have ... = image.sub.lincomb d ( $\eta \text{ ` } D$ ) by (rule image-eq)
  also have ... =  $0_2$  by (rule zero)
  finally show ?thesis .
qed
have linD-M: source.lincomb c D  $\in M$  using DM cR by (rule source.lincomb-closed)
  have linD-Ker: source.lincomb c D  $\in$  hom.Ker by (rule hom.Ker-memI[OF
map-zero linD-M])
  obtain a where a:  $a \in A \rightarrow_E R$ 
    source.lincomb c D = kernel.sub.lincomb a A
  using kernel.sub.basis-spanning[OF A] linD-Ker
  unfolding kernel.sub.spanning-def by blast
  have aR:  $\bigwedge v. v \in A \implies a \ v \in R$  using a(1) by auto
  have sourceA: kernel.sub.lincomb a A = source.lincomb a A
    by (rule kernel.sub-lincomb-eq[OF finA AKer aR])
  have eq-comb: source.lincomb c D = source.lincomb a A using a(2) sourceA by
simp
  have source-ind: source.mod.lin-indep C
    using source.basis-lin-indep[OF C] source.lin-indep-iff-mod-lin-indep[OF finC
CM] by simp
  have c-coeff: source.mod.coeffs-on c D using cR by (rule source.mod.coeffs-onI)
  have a-coeff: source.mod.coeffs-on a A using aR by (rule source.mod.coeffs-onI)
  obtain v where v:  $v \in D \ y = \eta \ v$  using y by blast
  have coord-eq: (if  $v \in D$  then c v else 0) = (if  $v \in A$  then a v else 0)
    by (rule source.mod.lin-indep-lincomb-unique-gen
[OF source-ind finD DC finA AC c-coeff a-coeff eq-comb])
  have c v = 0 using coord-eq v by (simp add: D-def)
  then show d y = 0 using v by (simp add: c-def)
qed
qed
qed

```

Rank-nullity follows by extending a basis of the kernel to a basis of the source. The complementary vectors map bijectively to a basis of the image.

theorem rank-nullity:

```

assumes B: source.basis B
shows plus kernel.sub.dimension image.sub.dimension = source.dimension
proof -
  obtain A where A: kernel.sub.basis A
    using kernel.subspace-basis-exists[OF B] by blast
  have AKer:  $A \subseteq$  hom.Ker using A by (simp add: kernel.sub.basis-def)

```

```

have source-ind: source.lin-indep A
  using kernel.sub.basis-lin-indep[OF A] kernel.sub-lin-indep-iff[OF AKer] by simp
obtain C where AC:  $A \subseteq C$  and C: source.basis C
  using source.basis-extension[OF source-ind B] by blast
have image-basis: image.sub.basis ( $\eta \restriction (C \setminus A)$ )
  by (rule image-basis-complement[OF A C AC])
have finC: finite C using C by (simp add: source.basis-def)
have inj: inj-on  $\eta \restriction (C \setminus A)$ 
  by (rule injective-on-basis-complement[OF A C AC])
have image-card:  $\text{card } (\eta \restriction (C \setminus A)) = \text{card } (C \setminus A)$ 
  by (rule card-image[OF inj])
have card-split:  $\text{plus } (\text{card } A) (\text{card } (C \setminus A)) = \text{card } C$ 
proof -
  have finA: finite A by (rule finite-subset[OF AC finC])
  have finD: finite  $(C \setminus A)$  using finC by simp
  have card  $(A \cup (C \setminus A)) = \text{plus } (\text{card } A) (\text{card } (C \setminus A))$ 
    by (rule card-Un-disjoint[OF finA finD]) simp
  moreover have  $A \cup (C \setminus A) = C$  using AC by auto
  ultimately show ?thesis by simp
qed
have kernel.sub.dimension = card A
  by (rule kernel.sub.dimension-eq-any-field[OF A])
moreover have image.sub.dimension = card  $(C \setminus A)$ 
  using image.sub.dimension-eq-any-field[OF image-basis] image-card by simp
moreover have source.dimension = card C
  by (rule source.dimension-eq-any-field[OF C])
ultimately show ?thesis using card-split by simp
qed

```

A bijective linear map preserves dimension. Its kernel is the trivial subspace, so the image of any source basis is a basis of the entire target.

```

theorem dimension-eq-of-bij-betw:
  assumes B: source.basis B and bij: bij-betw  $\eta \restriction M \restriction M'$ 
  shows source.dimension = target.dimension
proof -
  have injM: inj-on  $\eta \restriction M$  and imageM:  $\eta \restriction M = M'$ 
    using bij by (auto simp: bij-betw-def)
  have ker:  $\text{hom.Ker} = \{0_M\}$ 
    using injM hom.injective-iff-kernel-trivial by simp
  have empty-basis: kernel.sub.basis {}
    by (rule kernel.sub.basis-empty-trivial[OF ker])
  have image-basis: image.sub.basis ( $\eta \restriction B$ )
    using image-basis-complement[OF empty-basis B] by simp
  have target-basis: target.basis ( $\eta \restriction B$ ) using image-basis imageM by simp
  have BM:  $B \subseteq M$  using B by (simp add: source.basis-def)
  have injB: inj-on  $\eta \restriction B$  using injM BM by (rule inj-on-subset)
  have card-image:  $\text{card } (\eta \restriction B) = \text{card } B$  by (rule card-image[OF injB])
  show ?thesis
    using source.dimension-eq-any-field[OF B]

```

target.dimension-eq-any-field[*OF target-basis*] *card-image* **by** *simp*
qed

92.1 Finite-dimensional consequences

Rank-nullity immediately bounds both the kernel and the image by the source dimension. These named forms avoid making clients repeat arithmetic reasoning about the decomposition.

corollary *kernel-dimension-le*:
assumes *B*: *source.basis B*
shows *kernel.sub.dimension* ≤ *source.dimension*
using *rank-nullity*[*OF B*] **by** *presburger*

corollary *image-dimension-le*:
assumes *B*: *source.basis B*
shows *image.sub.dimension* ≤ *source.dimension*
using *rank-nullity*[*OF B*] **by** *presburger*

An injective map has trivial kernel, so rank-nullity identifies the source dimension with the image dimension.

theorem *image-dimension-eq-of-injective*:
assumes *B*: *source.basis B* **and** *inj*: *inj-on* η *M*
shows *image.sub.dimension* = *source.dimension*
proof –
have *ker*: *hom.Ker* = {**0**_{*M*}}
using *inj* *hom.injective-iff-kernel-trivial* **by** *simp*
have *ker-dim*: *kernel.sub.dimension* = 0
using *kernel.subspace-dimension-eq-zero-iff*[*OF B*] *ker* **by** *simp*
show *?thesis* **using** *rank-nullity*[*OF B*] *ker-dim* **by** *simp*
qed

Comparing the image of an injective map with the finite-dimensional target gives the expected dimension inequality.

theorem *source-dimension-le-of-injective*:
assumes *B*: *source.basis B* **and** *C*: *target.basis C* **and** *inj*: *inj-on* η *M*
shows *source.dimension* ≤ *target.dimension*
proof –
have *image-eq*: *image.sub.dimension* = *source.dimension*
by (rule *image-dimension-eq-of-injective*[*OF B inj*])
have *image.sub.dimension* ≤ *target.dimension*
by (rule *image.subspace-dimension-le*[*OF C*])
then show *?thesis* **using** *image-eq* **by** *simp*
qed

A surjective map has the whole target as its image, so the image bound from rank-nullity compares the target dimension directly with the source dimension.

theorem *target-dimension-le-of-surjective*:
assumes *B*: *source.basis B* **and** *surj*: η ‘ *M* = *M*’

```

    shows  $target.dimension \leq source.dimension$ 
  proof -
    have  $dim-image: image.sub.dimension = target.dimension$  using  $surj$  by  $simp$ 
    show  $?thesis$  using  $image-dimension-le[OF B] dim-image$  by  $simp$ 
  qed

```

For finite-dimensional spaces of equal dimension, a linear map is injective exactly when it is surjective. Injectivity gives an image of full target dimension; conversely, surjectivity and rank-nullity force the kernel to have dimension zero.

```

theorem injective-iff-surjective:
  assumes  $B: source.basis B$  and  $C: target.basis C$ 
    and  $dim: source.dimension = target.dimension$ 
  shows  $inj-on \eta M \longleftrightarrow \eta ' M = M'$ 
proof
  assume  $inj: inj-on \eta M$ 
  have  $image-source: image.sub.dimension = source.dimension$ 
    by (rule  $image-dimension-eq-of-injective[OF B inj]$ )
  have  $image-target: image.sub.dimension = target.dimension$ 
    using  $image-source dim$  by  $simp$ 
  show  $\eta ' M = M'$ 
    using  $image.subspace-eq-ambient-iff-dimension-eq[OF C] image-target$  by  $blast$ 
next
  assume  $surj: \eta ' M = M'$ 
  have  $image-target: image.sub.dimension = target.dimension$  using  $surj$  by  $simp$ 
  have  $image-source: image.sub.dimension = source.dimension$ 
    using  $image-target dim$  by  $simp$ 
  have  $ker-dim: kernel.sub.dimension = 0$ 
    using  $rank-nullity[OF B] image-source$  by  $presburger$ 
  have  $ker: hom.Ker = \{0_M\}$ 
    using  $kernel.subspace-dimension-eq-zero-iff[OF B] ker-dim$  by  $blast$ 
  show  $inj-on \eta M$ 
    using  $hom.injective-iff-kernel-trivial ker$  by  $simp$ 
qed

end

```

```

notation  $plus$  (infixl  $\langle + \rangle$  65)
notation  $minus$  (infixl  $\langle - \rangle$  65)
unbundle  $uminus-syntax$ 

```

```
end
```

93 Isomorphisms of finite-dimensional vector spaces

```

theory Finite-Dimensional-Isomorphism
  imports Rank-Nullity Free-Module-Universal
begin

```

Suppress HOL arithmetic syntax while binding the scalar-ring operations in the

locale header, as in the other module-homomorphism theories.

no-notation *plus* (**infixl** $\langle + \rangle$ 65)
no-notation *minus* (**infixl** $\langle - \rangle$ 65)
unbundle *no uminus-syntax*

A pair of vector spaces over the same scalar field is the ambient setting for constructing a linear isomorphism before a particular map has been chosen.

locale *paired-vector-spaces* =
source: *Vector-Space* R $(+)$ $(\cdot)$ **0** **1** $(\oplus)$ $\mathbf{0}_M$ M $(\odot)$ $+$
target: *Vector-Space* R $(+)$ $(\cdot)$ **0** **1** $(\oplus_2)$ $\mathbf{0}_2$ M' $(\odot_2)$
for R **and** *addition* (**infixl** $\langle + \rangle$ 65) **and** *multiplication* (**infixl** $\langle \cdot \rangle$ 70)
and *zero* $(\langle \mathbf{0} \rangle)$ **and** *unit* $(\langle \mathbf{1} \rangle)$
and M **and** *madd* (**infixl** $\langle \oplus \rangle$ 65) **and** *mzero* $(\langle \mathbf{0}_M \rangle)$
and *scale* (**infixr** $\langle \odot \rangle$ 75)
and M' **and** *madd'* (**infixl** $\langle \oplus_2 \rangle$ 65) **and** *mzero'* $(\langle \mathbf{0}_2 \rangle)$
and *scale'* (**infixr** $\langle \odot_2 \rangle$ 75)
begin

A bijection between finite bases extends uniquely to a module homomorphism. Its image contains the target basis and hence, by minimality of span, the whole target. Equal basis cardinalities then let rank-nullity turn this surjection into a bijection.

theorem *basis-isomorphism-exists*:
assumes B : *source.basis* B **and** C : *target.basis* C **and** *card-eq*: $\text{card } B = \text{card } C$
shows $\exists \eta$.
module-homomorphism R $(+)$ $(\cdot)$ **0** **1**
 M $(\oplus)$ $\mathbf{0}_M$ $(\odot)$ M' $(\oplus_2)$ $\mathbf{0}_2$ $(\odot_2)$ η
 $\wedge$ *bij-betw* η M $M' \wedge$ *bij-betw* η B C
proof –
have *finB*: *finite* B **and** *finC*: *finite* C **using** B C **by** (*auto simp: source.basis-def target.basis-def*)
obtain f **where** f : *bij-betw* f B C
using *finite-same-card-bij*[*OF finB finC card-eq*] **by** *blast*
have BM : $B \subseteq M$ **using** B **by** (*simp add: source.basis-def*)
have CM' : $C \subseteq M'$ **using** C **by** (*simp add: target.basis-def*)
interpret E : *free-module-ext* R $(+)$ $(\cdot)$ **0** **1**
 M $(\oplus)$ $\mathbf{0}_M$ $(\odot)$ M' $(\oplus_2)$ $\mathbf{0}_2$ $(\odot_2)$ B f
proof
show *source.mod.module-basis* B **by** (*rule source.basis-module-basis*[*OF B*])
show $\bigwedge v. v \in B \implies f v \in M'$
using f CM' **by** (*auto simp: bij-betw-def*)
qed
have *linear*: *linear-map* R $(+)$ $(\cdot)$ **0** **1**
 M $(\oplus)$ $\mathbf{0}_M$ $(\odot)$ M' $(\oplus_2)$ $\mathbf{0}_2$ $(\odot_2)$ E .*free-ext*
proof (*intro linear-map.intro*)
show *Vector-Space* R $(+)$ $(\cdot)$ **0** **1** $(\oplus)$ $\mathbf{0}_M$ M $(\odot)$
by (*rule source.Vector-Space-axioms*)
show *Vector-Space* R $(+)$ $(\cdot)$ **0** **1** $(\oplus_2)$ $\mathbf{0}_2$ M' $(\odot_2)$


```

    by (rule target.Vector-Space-axioms)
  show module-homomorphism R (+) (·) 0 1
    M (⊕) 0M (⊙) M' (⊕2) 02 (⊙2) E.free-ext
    by (rule E.free-ext-hom)
qed
interpret L: linear-map R (+) (·) 0 1
  M (⊕) 0M (⊙) M' (⊕2) 02 (⊙2) E.free-ext
  by (rule linear)
have basis-bij: bij-betw E.free-ext B C
proof (rule bij-betw-cong[THEN iffD2, OF - f])
  show  $\bigwedge x. x \in B \implies E.free-ext\ x = f\ x$  by (rule E.free-ext-extends)
qed
then have basis-image: E.free-ext ' B = C by (simp add: bij-betw-def)
have C-image: C  $\subseteq$  E.free-ext ' M using basis-image BM by blast
have span-image: target.mod.span C  $\subseteq$  E.free-ext ' M
  by (rule target.mod.span-minimal[OF L.hom.image-submodule C-image])
have target-spanning: target.mod.spanning C
  using target.basis-module-basis[OF C] by (rule target.mod.module-basis-spanning)
have target-span: target.mod.span C = M' using target-spanning by (simp add:
target.mod.spanning-def)
have image-closed: E.free-ext ' M  $\subseteq$  M' using L.hom.hom-closed by blast
have surj: E.free-ext ' M = M' using span-image target-span image-closed by blast
have dim: source.dimension = target.dimension
  using source.dimension-eq-any-field[OF B] target.dimension-eq-any-field[OF C]
card-eq by simp
have inj: inj-on E.free-ext M
  using L.injective-iff-surjective[OF B C dim] surj by simp
have carrier-bij: bij-betw E.free-ext M M'
  using inj surj by (simp add: bij-betw-def)
show ?thesis using E.free-ext-hom carrier-bij basis-bij by blast
qed

```

Thus any chosen bases of equal-dimensional finite vector spaces can be matched by a linear isomorphism.

corollary *isomorphism-exists-of-equal-dimension:*

assumes B: source.basis B **and** C: target.basis C

and dim: source.dimension = target.dimension

shows $\exists \eta.$

module-homomorphism R (+) (·) 0 1

M (⊕) 0_M (⊙) M' (⊕₂) 0₂ (⊙₂) η

$\wedge$ bij-betw η M M' $\wedge$ bij-betw η B C

proof –

have card B = card C

using source.dimension-eq-any-field[OF B] target.dimension-eq-any-field[OF C]

dim **by** simp

then show ?thesis **by** (rule basis-isomorphism-exists[OF B C])

qed

Combining existence with invariance of dimension gives the classification theorem: two finite-dimensional vector spaces over the same field are linearly iso-

morphic exactly when their dimensions agree.

theorem *isomorphism-exists-iff-dimension-eq*:

assumes *B*: *source.basis B* **and** *C*: *target.basis C*

shows $\exists \eta$.

module-homomorphism $R (+) (\cdot) \mathbf{0} \mathbf{1}$

$M (\oplus) \mathbf{0}_M (\odot) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \eta$

$\wedge \text{bij-betw } \eta \ M \ M'$

$\longleftrightarrow \text{source.dimension} = \text{target.dimension}$

proof

assume $\exists \eta$.

module-homomorphism $R (+) (\cdot) \mathbf{0} \mathbf{1}$

$M (\oplus) \mathbf{0}_M (\odot) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \eta$

$\wedge \text{bij-betw } \eta \ M \ M'$

then obtain η **where** *hom*:

module-homomorphism $R (+) (\cdot) \mathbf{0} \mathbf{1}$

$M (\oplus) \mathbf{0}_M (\odot) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \eta$

and *bij*: *bij-betw* $\eta \ M \ M'$ **by** *blast*

interpret *L*: *linear-map* $R (+) (\cdot) \mathbf{0} \mathbf{1}$

$M (\oplus) \mathbf{0}_M (\odot) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \eta$

proof (*intro linear-map.intro*)

show *Vector-Space* $R (+) (\cdot) \mathbf{0} \mathbf{1} (\oplus) \mathbf{0}_M \ M (\odot)$

by (*rule source.Vector-Space-axioms*)

show *Vector-Space* $R (+) (\cdot) \mathbf{0} \mathbf{1} (\oplus_2) \mathbf{0}_2 \ M' (\odot_2)$

by (*rule target.Vector-Space-axioms*)

show *module-homomorphism* $R (+) (\cdot) \mathbf{0} \mathbf{1}$

$M (\oplus) \mathbf{0}_M (\odot) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \eta$

by (*rule hom*)

qed

show *source.dimension* = *target.dimension*

by (*rule L.dimension-eq-of-bij-betw*[*OF B bij*])

next

assume *dim*: *source.dimension* = *target.dimension*

show $\exists \eta$.

module-homomorphism $R (+) (\cdot) \mathbf{0} \mathbf{1}$

$M (\oplus) \mathbf{0}_M (\odot) M' (\oplus_2) \mathbf{0}_2 (\odot_2) \eta$

$\wedge \text{bij-betw } \eta \ M \ M'$

using *isomorphism-exists-of-equal-dimension*[*OF B C dim*] **by** *blast*

qed

end

notation *plus* (**infixl** $\langle + \rangle$ 65)

notation *minus* (**infixl** $\langle - \rangle$ 65)

unbundle *uminus-syntax*

end

94 Complements in finite-dimensional vector spaces

```
theory Finite-Dimensional-Splitting
  imports Rank-Nullity Quotient-Module Module-Iso-Theorems
begin
```

94.1 Dimension of quotient spaces

```
no-notation plus (infixl <+> 65)
no-notation minus (infixl <-> 65)
unbundle no uminus-syntax
```

A quotient of a vector space by a subspace inherits the vector-space structure already constructed at the module level. The natural projection is therefore a linear map.

```
locale vector-quotient = vector-subspace
begin
```

```
sublocale factor: submodule-in-module
  R (+) (·) 0 1 (⊕) 0V V (⊙) W
proof
  show mod.submodule W by (rule W-submodule)
qed
```

```
lemma quotient-vector-space:
  Vector-Space R (+) (·) 0 1
  factor.qadd (factor.Qclass 0V) factor.Qcarrier factor.qscale
proof (intro Vector-Space.intro Vector-Space-axioms.intro)
  show Field R (+) (·) 0 1 by unfold-locales
  show Abelian-Group factor.Qcarrier factor.qadd (factor.Qclass 0V)
    by (rule factor.quotient.madd-group)
  show ∧ a X. [a ∈ R; X ∈ factor.Qcarrier] ⇒
    factor.qscale a X ∈ factor.Qcarrier
    by (rule factor.quotient.scale-closed)
  show ∧ a X Y. [a ∈ R; X ∈ factor.Qcarrier; Y ∈ factor.Qcarrier] ⇒
    factor.qscale a (factor.qadd X Y) = factor.qadd (factor.qscale a X) (factor.qscale
a Y)
    by (rule factor.quotient.scale-distrib-madd)
  show ∧ a b X. [a ∈ R; b ∈ R; X ∈ factor.Qcarrier] ⇒
    factor.qscale (a + b) X = factor.qadd (factor.qscale a X) (factor.qscale b X)
    by (rule factor.quotient.scale-distrib-add)
  show ∧ a b X. [a ∈ R; b ∈ R; X ∈ factor.Qcarrier] ⇒
    factor.qscale (a · b) X = factor.qscale a (factor.qscale b X)
    by (rule factor.quotient.scale-scale)
  show ∧ X. X ∈ factor.Qcarrier ⇒ factor.qscale 1 X = X
    by (rule factor.quotient.scale-one)
qed
```

```
sublocale quotient: Vector-Space
  R (+) (·) 0 1
```

```

factor.qadd factor.Qclass 0V factor.Qcarrier factor.qscale
by (rule quotient-vector-space)

```

sublocale *projection: linear-map*

```

R (+) (·) 0 1
V (⊕) 0V (⊙)
factor.Qcarrier factor.qadd factor.Qclass 0V factor.qscale factor.nat-proj
by unfold-locales

```

The dimension of a subspace plus the dimension of its quotient is the dimension of the ambient space. This is rank–nullity for the natural quotient projection.

theorem *quotient-dimension:*

```

assumes B: basis B
shows plus sub.dimension quotient.dimension = dimension
using projection.rank-nullity[OF B] factor.nat-proj-Ker factor.nat-proj-image by
simp

```

lemma *quotient-basis-exists:*

```

assumes B: basis B
shows ∃ C. quotient.basis C

```

proof –

```

obtain A where A: projection.kernel.sub.basis A
using projection.kernel.subspace-basis-exists[OF B] by blast
have AKer: A ⊆ factor.nat-proj.Ker
using A by (simp add: projection.kernel.sub.basis-def)
have indA: lin-indep A
using projection.kernel.sub.basis-lin-indep[OF A]
projection.kernel.sub-lin-indep-iff[OF AKer] by simp
obtain C where AC: A ⊆ C and C: basis C
using basis-extension[OF indA B] by blast
have projection.image.sub.basis (factor.nat-proj ‘ (C \ A))
by (rule projection.image-basis-complement[OF A C AC])
then have quotient.basis (factor.nat-proj ‘ (C \ A))
using factor.nat-proj-image by simp
then show ?thesis by blast

```

qed

end

94.2 Grassmann’s dimension formula

```

locale vector-subspace-pair = Vector-Space +
fixes U W :: 'b set
assumes U-submodule: mod.submodule U
and W-submodule: mod.submodule W
begin

```

abbreviation *sum-carrier* **where**

```

sum-carrier ≡ mod.submodule-sum U W

```

abbreviation *restricted-projection* **where**
restricted-projection $\equiv$ *mod.proj-res* *U* *W*

abbreviation *ambient-submodule* **where**
ambient-submodule $\equiv$ *mod.submodule*

lemmas *ambient-submodule-inter* = *mod.submodule-inter*
lemmas *ambient-sum-submodule* = *mod.submodule-sum-submodule*
lemmas *ambient-sum-incl-right* = *mod.submodule-sum-incl-right*
lemmas *ambient-submodule-subset* = *mod.submodule-subset*
lemmas *ambient-submodule-zero* = *mod.submodule-zero*
lemmas *ambient-submodule-add* = *mod.submodule-add*
lemmas *ambient-submodule-scale* = *mod.submodule-scale*
lemmas *ambient-submodule-vector-space* = *submodule-vector-space*
lemmas *ambient-proj-res-hom* = *mod.proj-res-hom*
lemmas *ambient-proj-res-Ker* = *mod.proj-res-Ker*
lemmas *ambient-proj-res-image* = *mod.proj-res-image*

sublocale *left: vector-subspace*
R (+) (·) **0** **1** ($\oplus$) **0**_{*V*} *V* ($\odot$) *U*

proof
show *ambient-submodule* *U* **by** (*rule U-submodule*)
qed

sublocale *right: vector-quotient*
R (+) (·) **0** **1** ($\oplus$) **0**_{*V*} *V* ($\odot$) *W*

proof
show *ambient-submodule* *W* **by** (*rule W-submodule*)
qed

sublocale *inter: vector-subspace*
R (+) (·) **0** **1** ($\oplus$) **0**_{*V*} *V* ($\odot$) *U* $\cap$ *W*

proof
show *ambient-submodule* (*U* $\cap$ *W*)
by (*rule ambient-submodule-inter*[*OF U-submodule W-submodule*])
qed

sublocale *sum: vector-subspace*
R (+) (·) **0** **1** ($\oplus$) **0**_{*V*} *V* ($\odot$) *sum-carrier*

proof
show *ambient-submodule* *sum-carrier*
by (*rule ambient-sum-submodule*[*OF U-submodule W-submodule*])
qed

The right summand is also a subspace of the sum.

sublocale *sum-quotient: vector-quotient*
R (+) (·) **0** **1** ($\oplus$) **0**_{*V*} *sum-carrier* ($\odot$) *W*

proof
show *sum.sub.mod.submodule* *W*
proof (*rule sum.sub.mod.submoduleI*)

```

show  $W \subseteq \text{sum-carrier}$ 
  by (rule ambient-sum-incl-right[OF U-submodule W-submodule])
show  $\mathbf{0}_V \in W$  by (rule ambient-submodule-zero[OF W-submodule])
show  $\bigwedge u v. \llbracket u \in W; v \in W \rrbracket \implies u \oplus v \in W$ 
  by (rule ambient-submodule-add[OF W-submodule])
show  $\bigwedge a v. \llbracket a \in R; v \in W \rrbracket \implies a \odot v \in W$ 
  by (rule ambient-submodule-scale[OF W-submodule])
qed
qed

```

Restricting the quotient projection by W to U gives a linear map whose kernel is $U \cap W$ and whose image consists of the classes represented in the sum.

```

lemma restricted-projection-hom:
  module-homomorphism R (+) (·)  $\mathbf{0}$   $\mathbf{1}$ 
  U ( $\oplus$ )  $\mathbf{0}_V$  ( $\odot$ )
  right.factor.Qcarrier right.factor.qadd (right.factor.Qclass  $\mathbf{0}_V$ )
  right.factor.qscale restricted-projection
by (rule ambient-proj-res-hom[OF U-submodule W-submodule])

```

```

lemma restricted-linear-map:
  linear-map R (+) (·)  $\mathbf{0}$   $\mathbf{1}$ 
  U ( $\oplus$ )  $\mathbf{0}_V$  ( $\odot$ )
  right.factor.Qcarrier right.factor.qadd (right.factor.Qclass  $\mathbf{0}_V$ )
  right.factor.qscale restricted-projection
proof (intro linear-map.intro)
  show Vector-Space R (+) (·)  $\mathbf{0}$   $\mathbf{1}$  ( $\oplus$ )  $\mathbf{0}_V$  U ( $\odot$ )
    by (rule ambient-submodule-vector-space[OF U-submodule])
  show Vector-Space R (+) (·)  $\mathbf{0}$   $\mathbf{1}$ 
    right.factor.qadd (right.factor.Qclass  $\mathbf{0}_V$ )
    right.factor.Qcarrier right.factor.qscale
    by (rule right.quotient-vector-space)
  show module-homomorphism R (+) (·)  $\mathbf{0}$   $\mathbf{1}$ 
    U ( $\oplus$ )  $\mathbf{0}_V$  ( $\odot$ )
    right.factor.Qcarrier right.factor.qadd (right.factor.Qclass  $\mathbf{0}_V$ )
    right.factor.qscale restricted-projection
    by (rule restricted-projection-hom)
qed

```

```

sublocale restricted: linear-map
  R (+) (·)  $\mathbf{0}$   $\mathbf{1}$ 
  U ( $\oplus$ )  $\mathbf{0}_V$  ( $\odot$ )
  right.factor.Qcarrier right.factor.qadd right.factor.Qclass  $\mathbf{0}_V$ 
  right.factor.qscale restricted-projection
  by (rule restricted-linear-map)

```

```

lemma quotient-class-eq:
  assumes  $x: x \in \text{sum-carrier}$ 
  shows  $\text{sum-quotient.factor.Qclass } x = \text{right.factor.Qclass } x$ 
proof -

```

```

have xV: x ∈ V
  using x ambient-submodule-subset
  [OF ambient-sum-submodule[OF U-submodule W-submodule]] by blast
show ?thesis
  using sum-quotient.factor.madd-sub.Class-is-Left-Coset[OF x]
  right.factor.madd-sub.Class-is-Left-Coset[OF xV] by simp
qed

lemma sum-quotient-carrier-eq:
  sum-quotient.factor.Qcarrier = right.factor.Qclass ' sum-carrier
proof
  show sum-quotient.factor.Qcarrier ⊆ right.factor.Qclass ' sum-carrier
  proof
    fix X assume X: X ∈ sum-quotient.factor.Qcarrier
    then obtain x where x: x ∈ sum-carrier
      X = sum-quotient.factor.Qclass x
      using sum-quotient.factor.madd-sub.representant-exists by blast
    then show X ∈ right.factor.Qclass ' sum-carrier
      using quotient-class-eq[OF x(1)] by blast
  qed
next
  show right.factor.Qclass ' sum-carrier ⊆ sum-quotient.factor.Qcarrier
  proof
    fix X assume X ∈ right.factor.Qclass ' sum-carrier
    then obtain x where x: x ∈ sum-carrier X = right.factor.Qclass x by blast
    have sum-quotient.factor.Qclass x ∈ sum-quotient.factor.Qcarrier
      using x(1) by simp
    then show X ∈ sum-quotient.factor.Qcarrier using x quotient-class-eq by simp
  qed
qed

lemma sum-quotient-carrier-eq-image:
  sum-quotient.factor.Qcarrier = restricted-projection ' U
  using sum-quotient-carrier-eq ambient-proj-res-image[OF U-submodule W-submodule]
  by simp

lemma quotient-add-eq:
  assumes X: X ∈ sum-quotient.factor.Qcarrier
  and Y: Y ∈ sum-quotient.factor.Qcarrier
  shows sum-quotient.factor.qadd X Y = right.factor.qadd X Y
proof -
  obtain x where x: x ∈ sum-carrier X = sum-quotient.factor.Qclass x
    using X sum-quotient.factor.madd-sub.representant-exists by blast
  obtain y where y: y ∈ sum-carrier Y = sum-quotient.factor.Qclass y
    using Y sum-quotient.factor.madd-sub.representant-exists by blast
  have xy: x ⊕ y ∈ sum-carrier
    by (rule ambient-submodule-add
      [OF ambient-sum-submodule[OF U-submodule W-submodule] x(1) y(1)])
  have xV: x ∈ V and yV: y ∈ V
    using x y ambient-submodule-subset

```

```

    [OF ambient-sum-submodule[OF U-submodule W-submodule]] by auto
  have sum-quotient.factor.qadd X Y = sum-quotient.factor.Qclass (x ⊕ y)
    using x y by (simp add: sum-quotient.factor.madd-sub.Class-commutes-with-composition)
  also have ... = right.factor.Qclass (x ⊕ y) by (rule quotient-class-eq[OF xy])
  also have ... = right.factor.qadd X Y
    using x y xV yV quotient-class-eq
    by (simp add: right.factor.madd-sub.Class-commutes-with-composition)
  finally show ?thesis .
qed

```

lemma quotient-scale-eq:

```

  assumes a: a ∈ R and X: X ∈ sum-quotient.factor.Qcarrier
  shows sum-quotient.factor.qscale a X = right.factor.qscale a X
proof –
  obtain x where x: x ∈ sum-carrier X = sum-quotient.factor.Qclass x
    using X sum-quotient.factor.madd-sub.representant-exists by blast
  have ax: a ⊙ x ∈ sum-carrier
    by (rule ambient-submodule-scale
      [OF ambient-sum-submodule[OF U-submodule W-submodule] a x(1)])
  have xV: x ∈ V
    using x ambient-submodule-subset
    [OF ambient-sum-submodule[OF U-submodule W-submodule]] by blast
  have sum-quotient.factor.qscale a X = sum-quotient.factor.Qclass (a ⊙ x)
    using a x by (simp add: sum-quotient.factor.qscale-Class)
  also have ... = right.factor.Qclass (a ⊙ x) by (rule quotient-class-eq[OF ax])
  also have ... = right.factor.qscale a X
    using a xV quotient-class-eq by (simp add: right.factor.qscale-Class)
  finally show ?thesis .
qed

```

definition quotient-embedding **where**

quotient-embedding = restrict id sum-quotient.factor.Qcarrier

lemma quotient-embedding-apply [simp]:

```

  X ∈ sum-quotient.factor.Qcarrier ⟹ quotient-embedding X = X
by (simp add: quotient-embedding-def)

```

lemma quotient-embedding-hom:

```

  module-homomorphism R (+) (·) 0 1
    sum-quotient.factor.Qcarrier sum-quotient.factor.qadd
    (sum-quotient.factor.Qclass 0_V) sum-quotient.factor.qscale
    (restricted-projection ‘ U) right.factor.qadd (right.factor.Qclass 0_V)
    right.factor.qscale quotient-embedding

```

proof

```

  show quotient-embedding ∈ sum-quotient.factor.Qcarrier →E restricted-projection
    ‘ U
    using sum-quotient-carrier-eq-image
    by (auto simp: quotient-embedding-def PiE-iff extensional-def)
  show ∧X Y. [X ∈ sum-quotient.factor.Qcarrier;
    Y ∈ sum-quotient.factor.Qcarrier] ⟹

```



```

    quotient-embedding (sum-quotient.factor.qadd X Y) =
    right.factor.qadd (quotient-embedding X) (quotient-embedding Y)
  proof -
    fix X Y assume X: X ∈ sum-quotient.factor.Qcarrier
    and Y: Y ∈ sum-quotient.factor.Qcarrier
    have XY: sum-quotient.factor.qadd X Y ∈ sum-quotient.factor.Qcarrier
    using X Y by (rule sum-quotient.factor.quotient.madd-closed)
    have quotient-embedding (sum-quotient.factor.qadd X Y) =
      sum-quotient.factor.qadd X Y
    by (rule quotient-embedding-apply[OF XY])
    also have ... = right.factor.qadd X Y by (rule quotient-add-eq[OF X Y])
    also have ... = right.factor.qadd (quotient-embedding X) (quotient-embedding Y)
    using X Y by simp
    finally show quotient-embedding (sum-quotient.factor.qadd X Y) =
      right.factor.qadd (quotient-embedding X) (quotient-embedding Y) .
  qed
show  $\bigwedge a X. \llbracket a \in R; X \in \text{sum-quotient.factor.Qcarrier} \rrbracket \implies$ 
  quotient-embedding (sum-quotient.factor.qscale a X) =
  right.factor.qscale a (quotient-embedding X)
proof -
  fix a X assume a: a ∈ R and X: X ∈ sum-quotient.factor.Qcarrier
  have aX: sum-quotient.factor.qscale a X ∈ sum-quotient.factor.Qcarrier
  using a X by (rule sum-quotient.quotient.scale-closed)
  have quotient-embedding (sum-quotient.factor.qscale a X) =
    sum-quotient.factor.qscale a X
  by (rule quotient-embedding-apply[OF aX])
  also have ... = right.factor.qscale a X by (rule quotient-scale-eq[OF a X])
  also have ... = right.factor.qscale a (quotient-embedding X) using X by simp
  finally show quotient-embedding (sum-quotient.factor.qscale a X) =
    right.factor.qscale a (quotient-embedding X) .
qed
qed

lemma quotient-embedding-linear-map:
  linear-map R (+) (·) 0 1
  sum-quotient.factor.Qcarrier sum-quotient.factor.qadd
  (sum-quotient.factor.Qclass 0V) sum-quotient.factor.qscale
  (restricted-projection ‘ U) right.factor.qadd (right.factor.Qclass 0V)
  right.factor.qscale quotient-embedding
proof (intro linear-map.intro)
  show Vector-Space R (+) (·) 0 1
    sum-quotient.factor.qadd (sum-quotient.factor.Qclass 0V)
    sum-quotient.factor.Qcarrier sum-quotient.factor.qscale
  by (rule sum-quotient.quotient-vector-space)
  show Vector-Space R (+) (·) 0 1
    right.factor.qadd (right.factor.Qclass 0V)
    (restricted-projection ‘ U) right.factor.qscale
  by (rule right.quotient.submodule-vector-space[OF restricted.hom.image-submodule])
  show module-homomorphism R (+) (·) 0 1
    sum-quotient.factor.Qcarrier sum-quotient.factor.qadd

```

```

      (sum-quotient.factor.Qclass  $\mathbf{0}_V$ ) sum-quotient.factor.qscale
      (restricted-projection '  $U$ ) right.factor.qadd (right.factor.Qclass  $\mathbf{0}_V$ )
      right.factor.qscale quotient-embedding
    by (rule quotient-embedding-hom)
  qed

```

```

sublocale quotient-embedding: linear-map
   $R$  (+) (·)  $\mathbf{0}$   $\mathbf{1}$ 
  sum-quotient.factor.Qcarrier sum-quotient.factor.qadd
  sum-quotient.factor.Qclass  $\mathbf{0}_V$  sum-quotient.factor.qscale
  restricted-projection '  $U$  right.factor.qadd right.factor.Qclass  $\mathbf{0}_V$ 
  right.factor.qscale quotient-embedding
  by (rule quotient-embedding-linear-map)

```

```

lemma quotient-embedding-bij:
  bij-betw quotient-embedding sum-quotient.factor.Qcarrier (restricted-projection '  $U$ )
  using sum-quotient-carrier-eq-image
  by (auto simp: quotient-embedding-def bij-betw-def inj-on-def)

```

Grassmann's dimension formula. The intersection dimension plus the sum dimension equals the sum of the two subspace dimensions.

```

theorem grassmann-dimension:
  assumes  $B$ : basis  $B$ 
  shows plus inter.sub.dimension sum.sub.dimension =
    plus left.sub.dimension right.sub.dimension
proof -
  obtain  $BU$  where  $BU$ : left.sub.basis  $BU$  using left.subspace-basis-exists[OF  $B$ ] by
  blast
  obtain  $BS$  where  $BS$ : sum.sub.basis  $BS$  using sum.subspace-basis-exists[OF  $B$ ] by
  blast
  obtain  $BQ$  where  $BQ$ : sum-quotient.quotient.basis  $BQ$ 
    using sum-quotient.quotient-basis-exists[OF  $BS$ ] by blast
  have rank: plus inter.sub.dimension restricted.image.sub.dimension = left.sub.dimension
proof -
  have ker: restricted.hom.Ker =  $U \cap W$ 
    unfolding restricted.hom.Ker-def
    using ambient-proj-res-Ker[OF  $U$ -submodule  $W$ -submodule] by simp
  have dim: restricted.kernel.sub.dimension = inter.sub.dimension
    using ker by simp
  show ?thesis using restricted.rank-nullity[OF  $BU$ ] dim by simp
qed
  have quotient: plus right.sub.dimension sum-quotient.quotient.dimension = sum.sub.dimension
    using sum-quotient.quotient-dimension[OF  $BS$ ] by simp
  have iso: sum-quotient.quotient.dimension = restricted.image.sub.dimension
    using quotient-embedding.dimension-eq-of-bij-betw[OF  $BQ$  quotient-embedding-bij]
  .
  show ?thesis using rank quotient iso by presburger
qed

```

```

corollary direct-sum-dimension:
  assumes B: basis B and disjoint:  $U \cap W = \{0_V\}$ 
  shows sum.sub.dimension = plus left.sub.dimension right.sub.dimension
proof –
  have empty-basis: inter.sub.basis {}
    by (rule inter.sub.basis-empty-trivial[OF disjoint])
  have inter.sub.dimension = 0
    using inter.sub.dimension-eq-any-field[OF empty-basis] by simp
  then show ?thesis using grassmann-dimension[OF B] by simp
qed

end

```

94.3 Complements in finite-dimensional vector spaces

```

context vector-subspace
begin

```

Every subspace of a finite-dimensional vector space has a complementary subspace. Extend a basis of the subspace to an ambient basis and take the span of the added basis vectors. Independence gives the trivial intersection, while span minimality gives the full sum.

```

theorem complementary-subspace-exists:
  assumes B: basis B
  shows  $\exists U. \text{mod.submodule } U \wedge W \cap U = \{0_V\} \wedge$ 
     $\text{mod.submodule-sum } W \ U = V$ 
proof –
  obtain A where A: sub.basis A and AW:  $A \subseteq W$ 
    using subspace-basis-exists[OF B] sub.basis-def by blast
  then obtain C where AC:  $A \subseteq C$  and C: basis C
    using B basis-extension sub.basis-lin-indep sub-lin-indep-iff by meson
  define D where  $D = C \setminus A$ 
  define U where  $U = \text{span } D$ 
  have CV:  $C \subseteq V$  using C by (simp add: basis-def)
  have DV:  $D \subseteq V$  using CV D-def by blast
  have U-submodule: mod.submodule U
    unfolding U-def by (rule mod.span-submodule[OF DV])

  have sub.mod.spanning A
    using sub.basis-module-basis[OF A] by (rule sub.mod.module-basis-spanning)
  then have W-span:  $W = \text{span } A$ 
    using sub.mod.spanning-def sub-span-eq by force
  have mod.lin-indep C
    using basis-module-basis[OF C] by (rule mod.module-basis-lin-indep)
  then have trivial-intersection:  $W \cap U = \{0_V\}$ 
    using AC D-def U-def W-span mod.span-inter-eq-trivial by force
  have sum-submodule: mod.submodule (mod.submodule-sum W U)
    by (rule mod.submodule-sum-submodule[OF W-submodule U-submodule])
  have A-sum:  $A \subseteq \text{mod.submodule-sum } W \ U$ 

```

```

    using AW mod.submodule-sum-incl-left[OF W-submodule U-submodule] by blast
  have D-U:  $D \subseteq U$ 
    unfolding U-def by (rule mod.span-incl[OF DV])
  have D-sum:  $D \subseteq \text{mod.submodule-sum } W \ U$ 
    using D-U mod.submodule-sum-incl-right[OF W-submodule U-submodule] by blast
  have C-sum:  $C \subseteq \text{mod.submodule-sum } W \ U$ 
    using A-sum D-def D-sum by auto
  have span-subset-sum:  $\text{span } C \subseteq \text{mod.submodule-sum } W \ U$ 
    by (rule mod.span-minimal[OF sum-submodule C-sum])
  have span C = V
    using C CV basis-spanning mod.spanningI mod.spanning-def spanning-span-all by
  force
  then have mod.submodule-sum W U = V
    using mod.submodule-def span-subset-sum sum-submodule by force
  then show ?thesis using U-submodule trivial-intersection by blast
qed

end

```

context *vector-quotient*
begin

Consequently every quotient of a finite-dimensional vector space is represented by a complementary subspace of the ambient space. The isomorphism is not merely existential: it is the natural quotient projection restricted to that complement.

theorem *complementary-quotient-isomorphism-exists:*

```

assumes B: basis B
shows  $\exists U. \text{mod.submodule } U$ 
   $\wedge W \cap U = \{\mathbf{0}_V\}$ 
   $\wedge \text{mod.submodule-sum } W \ U = V$ 
   $\wedge \text{module-homomorphism } R \ (+) \ (\cdot) \ \mathbf{0} \ \mathbf{1}$ 
     $U \ (\oplus) \ \mathbf{0}_V \ (\odot)$ 
     $\text{factor.Qcarrier factor.qadd (factor.Qclass } \mathbf{0}_V) \text{ factor.qscale}$ 
     $(\text{mod.proj-res } U \ W)$ 
   $\wedge \text{bij-betw } (\text{mod.proj-res } U \ W) \ U \ \text{factor.Qcarrier}$ 
proof –
  obtain U where U: mod.submodule U
  and disjoint:  $W \cap U = \{\mathbf{0}_V\}$ 
  and full: mod.submodule-sum W U = V
  using complementary-subspace-exists[OF B] by blast
  have mod.submodule-sum U W = V
    using full mod.submodule-sum-commute[OF U W-submodule] by simp
  then show ?thesis using U disjoint full
    using W-submodule mod.complement-quotient-isomorphism by blast
qed

end

```

```

notation plus (infixl <+> 65)
notation minus (infixl <-> 65)
unbundle uminus-syntax

```

```

end

```

95 Simple Algebraic Extensions inside the Complex Numbers

theory *Field-Extension*

```

imports Complex-Field-Extension HOL-Computational-Algebra.Field-as-Ring HOL-Computational-Algebra.Polynomial
begin

```

Towards the transitivity of the Galois group of a rational polynomial on its complex roots, we develop the base isomorphism $\mathbb{Q}(\alpha) \cong \mathbb{Q}(\beta)$ for two roots α, β of the same irreducible polynomial. We work concretely: rational polynomials are *rat poly*, evaluated at a complex point through the embedding *complex-of-rat*. The evaluation map *evalQ a* is a ring homomorphism $\text{rat poly} \Rightarrow \text{complex}$; its image is the subfield $\mathbb{Q}(a)$ and its kernel is the set of multiples of the minimal polynomial. Since *rat poly* is a principal ideal domain (a field-coefficient polynomial ring), the kernel characterization needs only Bezout's identity.

95.1 Evaluation of a rational polynomial at a complex point

definition *evalQ* :: $\text{complex} \Rightarrow \text{rat poly} \Rightarrow \text{complex}$ **where**
evalQ a p = *poly (map-poly of-rat p) a*

lemma *map-poly-of-rat-add*: $\text{map-poly of-rat } (p + q) = \text{map-poly of-rat } p + \text{map-poly of-rat } q$
by (*intro poly-eqI*) (*simp-all add: coeff-map-poly of-rat-add*)

lemma *map-poly-of-rat-mult*: $\text{map-poly of-rat } (p * q) = \text{map-poly of-rat } p * \text{map-poly of-rat } q$
by (*intro poly-eqI*) (*simp-all add: coeff-map-poly of-rat-mult of-rat-sum coeff-mult*)

lemma *evalQ-add*: $\text{evalQ } a \ (p + q) = \text{evalQ } a \ p + \text{evalQ } a \ q$
by (*simp add: evalQ-def map-poly-of-rat-add*)

lemma *evalQ-mult*: $\text{evalQ } a \ (p * q) = \text{evalQ } a \ p * \text{evalQ } a \ q$
by (*simp add: evalQ-def map-poly-of-rat-mult*)

lemma *evalQ-0*: $\text{evalQ } a \ 0 = 0$ **by** (*simp add: evalQ-def*)

lemma *evalQ-1*: $\text{evalQ } a \ 1 = 1$ **by** (*simp add: evalQ-def map-poly-1'*)

lemma *evalQ-diff*: $\text{evalQ } a \ (p - q) = \text{evalQ } a \ p - \text{evalQ } a \ q$
by (*metis eq-diff-eq evalQ-add*)

The constant rational polynomial evaluates to the embedded constant; in particular *evalQ* fixes the rationals.

lemma *evalQ-const*: $\text{evalQ } a \ [:c:] = \text{of-rat } c$
by (*simp add: evalQ-def map-poly-pCons*)

95.2 The kernel is generated by the minimal polynomial

If m is irreducible (over $\mathbb{Q}$) and both m and r vanish at a , then m divides r : by irreducibility m is coprime to any non-multiple, so Bezout would force $0 = 1$ under evaluation.

lemma *evalQ-kernel-dvd*:
assumes *m-irr*: *irreducible m* **and** *ma*: $\text{evalQ } a \ m = 0$ **and** *ra*: $\text{evalQ } a \ r = 0$
shows $m \text{ dvd } r$
proof (*rule ccontr*)
assume *ndvd*: $\neg m \text{ dvd } r$
have *pm*: *prime-elem m* **using** *m-irr* **by** (*rule field-poly-irreducible-imp-prime*)
have *coprime m r* **using** *pm ndvd* **by** (*meson coprimeI dvd-trans prime-elemD2*)
then have *g1*: $\text{gcd } m \ r = 1$ **by** *simp*
obtain *u v* **where** $u * m + v * r = \text{gcd } m \ r$
using *bezout-coefficients-fst-snd*[*of m r*] **by** *blast*
with g1 have *bz*: $u * m + v * r = 1$ **by** *simp*
have $\text{evalQ } a \ u * \text{evalQ } a \ m + \text{evalQ } a \ v * \text{evalQ } a \ r = \text{evalQ } a \ 1$
using *bz* **by** (*metis evalQ-add evalQ-mult*)
then have $(0::\text{complex}) = 1$ **using** *ma ra* **by** (*simp add: evalQ-1*)
then show *False* **by** *simp*
qed

Consequently two roots of the same irreducible polynomial annihilate exactly the same rational polynomials.

lemma *evalQ-same-kernel*:
assumes *m-irr*: *irreducible m* **and** *ma*: $\text{evalQ } a \ m = 0$ **and** *mb*: $\text{evalQ } b \ m = 0$
and *ra*: $\text{evalQ } a \ r = 0$
shows $\text{evalQ } b \ r = 0$
proof –
have $m \text{ dvd } r$ **using** *m-irr ma ra* **by** (*rule evalQ-kernel-dvd*)
then show *?thesis* **using** *mb*
using *evalQ-mult* **by** *auto*
qed

95.3 The simple extension $\mathbb{Q}(a)$ and the base isomorphism

The subfield $\mathbb{Q}(a)$ is the image of the evaluation map: every element is a rational polynomial expression in a .

definition $\text{Qx} :: \text{complex} \Rightarrow \text{complex set}$ **where**

$Qx\ a = \text{range}\ (\text{evalQ}\ a)$

lemma *Qx-mem-evalQ* [intro]: $\text{evalQ}\ a\ p \in Qx\ a$
by (*simp add: Qx-def*)

lemma *Qx-self*: $a \in Qx\ a$

proof –

have $\text{evalQ}\ a\ [:0, 1:] = a$ **by** (*simp add: evalQ-def map-poly-pCons*)

then show *?thesis* **by** (*metis Qx-mem-evalQ*)

qed

lemma *Qx-Rats*: $\text{of-rat}\ c \in Qx\ a$

using *evalQ-const* **by** (*metis Qx-mem-evalQ*)

For two roots a, b of the same irreducible polynomial m , the map sending $\text{evalQ}\ a\ p$ to $\text{evalQ}\ b\ p$ is well defined (independent of the representing polynomial p), by *evalQ-same-kernel*.

definition *simple-iso* :: $\text{rat poly} \Rightarrow \text{complex} \Rightarrow \text{complex} \Rightarrow \text{complex} \Rightarrow \text{complex}$ **where**
simple-iso $m\ a\ b\ y = \text{evalQ}\ b\ (\text{SOME}\ p.\ \text{evalQ}\ a\ p = y)$

context

fixes $m :: \text{rat poly}$ **and** $a\ b :: \text{complex}$

assumes *m-irr*: *irreducible* m **and** *ma*: $\text{evalQ}\ a\ m = 0$ **and** *mb*: $\text{evalQ}\ b\ m = 0$

begin

Well-definedness: on the image, *simple-iso* agrees with *evalQ* b .

lemma *simple-iso-evalQ*: $\text{simple-iso}\ m\ a\ b\ (\text{evalQ}\ a\ p) = \text{evalQ}\ b\ p$

proof –

let $?q = \text{SOME}\ q.\ \text{evalQ}\ a\ q = \text{evalQ}\ a\ p$

have $\text{evalQ}\ a\ ?q = \text{evalQ}\ a\ p$ **by** (*rule someI[of $\lambda q.\ \text{evalQ}\ a\ q = \text{evalQ}\ a\ p$]*) *rule*

then have $\text{evalQ}\ a\ (?q - p) = 0$ **by** (*simp add: evalQ-diff*)

then have $\text{evalQ}\ b\ (?q - p) = 0$ **by** (*rule evalQ-same-kernel[OF m-irr ma mb]*)

then have $\text{evalQ}\ b\ ?q = \text{evalQ}\ b\ p$ **by** (*simp add: evalQ-diff*)

then show *?thesis* **by** (*simp add: simple-iso-def*)

qed

lemma *simple-iso-add*:

$\llbracket x \in Qx\ a; y \in Qx\ a \rrbracket \Longrightarrow \text{simple-iso}\ m\ a\ b\ (x + y) = \text{simple-iso}\ m\ a\ b\ x + \text{simple-iso}\ m\ a\ b\ y$

by (*clarsimp simp: Qx-def*) (*metis simple-iso-evalQ evalQ-add*)

lemma *simple-iso-mult*:

$\llbracket x \in Qx\ a; y \in Qx\ a \rrbracket \Longrightarrow \text{simple-iso}\ m\ a\ b\ (x * y) = \text{simple-iso}\ m\ a\ b\ x * \text{simple-iso}\ m\ a\ b\ y$

by (*clarsimp simp: Qx-def*) (*metis simple-iso-evalQ evalQ-mult*)

lemma *simple-iso-1*: $\text{simple-iso}\ m\ a\ b\ 1 = 1$

using *simple-iso-evalQ*[of 1] **by** (*simp add: evalQ-1*)

```

lemma simple-iso-0: simple-iso m a b 0 = 0
  using simple-iso-evalQ[of 0] by (simp add: evalQ-0)

```

It fixes the rationals and sends a to b .

```

lemma simple-iso-Rats: simple-iso m a b (of-rat c) = of-rat c
  using simple-iso-evalQ[of [:c:]] by (simp add: evalQ-const)

```

```

lemma simple-iso-gen: simple-iso m a b a = b

```

```

proof –
  have evalQ a [:0, 1:] = a evalQ b [:0, 1:] = b by (simp-all add: evalQ-def map-poly-pCons)
  then show ?thesis using simple-iso-evalQ[of [:0, 1:]] by simp
qed

```

It maps $\mathbb{Q}(a)$ onto $\mathbb{Q}(b)$.

```

lemma simple-iso-image: simple-iso m a b ‘ Qx a = Qx b
proof
  show simple-iso m a b ‘ Qx a ⊆ Qx b
    by (auto simp: Qx-def simple-iso-evalQ)
  show Qx b ⊆ simple-iso m a b ‘ Qx a
    by (metis Qx-def imageI image-subsetI simple-iso-evalQ)
qed

```

end

end

96 The Galois correspondence between intermediate fields and subgroups

```

theory Galois-Correspondence
  imports Galois-Automorphism Order-Theory
begin

```

Let $F \subseteq K$ be subfields of an ambient type-class field. There are two natural operations between the *intermediate fields* of the extension and the *subgroups* of its Galois group *field-auto K F*:

- *fixing* a set of automorphisms, $H \mapsto K^H$, the subfield of K on which every element of H acts as the identity;
- *the relative Galois group* of an intermediate field, $E \mapsto \text{field-auto } K E$, the automorphisms of K fixing E pointwise.

Both are *inclusion-reversing*: a bigger group of automorphisms fixes fewer points, and a bigger intermediate field admits fewer automorphisms fixing it. So this is an *antitone* adjoint pair, and the way to see it as a *Galois-Connection* — which is defined for *monotone* adjoints — is to order one of the two lattices by reverse

inclusion. We order the subgroups by $\supseteq$; then both maps become isotone and the adjunction reads $K^H \subseteq E \longleftrightarrow H \supseteq \text{field-auto } K E$.

96.1 The fixed field of a set of automorphisms

The subfield of K fixed pointwise by every automorphism in H . We do not require H to be a subgroup.

The fixed field is a subfield by the generic closure theorem *field-auto-fixed-field-subfield*; the empty case is K itself.

theorem *fixed-field-subfield*:

assumes $Ksub$: Subfield K **and** H : $H \subseteq \text{field-auto } K F$
shows Subfield (fixed-field $K H$)
by (rule *field-auto-fixed-field-subfield*[$OF Ksub H$])

The base field lies inside every fixed field: by definition each automorphism in the Galois group fixes F pointwise.

lemma *base-subset-fixed-field*:

assumes H : $H \subseteq \text{field-auto } K F$ **and** FK : $F \subseteq K$
shows $F \subseteq \text{fixed-field } K H$
using *field-auto-base-subset-fixed-field*[$OF H FK$].

96.2 Intermediate fields and the relative Galois group

The intermediate fields of the extension $F \subseteq K$: the subfields lying between the two.

definition *inter-fields* :: 'a :: field set $\Rightarrow$ 'a set $\Rightarrow$ 'a set set

where *inter-fields* $K F \equiv \{E. \text{Subfield } E \wedge F \subseteq E \wedge E \subseteq K\}$

lemma *inter-fields-iff*:

$E \in \text{inter-fields } K F \longleftrightarrow \text{Subfield } E \wedge F \subseteq E \wedge E \subseteq K$
by (simp add: *inter-fields-def*)

The two ends of the extension are intermediate fields.

lemma *base-in-inter-fields*:

$\llbracket \text{Subfield } F; F \subseteq K \rrbracket \Longrightarrow F \in \text{inter-fields } K F$
by (simp add: *inter-fields-iff*)

lemma *top-in-inter-fields*:

$\llbracket \text{Subfield } K; F \subseteq K \rrbracket \Longrightarrow K \in \text{inter-fields } K F$
by (simp add: *inter-fields-iff*)

Taking the relative Galois group is inclusion-reversing: a bigger intermediate field imposes more conditions, so fewer automorphisms survive.

lemma *field-auto-antitone*: $E \subseteq E' \Longrightarrow \text{field-auto } K E' \subseteq \text{field-auto } K E$

by (auto simp: *field-auto-def*)

The relative Galois group of an intermediate field is a subgroup of the whole Galois group: it consists of automorphisms of K , and fixing E pointwise implies fixing the smaller F pointwise.

theorem *field-auto-subgroup-of-Galois-group:*

assumes $Ksub$: Subfield K **and** FK : $F \subseteq K$ **and** E : $E \in \text{inter-fields } K F$
shows Subgroup (field-auto $K E$) (field-auto $K F$) (compose K) (identity K)

proof –

from E **have** $Esub$: Subfield E **and** FE : $F \subseteq E$ **and** EK : $E \subseteq K$
by (auto simp: inter-fields-iff)

— Work inside the whole Galois group as the ambient group, so that *subgroupI* applies.

interpret W : Group field-auto $K F$ compose K identity K

using $Ksub FK$ **by** (rule field-auto-group)

— The relative group is also a subgroup of the symmetric group, which supplies inverses.

interpret S : Subgroup field-auto $K E$ transformations.Sym K compose K identity K

using $Ksub EK$ **by** (rule field-auto-subgroup)

have EF : field-auto $K E \subseteq \text{field-auto } K F$ **using** FE **by** (rule field-auto-antitone)

show ?thesis

proof (rule $W.subgroupI$ [OF EF])

show identity $K \in \text{field-auto } K E$ **using** $Ksub EK$ **by** (rule field-auto-identity)

show $\bigwedge g h. \llbracket g \in \text{field-auto } K E; h \in \text{field-auto } K E \rrbracket \implies \text{compose } K g h \in \text{field-auto } K E$

using $Ksub EK$ **by** (blast intro: field-auto-compose)

— Invertibility and the inverse are computed in the symmetric group; because *field-auto* $K E$ is a subgroup there, the inverse stays inside it, and the two ambient groups agree on inverses by *Subgroup.subgroup-inverse-equality*.

show $\bigwedge g. g \in \text{field-auto } K E \implies W.invertible g$

using EF **by** (blast intro: $W.invertible$)

show $W.inverse g \in \text{field-auto } K E$ **if** $g \in \text{field-auto } K E$ **for** g

proof –

have gW : $g \in \text{field-auto } K F$ **using** that EF **by** blast

have $inv\text{-}in$: $S.sub.inverse g \in \text{field-auto } K E$

using that **by** (simp add: $S.sub.invertible\text{-}inverse\text{-}closed$)

— The symmetric-group inverse is a two-sided inverse of g in the ambient Galois group too, so the two notions of inverse agree.

have $W.inverse g = S.sub.inverse g$

proof (rule $W.inverse\text{-}equality$)

show $S.sub.inverse g \in \text{field-auto } K F$ **using** $inv\text{-}in EF$ **by** blast

qed (use that gW in auto)

then show $W.inverse g \in \text{field-auto } K E$ **using** $inv\text{-}in$ **by** simp

qed

qed

qed

The set of subgroups of the Galois group, as the second of the two lattices.

definition *galois-subgroups* :: ' a :: field set $\Rightarrow$ ' a set $\Rightarrow$ (' $a \Rightarrow$ ' a) set set

where *galois-subgroups* $K F \equiv$

$\{H. \text{Subgroup } H (\text{field-auto } K F) (\text{compose } K) (\text{identity } K)\}$

lemma *galois-subgroups-iff*:
 $H \in \text{galois-subgroups } K F \longleftrightarrow \text{Subgroup } H \text{ (field-auto } K F) \text{ (compose } K) \text{ (identity } K)$
by (*simp add: galois-subgroups-def*)

lemma *galois-subgroups-subset*:
assumes $H \in \text{galois-subgroups } K F$ **shows** $H \subseteq \text{field-auto } K F$
using *assms* **by** (*meson Subgroup-def Submonoid.subset galois-subgroups-iff*)

96.3 The adjunction

locale *galois-extension* =
fixes $K F :: 'a :: \text{field set}$
assumes $K\text{sub}: \text{Subfield } K$ **and** $F\text{sub}: \text{Subfield } F$ **and** $FK: F \subseteq K$
begin

Fixing lands in the intermediate fields: the fixed field of a subgroup is a subfield lying between F and K .

lemma *fixed-field-in-inter-fields*:
assumes $H: H \in \text{galois-subgroups } K F$
shows $\text{fixed-field } K H \in \text{inter-fields } K F$
proof –
have $HG: H \subseteq \text{field-auto } K F$ **using** H **by** (*rule galois-subgroups-subset*)
show *?thesis*
unfolding *inter-fields-iff*
proof (*intro conjI*)
show $\text{Subfield } (\text{fixed-field } K H)$ **using** $K\text{sub } HG$ **by** (*rule fixed-field-subfield*)
show $F \subseteq \text{fixed-field } K H$ **using** $HG FK$ **by** (*rule base-subset-fixed-field*)
show $\text{fixed-field } K H \subseteq K$ **by** (*rule fixed-field-subset*)
qed
qed

The adjunction. Both sides of the biconditional say the same thing — *every* automorphism in H fixes *every* point of E — so the proof is a matter of unfolding. This is the antitone (or *polarity*) form of an adjunction: both maps reverse inclusion, and each side bounds one argument by the image of the other.

It is worth being precise about which biconditional is formal, because the other one is a deep theorem. The statement proved here bounds E *above* by a fixed field and H *above* by a Galois group. The reverse reading — $\text{fixed-field } K H \subseteq E \longleftrightarrow \text{field-auto } K E \subseteq H$ — is *not* a formal consequence: its left-to-right direction asserts that every automorphism fixing E already lies in H , which is Artin’s theorem and needs H finite.

theorem *fixed-field-galois*:
assumes $E: E \in \text{inter-fields } K F$ **and** $H: H \in \text{galois-subgroups } K F$
shows $E \subseteq \text{fixed-field } K H \longleftrightarrow H \subseteq \text{field-auto } K E$
proof
assume $E \subseteq \text{fixed-field } K H$

```

then have fixes-E:  $\bigwedge \sigma x. \llbracket \sigma \in H; x \in E \rrbracket \implies \sigma x = x$ 
  by (auto dest: fixed-field-memD)
show  $H \subseteq \text{field-auto } K E$ 
  using fixes-E galois-subgroups-subset [OF H]
  by (smt (cvc5, dec-internal-enum-inst-sum) field-auto-mem-iff subset-iff)
next
  assume  $H \subseteq \text{field-auto } K E$ 
  then show  $E \subseteq \text{fixed-field } K H$ 
  using E by (meson field-auto-base-subset-fixed-field inter-fields-iff)
qed

```

The two *unit* inequalities, which hold unconditionally and in both directions. These are the parts of the correspondence that need no hypotheses; it is their *converses* that require normality and separability.

```

lemma le-fixed-field-galois-group:
  assumes E:  $E \in \text{inter-fields } K F$ 
  shows  $E \subseteq \text{fixed-field } K (\text{field-auto } K E)$ 
  using E by (auto simp: inter-fields-iff fixed-field-def field-auto-def)

```

```

lemma le-galois-group-fixed-field:
  assumes H:  $H \in \text{galois-subgroups } K F$ 
  shows  $H \subseteq \text{field-auto } K (\text{fixed-field } K H)$ 
  using assms fixed-field-galois fixed-field-in-inter-fields by blast

```

96.4 The Galois connection

Both maps reverse inclusion, so to view the pair as a *Galois-Connection* — whose adjoints are *isotone* — we order the subgroup lattice by *reverse* inclusion. Then the adjunction *fixed-field-galois* is literally the *galois* axiom, with *field-auto* K as the lower adjoint out of the intermediate fields and *fixed-field* K as the upper adjoint back.

Note the direction: *field-auto* K is the *lower* adjoint here, even though it is the map *into* the groups, because the group side carries the reversed order. Inclusion is a partial order on any set of sets, and so is its converse.

Inclusion and reverse inclusion are partial orders on the two lattices.

```

lemma partial-order-inter-fields: Partial-Order (inter-fields  $K F$ ) ( $\subseteq$ )
  by unfold-locales auto

```

```

lemma partial-order-galois-subgroups: Partial-Order (galois-subgroups  $K F$ ) ( $\lambda H H'. H' \subseteq H$ )
  by unfold-locales auto

```

```

theorem galois-correspondence:
  Galois-Connection
    (inter-fields  $K F$ ) ( $\subseteq$ )
    (galois-subgroups  $K F$ ) ( $\lambda H H'. H' \subseteq H$ )
    (field-auto  $K$ ) (fixed-field  $K$ )

```

```

proof (intro partial-order-inter-fields partial-order-galois-subgroups Galois-Connection.intro)
  show Galois-Connection-axioms
    (inter-fields K F) ( $\subseteq$ ) (galois-subgroups K F) ( $\lambda H H'. H' \subseteq H$ )
    (field-auto K) (fixed-field K)
proof
  show field-auto K  $\in$  inter-fields K F  $\rightarrow$  galois-subgroups K F
  using Ksub FK field-auto-subgroup-of-Galois-group by (auto simp: galois-subgroups-iff)
  show fixed-field K  $\in$  galois-subgroups K F  $\rightarrow$  inter-fields K F
  using fixed-field-in-inter-fields by auto
  fix E H assume E  $\in$  inter-fields K F and H  $\in$  galois-subgroups K F
  then show H  $\subseteq$  field-auto K E  $\longleftrightarrow$  E  $\subseteq$  fixed-field K H
  using fixed-field-galois by blast
qed
qed

```

The consequences, specialised to the correspondence. As in *Ideal-Extension* these come from a local **interpretation** of the theorem rather than a **sublocale** declaration.

```

context
begin

```

```

interpretation GC: Galois-Connection
  inter-fields K F ( $\subseteq$ ) galois-subgroups K F  $\lambda H H'. H' \subseteq H$ 
  field-auto K fixed-field K
  by (rule galois-correspondence)

```

Both maps are inclusion-reversing on their lattices — the isotonicity of the connection, read back through the reversed order on subgroups.

```

corollary field-auto-mono:
   $\llbracket E \in \text{inter-fields } K F; E' \in \text{inter-fields } K F; E \subseteq E' \rrbracket$ 
   $\implies \text{field-auto } K E' \subseteq \text{field-auto } K E$ 
  using isotone-le[OF GC.lower-isotone] by simp

```

```

corollary fixed-field-mono:
   $\llbracket H \in \text{galois-subgroups } K F; H' \in \text{galois-subgroups } K F; H' \subseteq H \rrbracket$ 
   $\implies \text{fixed-field } K H \subseteq \text{fixed-field } K H'$ 
  using isotone-le[OF GC.upper-isotone] by simp

```

The *semi-inverse* laws: one round trip and back is the identity. These are the strongest fixed-point statements available without finiteness — they say the closure and kernel operators are idempotent, *not* that either is the identity.

```

corollary field-auto-fixed-field-field-auto:
  E  $\in$  inter-fields K F  $\implies \text{field-auto } K (\text{fixed-field } K (\text{field-auto } K E)) = \text{field-auto } K E$ 
  using GC.lower-semi-inverse by simp

```

```

corollary fixed-field-field-auto-fixed-field:
  H  $\in$  galois-subgroups K F  $\implies \text{fixed-field } K (\text{field-auto } K (\text{fixed-field } K H)) = \text{fixed-field } K H$ 

```

```
using GC.upper-semi-inverse by simp
```

```
end
```

```
end
```

Scope and downstream completion. This theory deliberately stops at the representation-independent, inclusion-reversing adjunction, packaged as a *Galois-Connection*. No separability, normality or finiteness hypothesis belongs in this layer: the unit, counit and semi-inverse laws are the reusable order-theoretic API.

```
end
```

97 Normal and separable complex field extensions

```
theory Galois-Normality
```

```
imports Iso-Extension-Tower Extension-Properties
```

```
begin
```

The generic extension predicates and their root-set API live in *Extension-Properties*. This theory supplies the complex-specific characteristic-zero and algebraically-closed-field arguments needed by the Galois development: normality is witnessed by root transfer, while separability follows from the nonvanishing characteristic. The root-transfer theorem below is the useful bridge: an isomorphism between two simple subextensions extends across the finite root set, and therefore becomes an automorphism of the complex splitting field.

In characteristic zero every algebraic element is separable. This is the only separability input needed here; keeping the Bezout argument local avoids importing the Galois-degree and symmetric-generation developments into the normality layer.

```
lemma (in Subfield) galois-normality-poly-over-pderiv:
```

```
assumes p ∈ poly-over K
```

```
shows pderiv p ∈ poly-over K
```

```
proof (rule poly-overI)
```

```
fix i
```

```
have coeff (pderiv p) i = of-nat (Suc i) * coeff p (Suc i)
```

```
by (rule coeff-pderiv)
```

```
moreover have of-nat (Suc i) ∈ K by (rule of-nat-closed)
```

```
moreover have coeff p (Suc i) ∈ K using assms by (rule poly-over-coeff)
```

```
ultimately show coeff (pderiv p) i ∈ K by (simp add: mult-closed)
```

```
qed
```

```
lemma rsquarefree-minpoly-char-0:
```

```
fixes F :: 'a :: field-char-0 set and a :: 'a
```

```
assumes sfF: Subfield F and alg: algebraic-over F a
```

```

shows rsquarefree (minpoly F a)
proof -
  define m where m = minpoly F a
  have ism: is-minpoly F a m
    unfolding m-def by (rule Subfield.is-minpoly-minpoly[OF sfF alg])
  have mF: m ∈ poly-over F
    unfolding m-def by (rule Subfield.minpoly-over[OF sfF alg])
  have mnz: m ≠ 0
    unfolding m-def by (rule Subfield.minpoly-nonzero[OF sfF alg])
  have mdeg: degree m > 0
    unfolding m-def
    by (rule Subfield.minpoly-degree-pos[OF sfF
      Subfield.is-minpoly-minpoly[OF sfF alg]])
  have pmF: pderiv m ∈ poly-over F
    using Subfield.galois-normality-poly-over-pderiv[OF sfF mF] .
  have pmnz: pderiv m ≠ 0
    using mdeg by (simp add: pderiv-eq-0-iff)
  have pmdeg: degree (pderiv m) < degree m
    using mdeg by (simp add: degree-pderiv)
  have ndvd: ¬ m dvd pderiv m
  proof
    assume m dvd pderiv m
    then have degree m ≤ degree (pderiv m)
      using pmnz by (rule dvd-imp-degree-le)
    then show False using pmdeg by simp
  qed
  obtain u v where u: u ∈ poly-over F and v: v ∈ poly-over F
    and bez: u * m + v * pderiv m = 1
    using Subfield.minpoly-bezout[OF sfF ism pmF ndvd] by blast
  have ¬ (poly m r = 0 ∧ poly (pderiv m) r = 0) for r
  proof
    assume poly m r = 0 ∧ poly (pderiv m) r = 0
    then have poly (u * m + v * pderiv m) r = 0
      by simp
    then have (1 :: 'a) = 0 using bez by simp
    then show False by simp
  qed
  then show ?thesis unfolding m-def [symmetric]
    using mnz by (simp add: rsquarefree-roots)
qed

```

In characteristic zero, the distinct complex roots of an algebraic element's minimal polynomial are counted by its simple extension degree. This elementary root-count bridge is kept with the separability API so later Galois results need not import the Galois-action machinery.

```

lemma card-minpoly-roots-eq-ext-degree:
  fixes F :: complex set and a :: complex
  assumes sfF: Subfield F and alg: algebraic-over F a
  shows card {r. poly (minpoly F a) r = 0} = ext-degree F a

```

```

proof –
  define m where m = minpoly F a
  have mnz: m ≠ 0
    unfolding m-def by (rule Subfield.minpoly-nonzero[OF sfF alg])
  have monic: lead-coeff m = 1
    unfolding m-def by (rule Subfield.minpoly-monic[OF sfF alg])
  have rsf: rsquarefree m
    unfolding m-def by (rule rsquarefree-minpoly-char-0[OF sfF alg])
  have finR: finite {z. poly m z = 0}
    using mnz by (rule poly-roots-finite)
  have decomp: smult (lead-coeff m) (∏ z | poly m z = 0. [:– z, 1:]) = m
    using rsf by (rule complex-poly-decompose-rsquarefree)
  have degree m = degree (∏ z | poly m z = 0. [:– z, 1:])
    using decomp monic by (metis smult-1-left)
  also have ... = (∑ z | poly m z = 0. degree [:– z, 1:])
    using finR by (subst degree-prod-eq-sum-degree) auto
  also have ... = card {z. poly m z = 0} by simp
  finally show ?thesis by (simp add: ext-degree-def m-def)
qed

```

Every complex extension is separable whenever the element under consideration is algebraic.

```

theorem complex-extension-separable:
  assumes sfF: complex-subfield F
  shows separable-extension K F
proof (rule separable-extensionI)
  fix a assume aK: a ∈ K and alg: algebraic-over F a
  have sfF': Subfield F using sfF
    by (simp add: complex-subfield-iff-subfield)
  show rsquarefree (minpoly F a)
    by (rule rsquarefree-minpoly-char-0[OF sfF' alg])
qed

```

The root-transfer argument needs only the elementary homomorphism facts from the transitivity development. They are kept here as a small local interface so that this normality layer does not depend on the later Galois-action theory.

```

lemma normality-field-hom-on-id:
  assumes Subfield K
  shows field-hom-on K (λx. x)
proof –
  interpret Subfield K by fact
  show ?thesis by unfold-locales auto
qed

```

```

lemma normality-map-poly-fix:
  fixes f :: 'a :: field ⇒ 'a
  assumes hom: field-hom-on K f and sfF: Subfield F
  and p: p ∈ poly-over F and fixF: ∧x. x ∈ F ⇒ f x = x
  shows map-poly f p = p

```



```

proof (rule poly-eqI)
  fix n
  have cF: coeff p n ∈ F using p by (rule Subfield.poly-over-coeff[OF sfF])
  have coeff (map-poly f p) n = f (coeff p n)
    by (rule field-hom-on.coeff-map-poly-f[OF hom])
  also have ... = coeff p n using cF by (rule fixF)
  finally show coeff (map-poly f p) n = coeff p n .
qed

```

lemma normality-hom-preserves-roots:

```

fixes f :: 'a :: field ⇒ 'a
assumes hom: field-hom-on K f and sfF: Subfield F and FK: F ⊆ K
and fixF: ⋀x. x ∈ F ⇒ f x = x
and Phi: Phi ∈ poly-over F and r: r ∈ K and root: poly Phi r = 0
shows poly Phi (f r) = 0

```

proof –

```

have PhiK: Phi ∈ poly-over K
  using Phi poly-over-mono[OF FK] by blast
have poly Phi (f r) = poly (map-poly f Phi) (f r)
  using normality-map-poly-fix[OF hom sfF Phi fixF] by simp
also have ... = f (poly Phi r)
  by (rule field-hom-on.poly-map-poly-hom[OF hom PhiK r])
also have ... = 0 using root field-hom-on.hom-0[OF hom] by simp
finally show ?thesis .

```

qed

Extend a conjugacy between two simple subextensions across the root set of a splitting field. The resulting map permutes that finite root set, hence maps the generated field onto itself.

theorem splitting-field-root-transfer:

```

fixes F K :: complex set and p m :: complex poly and a b :: complex
assumes split: splitting-field F p K
and sfF: Subfield F
and alg: algebraic-over F a
and aK: a ∈ K
and minp: is-minpoly F a m
and root: poly m b = 0
shows ∃σ ∈ field-auto K F. σ a = b

```

proof –

```

have pF: p ∈ poly-over F and pnz: p ≠ 0
  and Kgen: K = generate-field (F ∪ poly-root-set p)
  using split by (auto simp: splitting-fieldD)
have Kdef: K = gen-subfield (F ∪ poly-root-set p)
  using Kgen by (simp add: gen-subfield-eq-generate-field)
have sfKc: complex-subfield K

```

proof –

```

have sfKc0: complex-subfield (gen-subfield (F ∪ poly-root-set p))
  by (rule gen-subfield-is-subfield)
then show ?thesis using Kdef by blast

```

```

qed
have sfK: Subfield K using sfKc by (simp add: complex-subfield-iff-subfield)
have FK:  $F \subseteq K$ 
  using Kdef gen-subfield-subset by blast
have Rfin: finite (poly-root-set p)
  using pnz by (rule finite-poly-root-set)
have Ralg:  $\bigwedge r. r \in \text{poly-root-set } p \implies \text{algebraic-over } F \ r$ 
  by (intro splitting-field-root-algebraic[OF split])
interpret F: field-hom-on F ( $\lambda x. x$ )
  by (rule normality-field-hom-on-id[OF sfF])
have map-minp: map-poly ( $\lambda x. x$ ) m = m
  by (rule poly-eqI) simp
have root-image: poly (map-poly ( $\lambda x. x$ ) m) b = 0
  using root by (simp only: map-minp)
obtain g0 where g0: field-hom-on (eval-img F a) g0
  and g0a: g0 a = b and g0F:  $\forall x \in F. g0 \ x = x$ 
  using F.iso-extension[OF alg minp root-image] by blast
define E where E = eval-img F a
have sfE: Subfield E
  unfolding E-def by (rule Subfield.subfield-eval-img[OF sfF alg])
have FE:  $F \subseteq E$ 
  unfolding E-def using Subfield.eval-img-base[OF sfF] by blast
have g0E: field-hom-on E g0 using g0 by (simp add: E-def)
have RalgE:  $\bigwedge r. r \in \text{poly-root-set } p \implies \text{algebraic-over } E \ r$ 
  using Ralg FE by (blast intro: algebraic-over-mono)
have closed:  $\bigwedge q :: \text{complex poly}. 0 < \text{degree } q \implies \exists c. \text{poly } q \ c = 0$ 
  by (intro allI impI fundamental-theorem-of-algebra) (simp add: constant-degree)
obtain g where g: field-hom-on (generate-field (E  $\cup$  poly-root-set p)) g
  and gE:  $\forall x \in E. g \ x = g0 \ x$ 
  using field-hom-on-extend-finite[OF closed Rfin g0E RalgE] by blast
have Egen: E = generate-field (F  $\cup$  {a})
  unfolding E-def by (rule Subfield.eval-img-eq-generate-field[OF sfF alg])
have EsubK: E  $\subseteq$  K
proof -
  have FKa: F  $\cup$  {a}  $\subseteq$  K using FK aK by blast
  have generate-field (F  $\cup$  {a})  $\subseteq$  K
    by (rule generate-field-least[OF sfK FKa])
  then show ?thesis using Egen by simp
qed
have LsubK: generate-field (E  $\cup$  poly-root-set p)  $\subseteq$  K
proof -
  have EL: E  $\cup$  poly-root-set p  $\subseteq$  K
    using EsubK splitting-field-roots-subset[OF split] by blast
  show ?thesis by (rule generate-field-least[OF sfK EL])
qed
have KsubL: K  $\subseteq$  generate-field (E  $\cup$  poly-root-set p)
proof -
  have FRsub: F  $\cup$  poly-root-set p  $\subseteq$  generate-field (E  $\cup$  poly-root-set p)
    using FE subset-generate-field[of E  $\cup$  poly-root-set p] by blast
  have sFL0: Subfield (generate-field (E  $\cup$  poly-root-set p))

```

```

    by (rule subfield-generate-field)
  have sfl: complex-subfield (generate-field (E ∪ poly-root-set p))
    using sfl0 by (simp add: complex-subfield-iff-subfield)
  have geninc: gen-subfield (F ∪ poly-root-set p) ⊆
    generate-field (E ∪ poly-root-set p)
    by (rule gen-subfield-minimal[OF sfl FRsub])
  show ?thesis
proof
  fix x assume xK: x ∈ K
  have xgen: x ∈ gen-subfield (F ∪ poly-root-set p)
    using xK Kdef by (simp only: Kdef)
  then show x ∈ generate-field (E ∪ poly-root-set p)
    by (rule geninc[THEN subsetD])
qed
qed
have dom-eq: generate-field (E ∪ poly-root-set p) = K
  using LsubK KsubL by (rule antisym)
have homK: field-hom-on K g using g dom-eq by simp
interpret G: field-hom-on K g by (rule homK)
have gfixF: ⋀x. x ∈ F ⟹ g x = x
proof -
  fix x assume xF: x ∈ F
  have xE: x ∈ E using FE xF by blast
  have g x = g0 x using gE xE by blast
  also have g0 x = x using g0F xF by blast
  finally show g x = x .
qed
have RsubK: poly-root-set p ⊆ K
  using splitting-field-roots-subset[OF split] .
have pK: p ∈ poly-over K
  using pF poly-over-mono[OF FK] by blast
have p-fix: map-poly g p = p
  by (rule normality-map-poly-fix[OF homK sfF pF gfixF])
have gRsub: g ‘ poly-root-set p ⊆ poly-root-set p
proof
  fix y assume y ∈ g ‘ poly-root-set p
  then obtain r where rR: r ∈ poly-root-set p and y: y = g r by blast
  have rK: r ∈ K using RsubK rR by blast
  have rroot: poly p r = 0 using rR by (simp only: poly-root-set-iff)
  have groot: poly p (g r) = 0
  proof -
    have poly p (g r) = poly (map-poly g p) (g r)
      using p-fix by simp
    also have ... = g (poly p r)
      by (rule field-hom-on.poly-map-poly-hom[OF homK pK rK])
    also have g (poly p r) = g 0 by (simp only: rroot)
    also have ... = 0 by (rule field-hom-on.hom-0[OF homK])
    finally show ?thesis .
  qed
  show y ∈ poly-root-set p using y groot by (simp only: poly-root-set-iff)

```

```

qed
have gReq: g ' poly-root-set p = poly-root-set p
proof -
  have inj: inj-on g (poly-root-set p)
    using G.inj-on RsubK by (rule inj-on-subset)
  show ?thesis by (rule endo-inj-surj[OF Rfin gRsub inj])
qed
have gF-eq: g ' F = F
proof (rule subset-antisym)
  show g ' F  $\subseteq$  F
    using gfixF by (auto intro: image-eqI[where x=x])
  show F  $\subseteq$  g ' F
    using gfixF by (auto intro: image-eqI[where x=x])
qed
have FRK: F  $\cup$  poly-root-set p  $\subseteq$  K using FK RsubK by blast
have Kgen: K = generate-field (F  $\cup$  poly-root-set p)
  using Kdef gen-subfield-eq-generate-field by simp
have gK: g ' K = K
proof -
  have gUn: g ' (F  $\cup$  poly-root-set p) = F  $\cup$  poly-root-set p
    by (simp only: image-Un gF-eq gReq)
  have g ' K = g ' generate-field (F  $\cup$  poly-root-set p) by (simp add: Kgen)
  also have ... = generate-field (g ' (F  $\cup$  poly-root-set p))
    by (rule G.image-generate-field-eq[OF FRK])
  also have ... = generate-field (F  $\cup$  poly-root-set p)
    by (rule arg-cong[OF gUn])
  also have generate-field (F  $\cup$  poly-root-set p) = K
    using Kgen by simp
  finally show ?thesis .
qed
have aut: restrict g K  $\in$  field-auto K F
  using field-hom-on-imp-field-auto[OF homK gK gfixF FK] .
have aE: a  $\in$  E unfolding E-def by (rule Subfield.eval-img-self[OF sfF])
have restrict g K a = b
  using aK aE g gE g0a by simp
then show ?thesis using aut by blast
qed

```

A splitting field is normal in the root-theoretic sense. The proof does not assume that the polynomial used to construct the splitting field is irreducible: the finite root set is only the carrier across which the simple conjugacy is extended.

theorem *splitting-field-normal*:

assumes *split*: *splitting-field* F p K **and** *sfF*: *complex-subfield* F

shows *normal-extension* K F

proof (rule *normal-extensionI*)

fix q a

assume *qF*: q $\in$ *poly-over* F **and** *irr*: *irreducible-over* F q

and *aK*: a $\in$ K **and** *qa*: *poly* q a = 0

have *sfF'*: *Subfield* F **using** *sfF* **by** (simp add: *complex-subfield-iff-subfield*)

```

have alg: algebraic-over F a
  unfolding algebraic-over-def using qF qa irr by (auto simp: irreducible-over-def)
obtain m where minp: is-minpoly F a m
  using Subfield.minpoly-exists[OF sfF' alg] by blast
have mroot: poly m b = 0 if b ∈ poly-root-set q for b
proof -
  have qb: poly q b = 0 using that by (simp add: poly-root-set-def)
  show ?thesis
  by (rule Subfield.irreducible-over-root-is-minpoly-root[OF sfF' irr qF minp qa qb])
qed
show poly-root-set q ⊆ K
proof
  fix b assume bq: b ∈ poly-root-set q
  obtain σ where s: σ ∈ field-auto K F and sab: σ a = b
  using splitting-field-root-transfer[OF split sfF' alg aK minp mroot[OF bq]] by
blast
  have σ a ∈ K using s aK by (auto simp: field-auto-mem-iff)
  then show b ∈ K using sab by simp
qed
qed
end

```

98 Fixed fields of finite normal and separable extensions

```

theory Galois-Fixed-Field
  imports Galois-Correspondence Galois-Normality
begin

```

A splitting field remains a splitting field after enlarging its base to an intermediate field. This is the representation bridge needed below: the root set is unchanged, while the base part of the generated subfield grows from F to E .

```

lemma splitting-field-over-intermediate:
  fixes F E K :: complex set and p :: complex poly
  assumes split: splitting-field F p K
    and E: E ∈ inter-fields K F
  shows splitting-field E p K
proof -
  obtain FE: F ⊆ E and EK: E ⊆ K
    using E by (auto simp: inter-fields-iff)
  have Ksub': Subfield K
    by (rule splitting-field-subfield[OF split])
  have Ksub: complex-subfield K
    using Ksub' by (simp add: complex-subfield-iff-subfield)
  obtain pF: p ∈ poly-over F and pnz: p ≠ 0
    and Kdef: K = gen-subfield (F ∪ poly-root-set p)
    using split by (auto simp: splitting-fieldD gen-subfield-eq-generate-field)

```

```

have pE:  $p \in \text{poly-over } E$ 
  using pF poly-over-mono[OF FE] by blast
have RsubK:  $\text{poly-root-set } p \subseteq K$ 
  using splitting-field-roots-subset[OF split] .
have KdefE:  $K = \text{gen-subfield } (E \cup \text{poly-root-set } p)$ 
proof (rule antisym)
  show  $K \subseteq \text{gen-subfield } (E \cup \text{poly-root-set } p)$ 
  proof -
    have FER:  $F \cup \text{poly-root-set } p \subseteq E \cup \text{poly-root-set } p$ 
      by (rule Un-mono) (rule FE, rule subset-refl)
    have genE:  $E \cup \text{poly-root-set } p \subseteq \text{gen-subfield } (E \cup \text{poly-root-set } p)$ 
      by (rule gen-subfield-subset)
    have  $F \cup \text{poly-root-set } p \subseteq \text{gen-subfield } (E \cup \text{poly-root-set } p)$ 
      by (rule subset-trans[OF FER genE])
    then have  $\text{gen-subfield } (F \cup \text{poly-root-set } p) \subseteq$ 
       $\text{gen-subfield } (E \cup \text{poly-root-set } p)$ 
      by (rule gen-subfield-minimal[OF gen-subfield-is-subfield])
    then show ?thesis using Kdef by simp
  qed
show  $\text{gen-subfield } (E \cup \text{poly-root-set } p) \subseteq K$ 
proof (rule gen-subfield-minimal[OF Ksub])
  show  $E \cup \text{poly-root-set } p \subseteq K$ 
  proof
    fix y assume y:  $y \in E \cup \text{poly-root-set } p$ 
    show  $y \in K$ 
    proof (cases y  $\in E$ )
      case True
      then show  $y \in K$  using EK by blast
    next
      case False
      then have  $y \in \text{poly-root-set } p$  using y by simp
      then show  $y \in K$  using RsubK by blast
    qed
  qed
qed
show ?thesis
  unfolding splitting-field-def
  using pE pnz KdefE by (simp add: gen-subfield-eq-generate-field)
qed

lemma card-gt-one-imp-exists-mem-ne:
  fixes A :: 'a set and x :: 'a and n :: nat
  assumes xA:  $x \in A$ 
    and cardA:  $\text{card } A = n$ 
    and ngt:  $1 < n$ 
  shows  $\exists b. b \in A \wedge b \neq x$ 
proof (rule ccontr)
  assume no-b:  $\neg (\exists b. b \in A \wedge b \neq x)$ 
  have  $A \subseteq \{x\}$  using no-b by auto

```

```

then have Aeq:  $A = \{x\}$  using xA by auto
have card-one:  $\text{card } A = 1$  using Aeq by simp
have n-one:  $n = 1$  using card-one cardA by simp
have impossible:  $(1 :: \text{nat}) < 1$ 
  using ngt n-one by (simp add: n-one)
have impossible-false: False
  using impossible by (rule less-irrefl-nat)
from impossible-false show False .
qed

```

The closure half of the correspondence is most transparent as a separation statement. If an element of a finite splitting field is outside the intermediate field, its minimal polynomial has a second root: characteristic zero gives distinct roots, and degree one is exactly the criterion for already belonging to the base. The root-transfer theorem then supplies an automorphism fixing the intermediate field that moves the element.

The explicit *algebraic-extension* assumption records the finite-extension input without importing Artin or an algebraic-closure construction. A later finite-generation layer can discharge it for the intended classes of root-generated extensions.

```

theorem fixed-field-eq-of-splitting-field:
  fixes K F E :: complex set and p :: complex poly
  assumes split: splitting-field F p K
    and E:  $E \in \text{inter-fields } K F$ 
    and alg: algebraic-extension K E
  shows fixed-field K (field-auto K E) = E
proof (rule antisym)
  have Ksub': Subfield K
    by (rule splitting-field-subfield[OF split])
  have Ksub: complex-subfield K
    using Ksub' by (simp add: complex-subfield-iff-subfield)
  have sfE: Subfield E
    using E by (auto simp: inter-fields-iff)
  show fixed-field K (field-auto K E)  $\subseteq$  E
proof
  fix x assume xfix:  $x \in \text{fixed-field } K (\text{field-auto } K E)$ 
  have xK:  $x \in K$ 
    using xfix by (auto simp: fixed-field-def)
  have fixed:  $\forall \sigma \in \text{field-auto } K E. \sigma x = x$ 
    using xfix by (auto simp: fixed-field-def)
  show  $x \in E$ 
proof (rule ccontr)
  assume xnotE:  $x \notin E$ 
  have algx: algebraic-over E x
    by (rule algebraic-extensionD[OF alg xK])
  have splitE: splitting-field E p K
    by (rule splitting-field-over-intermediate[OF split E])
  have minp: is-minpoly E x (minpoly E x)
    by (rule Subfield.is-minpoly-minpoly[OF sfE algx])

```

```

have xroot: poly (minpoly E x) x = 0
  by (rule Subfield.minpoly-root[OF sfE algx])
have deg-gt: 1 < ext-degree E x
proof -
  have degree-ne: ext-degree E x ≠ 1
  proof
    assume ext-degree E x = 1
    then have x ∈ E
      using Subfield.ext-degree-eq-1-iff[OF sfE algx] by blast
    then show False using xnotE by simp
  qed
  have degree-pos: 0 < ext-degree E x
    by (rule Subfield.ext-degree-pos[OF sfE algx])
  then show ?thesis using degree-ne by simp
qed
define R where R = {r. poly (minpoly E x) r = 0}
have cardR: card R = ext-degree E x
  unfolding R-def by (rule card-minpoly-roots-eq-ext-degree[OF sfE algx])
have xR: x ∈ R
  unfolding R-def using xroot by simp
obtain b where bR: b ∈ R and bne: b ≠ x
  using card-gt-one-imp-exists-mem-ne[OF xR cardR deg-gt] by blast
have mb: poly (minpoly E x) b = 0
  using bR by (simp add: R-def)
obtain σ where s: σ ∈ field-auto K E and moved: σ x = b
  using splitting-field-root-transfer[OF splitE sfE algx xK minp mb] by blast
have σ x = x using fixed s by blast
then show False using moved bne by simp
qed
qed
show E ⊆ fixed-field K (field-auto K E)
  using E by (auto simp: inter-fields-iff fixed-field-def field-auto-def)
qed
end

```

99 Primitive elements of finite field extensions

theory *Primitive-Element*
imports *Algebraic-Transitivity Extension-Properties Field-Mult-Cyclic Galois-Transitivity*
begin

A primitive element is an element whose simple extension is the whole target field. The definition is deliberately just the equality with *eval-img*: the hypotheses that make this set a field belong to the surrounding extension theorem, rather than being hidden in the predicate itself.

definition *primitive-element* :: 'a :: field set ⇒ 'a set ⇒ 'a ⇒ bool
where *primitive-element* K L a ≡ L = eval-img K a

lemma *primitive-elementI*: $L = \text{eval-img } K \ a \implies \text{primitive-element } K \ L \ a$
by (*simp add: primitive-element-def*)

lemma *primitive-elementD*: $\text{primitive-element } K \ L \ a \implies L = \text{eval-img } K \ a$
by (*simp add: primitive-element-def*)

lemma
assumes *primitive-element* $K \ L \ a$ *Subfield* K
shows *primitive-element-mem*: $a \in L$ **and** *primitive-element-base*: $K \subseteq L$
using *assms Subfield.eval-img-base Subfield.eval-img-self primitive-elementD* **by** *blast+*

99.1 The two-generator theorem over an infinite base

The classical separating-linear-combination argument is the substantive core of the primitive element theorem. For separable algebraic elements a and b , only finitely many scalars c can make two distinct pairs of conjugates have the same value under $(x,y) \mapsto x + c*y$. An infinite base field therefore supplies a scalar outside that exceptional set. The resulting element $a + c*b$ recovers both generators.

theorem *primitive-element-two-generators*:
fixes $F :: 'a :: \text{alg-closed-field set}$
assumes *sfF*: *Subfield* F **and** *infF*: *infinite* F
and *alg-a*: *algebraic-over* $F \ a$ **and** *alg-b*: *algebraic-over* $F \ b$
and *sep-a*: *rsquarefree* (*minpoly* $F \ a$) **and** *sep-b*: *rsquarefree* (*minpoly* $F \ b$)
shows $\exists \theta. \text{eval-img } (\text{eval-img } F \ a) \ b = \text{eval-img } F \ \theta$
proof –
define p **where** $p \equiv \text{minpoly } F \ a$
define q **where** $q \equiv \text{minpoly } F \ b$
define A **where** $A \equiv \{x. \text{poly } p \ x = 0\}$
define B **where** $B \equiv \{y. \text{poly } q \ y = 0\}$
define C **where** $C \equiv \{(x - a) / (b - y) \mid x \ y. \ x \in A \wedge y \in B \wedge y \neq b\}$
obtain $p0$: $p \neq 0$ **and** $q0$: $q \neq 0$
using *p-def q-def rsquarefree-def sep-a sep-b* **by** *auto*
have *finA*: *finite* A **and** *finB*: *finite* B
unfolding *A-def B-def* **using** $p0 \ q0$ *poly-roots-finite* **by** *blast+*
then have *finC*: *finite* C
unfolding *C-def* *finite-subset* [*of* - ($\lambda(x,y). (x-a)/(b-y)$) ' ($A \times B$)]
by (*simp add: finite-image-set2*)
have $\neg F \subseteq C$
using *finC finite-subset infF* **by** *blast*
then obtain c **where** *cF*: $c \in F$ **and** *cnot*: $c \notin C$ **by** *blast*
define θ **where** $\theta = a + c * b$

have *unique*: $x = a \wedge y = b$
if xA : $x \in A$ **and** yB : $y \in B$ **and** *eq*: $\theta = x + c * y$ **for** $x \ y$
proof (*cases* $y = b$)
case *True*
then show *?thesis*
using *eq theta-def* **by** *force*

```

next
  case False
  then have den:  $b - y \neq 0$  by simp
  have  $c * (b - y) = x - a$ 
    using eq unfolding theta-def by algebra
  then have ceq:  $c = (x - a) / (b - y)$ 
    using den eq-divide-imp by blast
  then show ?thesis using cnot
    using C-def False xA yB by blast
qed

interpret Alg: Subfield algebraic-elements F
  by (rule subfield-algebraic-elements[OF sfF])
have cA:  $c \in \text{algebraic-elements } F$ 
  by (simp add: Subfield.algebraic-over-self algebraic-elements-def cF sfF)
then have alg-theta: algebraic-over F theta
  using alg-a alg-b algebraic-elements-def theta-def by blast
define E where  $E \equiv \text{eval-img } F \text{ theta}$ 
have sfE: Subfield E
  unfolding E-def by (rule Subfield.subfield-eval-img[OF sfF alg-theta])
have FE:  $F \subseteq E$ 
  unfolding E-def using Subfield.eval-img-base[OF sfF] by blast
have thetaE:  $\text{theta} \in E$ 
  unfolding E-def by (rule Subfield.eval-img-self[OF sfF])
have cE:  $c \in E$  using cF FE by blast
have alg-bE: algebraic-over E b
  by (rule algebraic-over-mono[OF alg-b FE])
define m where  $m \equiv \text{minpoly } E \text{ b}$ 
have m-min: is-minpoly E b m
  unfolding m-def by (rule Subfield.is-minpoly-minpoly[OF sfE alg-bE])
have mE:  $m \in \text{poly-over } E$  and mroot:  $\text{poly } m \text{ b} = 0$  and m0:  $m \neq 0$ 
  using m-min by (auto simp: is-minpoly-def)
have qE:  $q \in \text{poly-over } E$ 
  unfolding q-def
  using Subfield.minpoly-over[OF sfF alg-b] poly-over-mono[OF FE] by blast
have mq:  $m \text{ dvd } q$ 
  using Subfield.minpoly-dvd Subfield.minpoly-root alg-b m-min qE q-def sfE sfF by
blast

have every-root:  $r = b$  if mr:  $\text{poly } m \text{ r} = 0$  for r
proof -
  interpret Id: field-hom-on E  $\lambda x. x$  by (rule field-hom-on-id[OF sfE])
  have mapm:  $\text{map-poly } (\lambda x. x) \text{ m} = m$ 
    by (intro poly-eqI) (simp add: coeff-map-poly)
  then obtain h where h: field-hom-on (eval-img E b) h h b = r  $\forall x \in E. h \text{ x} = x$ 
    using mr Id.iso-extension[OF alg-bE m-min] by metis
  interpret H: field-hom-on eval-img E b h by (rule h(1))
  have thetaL:  $\text{theta} \in \text{eval-img } E \text{ b}$ 
    using Id.eval-img-base thetaE by auto
  have cL:  $c \in \text{eval-img } E \text{ b}$ 

```

```

    by (simp add: Id.eval-imag-base cE)
  have aeq:  $a = \text{theta} - c*b$  unfolding theta-def by algebra
  have bL:  $b \in \text{eval-imag } E$  by (rule Subfield.eval-imag-self[OF sfE])
  have aL:  $a \in \text{eval-imag } E$ 
    using aeq bL cL thetaL by blast
  have FL:  $F \subseteq \text{eval-imag } E$  using FE Id.eval-imag-base by blast
  have hfixF:  $\bigwedge x. x \in F \implies h\ x = x$  using h(3) FE by blast
  obtain pF:  $p \in \text{poly-over } F$  and pa:  $\text{poly } p\ a = 0$ 
    using Subfield.minpoly-over Subfield.minpoly-root alg-a p-def sfF by blast
  have proot-h:  $\text{poly } p\ (h\ a) = 0$ 
    using FL H.field-hom-on-axioms aL hfixF hom-preserves-roots pF pa sfF by blast
  have ha:  $\text{theta} - c*r = h\ a$ 
    by (simp add: H.hom-diff H.hom-mult H.mult-closed aeq bL cE cL h thetaE
thetaL)
  then have xA:  $\text{theta} - c*r \in A$ 
    using proot-h unfolding A-def by (metis mem-Collect-eq)
  obtain s where qs:  $q = m * s$  using mq by (elim dvdE)
  have qr:  $\text{poly } q\ r = 0$ 
    using mr qs by (metis mult-zero-left poly-mult)
  have rB:  $r \in B$ 
    using qr by (simp only: B-def mem-Collect-eq)
  have theta-eq:  $\text{theta} = (\text{theta} - c*r) + c*r$  by algebra
  then show  $r = b$ 
    using unique[OF xA rB theta-eq] by presburger
qed
have roots-m:  $\{r. \text{poly } m\ r = 0\} = \{b\}$ 
  by (auto simp: every-root mroot)
have card:  $\{r. \text{poly } m\ r = 0\} = \text{degree } m$ 
  using card-roots-eq-degree-alg-closed m0 mq q-def rsquarefree-dvd sep-b by blast
then have deg-m:  $\text{degree } m = 1$ 
  by (simp add: roots-m)
have ext-degree  $E\ b = 1$ 
  using deg-m by (simp add: ext-degree-def m-def)
then have bE:  $b \in E$ 
  using Subfield.ext-degree-eq-1-iff alg-bE sfE by blast
have diffE:  $\text{theta} - c*b \in E$ 
  by (simp add: Subfield.diff-closed Subfield.mult-closed bE cE sfE thetaE)
then have aE:  $a \in E$ 
  using theta-def by fastforce
have double-gen:  $\text{eval-imag } (\text{eval-imag } F\ a)\ b = \text{generate-field } (F \cup \{a, b\})$ 
proof -
  have sfFa: Subfield (eval-imag  $F\ a$ )
    by (rule Subfield.subfield-eval-imag[OF sfF alg-a])
  have alg-bFa: algebraic-over (eval-imag  $F\ a$ )  $b$ 
    by (metis alg-b sfF algebraic-over-mono primitive-elementI primitive-element-base)
  have eval-imag (eval-imag  $F\ a$ )  $b = \text{generate-field } (\text{generate-field } (F \cup \{a\}) \cup \{b\})$ 
    by (metis alg-a alg-bFa sfF sfFa Subfield.eval-imag-eq-generate-field)
  also have  $\dots = \text{generate-field } (F \cup \{a, b\})$ 
    by (metis Un-insert-left Un-insert-right generate-field-Un-collapse1 sup-bot.right-neutral)
  finally show ?thesis .

```

```

qed
have gen-subset-E: generate-field (F ∪ {a,b}) ⊆ E
  by (rule generate-field-least[OF sfE]) (use FE aE bE in auto)
have theta-double: theta ∈ generate-field (F ∪ {a,b})
  using cF theta-def by blast
have E-subset-gen: E ⊆ generate-field (F ∪ {a,b})
proof -
  have E = generate-field (F ∪ {theta})
    by (simp add: E-def Subfield.eval-img-eq-generate-field alg-theta sfF)
  also have ... ⊆ generate-field (F ∪ {a,b})
    using theta-double
    by (intro generate-field-least[OF subfield-generate-field]) auto
  finally show ?thesis .
qed
then show ?thesis unfolding double-gen E-def
  using E-def gen-subset-E by blast
qed

```

99.2 Finite separable extensions are simple

Over an infinite base, a finite algebraic generating set can be collapsed one generator at a time. Separability is asked of the containing extension, so it applies both to the next generator and to the primitive element already obtained for the preceding generators.

```

lemma finite-separable-generators-primitive:
  fixes K L A :: 'a :: alg-closed-field set
  assumes T: finite-subfield-tower K L
    and infK: infinite K and finA: finite A and AL: A ⊆ L
    and sep: separable-extension L K
  shows ∃ theta ∈ L. generate-field (K ∪ A) = eval-img K theta
proof -
  interpret T: finite-subfield-tower K L by (rule T)
  from finA AL show ?thesis
proof (induction A rule: finite-induct)
  case empty
  have eval-img K 0 = generate-field (K ∪ {0})
    by (simp add: T.base.eval-img-eq-generate-field T.base.algebraic-over-self)
  also have ... = K
    by (simp add: T.base.generate-field-self insert-absorb)
  finally have eval0: eval-img K 0 = K .
  show ?case using T.ext.zero-closed T.base.generate-field-self eval0 by auto
next
  case (insert a A)
  have aL: a ∈ L and AL: A ⊆ L using insert.prem by auto
  obtain theta where thetaL: theta ∈ L
    and genA: generate-field (K ∪ A) = eval-img K theta
    using insert.IH[OF AL] by blast
  have alg-a: algebraic-over K a by (rule T.finite-extension-algebraic[OF aL])
  have alg-theta: algebraic-over K theta

```

```

    by (rule T.finite-extension-algebraic[OF thetaL])
  have sep-a: rsquarefree (minpoly K a)
    by (rule separable-extensionD[OF sep aL alg-a])
  have sep-theta: rsquarefree (minpoly K theta)
    by (rule separable-extensionD[OF sep thetaL alg-theta])
  obtain z where pair: eval-img (eval-img K theta) a = eval-img K z
    using primitive-element-two-generators[OF T.base.Subfield-axioms infK alg-theta
alg-a
    sep-theta sep-a] by blast
  have sfKtheta: Subfield (eval-img K theta)
    by (rule T.base.subfield-eval-img[OF alg-theta])
  have alg-a-theta: algebraic-over (eval-img K theta) a
    using T.base.eval-img-base alg-a algebraic-over-mono genA generate-field-mono
sup-ge1 by blast
  have K ∪ insert a A = (K ∪ A) ∪ {a} by auto
  then have generate-field (K ∪ insert a A) = generate-field (generate-field (K ∪
A) ∪ {a})
    by (metis generate-field-Un-collapse1)
  also have ... = eval-img (eval-img K theta) a
    by (simp add: Subfield.eval-img-eq-generate-field alg-a-theta genA sfKtheta)
  finally have generated-insert:
    generate-field (K ∪ insert a A) = eval-img (eval-img K theta) a .
  have generated-subset-L: generate-field (K ∪ insert a A) ⊆ L
    using generate-field-least[OF T.ext.Subfield-axioms] T.base-subset aL AL by
simp
  have z ∈ eval-img K z by (rule T.base.eval-img-self)
  then show ?case
    using pair generated-insert generated-subset-L pair by blast
qed
qed

```

Finite fields have a particularly clean primitive-element proof. The multiplicative group of the target field is cyclic, so a generator of its nonzero elements already generates the field as a simple extension over every subfield. This is the finite-field instance of the primitive element theorem and is the useful bridge for the concrete finite-field theories.

```

theorem finite-field-extension-is-simple:
  assumes KL: subfield-tower K L and finL: finite L
  shows ∃ a ∈ L. primitive-element K L a
proof -
  interpret KL: subfield-tower K L by (rule KL)
  interpret Lf: Field L (+) (*) 0 1
    by (rule Subfield.sf-field[OF KL.ext.Subfield-axioms])
  interpret G: Group Lf.Fstar (*) 1 by (rule Lf.Group-Fstar)
  have finstar: finite Lf.Fstar
    by (simp add: Lf.Fstar-def finL)
  obtain g where gstar: g ∈ Lf.Fstar
    and cyc: G.cyclic-subgroup g = Lf.Fstar
    using Lf.finite-field-mult-cyclic[OF finstar] by blast

```

```

have gL:  $g \in L$  using gstar by (simp add: Lf.Fstar-def)
have gpow:  $G.\text{power } g \ n \in \text{eval-img } K \ g$  for  $n$ 
  by (induction n) (auto simp: KL.base.eval-img-1 KL.base.eval-img-mult KL.base.eval-img-self)
have L = eval-img K g
proof (intro antisym subsetI)
  fix x assume xL:  $x \in L$ 
  show  $x \in \text{eval-img } K \ g$ 
  proof (cases  $x = 0$ )
    case True
    then show ?thesis by (simp add: Subfield.eval-img-0[OF KL.base.Subfield-axioms])
  next
    case False
    then obtain n where  $x = G.\text{power } g \ n$ 
      using G.cyclic-subgroup-eq-range[OF finstar gstar] Lf.Fstar-iff cyc xL by blast
    then show ?thesis using gpow by simp
  qed
next
  fix x assume xE:  $x \in \text{eval-img } K \ g$ 
  obtain p where pK:  $p \in \text{poly-over } K$  and  $x: x = \text{poly } p \ g$ 
    using xE by (rule eval-imgE)
  have xG:  $x \in \text{generate-field } (K \cup \{g\})$ 
    unfolding x
    by (rule Subfield.eval-img-subset-generate-field[OF KL.base.Subfield-axioms pK])
  have KG:  $K \cup \{g\} \subseteq L$ 
    using KL.base-subset gL by blast
  show  $x \in L$  using generate-field-least[OF KL.ext.Subfield-axioms KG] xG by blast
  qed
then show ?thesis using gL primitive-elementI by blast
qed

```

The full primitive-element theorem now follows from a basis. If the base is finite, the finite coordinate space makes the target field finite and the multiplicative-cyclic proof applies. If the base is infinite, the preceding finite-generator theorem collapses the basis to one element.

```

theorem finite-separable-extension-is-simple:
  fixes K L :: 'a :: alg-closed-field set
  assumes T: finite-subfield-tower K L and sep: separable-extension L K
  shows  $\exists a \in L. \text{primitive-element } K \ L \ a$ 
proof -
  interpret T: finite-subfield-tower K L by (rule T)
  show ?thesis
  proof (cases finite K)
    case True
    then show ?thesis
      by (simp add: T.finite-carrier-of-finite-base T.subfield-tower-axioms
        finite-field-extension-is-simple)
  next
    case False
    obtain B where basis: T.vs.basis B and finB: finite B and BL:  $B \subseteq L$ 

```

```

    using T.finite-basis by (auto simp: T.vs.basis-def)
  have L = generate-field (K ∪ B)
    using T.generate-field-basis basis by auto
  then show ?thesis using primitive-elementI
    using finite-separable-generators-primitive[OF T False finB BL sep] by blast
qed
qed

lemma (in finite-subfield-tower) primitive-element-degree:
  assumes primitive-element K L a
  shows extension-degree = ext-degree K a
proof -
  have aL: a ∈ L
    using primitive-element-mem[OF assms base.Subfield-axioms] .
  with assms show ?thesis
    by (simp add: extension-degree-eq-ext-degree finite-extension-algebraic primitive-elementD)
qed

end

```

100 The order of the Galois group of a finite normal separable extension

```

theory Galois-Degree
  imports Galois-Action Extension-Properties Primitive-Element
begin

```

For a *simple normal* extension $K = F(a)$ of complex fields (all conjugates of a — the roots of its minimal polynomial — already lying in K), the order of the Galois group equals the number of those conjugates, and, in characteristic 0 where the minimal polynomial is separable, this equals *ext-degree* F a . The argument is the classical bijection $\lambda\sigma. \sigma a$ between *field-auto* K F and the conjugates of a : it is injective (an F -automorphism of $F a$ is determined by where it sends a), lands among the conjugates (*field-auto-maps-root*), and is onto (the Galois action is transitive on the roots of an irreducible polynomial). The simple-extension calculation is the reusable core. The final theorem combines it with the primitive-element theorem and the basis definition of extension degree to obtain $|Gal(K/F)| = [K:F]$ for every finite normal separable extension in an algebraically closed ambient field.

100.1 The minimal polynomial is irreducible over the base

A minimal polynomial is irreducible over F : it is nonzero of positive degree and, by minimality, admits no factorisation over F into two positive-degree factors.

```

lemma irreducible-over-minpoly:
  assumes sfF: Subfield F and alg: algebraic-over F a

```

shows *irreducible-over* F (*minpoly* F a)
proof –
have *minpoly* F $a \neq 0$ **by** (rule *Subfield.minpoly-nonzero*[*OF sfF alg*])
moreover **have** *degree* (*minpoly* F a) > 0
using *Subfield.is-minpoly-minpoly* *Subfield.minpoly-degree-pos* *alg sfF* **by** *blast*
ultimately show *?thesis*
unfolding *irreducible-over-def*
using *alg sfF* **by** (*metis Subfield.is-minpoly-minpoly Subfield.minpoly-no-proper-factor*)
qed

100.2 An F -automorphism of $F(a)$ is determined by its value at a

Two F -automorphisms of the simple extension *eval-img* F a that agree at a agree everywhere. Every element of *eval-img* F a is *poly* p a for some $p \in \text{poly-over } F$, and an F -automorphism commutes with such evaluation (*field-hom-on.poly-map-poly-hom*, together with *map-poly-fix* since the coefficients lie in F); so the common value at a fixes the value everywhere.

lemma *field-auto-determined-by-gen*:

assumes *sfF*: *Subfield* F **and** *alg*: *algebraic-over* F a
and s : $\sigma \in \text{field-auto } (\text{eval-img } F \ a) \ F$ **and** t : $\tau \in \text{field-auto } (\text{eval-img } F \ a) \ F$
and *eq*: $\sigma \ a = \tau \ a$
shows $\sigma = \tau$
proof –
define E **where** $E \equiv \text{eval-img } F \ a$
have *sfE*: *Subfield* E **unfolding** *E-def* **by** (rule *Subfield.subfield-eval-img*[*OF sfF alg*])
have *FE*: $F \subseteq E$ **unfolding** *E-def* **using** *Subfield.eval-img-base*[*OF sfF*] **by** *blast*
have *aE*: $a \in E$ **unfolding** *E-def* **by** (rule *Subfield.eval-img-self*[*OF sfF*])
have *homs*: *field-hom-on* E σ
using *E-def field-auto-imp-field-hom-on s sfE* **by** *blast*
have *homt*: *field-hom-on* E τ
using *E-def field-auto-imp-field-hom-on sfE t* **by** *blast*
have *fixFs*: $\bigwedge x. x \in F \implies \sigma \ x = x$ **using** s **by** (*auto simp: field-auto-mem-iff*)
have *fixFt*: $\bigwedge x. x \in F \implies \tau \ x = x$ **using** t **by** (*auto simp: field-auto-mem-iff*)
— On the shared carrier E the two maps agree, elementwise.
have *agree*: $\sigma \ x = \tau \ x$ **if** $x \in E$ **for** x
proof –
obtain p **where** $p \in \text{poly-over } F$ **and** $x \equiv \text{poly } p \ a$ **and** pE : $p \in \text{poly-over } E$
using *xE poly-over-mono*[*OF FE*] **by** (*auto simp: E-def eval-img-def*)
— σ commutes with evaluation, and fixes the (F -)coefficients.
have $\sigma \ x = \text{poly } (\text{map-poly } \sigma \ p) \ (\sigma \ a)$
using *field-hom-on.poly-map-poly-hom*[*OF homs pE aE*] **by** (*simp add: xp*)
also have $\dots = \text{poly } (\text{map-poly } \tau \ p) \ (\tau \ a)$
using *eq fixFs fixFt homs homt p sfF* **by** (*metis map-poly-fix*)
also have $\dots = \tau \ x$
using *field-hom-on.poly-map-poly-hom*[*OF homt pE aE*] **by** (*simp add: xp*)

finally show *?thesis* .
qed
 — Both maps are extensional on E , so agreeing on E makes them equal.
then show *?thesis*
 using s t **unfolding** E -def $field$ -auto-mem-iff **by** ($meson$ PiE -ext)
qed

100.3 The order of the Galois group of a simple normal extension

Normality of the simple extension: all conjugates of a lie in $eval\text{-}img\ F\ a$. Under this hypothesis, $\lambda\sigma$. $\sigma\ a$ is a bijection from $field\text{-}auto\ (eval\text{-}img\ F\ a)\ F$ to the conjugate set, so the group order equals the number of conjugates.

theorem *galois-simple-normal-card*:

fixes $F :: complex\ set$ **and** $a :: complex$
assumes sfF : *Subfield* F **and** alg : *algebraic-over* $F\ a$
 and $normal$: $\{r. poly\ (minpoly\ F\ a)\ r = 0\} \subseteq eval\text{-}img\ F\ a$
shows $card\ (field\text{-}auto\ (eval\text{-}img\ F\ a)\ F) = card\ \{r. poly\ (minpoly\ F\ a)\ r = 0\}$
proof —
define m **where** $m \equiv minpoly\ F\ a$
define E **where** $E \equiv eval\text{-}img\ F\ a$
define R **where** $R \equiv \{r. poly\ m\ r = 0\}$
have sfE : *Subfield* E **unfolding** E -def **by** ($rule\ Subfield.subfield\text{-}eval\text{-}img[OF\ sfF\ alg]$)
have mF : $m \in poly\text{-}over\ F$ **unfolding** m -def **by** ($rule\ Subfield.minpoly\text{-}over[OF\ sfF\ alg]$)
have mnz : $m \neq 0$ **unfolding** m -def **by** ($rule\ Subfield.minpoly\text{-}nonzero[OF\ sfF\ alg]$)
have $mroot$: $poly\ m\ a = 0$ **unfolding** m -def **by** ($rule\ Subfield.minpoly\text{-}root[OF\ sfF\ alg]$)
have irr : *irreducible-over* $F\ m$ **unfolding** m -def **by** ($rule\ irreducible\text{-}over\text{-}minpoly[OF\ sfF\ alg]$)
have aR : $a \in R$ **using** $mroot$ **by** ($simp\ add$: R -def)
have aE : $a \in E$ **unfolding** E -def **by** ($rule\ Subfield.eval\text{-}img\text{-}self[OF\ sfF]$)
have FE : $F \subseteq E$ **unfolding** E -def **using** $Subfield.eval\text{-}img\text{-}base[OF\ sfF]$ **by** *blast*
have RE : $R \subseteq E$ **using** $normal$ **by** ($simp\ add$: R -def m -def E -def)
have $finR$: *finite* R **unfolding** R -def **using** mnz **by** ($rule\ poly\text{-}roots\text{-}finite$)
 — Normality makes E the splitting field $generate\text{-}field\ (F \cup R)$.
have E -gen: $E = generate\text{-}field\ (F \cup R)$
proof ($rule\ antisym$)
have $F \cup \{a\} \subseteq F \cup R$ **using** aR **by** *blast*
then show $E \subseteq generate\text{-}field\ (F \cup R)$
 unfolding E -def
using $Subfield.eval\text{-}img\text{-}eq\text{-}generate\text{-}field[OF\ sfF\ alg]$ **by** ($metis\ generate\text{-}field\text{-}mono$)
show $generate\text{-}field\ (F \cup R) \subseteq E$
 using $FE\ RE$ **by** ($intro\ generate\text{-}field\text{-}least[OF\ sfE]$) *auto*
qed
 — Fundamental theorem of algebra: every positive-degree complex polynomial has a root.
have $closed$: $\exists b. poly\ q\ b = 0$ **if** $0 < degree\ q$ **for** $q :: complex\ poly$

```

    using that by (intro fundamental-theorem-of-algebra) (simp add: constant-degree)
  — The bijection  $\lambda\sigma. \sigma a$  from the Galois group to the conjugate set.
  have bij-betw ( $\lambda\sigma. \sigma a$ ) (field-auto  $E F$ )  $R$ 
    unfolding bij-betw-def
  proof
    show inj: inj-on ( $\lambda\sigma. \sigma a$ ) (field-auto  $E F$ )
      unfolding inj-on-def
      using field-auto-determined-by-gen[OF sfF alg] by (simp add: E-def)
    show ( $\lambda\sigma. \sigma a$ ) ' field-auto  $E F = R$ 
      proof (rule antisym)
        — Well-defined: an automorphism sends  $a$  to a conjugate.
        show ( $\lambda\sigma. \sigma a$ ) ' field-auto  $E F \subseteq R$ 
          using FE R-def aE field-auto-maps-root mF mroot sfE sfF by blast
        — Onto: transitivity of the Galois action on the roots of the irreducible  $m$ .
        show  $R \subseteq (\lambda\sigma. \sigma a)$  ' field-auto  $E F$ 
          using galois-action-transitive[OF sfF finR mF irr - closed aR] E-gen aR
          by (force simp add: R-def)
      qed
    qed
  qed
  then show ?thesis
    using E-def R-def bij-betw-same-card m-def by blast
  qed

```

100.4 Separability in characteristic 0: the number of conjugates is the degree

poly-over is closed under formal differentiation: the coefficients of *pderiv* p are integer multiples of coefficients of p (*coeff-pderiv*), and a subfield contains all *of-nat* n .

```

lemma (in Subfield) poly-over-pderiv:
  assumes  $p \in \text{poly-over } K$ 
  shows  $pderiv\ p \in \text{poly-over } K$ 
    using poly-overI assms mult-closed of-nat-closed poly-over-coeff by (metis coeff-pderiv)

```

In characteristic 0 the minimal polynomial is *separable* (squarefree): it shares no root with its derivative. As *pderiv* m has smaller degree than the irreducible m , it is not a multiple of m , so Bézout over F gives $u * m + v * pderiv\ m = 1$; evaluating this *polynomial identity* at any root r of m yields $\text{poly } v\ r * \text{poly } (pderiv\ m)\ r = 1$, so $\text{poly } (pderiv\ m)\ r \neq 0$.

```

lemma rsquarefree-minpoly:
  fixes  $F :: 'a :: \text{field-char-0 set}$  and  $a :: 'a$ 
  assumes sfF: Subfield  $F$  and alg: algebraic-over  $F\ a$ 
  shows rsquarefree (minpoly  $F\ a$ )
  proof —
    define  $m$  where  $m \equiv \text{minpoly } F\ a$ 
    have ism: is-minpoly  $F\ a\ m$  unfolding m-def by (rule Subfield.is-minpoly-minpoly[OF sfF alg])
  
```

```

have mdeg: degree m > 0 unfolding m-def
  by (rule Subfield.minpoly-degree-pos[OF sfF Subfield.is-minpoly-minpoly[OF sfF
alg]])
have pmF: pderiv m ∈ poly-over F
  using Subfield.poly-over-pderiv is-minpoly-def ism sfF by blast
— pderiv m is nonzero (positive degree, char 0) of degree below m.
have pmnz: pderiv m ≠ 0 using mdeg by (simp add: pderiv-eq-0-iff)
have ndvd: ¬ m dvd pderiv m
  by (simp add: mdeg)
— Bézout over F: the coefficient identity holds as polynomials.
obtain u v where u: u ∈ poly-over F and v: v ∈ poly-over F
  and bez: u * m + v * pderiv m = 1
  using Subfield.minpoly-bezout[OF sfF ism pmF ndvd] by blast
— No root of m is a root of pderiv m, so m is squarefree.
have ¬ (poly m r = 0 ∧ poly (pderiv m) r = 0) for r
  using bez pmnz by (metis mult-cancel-left1 poly-squarefree-decomp rsquarefree-roots)
then show ?thesis
  using m-def rsquarefree-roots by blast
qed

```

Hence, over *complex*, a separable minimal polynomial that splits has exactly *degree* (*minpoly* *F* *a*) distinct roots: it decomposes as a product of the distinct linear factors $[-z, 1:]$ (*complex-poly-decompose-rsquarefree*), and the degree of that product is the number of factors.

```

lemma card-conjugates-eq-degree:
  fixes F :: complex set and a :: complex
  assumes sfF: Subfield F and alg: algebraic-over F a
  shows card {r. poly (minpoly F a) r = 0} = ext-degree F a
proof —
  define m where m ≡ minpoly F a
  have mnz: m ≠ 0
    by (simp add: Subfield.minpoly-nonzero alg m-def sfF)
  have monic: lead-coeff m = 1
    by (simp add: Subfield.minpoly-monic alg m-def sfF)
  have rsquarefree m
    by (simp add: alg m-def rsquarefree-minpoly sfF)
  — Squarefree decomposition into distinct monic linear factors.
  then have decomp: smult (lead-coeff m) (∏ z | poly m z = 0. [-z, 1:]) = m
    by (intro complex-poly-decompose-rsquarefree)
  have degree m = degree (∏ z | poly m z = 0. [-z, 1:])
    using decomp monic by (metis smult-1-left)
  also have ... = card {z. poly m z = 0}
    using mnz by (simp add: poly-roots-finite degree-prod-eq-sum-degree)
  finally show ?thesis by (simp add: ext-degree-def m-def)
qed

```

100.5 The headline: order of the Galois group of a simple normal extension

The order of the Galois group of a simple normal extension equals the extension degree. Combining the bijection *galois-simple-normal-card* with separability *card-conjugates-eq-degree*: for a simple normal complex extension $F a$,

$$|\text{field_auto } (F(a)) \ F| = \text{ext_degree } F \ a = \deg(\text{minpoly } F \ a).$$

This is $|\text{Gal}(K/F)| = [K:F]$ in the simple normal (separable) case.

theorem *galois-simple-normal-degree*:

fixes $F :: \text{complex set}$ **and** $a :: \text{complex}$
assumes sfF : Subfield F **and** alg : algebraic-over $F \ a$
and normal : $\{r. \text{poly } (\text{minpoly } F \ a) \ r = 0\} \subseteq \text{eval-img } F \ a$
shows $\text{card } (\text{field-auto } (\text{eval-img } F \ a) \ F) = \text{ext-degree } F \ a$
using *galois-simple-normal-card*[$OF \ \text{sfF} \ \text{alg} \ \text{normal}$] *card-conjugates-eq-degree*[$OF \ \text{sfF} \ \text{alg}$]
by *simp*

100.6 Generic finite normal and separable extensions

Normality of a simple extension identifies its automorphisms with the roots of the minimal polynomial. The proof is the generic root-action argument: an automorphism is determined by the generator, maps the generator to a root, and transitivity supplies an automorphism for each root.

theorem *galois-simple-normal-card-alg-closed*:

fixes $F :: 'a :: \text{alg-closed-field set}$ **and** $a :: 'a$
assumes sfF : Subfield F **and** alg : algebraic-over $F \ a$
and normal : $\{r. \text{poly } (\text{minpoly } F \ a) \ r = 0\} \subseteq \text{eval-img } F \ a$
shows $\text{card } (\text{field-auto } (\text{eval-img } F \ a) \ F) = \text{card } \{r. \text{poly } (\text{minpoly } F \ a) \ r = 0\}$
proof –
define m **where** $m \equiv \text{minpoly } F \ a$
define E **where** $E \equiv \text{eval-img } F \ a$
define R **where** $R \equiv \{r. \text{poly } m \ r = 0\}$
have sfE : Subfield E
unfolding $E\text{-def}$ **by** (rule *Subfield.subfield-eval-img*[$OF \ \text{sfF} \ \text{alg}$])
have mF : $m \in \text{poly-over } F$
unfolding $m\text{-def}$ **by** (rule *Subfield.minpoly-over*[$OF \ \text{sfF} \ \text{alg}$])
have $m\text{root}$: $\text{poly } m \ a = 0$
unfolding $m\text{-def}$ **by** (rule *Subfield.minpoly-root*[$OF \ \text{sfF} \ \text{alg}$])
have irr : irreducible-over $F \ m$
unfolding $m\text{-def}$ **by** (rule *irreducible-over-minpoly*[$OF \ \text{sfF} \ \text{alg}$])
have aE : $a \in E$
unfolding $E\text{-def}$ **by** (rule *Subfield.eval-img-self*[$OF \ \text{sfF}$])
have FE : $F \subseteq E$
unfolding $E\text{-def}$ **using** *Subfield.eval-img-base*[$OF \ \text{sfF}$] **by** *blast*
have RE : $R \subseteq E$
using normal **by** (*simp add*: $R\text{-def } m\text{-def } E\text{-def}$)

```

have finR: finite R
  using R-def irr irreducible-over-def poly-roots-finite by blast
have E-gen: E = generate-field (F ∪ R)
proof (rule antisym)
  show E ⊆ generate-field (F ∪ R)
    using Subfield.eval-img-eq-generate-field[OF sfF alg] mroot
    by (simp add: E-def R-def generate-field-mono)
  show generate-field (F ∪ R) ⊆ E
    using FE RE by (intro generate-field-least[OF sfE]) auto
qed
have action-bij: bij-betw (λσ. σ a) (field-auto E F) R
  unfolding bij-betw-def
proof
  show inj-on (λσ. σ a) (field-auto E F)
    unfolding inj-on-def
    using field-auto-determined-by-gen[OF sfF alg] by (simp add: E-def)
  show (λσ. σ a) ' field-auto E F = R
proof
  show (λσ. σ a) ' field-auto E F ⊆ R
    using FE R-def aE field-auto-maps-root mF mroot sfE sfF by blast
  show R ⊆ (λσ. σ a) ' field-auto E F
    using E-gen mroot galois-action-transitive[OF sfF finR mF irr - alg-closed-imp-poly-has-root]

    by (force simp: R-def)
qed
qed
then show ?thesis
  using E-def R-def bij-betw-same-card m-def by blast
qed

```

The simple normal argument is now available over every algebraically closed field. The normality hypothesis says that all roots of the minimal polynomial lie in the simple extension; separability supplies squarefreeness, and the generic root action supplies the matching automorphism count.

```

theorem galois-simple-normal-separable-degree:
  fixes F :: 'a :: alg-closed-field set and a :: 'a
  assumes sfF: Subfield F and alg: algebraic-over F a
    and normal: {r. poly (minpoly F a) r = 0} ⊆ eval-img F a
    and sep: separable-extension (eval-img F a) F
  shows card (field-auto (eval-img F a) F) = ext-degree F a
proof -
  have card-group-roots:
    card (field-auto (eval-img F a) F) = card {r. poly (minpoly F a) r = 0}
  by (rule galois-simple-normal-card-alg-closed[OF sfF alg normal])
  have sep-minpoly: rsquarefree (minpoly F a)
    using Subfield.eval-img-self alg sep separable-extension-def sfF by blast
  have minpoly0: minpoly F a ≠ 0
    by (rule Subfield.minpoly-nonzero[OF sfF alg])
  then have card-roots-degree:

```

```

      card {r. poly (minpoly F a) r = 0} = ext-degree F a
    by (metis sep-minpoly card-roots-eq-degree-alg-closed ext-degree-def)
  show ?thesis using card-group-roots card-roots-degree by simp
qed

```

A finite normal separable extension is simple by the primitive-element theorem. Normality then puts every conjugate of the primitive element back in the target field, so the preceding simple theorem computes the whole automorphism group. The basis theorem identifies the simple degree with the intrinsic extension degree.

context

```

  fixes F K :: 'a :: alg-closed-field set
  assumes finite: finite-subfield-tower F K
begin

```

interpretation T: finite-subfield-tower F K **by** (rule finite)

theorem finite-normal-separable-field-auto:

```

  assumes normal: normal-extension K F and sep: separable-extension K F
  shows finite (field-auto K F)

```

proof –

```

  obtain a where aK: a ∈ K and prim: primitive-element F K a
  using finite-separable-extension-is-simple[OF finite sep] by blast
  have Kdef: K = eval-img F a by (rule primitive-elementD[OF prim])
  have alg: algebraic-over F a by (rule T.finite-extension-algebraic[OF aK])
  define p where p ≡ minpoly F a
  define R where R ≡ {r. poly p r = 0}
  have pF: p ∈ poly-over F
    unfolding p-def by (rule T.base.minpoly-over[OF alg])
  have p0: p ≠ 0
    unfolding p-def by (rule T.base.minpoly-nonzero[OF alg])
  have aR: a ∈ R
    unfolding R-def p-def using T.base.minpoly-root[OF alg] by simp
  have finR: finite R unfolding R-def using p0 by (rule poly-roots-finite)
  have roots-min: poly-root-set (minpoly F a) ⊆ K
    by (rule normal-extensionD[OF normal T.base.minpoly-over[OF alg]
      irreducible-over-minpoly[OF T.base.Subfield-axioms alg]
      aK T.base.minpoly-root[OF alg]])
  have rootsK: R ⊆ K
    using roots-min unfolding R-def p-def poly-root-set-def by simp
  have Kgen: K = generate-field (F ∪ R)
  proof (rule subset-antisym)
    have K = generate-field (F ∪ {a})
      using Kdef T.base.eval-img-eq-generate-field[OF alg] by simp
    also have ... ⊆ generate-field (F ∪ R)
      by (rule generate-field-mono) (use aR in auto)
    finally show K ⊆ generate-field (F ∪ R) .
  show generate-field (F ∪ R) ⊆ K
    by (simp add: T.base-subset T.ext.Subfield-axioms generate-field-least rootsK)

```

```

qed
show ?thesis
  using finite-galois-group[OF T.ext.Subfield-axioms T.base.Subfield-axioms T.base-subset
pF]
  using Kgen R-def finR by blast
qed

theorem finite-normal-separable-galois-degree:
  assumes normal: normal-extension K F and sep: separable-extension K F
  shows card (field-auto K F) = T.extension-degree
proof -
  obtain a where aK: a ∈ K and prim: primitive-element F K a
  using finite-separable-extension-is-simple[OF finite sep] by blast
  have Kdef: K = eval-img F a by (rule primitive-elementD[OF prim])
  have alg: algebraic-over F a by (rule T.finite-extension-algebraic[OF aK])
  have irr: irreducible-over F (minpoly F a)
    by (rule irreducible-over-minpoly[OF T.base.Subfield-axioms alg])
  have poly (minpoly F a) a = 0
    by (rule T.base.minpoly-root[OF alg])
  then have roots-simple: {r. poly (minpoly F a) r = 0} ⊆ eval-img F a
  using Kdef T.base.minpoly-over aK alg irr normal normal-extensionD poly-root-set-def
  by blast
  have card-simple: card (field-auto (eval-img F a) F) = ext-degree F a
  using Kdef T.base.Subfield-axioms alg galois-simple-normal-separable-degree roots-simple
  sep
  by blast
  show ?thesis
    using Kdef T.primitive-element-degree card-simple prim by presburger
qed

end

end

```

101 Field-side inverse and degree formulas for simple normal extensions

```

theory Galois-Simple-Degree
  imports Galois-Fixed-Field Algebraic-Transitivity Galois-Degree
begin

```

101.1 Algebraicity of generated splitting fields

The generic algebraicity closure for generated fields lives in *Algebraic-Transitivity*. This complex-specific wrapper applies it to the root set of a splitting-field presentation.

Complex splitting fields are algebraic over their base. Their generators are the

root set of the defining nonzero polynomial, and the preceding closure result applies to that set directly (in fact, it does not need the root set's finiteness).

lemma *splitting-field-algebraic-extension*:
fixes $F K :: \text{complex set}$ **and** $q :: \text{complex poly}$
assumes *split*: *splitting-field* $F q K$ **and** *sfF*: *complex-subfield* F
shows *algebraic-extension* $K F$
proof (*rule algebraic-extensionI*)
fix x **assume** xK : $x \in K$
have *sfF'*: *Subfield* F
using *sfF* **by** (*simp add: complex-subfield-iff-subfield*)
have *algR*: $\bigwedge r. r \in \text{poly-root-set } q \implies \text{algebraic-over } F r$
by (*rule splitting-field-root-algebraic[OF split]*)
have *algK*: $\bigwedge x. x \in \text{generate-field } (F \cup \text{poly-root-set } q) \implies \text{algebraic-over } F x$
by (*rule generated-algebraic[OF sfF' algR]*)
have *Kdef*: $K = \text{generate-field } (F \cup \text{poly-root-set } q)$
using *splitting-fieldD[OF split]* *gen-subfield-eq-generate-field* **by** *simp*
show *algebraic-over* $F x$
using *algK xK* **by** (*simp add: Kdef*)
qed

theorem *splitting-field-normal-separable*:
fixes $F K :: \text{complex set}$ **and** $q :: \text{complex poly}$
assumes *split*: *splitting-field* $F q K$ **and** *sfF*: *complex-subfield* F
shows *normal-extension* $K F \wedge \text{separable-extension } K F$
proof (*intro conjI*)
show *normal-extension* $K F$
by (*rule splitting-field-normal[OF split sfF]*)
show *separable-extension* $K F$
by (*rule complex-extension-separable[OF sfF]*)
qed

With the finite-generation bridge in place, the closure theorem no longer needs an explicit algebraicity premise for an intermediate field of a complex splitting field.

theorem *fixed-field-eq-of-complex-splitting-field*:
fixes $F E K :: \text{complex set}$ **and** $q :: \text{complex poly}$
assumes *split*: *splitting-field* $F q K$
and *sfF*: *complex-subfield* F
and E : $E \in \text{inter-fields } K F$
shows *fixed-field* K (*field-auto* $K E$) = E
proof –
have *FE*: $F \subseteq E$
using E **by** (*simp add: inter-fields-iff*)
have *algKF*: *algebraic-extension* $K F$
by (*rule splitting-field-algebraic-extension[OF split sfF]*)
have *algKE*: *algebraic-extension* $K E$
proof (*rule algebraic-extensionI*)
fix x **assume** xK : $x \in K$
have *algx*: *algebraic-over* $F x$


```

    by (rule algebraic-extensionD[OF algKF xK])
  show algebraic-over E x
    by (rule algebraic-over-mono[OF algx FE])
qed
show ?thesis
  by (rule fixed-field-eq-of-splitting-field[OF split E algKE])
qed

```

The simple normal case is the degree-bearing part of the field-side correspondence. The existing Galois-degree theorem counts the automorphisms; this theory connects its hypotheses to the splitting-field presentation used by the fixed-field closure theorem.

```

lemma splitting-field-finite-galois-group:
  fixes F K :: complex set and q :: complex poly
  assumes split: splitting-field F q K and sfF: complex-subfield F
  shows finite (field-auto K F)
proof -
  have sfF': Subfield F
    using sfF by (simp add: complex-subfield-iff-subfield)
  have sfK': Subfield K
    using splitting-field-subfield[OF split]
    by (simp add: complex-subfield-iff-subfield)
  have FK: F  $\subseteq$  K
    by (rule splitting-field-base-subset[OF split])
  have qF: q  $\in$  poly-over F
    using split by (auto simp: splitting-fieldD)
  have pnz: q  $\neq$  0
    using split by (auto simp: splitting-fieldD)
  have Xdef: poly-root-set q = {r. poly q r = 0}
    by (simp add: poly-root-set-def)
  have XK: poly-root-set q  $\subseteq$  K
    by (rule splitting-field-roots-subset[OF split])
  have finX: finite (poly-root-set q)
    by (rule finite-poly-root-set[OF pnz])
  have Kgen: K = gen-subfield (F  $\cup$  poly-root-set q)
    using split by (auto simp: splitting-fieldD gen-subfield-eq-generate-field)
  show finite (field-auto K F)
    by (rule finite-field-auto[OF sfK' sfF' FK qF Xdef XK finX Kgen])
qed

```

```

lemma simple-normal-splitting-field:
  fixes F :: complex set and a :: complex
  assumes sfF: Subfield F and alg: algebraic-over F a
  and normal: {r. poly (minpoly F a) r = 0}  $\subseteq$  eval-img F a
  shows splitting-field F (minpoly F a) (eval-img F a)
proof -
  define R where R = {r. poly (minpoly F a) r = 0}
  have sfE: Subfield (eval-img F a)
    by (rule Subfield.subfield-eval-img[OF sfF alg])

```

```

have FE:  $F \subseteq \text{eval-img } F \ a$ 
  using Subfield.eval-img-base[OF sfF] by blast
have pF:  $\text{minpoly } F \ a \in \text{poly-over } F$ 
  by (rule Subfield.minpoly-over[OF sfF alg])
have pnz:  $\text{minpoly } F \ a \neq 0$ 
  by (rule Subfield.minpoly-nonzero[OF sfF alg])
have mroot:  $\text{poly } (\text{minpoly } F \ a) \ a = 0$ 
  by (rule Subfield.minpoly-root[OF sfF alg])
have aR:  $a \in R$ 
  using mroot by (simp add: R-def)
have RE:  $R \subseteq \text{eval-img } F \ a$ 
  using normal by (simp add: R-def)
have Egen:  $\text{eval-img } F \ a = \text{generate-field } (F \cup R)$ 
proof (rule antisym)
  have  $F \cup \{a\} \subseteq F \cup R$  using aR by blast
  then have  $\text{generate-field } (F \cup \{a\}) \subseteq \text{generate-field } (F \cup R)$ 
    by (rule generate-field-mono)
  then show  $\text{eval-img } F \ a \subseteq \text{generate-field } (F \cup R)$ 
    using Subfield.eval-img-eq-generate-field[OF sfF alg] by simp
  show  $\text{generate-field } (F \cup R) \subseteq \text{eval-img } F \ a$ 
    using FE RE by (intro generate-field-least[OF sfE]) auto
qed
show ?thesis
  unfolding splitting-field-def
  using pF pnz Egen gen-subfield-eq-generate-field
  by (simp add: R-def poly-root-set-def)
qed

```

For a simple normal complex extension, the field-side inverse law and the exact order formula can be consumed together. The subgroup-side inverse law is deliberately not reproved here; it is the separate Artin theorem.

```

theorem simple-normal-field-inverse-and-degree:
  fixes F :: complex set and a :: complex
  assumes sfF: complex-subfield F and alg: algebraic-over F a
    and normal:  $\{r. \text{poly } (\text{minpoly } F \ a) \ r = 0\} \subseteq \text{eval-img } F \ a$ 
  shows fixed-field (eval-img F a) (field-auto (eval-img F a) F) = F  $\wedge$ 
    card (field-auto (eval-img F a) F) = ext-degree F a
proof -
  have sfF': Subfield F
    using sfF by (simp add: complex-subfield-iff-subfield)
  have split: splitting-field F (minpoly F a) (eval-img F a)
    by (rule simple-normal-splitting-field[OF sfF' alg normal])
  have FE:  $F \subseteq \text{eval-img } F \ a$ 
    using Subfield.eval-img-base[OF sfF'] by blast
  have Eint:  $F \in \text{inter-fields } (\text{eval-img } F \ a) \ F$ 
    by (rule base-in-inter-fields[OF sfF' FE])
  have fixed:
    fixed-field (eval-img F a) (field-auto (eval-img F a) F) = F
    by (rule fixed-field-eq-of-complex-splitting-field[OF split sfF Eint])

```

```

have degree:
  card (field-auto (eval-img F a) F) = ext-degree F a
  by (rule galois-simple-normal-degree[OF sfF' alg normal])
show ?thesis using fixed degree by blast
qed

end

```

102 Generic restriction of field automorphisms

```

theory Galois-Restriction
  imports Galois-Automorphism Solvable-Transfer Galois-Action
begin

```

For a base subset $F \subseteq E \subseteq K$, where E and K are subfields, stability of the intermediate carrier under the base-field automorphisms makes restriction a homomorphism between the two corresponding automorphism groups. This theory contains the reusable carrier-set bookkeeping; splitting-field and polynomial-extension arguments remain in the theories that consume it.

```

definition galois-restriction ::
  'a :: field set  $\Rightarrow$  'a set  $\Rightarrow$  'a set  $\Rightarrow$ 
  (('a  $\Rightarrow$  'a)  $\Rightarrow$  ('a  $\Rightarrow$  'a))
where galois-restriction (K :: 'a set) E F = restrict ( $\lambda\sigma$ . restrict  $\sigma$  E) (field-auto K F)

```

```

lemma galois-restriction-apply [simp]:
   $\sigma \in \text{field-auto } K \ F \Longrightarrow$ 
  galois-restriction K E F  $\sigma$  = restrict  $\sigma$  E
by (simp add: galois-restriction-def)

```

```

context
  fixes K E F :: 'a :: field set
  assumes K: Subfield K and sfE: Subfield E
  and FE:  $F \subseteq E$  and EK:  $E \subseteq K$ 
  and stable:  $\bigwedge \sigma. \sigma \in \text{field-auto } K \ F \Longrightarrow \sigma ` E = E$ 
begin

```

```

lemma field-auto-bij:  $\sigma \in \text{field-auto } K \ F \Longrightarrow \text{bij-betw } \sigma \ E \ E$ 
by (metis EK bij-betw-subset field-auto-mem-iff stable)

```

Restriction inherits the field laws and fixed-base condition, while stability supplies the required bijection of the intermediate carrier.

```

lemma gal-res-mem:
  assumes s:  $\sigma \in \text{field-auto } K \ F$ 
  shows restrict  $\sigma$  E  $\in \text{field-auto } E \ F$ 
proof -
  have sE:  $\sigma ` E = E$  using s by (rule stable)
  have addh:  $\bigwedge x \ y. x \in K \Longrightarrow y \in K \Longrightarrow \sigma (x + y) = \sigma x + \sigma y$ 

```

```

and multH:  $\bigwedge x y. x \in K \implies y \in K \implies \sigma (x * y) = \sigma x * \sigma y$ 
and one:  $\sigma 1 = 1$  and fixF:  $\bigwedge x. x \in F \implies \sigma x = x$ 
using s by (auto simp: field-auto-mem-iff)
interpret E: Subfield E by (rule sfE)
show ?thesis
  unfolding field-auto-mem-iff
proof (intro conjI ballI)
  fix x y assume xy:  $x \in E \ y \in E$ 
  have xyK:  $x \in K \ y \in K$  using xy EK by auto
  show restrict  $\sigma \ E \ (x + y) = \text{restrict } \sigma \ E \ x + \text{restrict } \sigma \ E \ y$ 
    using xy xyK E.add-closed addh by simp
  show restrict  $\sigma \ E \ (x * y) = \text{restrict } \sigma \ E \ x * \text{restrict } \sigma \ E \ y$ 
    using xy xyK E.mult-closed multH by simp
  qed (use field-auto-bij[OF s] sE fixF FE one in auto)
qed

```

Restriction turns composition in K into composition in E .

lemma *gal-res-compose*:

```

assumes s:  $\sigma \in \text{field-auto } K \ F$  and t:  $\tau \in \text{field-auto } K \ F$ 
shows restrict (compose  $K \ \sigma \ \tau$ )  $E = \text{compose } E \ (\text{restrict } \sigma \ E) \ (\text{restrict } \tau \ E)$ 
proof (rule ext)
  fix x
  show restrict (compose  $K \ \sigma \ \tau$ )  $E \ x = \text{compose } E \ (\text{restrict } \sigma \ E) \ (\text{restrict } \tau \ E) \ x$ 
  proof (cases x ∈ E)
    case True
    with EK have xK:  $x \in K$  by blast
    have txE:  $\tau \ x \in E$ 
    proof –
      have  $\tau \ x \in \tau \ ` \ E$  by (rule imageI[OF True])
      then show ?thesis using stable[OF t] by simp
    qed
    show ?thesis by (simp add: compose-def txE xK)
  qed (simp add: compose-def restrict-def)
qed

```

Restriction sends the identity of K to the identity of E .

lemma *gal-res-unit*: *restrict* (*identity* K) $E = \text{identity } E$
using *EK* **by** *fastforce*

theorem *galois-restriction-hom*:

```

group-homomorphism (restrict ( $\lambda \sigma. \text{restrict } \sigma \ E$ ) (field-auto  $K \ F$ ))
  (field-auto  $K \ F$ ) (compose  $K$ ) (identity  $K$ )
  (field-auto  $E \ F$ ) (compose  $E$ ) (identity  $E$ )
proof –
  have FK:  $F \subseteq K$  using FE EK by blast
  interpret GK: Group field-auto K F compose K identity K
    by (rule field-auto-group[OF K FK])
  interpret GE: Group field-auto E F compose E identity E
    by (rule field-auto-group[OF sfE FE])

```

```

show ?thesis
proof qed (use gal-res-mem gal-res-unit in ⟨auto simp: gal-res-compose⟩)
qed

theorem galois-restriction-epimorphism:
  assumes onto: galois-restriction K E F ' field-auto K F = field-auto E F
  shows group-epimorphism (galois-restriction K E F)
    (field-auto K F) (compose K) (identity K)
    (field-auto E F) (compose E) (identity E)
proof -
  have hom: group-homomorphism (galois-restriction K E F)
    (field-auto K F) (compose K) (identity K)
    (field-auto E F) (compose E) (identity E)
  by (unfold galois-restriction-def; rule galois-restriction-hom)
  interpret H: group-homomorphism galois-restriction K E F
    field-auto K F compose K identity K
    field-auto E F compose E identity E
  by (rule hom)
  interpret S: surjective-map galois-restriction K E F
    field-auto K F field-auto E F
  by unfold-locales (rule onto)
  show ?thesis
proof (rule group-epimorphism.intro)
  show group-homomorphism (galois-restriction K E F)
    (field-auto K F) (compose K) (identity K)
    (field-auto E F) (compose E) (identity E)
  by (rule hom)
  show Monoid-epimorphism (galois-restriction K E F)
    (field-auto K F) (compose K) (identity K)
    (field-auto E F) (compose E) (identity E)
  by (rule Monoid-epimorphism.intro[OF H.Monoid-homomorphism-axioms S.surjective-map-axioms])
qed
qed

theorem galois-restriction-Ker:
  group-homomorphism.Ker (restrict (λσ. restrict σ E) (field-auto K F))
    (field-auto K F) (identity E) = field-auto K E
proof -
  interpret H: group-homomorphism restrict (λσ. restrict σ E) (field-auto K F)
    field-auto K F compose K identity K field-auto E F compose E identity E
  by (rule galois-restriction-hom)
  show ?thesis
proof (rule set-eqI, rule iffI)
  fix σ assume mem: σ ∈ H.Ker
  then have sF: σ ∈ field-auto K F by (rule H.Ker-closed)
  have restrict (λσ. restrict σ E) (field-auto K F) σ = identity E
    using mem by (rule H.Ker-image)
  then have restrict σ E = identity E using sF by simp
  then show σ ∈ field-auto K E
    using sF unfolding field-auto-mem-iff by (metis restrict-apply')

```

```

next
  fix  $\sigma$  assume  $sE$ :  $\sigma \in \text{field-auto } K \ E$ 
  have  $\text{fix}E$ :  $\bigwedge x. x \in E \implies \sigma x = x$ 
    using  $sE$  by (auto simp: field-auto-mem-iff)
  have  $sF$ :  $\sigma \in \text{field-auto } K \ F$ 
    using  $sE \ FE$  by (auto simp: field-auto-mem-iff)
  then show  $\sigma \in H.Ker$ 
    by (metis (no-types, lifting)  $H.Ker\text{-mem}I$   $\text{fix}E$  restrict-apply' restrict-ext)
qed
qed

```

The standard first-isomorphism theorem applies to this generic restriction map.

```

interpretation restriction-core: group-homomorphism
  galois-restriction  $K \ E \ F$ 
  field-auto  $K \ F$  compose  $K$  identity  $K$ 
  field-auto  $E \ F$  compose  $E$  identity  $E$ 
proof –
  show group-homomorphism (galois-restriction  $K \ E \ F$ )
    (field-auto  $K \ F$ ) (compose  $K$ ) (identity  $K$ )
    (field-auto  $E \ F$ ) (compose  $E$ ) (identity  $E$ )
  by (unfold galois-restriction-def; rule galois-restriction-hom)
qed

```

```

theorem galois-restriction-factor-isomorphism:
  (restriction-core.kernel.Factor-Group,
   restriction-core.kernel.quotient-composition,
   restriction-core.kernel.Class (identity  $K$ ))  $\cong_G$ 
  (galois-restriction  $K \ E \ F$  ‘ field-auto  $K \ F$ , compose  $E$ , identity  $E$ )
  by (rule restriction-core.first-isomorphism)

```

102.1 The inductive step: solvability climbs the tower

With E stable, solvability of the relative kernel and the intermediate-field automorphism group implies solvability of the full automorphism group. The restriction map is corestricted to its image; the group quotient is handled by the standard first-isomorphism machinery.

```

theorem galois-solvable-tower-step:
  assumes  $\text{solv}KE$ : Group.solvable (field-auto  $K \ E$ ) (compose  $K$ ) (identity  $K$ )
  and  $\text{solv}EF$ : Group.solvable (field-auto  $E \ F$ ) (compose  $E$ ) (identity  $E$ )
  shows Group.solvable (field-auto  $K \ F$ ) (compose  $K$ ) (identity  $K$ )
proof –
  define  $\eta$  where  $\eta = \text{galois-restriction } K \ E \ F$ 
  interpret  $H$ : group-homomorphism  $\eta$ 
    field-auto  $K \ F$  compose  $K$  identity  $K$  field-auto  $E \ F$  compose  $E$  identity  $E$ 
  unfolding  $\eta$ -def galois-restriction-def by (rule galois-restriction-hom)
  interpret  $I$ : group-epimorphism  $\eta$ 
    field-auto  $K \ F$  compose  $K$  identity  $K$   $\eta$  ‘ field-auto  $K \ F$  compose  $E$  identity  $E$ 
  by unfold-locales (auto simp:  $H$ .commutes-with-composition)
  have  $Ker\text{-eq}$ :  $I.Ker = \text{field-auto } K \ E$ 

```

```

    using galois-restriction-Ker
    unfolding  $\eta$ -def galois-restriction-def I.Ker-def H.Ker-def by simp
    have solvKer: Group.solvable I.Ker (compose K) (identity K)
    using Ker-eq solvKE by simp
    have solvImage: Group.solvable ( $\eta$  ' field-auto K F) (compose E) (identity E)
    proof (rule Group.solvable-subgroup)
      show Group (field-auto E F) (compose E) (identity E)
      by (rule field-auto-group[OF sfE FE])
      show Subgroup ( $\eta$  ' field-auto K F) (field-auto E F)
      (compose E) (identity E)
      by (rule H.image.Subgroup-axioms)
      show Group.solvable (field-auto E F) (compose E) (identity E)
      by (rule solvEF)
    qed
    show ?thesis
    using I.solvable-extension solvImage solvKer
    by blast
  qed

end

```

102.2 Complex splitting-field extension of generic restriction

Surjectivity of restriction for a generated root field. Let $F \subseteq E \subseteq K$ with K generated by E and the roots of an F -polynomial. The finite homomorphism-extension theorem supplies an extension of every F -automorphism of E .

theorem *galois-restriction-surjective*:

```

fixes K E F :: 'a :: field set and Phi :: 'a poly
assumes sfE: Subfield E and sfF: Subfield F and FE:  $F \subseteq E$ 
and Phi:  $\Phi \in \text{poly-over } F$ 
and Kdef:  $K = \text{generate-field } (E \cup \{r. \text{poly } \Phi r = 0\})$ 
and finR:  $\text{finite } \{r. \text{poly } \Phi r = 0\}$ 
and algR:  $\bigwedge r. \text{poly } \Phi r = 0 \implies \text{algebraic-over } E r$ 
and closed:  $\bigwedge q :: 'a \text{ poly}. 0 < \text{degree } q \implies \exists b. \text{poly } q b = 0$ 
and t:  $\tau \in \text{field-auto } E F$ 
shows  $\exists \sigma \in \text{field-auto } K F. \forall x \in E. \sigma x = \tau x$ 

```

proof —

```

define R where  $R = \{r. \text{poly } \Phi r = 0\}$ 
have EK:  $E \subseteq K$  using Kdef by (auto simp: subset-generate-field)
have sfK: Subfield K unfolding Kdef by (rule subfield-generate-field)
have FK:  $F \subseteq K$  using FE EK by simp
have homt: field-hom-on E  $\tau$  using sfE t by (rule field-auto-imp-field-hom-on)
have fixFt:  $\bigwedge x. x \in F \implies \tau x = x$  using t by (auto simp: field-auto-mem-iff)
— Extend  $\tau$  over the finite root set to a homomorphism  $g$  on  $K$ .
have algRE:  $\bigwedge r. r \in R \implies \text{algebraic-over } E r$  using algR by (simp add: R-def)
obtain g where homg: field-hom-on (generate-field (E  $\cup$  R))  $g$ 
and gE:  $\bigwedge x. x \in E \implies g x = \tau x$ 
using field-hom-on-extend-finite[OF closed finR[folded R-def] homt algRE] by blast

```

```

have homgK: field-hom-on K g using homg by (simp add: Kdef R-def)
have fixFg:  $\bigwedge x. x \in F \implies g x = x$  using gE fixFt FE by auto
have RK:  $R \subseteq K$  using EK unfolding Kdef R-def using subset-generate-field by
blast
have gR-sub:  $g \restriction R \subseteq R$ 
  using hom-preserves-roots[OF homgK sF FK fixFg Phi] RK
  by (auto simp: R-def)
have injK: inj-on g K by (rule field-hom-on.inj-on[OF homgK])
have gR:  $g \restriction R = R$ 
  by (metis RK R-def endo-inj-surj finR gR-sub injK inj-on-subset)
have gE-onto:  $g \restriction E = E$ 
  by (metis bij-betw-imp-surj-on field-auto-mem-iff gE image-cong t)
have gUn:  $g \restriction (E \cup R) = E \cup R$  using gE-onto gR by (simp add: image-Un)
have gK:  $g \restriction K = K$ 
  by (metis EK Kdef RK R-def field-hom-on.image-generate-field-eq gUn homgK
sup.boundedI)
have aut: restrict g K  $\in$  field-auto K F
  by (rule field-hom-on-imp-field-auto[OF homgK gK fixFg FK])
then show ?thesis using aut
  by (metis EK gE in-mono restrict-def)
qed

```

102.3 A normal solvable tower has a solvable Galois group

Given a fixed complex splitting field K and a chain of intermediate fields $F = F_0 \subseteq F_1 \subseteq \dots \subseteq F_n = K$ in which each F_i is *stable* under $\text{field-auto } K F$ and each step's relative Galois group $\text{field-auto } F_{i+1} F_i$ is solvable, the whole Galois group $\text{field-auto } K F$ is solvable. This chains the inductive step *galois-solvable-tower-step* down the tower, the base case being the trivial group $\text{field-auto } K K$.

102.3.1 The trivial Galois group

An automorphism of K fixing all of K is the identity, so $\text{field-auto } K K$ is trivial.

lemma *field-auto-self-trivial*:

```

fixes K :: complex set
assumes K: Subfield K
shows field-auto K K = {identity K}

```

proof –

```

have cK: complex-subfield K using K by (simp add: complex-subfield-iff-subfield)
have  $\sigma \in \{\text{identity } K\}$  if  $\sigma \in \text{field-auto } K K$  for  $\sigma$ 
  by (metis that PiE-restrict field-auto-mem-iff insertI1 restrict-ext)
then show ?thesis
  using cK identity-field-auto by auto

```

qed

lemma *solvable-field-auto-self*:

```

fixes K :: complex set

```



```

assumes  $K$ : Subfield  $K$ 
shows  $\text{Group.solvable } (\text{field-auto } K K) (\text{compose } K) (\text{identity } K)$ 
proof –
  have  $cK$ : complex-subfield  $K$  using  $K$  by (simp add: complex-subfield-iff-subfield)
  interpret  $G$ : Group field-auto  $K K$  compose  $K$  identity  $K$ 
    by (rule Galois-group-Group[OF  $cK$  subset-refl])
  show ?thesis unfolding  $G.\text{solvable-def}$ 
    using  $G.\text{derivedSeries.simps}(1)$  assms field-auto-self-trivial by blast
qed

```

102.3.2 Normal solvable towers

solvable-tower $K F F_s$: the chain starting at F and following F_s s, with fixed top K , in which every field is a subfield of K , every intermediate field is stable under *field-auto* $K F$, and every step has a solvable relative Galois group. (The empty tower requires $F = K$.)

```

fun solvable-tower :: complex set  $\Rightarrow$  complex set  $\Rightarrow$  complex set list  $\Rightarrow$  bool where
  solvable-tower  $K F [] \iff F = K$ 
| solvable-tower  $K F (G \# G_s) \iff$ 
  Subfield  $F \wedge$  Subfield  $G \wedge F \subseteq G \wedge G \subseteq K \wedge$ 
   $(\forall \sigma \in \text{field-auto } K F. \sigma \restriction G = \text{id}_G) \wedge$ 
  Group.solvable (field-auto  $G F$ ) (compose  $G$ ) (identity  $G$ )  $\wedge$ 
  solvable-tower  $K G G_s$ 

```

102.3.3 The assembly

A normal solvable tower has a solvable Galois group. By induction down the tower: the top K has the trivial (hence solvable) group *field-auto* $K K$; each step combines the solvable relative group *field-auto* $G F$ with the solvable *field-auto* $K G$ (from the induction hypothesis) via *galois-solvable-tower-step*, using stability of G under *field-auto* $K F$.

```

theorem solvable-tower-imp-solvable:
  assumes  $K$ : Subfield  $K$ 
  shows solvable-tower  $K F F_s \implies$  Subfield  $F$ 
     $\implies \text{Group.solvable } (\text{field-auto } K F) (\text{compose } K) (\text{identity } K)$ 
proof (induction  $F_s$  arbitrary:  $F$ )
  case Nil then show ?case
    by (metis solvable-field-auto-self solvable-tower.simps(1))
next
  case Cons
  then show ?case
    by (metis assms galois-solvable-tower-step solvable-tower.simps(2))
qed

end

```

103 Restriction along a stable intermediate field of a splitting field

```

theory Galois-Restriction-Splitting
imports Galois-Simple-Degree Galois-Restriction
begin

```

The general restriction theory exposes the homomorphism and its kernel, but leaves the surjectivity argument in terms of a generated root field. This theory supplies the missing representation bridge for the public *splitting-field* interface. The stability hypothesis is essential: without it, restricting an F -automorphism of K need not map the intermediate field E to itself, so the restriction does not land in *field-auto* $E F$.

The roots of a splitting-field polynomial give exactly the generated-field presentation required by the finite homomorphism-extension theorem.

```

lemma splitting-field-generate-field-over:
  fixes  $K E :: \text{complex set}$  and  $p :: \text{complex poly}$ 
  assumes split: splitting-field  $E p K$ 
  shows  $K = \text{generate-field } (E \cup \{r. \text{poly } p \ r = 0\})$ 
proof –
  have  $K = \text{generate-field } (E \cup \text{poly-root-set } p)$ 
    using splitting-fieldD[OF split] by blast
  then show ?thesis
    by (simp add: gen-subfield-eq-generate-field poly-root-set-def)
qed

```

For a splitting field over E , restriction onto E is surjective onto the F -automorphisms of E , provided the defining polynomial is already over F . This is the form used by the radical-solvability descent: the larger group fixes F , while the splitting-field generators are algebraic over E .

```

theorem galois-restriction-image-of-splitting-field:
  fixes  $K E F :: \text{complex set}$  and  $p :: \text{complex poly}$ 
  assumes split: splitting-field  $E p K$ 
  and sfE: Subfield  $E$  and sfF: Subfield  $F$  and FE:  $F \subseteq E$ 
  and pF:  $p \in \text{poly-over } F$ 
  and stable:  $\bigwedge \sigma. \sigma \in \text{field-auto } K F \implies \sigma ' E = E$ 
  shows galois-restriction  $K E F$  ' field-auto  $K F = \text{field-auto } E F$ 
proof –
  have sfK: Subfield  $K$ 
    using splitting-field-subfield[OF split]
    by (simp add: complex-subfield-iff-subfield)
  have EK:  $E \subseteq K$ 
    by (rule splitting-field-base-subset[OF split])
  have pnz:  $p \neq 0$ 
    using split by (auto simp: splitting-fieldD)
  have Kdef:  $K = \text{generate-field } (E \cup \{r. \text{poly } p \ r = 0\})$ 
    by (rule splitting-field-generate-field-over[OF split])

```

```

have finR: finite {r. poly p r = 0}
  using pnz by (rule poly-roots-finite)
have algR:  $\bigwedge r. \text{poly } p \ r = 0 \implies \text{algebraic-over } E \ r$ 
proof -
  fix r assume root: poly p r = 0
  have r  $\in$  poly-root-set p
    using root by (simp add: poly-root-set-def)
  then show algebraic-over E r
    by (rule splitting-field-root-algebraic[OF split])
qed
have closed:  $\bigwedge q :: \text{complex poly. } 0 < \text{degree } q \implies \exists b. \text{poly } q \ b = 0$ 
  using fundamental-theorem-of-algebra by (simp add: constant-degree)
have image-subset:
  galois-restriction K E F ' field-auto K F  $\subseteq$  field-auto E F
  using gal-res-mem[OF sfK sfE FE EK stable]
  by (auto simp: galois-restriction-apply)
have target-subset:
  field-auto E F  $\subseteq$  galois-restriction K E F ' field-auto K F
proof
  fix  $\tau$  assume tau:  $\tau \in \text{field-auto } E \ F$ 
  obtain  $\sigma$  where sigma:  $\sigma \in \text{field-auto } K \ F$ 
    and agree:  $\forall x \in E. \sigma \ x = \tau \ x$ 
    using galois-restriction-surjective[OF sfE sfF FE pF Kdef finR algR closed tau]
by blast
  have restrict-eq': restrict  $\sigma$  E = restrict  $\tau$  E
    by (rule restrict-ext) (use agree in simp)
  have tauPiE:  $\tau \in E \rightarrow_E E$ 
    using tau by (simp only: field-auto-mem-iff)
  have tau-restrict: restrict  $\tau$  E =  $\tau$ 
    using PiE-restrict[OF tauPiE] .
  have restrict-eq: restrict  $\sigma$  E =  $\tau$ 
    using restrict-eq' tau-restrict by (rule trans)
  have restriction: galois-restriction K E F  $\sigma = \tau$ 
    using sigma by (simp add: galois-restriction-apply restrict-eq)
  show  $\tau \in \text{galois-restriction } K \ E \ F \ ' \text{field-auto } K \ F$ 
    using sigma restriction by blast
qed
show ?thesis using image-subset target-subset by blast
qed

```

Package the preceding image equality as an epimorphism.

```

theorem galois-restriction-epimorphism-of-splitting-field:
  fixes K E F :: complex set and p :: complex poly
  assumes split: splitting-field E p K
    and sfE: Subfield E and sfF: Subfield F and FE: F  $\subseteq$  E
    and pF: p  $\in$  poly-over F
    and stable:  $\bigwedge \sigma. \sigma \in \text{field-auto } K \ F \implies \sigma \ ' E = E$ 
  shows group-epimorphism (galois-restriction K E F)
    (field-auto K F) (compose K) (identity K)
    (field-auto E F) (compose E) (identity E)

```

```

proof –
  have Ksub: Subfield K
    using splitting-field-subfield[OF split]
    by (simp add: complex-subfield-iff-subfield)
  have EK:  $E \subseteq K$ 
    by (rule splitting-field-base-subset[OF split])
  have image-eq: galois-restriction K E F ‘ field-auto K F = field-auto E F
    by (rule galois-restriction-image-of-splitting-field[OF split sfE sfF FE pF stable])
  show ?thesis
    by (rule galois-restriction-epimorphism[OF Ksub sfE FE EK stable image-eq])
qed

```

```

context
fixes K E F :: complex set and p :: complex poly
assumes split: splitting-field F p K
  and sfF: complex-subfield F
  and E:  $E \in \text{inter-fields } K F$ 
  and stable:  $\bigwedge \sigma. \sigma \in \text{field-auto } K F \implies \sigma \text{ ‘ } E = E$ 
begin

```

Now specialise to an intermediate field of a splitting field over the full base.

```

lemma splitting-field-restriction-image:
  galois-restriction K E F ‘ field-auto K F = field-auto E F
proof –
  obtain FE:  $F \subseteq E$ 
    using E by (auto simp: inter-fields-iff)
  have sfE: Subfield E
    using E by (auto simp: inter-fields-iff complex-subfield-iff-subfield)
  have sfF': Subfield F
    using sfF by (simp add: complex-subfield-iff-subfield)
  have splitE: splitting-field E p K
    by (rule splitting-field-over-intermediate[OF split E])
  have pF:  $p \in \text{poly-over } F$ 
    using split by (auto simp: splitting-fieldD)
  show ?thesis
    by (rule galois-restriction-image-of-splitting-field[OF splitE sfE sfF' FE pF stable])
qed

```

```

lemma splitting-field-restriction-kernel:
  group-homomorphism.Ker (galois-restriction K E F)
    (field-auto K F) (identity E) = field-auto K E
proof –
  obtain FE:  $F \subseteq E$  and EK:  $E \subseteq K$ 
    using E by (auto simp: inter-fields-iff)
  have sfE: Subfield E
    using E by (auto simp: inter-fields-iff complex-subfield-iff-subfield)
  have Ksub: Subfield K
    using splitting-field-subfield[OF split]
    by (simp add: complex-subfield-iff-subfield)
  show ?thesis

```

by (unfold galois-restriction-def; rule galois-restriction-Ker[OF Ksub sfE FE EK stable])
qed

interpretation restriction: group-epimorphism

galois-restriction K E F
field-auto K F compose K identity K
field-auto E F compose E identity E

proof –

obtain FE: $F \subseteq E$ and EK: $E \subseteq K$
using E by (auto simp: inter-fields-iff)
have sfE: Subfield E
using E by (auto simp: inter-fields-iff complex-subfield-iff-subfield)
have sfF': Subfield F
using sfE by (simp add: complex-subfield-iff-subfield)
have splitE: splitting-field E p K
by (rule splitting-field-over-intermediate[OF split E])
have pF: $p \in \text{poly-over } F$
using split by (auto simp: splitting-fieldD)
show group-epimorphism (galois-restriction K E F)
(field-auto K F) (compose K) (identity K)
(field-auto E F) (compose E) (identity E)
by (rule galois-restriction-epimorphism-of-splitting-field
[OF splitE sfE sfF' FE pF stable])

qed

The first isomorphism theorem identifies the quotient by the relative automorphism group with the Galois group of the stable intermediate field.

theorem splitting-field-restriction-factor-isomorphism:

(restriction.kernel.Factor-Group,
restriction.kernel.quotient-composition,
restriction.kernel.Class (identity K)) $\cong_G$
(field-auto E F, compose E, identity E)

proof –

have image-eq:
galois-restriction K E F ' field-auto K F = field-auto E F
using splitting-field-restriction-image .
have iso:
(restriction.kernel.Factor-Group,
restriction.kernel.quotient-composition,
restriction.kernel.Class (identity K)) $\cong_G$
(galois-restriction K E F ' field-auto K F, compose E, identity E)
by (rule restriction.first-isomorphism)
show ?thesis using iso image-eq by simp

qed

For a finite splitting-field Galois group, the quotient isomorphism gives the expected multiplicative cardinality law. This is a group-order statement; identifying it with a general extension dimension is a separate finite-dimensional theorem.

```

theorem splitting-field-restriction-card:
  card (field-auto K F) =
    card (field-auto K E) * card (field-auto E F)
proof –
  have sfF': complex-subfield F by (rule sfF)
  have finite-source: finite (field-auto K F)
    by (rule splitting-field-finite-galois-group[OF split sfF'])
  have lagrange:
    card (field-auto K F) =
      card restriction.Ker * restriction.kernel.index
  using restriction.kernel.lagrange[OF finite-source] by simp
  have quotient-card: card restriction.kernel.Factor-Group = restriction.kernel.index
  using restriction.kernel.card-Factor-Group[OF finite-source] by simp
  have f-exists:
    ∃ f. group-isomorphism f restriction.kernel.Factor-Group
      restriction.kernel.quotient-composition (restriction.kernel.Class (identity K))
      (field-auto E F) (compose E) (identity E)
  using splitting-field-restriction-factor-isomorphism
  by (simp add: isomorphic-as-groups-def)
  obtain f where f-iso:
    group-isomorphism f restriction.kernel.Factor-Group
      restriction.kernel.quotient-composition (restriction.kernel.Class (identity K))
      (field-auto E F) (compose E) (identity E)
  using f-exists by blast
  interpret I: group-isomorphism f restriction.kernel.Factor-Group
    restriction.kernel.quotient-composition restriction.kernel.Class (identity K)
    field-auto E F compose E identity E
  by (rule f-iso)
  have quotient-target-card: card restriction.kernel.Factor-Group = card (field-auto E
F)
  using I.bijjective by (rule bij-betw-same-card)
  have card-source:
    card (field-auto K F) = card restriction.Ker * card (field-auto E F)
  using lagrange quotient-card quotient-target-card by simp
  show ?thesis
  using card-source splitting-field-restriction-kernel by simp
qed

end

end

```

104 Artin's lemma: a finite automorphism group bounds the degree over its fixed field

```

theory Artin
imports Galois-Correspondence Linear-System Iso-Extension-Tower
begin

```

Let H be a *finite* group of automorphisms of a subfield K of a field, and let *fixed-field* K^H be the subfield it fixes pointwise. *Artin's lemma* bounds the extension K over that fixed field by the order of the group: any $\text{card } H + 1$ elements of K are linearly dependent over *fixed-field* K^H .

The proof is the linear-algebra pigeonhole principle. Given elements $x_1, \dots, x_n$ of K with $n > \text{card } H$, consider the homogeneous system

- one equation per automorphism $\sigma \in H$, namely $\sum_j \sigma(x_j) c_j = 0$, in the unknowns c_j . There are more unknowns than equations, so *Subfield.underdetermined-solution* supplies a nontrivial solution in K — and the equation belonging to the identity automorphism is precisely the required dependence $\sum_j x_j c_j = 0$.

104.1 Artin's lemma

The system to be solved has one equation per automorphism and one unknown per element, so its coefficient matrix is $\lambda \sigma \ j. \ \sigma \ (x \ j)$: a genuinely two-index family, indexed by *functions* on one side and by the element index on the other. This is exactly the shape *Subfield.underdetermined-solution* was stated for, indices ranging over arbitrary finite sets rather than initial segments.

theorem *artin-lemma*:

fixes $H :: ('a :: \text{field} \Rightarrow 'a) \text{ set}$ **and** $x :: 'j \Rightarrow 'a$

assumes K : *Subfield* K

and $\text{fin}H$: *finite* H **and** HG : $H \subseteq \text{field-auto } K$

and $\text{fin}J$: *finite* J **and** card : $\text{card } H < \text{card } J$

and xK : $\bigwedge j. j \in J \implies x \ j \in K$

shows $\exists c. (\forall j. c \ j \in K) \wedge (\exists j \in J. c \ j \neq 0) \wedge (\forall \sigma \in H. (\sum j \in J. \sigma \ (x \ j) * c \ j) = 0)$

proof —

interpret K : *Subfield* K **by** (*rule* K)

— The coefficients $\sigma \ (x \ j)$ lie in K , since an automorphism maps K onto itself. Stated as an explicit $\forall$ rather than with $\bigwedge$: as a rule with schematic variables it unifies with the closure premise in more than one way, which *OF* rejects outright (*OF*: *multiple unifiers*).

have *entries*: $\forall \sigma \in H. \forall j \in J. \sigma \ (x \ j) \in K$

using $HG \ xK$ **by** (*blast intro: field-auto-closed*)

— Every index of the hypothesis is pinned explicitly, or the matrix argument x comes back applied to an unknown index function — see *Linear-System*.

show *?thesis*

using $K.\text{underdetermined-solution}$

[**where** $A = \lambda \sigma \ j. \ \sigma \ (x \ j)$ **and** $I = H$ **and** $J = J$, *OF* $\text{fin}H \ \text{fin}J \ \text{card}$]

using *entries* **by** *blast*

qed

The advertised form: more than $\text{card } H$ elements of K admit a nontrivial vanishing K -linear combination against *every* automorphism of H at once, and in

particular — taking the identity, which lies in every Galois group — against the elements themselves.

corollary *artin-lemma-dependence*:

fixes $H :: ('a :: \text{field} \Rightarrow 'a) \text{ set}$ **and** $x :: 'j \Rightarrow 'a$

assumes K : *Subfield* K

and $\text{fin}H$: *finite* H **and** HG : $H \subseteq \text{field-auto } K \ F$ **and** id : *identity* $K \in H$

and $\text{fin}J$: *finite* J **and** card : $\text{card } H < \text{card } J$

and xJK : $x \text{ ' } J \subseteq K$

shows $\exists c. (\forall j. c \ j \in K) \wedge (\exists j \in J. c \ j \neq 0) \wedge (\sum j \in J. x \ j * c \ j) = 0$

proof —

have xK : $\bigwedge j. j \in J \Rightarrow x \ j \in K$ **using** xJK **by** *blast*

obtain c **where** cK : $\forall j. c \ j \in K$ **and** cnz : $\exists j \in J. c \ j \neq 0$

and ceq : $\forall \sigma \in H. (\sum j \in J. \sigma \ (x \ j) * c \ j) = 0$

using *artin-lemma*[**where** $x = x$, *OF* $K \ \text{fin}H \ HG \ \text{fin}J \ \text{card } xK$] **by** *blast*

— The equation belonging to the identity automorphism is the dependence we want.

have $(\sum j \in J. \text{identity } K \ (x \ j) * c \ j) = 0$ **using** ceq id **by** *blast*

moreover **have** $\bigwedge j. j \in J \Rightarrow \text{identity } K \ (x \ j) = x \ j$ **using** xK **by** (*simp add: identity-apply*)

ultimately **have** $(\sum j \in J. x \ j * c \ j) = 0$ **by** *simp*

then show *?thesis* **using** $cK \ \text{cnz}$ **by** *blast*

qed

For a genuine Galois subgroup the identity premise is automatic. This is the convenient interface for the later fixed-field argument, while *artin-lemma* deliberately retains the extra generality of an arbitrary finite automorphism family.

corollary *artin-lemma-subgroup-dependence*:

fixes $H :: ('a :: \text{field} \Rightarrow 'a) \text{ set}$ **and** $x :: 'j \Rightarrow 'a$

assumes K : *Subfield* K

and H : $H \in \text{galois-subgroups } K \ F$ **and** $\text{fin}H$: *finite* H

and $\text{fin}J$: *finite* J **and** card : $\text{card } H < \text{card } J$

and xJK : $x \text{ ' } J \subseteq K$

shows $\exists c. (\forall j. c \ j \in K) \wedge (\exists j \in J. c \ j \neq 0) \wedge (\sum j \in J. x \ j * c \ j) = 0$

proof —

interpret H : *Subgroup* $H \ \text{field-auto } K \ F$ *compose* $K \ \text{identity } K$

using H **by** (*simp add: galois-subgroups-iff*)

show *?thesis*

by (*rule artin-lemma-dependence*[*OF* $K \ \text{fin}H \ H.\text{subset } H.\text{sub-unit-closed } \text{fin}J \ \text{card } xJK$])

qed

104.2 Transporting a dependence along an automorphism

The step that will drive the descent to the fixed field: an automorphism carries a vanishing linear combination to another vanishing linear combination, over the transported coefficients. Multiplicativity handles each summand, additivity the sum, and *field-auto-zero* the right-hand side.

This is the mechanism behind the minimality argument sketched at the end. If c is a dependence among the $x \ j$ with the fewest nonzero coefficients, normalised

so that some $c \cdot k = 1$, then $\lambda j. \sigma (c \cdot j)$ is another dependence, also with $\sigma (c \cdot k) = 1$ and supported in the same places. Their difference is a shorter dependence, so minimality forces it to vanish, i.e. $\sigma (c \cdot j) = c \cdot j$ for every $\sigma \in H$ — which is exactly to say that the coefficients lie in *fixed-field* $K \cdot H$.

lemma *field-auto-transport-dependence*:

assumes K : Subfield K **and** s : $\sigma \in \text{field-auto } K \ F$
and finJ : finite J
and xK : $\bigwedge j. j \in J \implies x \cdot j \in K$ **and** cK : $\bigwedge j. j \in J \implies c \cdot j \in K$
and dep : $(\sum j \in J. x \cdot j * c \cdot j) = 0$
shows $(\sum j \in J. \sigma (x \cdot j) * \sigma (c \cdot j)) = 0$

proof –

interpret K : Subfield K **by** (rule K)
have $(\sum j \in J. \sigma (x \cdot j) * \sigma (c \cdot j)) = (\sum j \in J. \sigma (x \cdot j * c \cdot j))$
using $xK \ cK$ **by** (simp add: field-auto-mult[OF s])
also have $\dots = \sigma (\sum j \in J. x \cdot j * c \cdot j)$
using $xK \ cK$ **by** (intro field-auto-sum[OF $K \ s$, symmetric]) (blast intro: $K.\text{mult-closed}$)
also have $\dots = 0$ **using** dep field-auto-zero[OF $K \ s$] **by** simp
finally show ?thesis .

qed

Applying one subgroup automorphism to the coefficients preserves the simultaneous system. For a target row ϱ , apply σ to the row indexed by *compose* K (*inverse* σ) ϱ ; the group laws reduce the transformed row back to ϱ .

lemma *field-auto-transport-artin-solution*:

fixes $H :: ('a :: \text{field} \Rightarrow 'a) \text{ set}$
assumes K : Subfield K **and** H : $H \in \text{galois-subgroups } K \ F$
and s : $\sigma \in H$ **and** finJ : finite J
and xK : $\bigwedge j. j \in J \implies x \cdot j \in K$
and cK : $\bigwedge j. j \in J \implies c \cdot j \in K$
and equations : $\forall \varrho \in H. (\sum j \in J. \varrho (x \cdot j) * c \cdot j) = 0$
and $\varrho \in H$
shows $(\sum j \in J. \varrho (x \cdot j) * \sigma (c \cdot j)) = 0$

proof –

interpret H : Subgroup H field-auto $K \ F$ *compose* K *identity* K
using H **by** (simp add: galois-subgroups-iff)
let $? \tau = \text{compose } K (H.\text{sub.inverse } \sigma) \varrho$
have invH : $H.\text{sub.inverse } \sigma \in H$
using s **by** simp
have t : $? \tau \in H$
using invH $\langle \varrho \in H \rangle$ **by** (rule $H.\text{sub.composition-closed}$)
have dep : $(\sum j \in J. ? \tau (x \cdot j) * c \cdot j) = 0$
using $\text{equations } t$ **by** blast
have txK : $\bigwedge j. j \in J \implies ? \tau (x \cdot j) \in K$
using $H.\text{subset } t \ xK$ **by** (blast intro: field-auto-closed)
have $s\text{Auto}$: $\sigma \in \text{field-auto } K \ F$
using $H.\text{subset } s$ **by** blast
have transported : $(\sum j \in J. \sigma (? \tau (x \cdot j)) * \sigma (c \cdot j)) = 0$
using field-auto-transport-dependence[OF $K \ s\text{Auto } \text{finJ } \text{txK } cK \ \text{dep}$] .
have $\bigwedge j. j \in J \implies \sigma (? \tau (x \cdot j)) = \varrho (x \cdot j)$

```

    using  $H.sub.invertible$   $H.sub.invertible-right-inverse2$   $\langle \varrho \in H \rangle s xK$  by (metis
compose-eq)
    with transported show  $(\sum j \in J. \varrho (x j) * \sigma (c j)) = 0$ 
    by simp
qed

```

104.3 Descending the coefficients to the fixed field

The coefficients of a dependence can be taken to lie in the fixed field, not merely in K .

The device is minimality. Among all nontrivial dependences of the $x j$ choose one whose *support* $\{j \in J. c j \neq 0\}$ is smallest, and scale it so that some distinguished coefficient equals 1. For $\sigma \in H$ the transported family $\lambda j. \sigma (c j)$ is again a dependence — this is *field-auto-transport-dependence*, and it needs σ to fix the $x j$, which is why the $x j$ are taken in the fixed field's *ambient* position below.

lemma *dependence-difference*:

```

fixes  $c d :: 'j \Rightarrow 'a :: field$ 
assumes  $finJ$ :  $finite J$ 
    and  $dep-c$ :  $(\sum j \in J. x j * c j) = 0$  and  $dep-d$ :  $(\sum j \in J. x j * d j) = 0$ 
shows  $(\sum j \in J. x j * (c j - d j)) = 0$ 
by (simp add:  $dep-c$   $dep-d$   $right-diff-distrib'$   $sum-subtractf$ )

```

The simultaneous system version of the minimal-support descent. Unlike the specialized fixed-vector descent below, this does not assume that the vectors are fixed by H : the equations contain every row $\varrho \in H$, and *field-auto-transport-artin-solution* transports a solution to another solution of that same system. This is the form needed for the genuine Artin bound.

theorem *artin-solution-over-fixed-field*:

```

fixes  $H :: ('a :: field \Rightarrow 'a) set$  and  $x :: 'j \Rightarrow 'a$ 
assumes  $K$ : Subfield  $K$  and  $H$ :  $H \in galois-subgroups K F$ 
    and  $finJ$ :  $finite J$  and  $xK$ :  $\bigwedge j. j \in J \implies x j \in K$ 
    and  $exsol$ :  $\exists c. (\forall j. c j \in K) \wedge (\exists j \in J. c j \neq 0) \wedge (\forall \varrho \in H. (\sum j \in J. \varrho (x j) * c j) = 0)$ 
shows  $\exists c. (\forall j \in J. c j \in fixed-field K H) \wedge (\exists j \in J. c j \neq 0) \wedge (\forall \varrho \in H. (\sum j \in J. \varrho (x j) * c j) = 0)$ 

```

proof –

```

interpret  $KS$ : Subfield  $K$  by (rule  $K$ )

```

```

define  $Sol$  where

```

```

 $Sol \equiv (\lambda c :: 'j \Rightarrow 'a. (\forall j. c j \in K) \wedge (\exists j \in J. c j \neq 0) \wedge (\forall \varrho \in H. (\sum j \in J. \varrho (x j) * c j) = 0))$ 

```

```

define  $supp$  where  $supp \equiv (\lambda c :: 'j \Rightarrow 'a. \{j \in J. c j \neq 0\})$ 

```

```

define  $Q$  where  $Q \equiv (\lambda n. \exists c. Sol c \wedge card (supp c) = n)$ 

```

```

have  $Q (LEAST n. Q n)$  using  $exsol$  by (metis LeastI  $Sol-def$   $Q-def$ )

```

```

then obtain  $c$  where  $solc$ :  $Sol c$  and  $cmin$ :  $card (supp c) = (LEAST n. Q n)$ 

```

```

unfolding  $Q-def$  by blast

```

```

have  $minimal$ :  $card (supp c) \leq card (supp d)$  if  $Sol d$  for  $d$ 

```

```

by (metis Least-le  $Q-def$   $cmin$  that)

```

```

from solc have cK:  $\bigwedge j. c\ j \in K$  and ceq:  $\bigwedge \varrho. \varrho \in H \implies (\sum j \in J. \varrho\ (x\ j) * c\ j)$ 
= 0
by (simp-all add: Sol-def)
from solc obtain k where kJ:  $k \in J$  and cknz:  $c\ k \neq 0$ 
by (auto simp: Sol-def)
define e where  $e \equiv (\lambda j. c\ j / c\ k)$ 
have eK:  $\bigwedge j. e\ j \in K$  using cK by (simp add: e-def KS.divide-closed)
have ekeq1:  $e\ k = 1$  using cknz by (simp add: e-def)
have esupp:  $\text{supp } e = \text{supp } c$  using cknz by (auto simp: supp-def e-def)
have eeq:  $(\sum j \in J. \varrho\ (x\ j) * e\ j) = 0$  if  $r: \varrho \in H$  for  $\varrho$ 
using ceq[OF r] by (simp add: e-def divide-simps flip: sum-divide-distrib)
have depe: Sol e
unfolding Sol-def using eK ekeq1 kJ eeq by (intro conjI allI beqI[of - k]) (simp-all)
have fixed:  $\sigma\ (e\ j) = e\ j$  if  $s: \sigma \in H$  and  $jJ: j \in J$  for  $\sigma$   $j$ 
proof (rule ccontr)
assume ne:  $\sigma\ (e\ j) \neq e\ j$ 
have sG:  $\sigma \in \text{field-auto } K\ F$ 
using H galois-subgroups-subset s by blast
define d where  $d = (\lambda i. e\ i - \sigma\ (e\ i))$ 
have eqe:  $\forall \varrho \in H. (\sum i \in J. \varrho\ (x\ i) * e\ i) = 0$ 
using depe by (simp add: Sol-def)
have transported-all:
 $\forall \varrho \in H. (\sum i \in J. \varrho\ (x\ i) * \sigma\ (e\ i)) = 0$ 
using field-auto-transport-artin-solution[OF K H s finJ xK eK eqe] by blast
have dep-d:  $(\sum i \in J. \varrho\ (x\ i) * d\ i) = 0$  if  $r: \varrho \in H$  for  $\varrho$ 
using r eeq transported-all unfolding d-def
by (simp add: sum-subtractf right-diff-distrib)
have dK:  $\bigwedge i. d\ i \in K$ 
unfolding d-def using eK sG by (blast intro: KS.diff-closed field-auto-closed)
have depd: Sol d
using Sol-def dK d-def dep-d jJ ne by force
have dk:  $d\ k = 0$  using ekeq1 field-auto-one[OF sG] by (simp add: d-def)
have sub:  $\text{supp } d \subseteq \text{supp } e - \{k\}$ 
using field-auto-zero K d-def dk sG supp-def by fastforce
have fin-e: finite ( $\text{supp } e$ ) using finJ by (simp add: supp-def)
have k-e:  $k \in \text{supp } e$  using kJ ekeq1 by (simp add: supp-def)
have card ( $\text{supp } d$ )  $\leq \text{card } (\text{supp } e - \{k\})$  using sub fin-e by (intro card-mono)
blast
also have  $\dots < \text{card } (\text{supp } e)$ 
using fin-e k-e by (meson card-Diff1-less)
finally show False using minimal[OF depd] esupp by simp
qed
show ?thesis
using Sol-def depe fixed by (smt (cvc5, dec-internal-enum-inst-sum) fixed-field-memI)
qed

```

Note that the $x\ j$ are *fixed* by every $\sigma \in H$. This is essential and not a technicality: *field-auto-transport-dependence* turns a dependence of the $x\ j$ into a dependence of the images $\sigma\ (x\ j)$, and only when those images are the $x\ j$ again is the result a dependence of the *same* family, so that subtracting it from the original stays

inside the system. In the intended application the roles are reversed from what one might expect: it is the *vectors* that are taken over the fixed field and the *coefficients* that are unknown and must be shown to descend.

Also, H is closed under composition — a subgroup of the Galois group — which is what makes $\sigma(cj) = c\sigma(j)$ for all $\sigma \in H$ equivalent to $cj \in \text{fixed-field } K^H$. In fact only membership of $\sigma \in H \subseteq \text{field-auto } K/F$ is used below; the group structure is what makes the conclusion *mean* what it says.

theorem *dependence-over-fixed-field:*

fixes $H :: ('a :: \text{field} \Rightarrow 'a) \text{ set}$ **and** $x :: 'j \Rightarrow 'a$
assumes $K: \text{Subfield } K$
and $HG: H \subseteq \text{field-auto } K/F$
and $\text{fin}J: \text{finite } J$
and $xK: \bigwedge j. j \in J \Rightarrow xj \in K$
and $\text{xfix}: \bigwedge \sigma j. [\sigma \in H; j \in J] \Rightarrow \sigma(xj) = xj$
and $\text{exdep}: \exists c. (\forall j. cj \in K) \wedge (\exists j \in J. cj \neq 0) \wedge (\sum j \in J. xj * cj) = 0$
shows $\exists c. (\forall j \in J. cj \in \text{fixed-field } K^H) \wedge (\exists j \in J. cj \neq 0)$
 $\wedge (\sum j \in J. xj * cj) = 0$

proof —

interpret $K: \text{Subfield } K$ **by** (rule K)

— The dependences of the xj , and the sizes of their supports.

define Dep **where**

$\text{Dep} \equiv (\lambda c :: 'j \Rightarrow 'a. (\forall j. cj \in K) \wedge (\exists j \in J. cj \neq 0) \wedge (\sum j \in J. xj * cj) = 0)$

define supp **where** $\text{supp} \equiv (\lambda c :: 'j \Rightarrow 'a. \{j \in J. cj \neq 0\})$

define Q **where** $Q \equiv (\lambda n. \exists c. \text{Dep } c \wedge \text{card } (\text{supp } c) = n)$

— Choose a dependence of least support, exactly as *minpoly-exists* chooses an annihilator of least degree.

have Q (LEAST $n. Q n$) **using** exdep **by** (metis LeastI Dep-def Q-def)

then obtain c **where** $\text{depc}: \text{Dep } c$ **and** $\text{cmin}: \text{card } (\text{supp } c) = (\text{LEAST } n. Q n)$

unfolding $Q\text{-def}$ **by** blast

have $\text{minimal}: \text{card } (\text{supp } c) \leq \text{card } (\text{supp } d)$ **if** $\text{Dep } d$ **for** d

by (metis Least-le Q-def cmin that)

from depc **have** $cK: \bigwedge j. cj \in K$ **and** $\text{ceq}: (\sum j \in J. xj * cj) = 0$

by (simp-all add: Dep-def)

from depc **obtain** k **where** $kJ: k \in J$ **and** $\text{cknz}: ck \neq 0$ **by** (auto simp: Dep-def)

— Normalise: divide through by ck . This changes neither the support nor the property of being a dependence, and it makes the distinguished coefficient 1.

define e **where** $e = (\lambda j. cj / ck)$

have $eK: \bigwedge j. ej \in K$ **using** cK **by** (simp add: e-def K.divide-closed)

have $ek: ek = 1$ **using** cknz **by** (simp add: e-def)

have $\text{esupp}: \text{supp } e = \text{supp } c$ **using** cknz **by** (auto simp: supp-def e-def)

have $\text{eeq}: (\sum j \in J. xj * ej) = 0$

using ceq $e\text{-def}$

by (metis (mono-tags, lifting) div-0 sum.cong sum-divide-distrib times-divide-eq-right)

have $\text{depe}: \text{Dep } e$

unfolding Dep-def **using** eK eeq **by** (intro conjI allI bexI[of - k]) (simp-all add: ek kJ)

— The heart of it: every σ in H fixes every coefficient of e . Suppose not, and let d be

the difference of e and its transport. Then d is a dependence, it is nonzero somewhere (where σ moved a coefficient), and its support omits k because both families have 1 there. So d is a dependence of strictly smaller support, contradicting minimality.

have fixed: $\sigma (e j) = e j$ **if** $s: \sigma \in H$ **and** $jJ: j \in J$ **for** σj
proof (rule ccontr)
assume $ne: \neg ?thesis$
have $sG: \sigma \in \text{field-auto } K F$ **using** $s HG$ **by** $blast$
define d **where** $d = (\lambda i. e i - \sigma (e i))$
 — The transport is a dependence of the $x i$ themselves, because σ fixes them.
have $(\sum i \in J. \sigma (x i) * \sigma (e i)) = 0$
using $eeq eK xK$ **by** (intro field-auto-transport-dependence[OF K sG finJ]) *auto*
then have $dep-t: (\sum i \in J. x i * \sigma (e i)) = 0$ **using** $xfix[OF s]$ **by** *simp*
have $dep-d: (\sum i \in J. x i * d i) = 0$
unfolding $d\text{-def}$ **using** $eeq dep-t$ **by** (rule dependence-difference[OF finJ])
have $dK: \bigwedge i. d i \in K$
unfolding $d\text{-def}$ **using** $eK sG$ **by** (blast intro: K.diff-closed field-auto-closed)
have $dnz: d j \neq 0$ **using** ne **by** (simp add: d-def)
have $depd: Dep d$ **using** $dK dnz jJ dep-d$ **by** (auto simp: Dep-def)
 — k drops out: both families take the value 1 there.
have $dk: d k = 0$ **using** $ek \text{field-auto-one}[OF sG]$ **by** (simp add: d-def)
have $sub: \text{supp } d \subseteq \text{supp } e - \{k\}$
using $\text{field-auto-zero Diff-iff } K d\text{-def } dk \text{empty-iff insert-iff mem-Collect-eq}$
right-minus-eq
sG supp-def **by** *fastforce*
have $fin-e: \text{finite } (\text{supp } e)$ **using** $finJ$ **by** (simp add: supp-def)
have $k-e: k \in \text{supp } e$ **using** $kJ ek$ **by** (simp add: supp-def)
 — Removing an element of a finite set drops the cardinality; $\text{supp } d$ is inside what is left.
have $\text{card } (\text{supp } d) \leq \text{card } (\text{supp } e - \{k\})$ **using** $sub fin-e$ **by** (intro card-mono)
blast
also have $\dots < \text{card } (\text{supp } e)$
using $fin-e k-e$ **by** (meson card-Diff1-less-iff)
finally show $False$ **using** $minimal[OF depd] esupp$ **by** *simp*
qed
 — Fixed by every element of H and lying in K is exactly membership of the fixed field.
show $?thesis$
using $Dep\text{-def } depe \text{fixed}$ **by** (metis (lifting) fixed-field-memI)
qed

104.4 Dedekind's lemma: distinct automorphisms are linearly independent

The reverse direction. Distinct field automorphisms are linearly independent as functions: if $\sum \sigma a_\sigma \sigma(z) = 0$ for every z in K , with the σ pairwise distinct, then all the coefficients a_σ vanish.

The proof is by induction on the number of automorphisms involved, and the inductive step is a small piece of cunning. Suppose the relation holds for a set $\text{insert } \sigma_0 S$ and, for a contradiction, that it is nontrivial. Pick τ in S and, since

τ and σ_0 are distinct automorphisms, an element y of K with $\sigma_0 y \neq \tau y$. Now evaluate the relation at $y * z$ rather than at z : by multiplicativity this multiplies the σ -th term by σy . Subtracting $\sigma_0 y$ times the original relation annihilates the σ_0 term, leaving a relation on S alone whose τ -coefficient is $a \tau * (\tau y - \sigma_0 y) \neq 0$.

lemma *character-relation-scaled*:

assumes K : Subfield K **and** $\text{fin}S$: finite S
and SG : $S \subseteq \text{field-auto } K F$
and rel : $\bigwedge z. z \in K \implies (\sum \sigma \in S. a \sigma * \sigma z) = 0$
and y : $y \in K$ **and** z : $z \in K$
shows $(\sum \sigma \in S. (a \sigma * \sigma y) * \sigma z) = 0$

proof –

interpret K : Subfield K **by** (rule K)
have mult : $\sigma (y * z) = \sigma y * \sigma z$ **if** $\sigma \in S$ **for** σ
using that SG $y z$ **by** (blast intro: field-auto-mult)
have $(\sum \sigma \in S. (a \sigma * \sigma y) * \sigma z) = (\sum \sigma \in S. a \sigma * \sigma (y * z))$
by (simp add: mult mult.assoc)
also have $\dots = 0$ **using** $\text{rel } y z$ **by** (simp add: $K.\text{mult-closed}$)
finally show ?thesis .

qed

Dedekind's lemma. The coefficients are arbitrary field elements; no subfield hypothesis on them is needed, and none is available in the intended application.

theorem *dedekind-independence*:

fixes $S :: ('a :: \text{field} \Rightarrow 'a)$ set **and** $a :: ('a \Rightarrow 'a) \Rightarrow 'a$
assumes K : Subfield K
shows $\llbracket \text{finite } S; S \subseteq \text{field-auto } K F; \bigwedge z. z \in K \implies (\sum \sigma \in S. a \sigma * \sigma z) = 0 \rrbracket$
 $\implies \forall \sigma \in S. a \sigma = 0$

proof (induction S arbitrary: a rule: finite-induct)

case empty then show ?case **by** simp

next

case (insert σ_0 S)

interpret K : Subfield K **by** (rule K)

— *finite-induct* consumes the finiteness premise, so the remaining two are the subset and the relation. Named rather than indexed, since the offset is easy to get wrong.

note $\text{ins}G = \text{insert.prem}(1)$ **and** $\text{ins}rel = \text{insert.prem}(2)$

have $s0G$: $\sigma_0 \in \text{field-auto } K F$ **and** SG : $S \subseteq \text{field-auto } K F$ **using** $\text{ins}G$ **by** auto

have rel : $\bigwedge z. z \in K \implies a \sigma_0 * \sigma_0 z + (\sum \sigma \in S. a \sigma * \sigma z) = 0$

using $\text{ins}rel$ insert.hyps **by** simp

— Every coefficient indexed by S vanishes. Given that, the σ_0 coefficient vanishes too, by evaluating the relation at 1.

have $S\text{-zero}$: $a \tau = 0$ **if** $\tau \in S$ **for** τ

proof (rule ccontr)

assume atnz : $\neg ?thesis$

have tG : $\tau \in \text{field-auto } K F$ **using** $\langle \tau \in S \rangle SG$ **by** blast

— Distinct automorphisms differ at some point of K : both are extensional on K , so agreeing there would make them equal.

have tne : $\tau \neq \sigma_0$ **using** $\langle \tau \in S \rangle \text{insert.hyps}$ **by** blast

obtain y **where** yK : $y \in K$ **and** diff : $\sigma_0 y \neq \tau y$

using *s0G tG tne* **by** (*metis (no-types, lifting) ext PiE-E field-auto-mem-iff*)
 — Two relations on *insert* σ_0 *S*: the original scaled by σ_0 *y*, and the one obtained
 by evaluating at $y * z$. Both have the same σ_0 term.
define *b* **where** $b \equiv (\lambda \sigma. a \sigma * (\sigma y - \sigma_0 y))$
have $(\sum \sigma \in S. b \sigma * \sigma z) = 0$ **if** zK : $z \in K$ **for** *z*
proof —
 have *scaled*: $(\sum \sigma \in \text{insert } \sigma_0 S. (a \sigma * \sigma y) * \sigma z) = 0$
 by (*rule character-relation-scaled[OF K - insG insrel yK zK]*) (*simp add:*
 insert.hyps)
 have $(\sum \sigma \in \text{insert } \sigma_0 S. (a \sigma * \sigma_0 y) * \sigma z) = \sigma_0 y * (\sum \sigma \in \text{insert } \sigma_0 S. a$
 $\sigma * \sigma z)$
 by (*simp add: sum-distrib-left mult.commute mult.left-commute*)
 also have $\dots = 0$ **using** *insrel zK* **by** *simp*
 finally have *orig*: $(\sum \sigma \in \text{insert } \sigma_0 S. (a \sigma * \sigma_0 y) * \sigma z) = 0$.
 have $(\sum \sigma \in \text{insert } \sigma_0 S. b \sigma * \sigma z)$
 $= (\sum \sigma \in \text{insert } \sigma_0 S. (a \sigma * \sigma y) * \sigma z) - (\sum \sigma \in \text{insert } \sigma_0 S. (a \sigma * \sigma_0$
 $y) * \sigma z)$
 unfolding *b-def* **by** (*simp add: sum-subtractf left-diff-distrib right-diff-distrib*
 mult.assoc)
 also have $\dots = 0$ **using** *scaled orig* **by** *simp*
 finally have $(\sum \sigma \in \text{insert } \sigma_0 S. b \sigma * \sigma z) = 0$.
 then show *?thesis* **using** *insert.hyps*
 using *b-def* **by** *force*
qed
 — So the induction hypothesis applies to *S* with coefficients *b* — but *b* τ is a
 product of two nonzero factors.
then show *False* **using** *insert.IH[where a = b, OF SG]* $\langle \tau \in S \rangle$ *atnz diff*
 by (*force simp: b-def*)
qed
 — Finally the leading coefficient.
have $a \sigma_0 * \sigma_0 1 + (\sum \sigma \in S. a \sigma * \sigma 1) = 0$ **using** *rel K.one-closed* **by** *blast*
moreover have $(\sum \sigma \in S. a \sigma * \sigma 1) = 0$ **using** *S-zero* **by** *simp*
ultimately have $a \sigma_0 = 0$
 using *field-auto-one[OF s0G]* **by** *simp*
then show *?case* **using** *S-zero* **by** *blast*
qed

This theory supplies the two difficult ingredients for Artin's theorem: the pi-
 geonhole dependence and the descent of its coefficients into the fixed field.
 The vector-space degree assembly and the final finite-group equality live in
Artin-Degree.

The older *dependence-over-fixed-field* remains a useful specialization for a fam-
 ily of vectors already fixed by *H*. The corrected simultaneous-system theorem
artin-solution-over-fixed-field is the interface used by the degree layer, so the
 public *artin-theorem* no longer exposes either that fixed-vector premise or an
 explicit basis premise.

end

105 Artin's degree bound: the fixed field of a finite group has small index

```
theory Artin-Degree
  imports Artin Algebraic-Transitivity
begin
```

The results of *Artin* are phrased with explicit sums over an index set. Here they are transported into the vector-space language of *Extension-Vector-Space*, where *fixed-field* $K \subseteq H \subseteq K$ is a *subfield-tower* and the size of the extension is measured by *Vector-Space.lin-indep* and *Vector-Space.dimension*. The bridge in both directions is *subfield-tower.vs-lincomb-eq-sum*: an abstract linear combination over a finite set of vectors is an ordinary field sum.

The conclusion is *Artin's bound*: over the fixed field of a finite group H of automorphisms, no more than $\text{card } H$ elements of K can be linearly independent. Together with Dedekind's lemma — which gives the matching lower bound — this is what pins the order of the Galois group of *fixed-field* $K \subseteq H$ to exactly $\text{card } H$.

```
notation plus (infixl <+> 65)
notation minus (infixl <-> 65)
unbundle uminus-syntax
```

105.1 The fixed field sits below K as a tower

Both fields are native set-based subfields and one contains the other, so they form a *subfield-tower* — the scalars being the fixed field and the vectors K .

```
lemma fixed-field-tower:
  assumes  $K$ : Subfield  $K$  and  $HG$ :  $H \subseteq \text{field-auto } K \ F$ 
  shows subfield-tower (fixed-field  $K \ H$ )  $K$ 
  using  $HG \ K$ 
  by (meson field-auto-fixed-field-subfield fixed-field-subset subfield-tower-axioms.intro
      subfield-tower-def)
```

105.2 Artin's bound

No set of more than $\text{card } H$ elements of K is linearly independent over the fixed field. The argument is the one sketched in *Artin*, now assembled: *artin-lemma* produces a simultaneous dependence with coefficients in K , and *artin-solution-over-fixed-field* pushes those coefficients down into the fixed field. The simultaneous system is the key correction: no assumption that the vectors themselves are fixed by H is needed.

Care is needed about which set indexes what. A *lin-indep* set is a set of *vectors*, and the dependence lemmas index by an arbitrary type; we index the vectors by themselves, taking $J = B$ and $x = \text{id}$.

The statements live in a *context* fixing K and H and interpreting the tower. The *Vector-Space* locale has nine parameters, so spelling *lin-indep* out at the

top level would mean writing all of them; inside the context the interpretation supplies them and the qualified *T.vs.* prefix works.

context

fixes $K\ F :: 'a :: \text{field set}$ **and** $H :: ('a \Rightarrow 'a) \text{ set}$
assumes K : *Subfield* K **and** H sub: $H \in \text{galois-subgroups } K\ F$
begin

lemma *H-auto*: $H \subseteq \text{field-auto } K\ F$ **using** H sub **by** (*rule galois-subgroups-subset*)

interpretation T : *subfield-tower fixed-field* $K\ H\ K$ **by** (*rule fixed-field-tower*[*OF* $K\ H$ -auto])

theorem *artin-bound*:

assumes *finH*: *finite* H **and** BK : $B \subseteq K$
and *finB*: *finite* B **and** *lt*: $\text{card } H < \text{card } B$
shows $\neg T.\text{vs.lin-indep } B$

proof

assume *indep*: $T.\text{vs.lin-indep } B$
have *idH*: *identity* $K \in H$
using H sub **by** (*metis Subgroup-def Submonoid.sub-unit-closed galois-subgroups-iff*)
— A simultaneous nontrivial dependence with coefficients in K , indexed by the vectors themselves.

have xK : $\bigwedge j. j \in B \implies \text{id } j \in K$

using BK **by** *force*

have *dep*: $\exists c. (\forall j. c\ j \in K) \wedge (\exists j \in B. c\ j \neq 0) \wedge$
 $(\forall \sigma \in H. (\sum j \in B. \sigma (\text{id } j) * c\ j) = 0)$

using *artin-lemma*[**where** $x = \text{id}$ **and** $J = B$, *OF* $K\ \text{finH}\ H\text{-auto}\ \text{finB}\ \text{lt}\ xK$] .

have *descended*:

$\exists c. (\forall j \in B. c\ j \in \text{fixed-field } K\ H) \wedge (\exists j \in B. c\ j \neq 0) \wedge (\forall \sigma \in H. (\sum j \in B. \sigma (\text{id } j) * c\ j) = 0)$

by (*intro* $xK\ K\ \text{dep}\ \text{finB}\ H$ sub *artin-solution-over-fixed-field*[**where** $F = F$])

then obtain c **where** *cfix*: $\bigwedge j. j \in B \implies c\ j \in \text{fixed-field } K\ H$

and *cnz*: $\exists j \in B. c\ j \neq 0$

and *ceq-all*: $\forall \sigma \in H. (\sum j \in B. \sigma (\text{id } j) * c\ j) = 0$ **by** *blast*

have *ceq*: $(\sum j \in B. j * c\ j) = 0$

proof —

have *row*: $(\sum j \in B. \text{identity } K (\text{id } j) * c\ j) = 0$

using *ceq-all idH* **by** *blast*

moreover have $(\sum j \in B. \text{identity } K (\text{id } j) * c\ j) = (\sum j \in B. j * c\ j)$

using xK **by** *force*

ultimately show *?thesis* **by** *simp*

qed

— Re-read the sum as an abstract linear combination. *Vector-Space.lin-indep* demands a coefficient function that is *extensional* on B , so restrict c .

define e **where** $e \equiv \text{restrict } c\ B$

have eF : $e \in B \rightarrow_E \text{fixed-field } K\ H$ **using** *cfix* **by** (*simp add: e-def*)

have $T.\text{vs.lincomb } e\ B = (\sum j \in B. e\ j * j)$

using *finB BK eF* **by** (*intro T.vs-lincomb-eq-sum*) *auto*

also have $\dots = 0$

```

    by (simp add: ceq e-def mult.commute)
  finally have lc0:  $T.vs.lincomb\ e\ B = 0$  .
  — Independence would force every coefficient to vanish, contradicting nontriviality.
  have  $\forall v \in B. e\ v = 0$  using indep eF lc0 by (simp add: T.vs.lin-indep-def)
  then show False using cnz by (auto simp: e-def)
qed

```

Consequently a basis of K over the fixed field — if there is one — has at most $\text{card } H$ elements, so the degree is bounded by the group order.

```

corollary artin-dimension-le:
  assumes finH: finite H and basis:  $T.vs.basis\ B$ 
  shows  $\text{card } B \leq \text{card } H$ 
  using T.vs.basis-def T.vs.basis-lin-indep artin-bound assms
  by (meson linorder-le-less-linear)

```

And the same bound on the degree itself, since the dimension is the size of any basis (*Vector-Space.dimension-eq-any-field* — no finiteness of the base field needed).

```

corollary artin-degree-le:
  assumes finH: finite H and basis:  $T.vs.basis\ B$ 
  shows  $T.vs.dimension \leq \text{card } H$ 
  using artin-dimension-le[where  $B = B$ , OF finH basis]
  T.vs.dimension-eq-any-field[OF basis] by simp

```

end

105.3 The reverse bound: the Galois group is no larger than the degree

Now the other inequality, and the one that needs Dedekind's lemma. Let E be an intermediate field and B a basis of K over E . Then *field-auto* $K\ E$ has at most $\text{card } B$ elements.

The argument is again the linear pigeonhole, with the roles of the two index sets exchanged. Suppose there were more than $\text{card } B$ automorphisms, and take a finite set S of them of size $\text{card } B + 1$. Consider the system with *one equation per basis vector* $v \in B$ and one unknown per automorphism: $\sum_{\sigma} a_{\sigma} \sigma(v) = 0$. It has more unknowns than equations, so *Subfield.underdetermined-solution* gives a nontrivial solution.

A solution of those finitely many equations already satisfies the corresponding relation at *every* point of K : an arbitrary $z \in K$ is an E -combination of B , and each automorphism is E -linear (it is additive, multiplicative, and fixes E), so the relation propagates from the basis to all of K by linearity. But then Dedekind's lemma forces every coefficient to vanish — contradicting nontriviality.

The propagation step is the only real work, and it is where the basis hypothesis is used.

As before the statements sit inside a context, so that the tower's interpretation

supplies the nine parameters of *Vector-Space* and *basis* can be written qualified. The tower is *interpreted* here rather than passed as an assumption: a hypothesis *subfield-tower* $E\ K$ would not bring $T.vs.basis$ into scope for the statement itself.

context

fixes $K\ E :: 'a :: \text{field set}$

assumes $K: \text{Subfield } K$ **and** $Ebase: \text{subfield-tower } E\ K$

begin

interpretation $T: \text{subfield-tower } E\ K$ **by** (*rule* $Ebase$)

The base field lies inside the extension — the tower's own axiom.

lemma $EK: E \subseteq K$ **by** (*rule* $T.base-subset$)

An automorphism fixing E is E -linear on linear combinations of a finite set of vectors: it commutes with the sum, with the scalar products, and fixes the scalars.

lemma *field-auto-lincomb*:

assumes $s: \sigma \in \text{field-auto } K\ E$ **and** $finB: \text{finite } B$ **and** $BK: B \subseteq K$

and $c: \bigwedge v. v \in B \implies c\ v \in E$

shows $\sigma (\sum v \in B. c\ v * v) = (\sum v \in B. c\ v * \sigma\ v)$

proof —

have $cK: \bigwedge v. v \in B \implies c\ v \in K$ **using** $c\ EK$ **by** *blast*

have $\sigma (\sum v \in B. c\ v * v) = (\sum v \in B. \sigma (c\ v * v))$

using $cK\ BK$ **by** (*intro* $\text{field-auto-sum}[OF\ K\ s]$) (*blast* *intro*: $\text{Subfield.mult-closed}[OF\ K]$)

also have $\dots = (\sum v \in B. c\ v * \sigma\ v)$

using $BK\ c\ cK\ s$ **by** (*smt* (*cvc5*, *interleave*) *field-auto-mem-iff subset-iff sum.cong*)

finally show *?thesis* .

qed

Hence a relation among automorphisms that holds on a basis holds throughout K .

lemma *character-relation-from-basis*:

assumes $finS: \text{finite } S$ **and** $SG: S \subseteq \text{field-auto } K\ E$

and $basis: T.vs.basis\ B$

and $rel: \forall v \in B. (\sum \sigma \in S. a\ \sigma * \sigma\ v) = 0$

and $z: z \in K$

shows $(\sum \sigma \in S. a\ \sigma * \sigma\ z) = 0$

proof —

have $finB: \text{finite } B$ **and** $BK: B \subseteq K$ **using** $basis$ **by** (*auto simp*: $T.vs.basis-def$)

— Coordinates of z with respect to the basis.

have $T.vs.spanning\ B$ **using** $basis$ **by** (*rule* $T.vs.basis-spanning$)

then obtain c **where** $cE: c \in B \rightarrow_E E$ **and** $zeq: z = T.vs.lincomb\ c\ B$

using z **by** (*auto simp*: $T.vs.spanning-def$)

have $cEv: \bigwedge v. v \in B \implies c\ v \in E$ **using** cE **by** *auto*

have $zsum: z = (\sum v \in B. c\ v * v)$

using $zeq\ cE\ T.vs.lincomb-eq-sum[OF\ finB\ BK]$ **by** *auto*

— Expand each $\sigma\ z$ by linearity and exchange the two sums.

```

have (∑ σ ∈ S. a σ * σ z) = (∑ σ ∈ S. a σ * (∑ v ∈ B. c v * σ v))
  using SG finB BK cEv by (simp add: zsum field-auto-lincomb subset-iff)
also have ... = (∑ σ ∈ S. ∑ v ∈ B. c v * (a σ * σ v))
  by (simp add: sum-distrib-left mult.commute mult.left-commute)
also have ... = (∑ v ∈ B. c v * (∑ σ ∈ S. a σ * σ v))
  using sum.swap by (simp add: sum-distrib-left)
also have ... = 0 using rel by simp
finally show ?thesis .
qed

```

The reverse bound. With a basis of size $\text{card } B$, there cannot be more than $\text{card } B$ distinct automorphisms of K fixing E .

```

theorem galois-group-card-le:
  assumes basis: T.vs.basis B
    and finS: finite S and SG: S ⊆ field-auto K E
  shows card S ≤ card B
proof (rule ccontr)
  assume non: ¬ card S ≤ card B
  have finB: finite B and BK: B ⊆ K using basis by (auto simp: T.vs.basis-def)
  — One equation per basis vector, one unknown per automorphism: more unknowns
  than equations. The coefficient matrix  $\lambda v \sigma. \sigma v$  is indexed by vectors then automor-
  phisms.
  have entries: ∀ v ∈ B. ∀ σ ∈ S. σ v ∈ K
    using SG BK by (blast intro: field-auto-closed)
  have ∃ a. (∀ σ. a σ ∈ K) ∧ (∃ σ ∈ S. a σ ≠ 0) ∧ (∀ v ∈ B. (∑ σ ∈ S. σ v * a σ)
    = 0)
    using Subfield.underdetermined-solution[OF K, where A = λ v σ. σ v and I =
    B and J = S]
    using entries finB finS non by force
  then obtain a where anz: ∃ σ ∈ S. a σ ≠ 0 and arel: ∀ v ∈ B. (∑ σ ∈ S. σ v *
    a σ) = 0
    by fastforce
  — Rewrite into the orientation Dedekind's lemma expects.
  have arel': ∀ v ∈ B. (∑ σ ∈ S. a σ * σ v) = 0
    using arel by (simp add: mult.commute)
  — The relation extends from the basis to all of K, so Dedekind applies.
  have ∧ z. z ∈ K ⟹ (∑ σ ∈ S. a σ * σ z) = 0
    by (rule character-relation-from-basis[OF finS SG basis arel'])
  then show False using anz
    using dedekind-independence[OF K, of S] finS SG by blast
qed
end

```

105.4 Artin's theorem

The two bounds now meet. Let H be a finite group of automorphisms of K . The uniform bound first extracts a basis B of K over *fixed-field* K^H ; write $G = \text{field-auto } K \text{ (fixed-field } K^H)$ for the Galois group of the fixed field.

- *artin-dimension-le* gives $\text{card } B \leq \text{card } H$: were there more basis vectors than automorphisms, Artin's bound would make them dependent;
- *galois-group-card-le* gives $\text{card } G \leq \text{card } B$: were there more automorphisms than basis vectors, Dedekind's lemma would be contradicted;
- *galois-extension.le-galois-group-fixed-field* — the unit of the adjunction, and the one part needing no hypotheses — gives $H \subseteq G$.

Chaining the first two bounds gives $\text{card } G \leq \text{card } H$, and an inclusion between finite sets of equal cardinality is an equality. So the kernel operator of the Galois correspondence is the identity on finite subgroups: *every* automorphism fixing the fixed field of H already belongs to H .

This is the converse half of the correspondence, and the hypothesis that does the work is *finiteness of H* ; no basis, separability, or normality is assumed by the public theorem.

Again a context, for the same reason as before: the locally extracted *basis* is a *Vector-Space* constant and can only be written qualified where an interpretation of the tower is in scope. Here the tower is the one built from H itself.

context

fixes $K F :: 'a :: \text{field set}$ **and** $H :: ('a \Rightarrow 'a) \text{ set}$
assumes K : *Subfield* K **and** F_{sub} : *Subfield* F **and** FK : $F \subseteq K$
and H_{sub} : $H \in \text{galois-subgroups } K F$

begin

lemma HG : $H \subseteq \text{field-auto } K F$ **using** H_{sub} **by** (*rule galois-subgroups-subset*)

interpretation T : *subfield-tower fixed-field* $K H K$
by (*rule fixed-field-tower*[$OF K HG$])

theorem *artin-fixed-field-degree-data*:

assumes finH : *finite* H
shows $\exists B. T.\text{vs.basis } B \wedge \text{card } B = \text{card } H \wedge$
 $\text{finite } (\text{field-auto } K (\text{fixed-field } K H)) \wedge$
 $\text{card } (\text{field-auto } K (\text{fixed-field } K H)) = \text{card } H$

proof —

interpret GE : *galois-extension* $K F$
by (*rule galois-extension.intro*[$OF K F_{\text{sub}} FK$])
have tower : *subfield-tower* (*fixed-field* $K H$) K **by** (*rule fixed-field-tower*[$OF K HG$])

— First obtain a basis from the uniform Artin bound. The bound applies to every independent set, so the maximum-cardinality extraction in *Vector-Space.basis-exists-of-independent-card-bound* supplies a basis without assuming one in the theorem statement.

have indep-bound : $\bigwedge A. T.\text{vs.lin-indep } A \implies \text{card } A \leq \text{card } H$
using $H_{\text{sub}} K T.\text{vs.lin-indep-def finH}$ **by** (*metis artin-bound linorder-not-less*)
obtain B **where** basis : $T.\text{vs.basis } B$
using $T.\text{vs.basis-exists-of-independent-card-bound}$ [**where** $n = \text{card } H$] indep-bound

by *blast*

— Upper bound: the extracted basis is no larger than the group.

```

have le1: card B ≤ card H
  by (rule artin-dimension-le[OF K Hsub finH basis])
  — Lower bound: the Galois group of the fixed field is no larger than the basis. Its
  finiteness comes from the same bound, since a set exceeding card B could not inject.
have finG: finite (field-auto K (fixed-field K H))
  using galois-group-card-le[OF K tower basis] infinite-arbitrarily-large
  by (meson finite-if-finite-subsets-card-bdd)
have le2: card (field-auto K (fixed-field K H)) ≤ card B
  by (rule galois-group-card-le[OF K tower basis finG subset-refl])
have card-le: card (field-auto K (fixed-field K H)) ≤ card H using le1 le2 by simp
have card-ge: card H ≤ card (field-auto K (fixed-field K H))
  by (simp add: GE.le-galois-group-fixed-field Hsub card-mono finG)
have cardG: card (field-auto K (fixed-field K H)) = card H
  using card-le card-ge by simp
have cardB: card B = card H using le1 le2 card-ge by simp
show ?thesis using basis cardB finG cardG by blast
qed

```

theorem *artin-theorem*:

```

assumes finH: finite H
shows field-auto K (fixed-field K H) = H
proof —
obtain B where basis: T.vs.basis B and cardB: card B = card H
  and finG: finite (field-auto K (fixed-field K H))
  and cardG: card (field-auto K (fixed-field K H)) = card H
  using artin-fixed-field-degree-data[OF finH] by blast
interpret GE: galois-extension K F
  by (rule galois-extension.intro[OF K Fsub FK])
show ?thesis
  using GE.le-galois-group-fixed-field Hsub cardG finG by (metis card-subset-eq)
qed

```

end

With *artin-theorem* the kernel operator of the Galois correspondence is the identity on finite subgroups. Together with the semi-inverse laws already in *Galois-Correspondence*, the correspondence restricts to a *bijection* between the finite subgroups of the Galois group and those intermediate fields that arise as fixed fields.

The classical finite correspondence is assembled downstream in *Galois-Finite-Correspondence*. There separability and normality show that every intermediate field is recovered from its relative automorphism group, while this theorem supplies the inverse law for every subgroup.

end

106 The fundamental theorem for finite Galois extensions

theory *Galois-Finite-Correspondence*

imports *Artin-Degree Galois-Degree Finite-Extension-Tower Galois-Restriction*
begin

Separability survives enlargement of the base field. The new minimal polynomial divides the old one, and root-squarefreeness passes to divisors.

theorem *finite-separable-extension-base-change*:

fixes $F E K :: 'a :: \text{field set}$
assumes $T: \text{finite-subfield-tower } F K$ **and** $\text{sep}: \text{separable-extension } K F$
and $\text{sfE}: \text{Subfield } E$ **and** $\text{FE}: F \subseteq E$ **and** $\text{EK}: E \subseteq K$
shows *separable-extension* $K E$
proof (*rule separable-extensionI*)
interpret $T: \text{finite-subfield-tower } F K$ **by** (*rule T*)
fix x **assume** $xK: x \in K$ **and** $\text{algE}: \text{algebraic-over } E x$
have $\text{algF}: \text{algebraic-over } F x$ **by** (*rule T.finite-extension-algebraic* [$OF xK$])
have $\text{sepF}: \text{rsquarefree } (\text{minpoly } F x)$
by (*rule separable-extensionD* [$OF \text{sep } xK \text{algF}$])
have $\text{minE}: \text{is-minpoly } E x (\text{minpoly } E x)$
by (*rule Subfield.is-minpoly-minpoly* [$OF \text{sfE algE}$])
have $\text{minpoly } F x \in \text{poly-over } E$
using $T.\text{base.minpoly-over}$ [$OF \text{algF}$] poly-over-mono [$OF FE$] **by** *blast*
then show $\text{rsquarefree } (\text{minpoly } E x)$
using $\text{Subfield.minpoly-dvd } T.\text{base.minpoly-root } \text{algF } \text{minE } \text{rsquarefree-dvd } \text{sepF } \text{sfE}$
by *blast*
qed

Normality also survives enlargement of the base inside a finite extension. The minimal polynomial over the larger base divides the old minimal polynomial, so every new conjugate is already an old conjugate.

theorem *finite-normal-extension-base-change*:

fixes $F E K :: 'a :: \text{field set}$
assumes $T: \text{finite-subfield-tower } F K$ **and** $\text{normal}: \text{normal-extension } K F$
and $\text{sfE}: \text{Subfield } E$ **and** $\text{FE}: F \subseteq E$ **and** $\text{EK}: E \subseteq K$
shows *normal-extension* $K E$
proof (*rule normal-extensionI*)
interpret $T: \text{finite-subfield-tower } F K$ **by** (*rule T*)
fix $p a$
assume $pE: p \in \text{poly-over } E$ **and** $\text{irr}: \text{irreducible-over } E p$
and $aK: a \in K$ **and** $pa: \text{poly } p a = 0$
have $\text{algF}: \text{algebraic-over } F a$
by (*rule T.finite-extension-algebraic* [$OF aK$])
have $\text{algE}: \text{algebraic-over } E a$
by (*rule algebraic-over-mono* [$OF \text{algF } FE$])
then have $\text{minE}: \text{is-minpoly } E a (\text{minpoly } E a)$
by (*rule Subfield.is-minpoly-minpoly* [$OF \text{sfE}$])

```

have minF-E: minpoly F a ∈ poly-over E
  using T.base.minpoly-over[OF algF] poly-over-mono[OF FE] by blast
have dvd: minpoly E a dvd minpoly F a
  by (rule Subfield.minpoly-dvd[OF sfE minE minF-E T.base.minpoly-root[OF algF]])
have poly-root-set (minpoly F a) ⊆ K
  using T.base.Subfield-axioms T.base.minpoly-over T.base.minpoly-root aK algF
  irreducible-over-minpoly normal normal-extensionD by blast
then show poly-root-set p ⊆ K
  using Subfield.irreducible-over-root-is-minpoly-root dvd irr minE pE pa sfE
  by fastforce
qed

```

106.1 The finite correspondence

context

```

fixes F K :: 'a :: alg-closed-field set
assumes finite: finite-subfield-tower F K
  and normal: normal-extension K F
  and separable: separable-extension K F
begin

```

interpretation T: finite-subfield-tower F K **by** (rule finite)

interpretation GE: galois-extension K F

```

  by (rule galois-extension.intro[OF T.ext.Subfield-axioms T.base.Subfield-axioms
    T.base-subset])

```

Any two conjugates over the base are related by an automorphism of a finite Galois extension. A primitive element presents the extension as the field generated by the roots of its minimal polynomial; multiplying that polynomial by the minimal polynomial lets the generic transitivity theorem act on both roots without changing the generated field.

theorem finite-galois-root-transfer:

```

  assumes aK: a ∈ K and mb: poly (minpoly F a) b = 0
  shows ∃σ∈field-auto K F. σ a = b

```

proof –

```

  have alga: algebraic-over F a

```

```

    by (rule T.finite-extension-algebraic[OF aK])

```

```

  obtain theta where thetaK: theta ∈ K and prim: primitive-element F K theta and
  algebraic-over F theta

```

```

    by (meson T.finite-extension-algebraic finite separable finite-separable-extension-is-simple)

```

```

  have algheta: algebraic-over F theta

```

```

    by (rule T.finite-extension-algebraic[OF thetaK])

```

```

  define q where q ≡ minpoly F theta

```

```

  define m where m ≡ minpoly F a

```

```

  define Phi where Phi ≡ q * m

```

```

  define R where R ≡ {r. poly Phi r = 0}

```

```

  obtain qF: q ∈ poly-over F and mF: m ∈ poly-over F

```

```

    by (simp add: T.base.minpoly-over alga algheta m-def q-def)

```



```

obtain  $q0$ :  $q \neq 0$  and  $m0$ :  $m \neq 0$ 
  using  $T.base.minpoly-nonzero$   $alga$   $algheta$   $m-def$   $q-def$  by  $presburger$ 
have  $\Phi F$ :  $\Phi \in poly-over\ F$ 
  unfolding  $\Phi-def$  by (rule  $T.base.poly-over-mult[OF\ qF\ mF]$ )
have  $\Phi 0$ :  $\Phi \neq 0$  unfolding  $\Phi-def$  using  $q0\ m0$  by  $simp$ 
have  $finR$ :  $finite\ R$ 
  unfolding  $R-def$  by (rule  $poly-roots-finite[OF\ \Phi 0]$ )
have  $groots$ :  $poly-root-set\ q \subseteq K$ 
  using  $T.base.Subfield-axioms$   $T.base.minpoly-root$   $algheta$   $irreducible-over-minpoly$ 
normal
   $normal-extensionD\ qF\ q-def\ thetaK$  by  $blast$ 
have  $mroots$ :  $poly-root-set\ m \subseteq K$ 
  using  $normal\ mF$  unfolding  $normal-extension-def\ m-def$ 
  using  $GE.Fsub\ T.base.minpoly-root\ aK$   $alga$   $irreducible-over-minpoly$  by  $blast$ 
have  $RsubK$ :  $R \subseteq K$ 
  using  $groots\ mroots$ 
  unfolding  $R-def\ \Phi-def\ poly-root-set-def$  by (auto  $simp$ :  $poly-mult$ )
have  $algR$ :  $\bigwedge r. r \in R \implies algebraic-over\ F\ r$ 
  by (rule  $T.finite-extension-algebraic$ ) (use  $RsubK$  in  $blast$ )
obtain  $thetaR$ :  $theta \in R$  and  $aR$ :  $a \in R$  and  $bR$ :  $b \in R$ 
  unfolding  $R-def\ \Phi-def\ q-def$ 
  by ( $simp$   $add$ :  $T.base.minpoly-root\ alga\ algheta\ m-def\ mb$ )
have  $Kgen$ :  $K = generate-field\ (F \cup R)$ 
proof
  show  $K \subseteq generate-field\ (F \cup R)$ 
    using  $primitive-elementD[OF\ prim]$   $T.base.eval-img-eq-generate-field[OF\ algheta]$ 
    by ( $simp$   $add$ :  $generate-field-mono\ thetaR$ )
  show  $generate-field\ (F \cup R) \subseteq K$ 
    using  $RsubK\ T.base-subset$  by ( $simp$   $add$ :  $T.ext.Subfield-axioms\ generate-field-least$ )

qed
have  $min-a$ :  $is-minpoly\ F\ a$  ( $minpoly\ F\ a$ )
  by (rule  $T.base.is-minpoly-minpoly[OF\ alga]$ )
obtain  $g$  where  $homg$ :  $field-hom-on\ (generate-field\ (F \cup R))\ g$ 
  and  $gab$ :  $g\ a = b$  and  $fixg$ :  $\forall x \in F. g\ x = x$ 
  and  $imageg$ :  $g\ ` generate-field\ (F \cup R) = generate-field\ (F \cup R)$ 
  using  $galois-transitive[OF\ T.base.Subfield-axioms\ finR\ algR]$ 
   $\Phi F\ R-def\ alg-closed-imp-poly-has-root\ aR\ bR\ min-a\ mb$ 
  by  $blast$ 
have  $aut$ :  $restrict\ g\ K \in field-auto\ K\ F$ 
  using  $Kgen\ field-hom-on-imp-field-auto\ fixg\ homg\ imageg$  by  $blast$ 
have  $restrict\ g\ K\ a = b$  using  $aK\ gab$  by  $simp$ 
then show  $?thesis$  using  $aut$  by  $blast$ 
qed

```

Every intermediate field is recovered as the fixed field of its relative Galois group. The upper extension remains finite, normal and separable. Its degree equals the order of the relative group; Artin's theorem computes the degree over that group's fixed field. The tower law then makes the intervening degree one.

theorem *finite-galois-fixed-field*:
assumes $E: E \in \text{inter-fields } K \ F$
shows $\text{fixed-field } K \ (\text{field-auto } K \ E) = E$
proof –
have $\text{sfE}: \text{Subfield } E$ **and** $\text{FE}: F \subseteq E$ **and** $\text{EK}: E \subseteq K$
using E **by** (*auto simp: inter-fields-iff*)
have $\text{TFE-raw}: \text{finite-subfield-tower } F \ E$
by (*rule finite-subfield-tower-intermediate-left[OF finite sfE FE EK]*)
then interpret $\text{TFE}: \text{finite-subfield-tower } F \ E$.
have $\text{TEK-raw}: \text{finite-subfield-tower } E \ K$
by (*rule finite-subfield-tower-intermediate-right[OF finite sfE FE EK]*)
then interpret $\text{TEK}: \text{finite-subfield-tower } E \ K$.
have $\text{normalEK}: \text{normal-extension } K \ E$
by (*rule finite-normal-extension-base-change[OF finite normal sfE FE EK]*)
have $\text{sepEK}: \text{separable-extension } K \ E$
by (*rule finite-separable-extension-base-change[OF finite separable sfE FE EK]*)

have $\text{Hsub-raw}: \text{Subgroup } (\text{field-auto } K \ E) \ (\text{field-auto } K \ F) \ (\text{compose } K) \ (\text{identity } K)$
by (*rule field-auto-subgroup-of-Galois-group[OF T.ext.Subfield-axioms T.base-subset E]*)
have $\text{Hsub}: \text{field-auto } K \ E \in \text{galois-subgroups } K \ F$
using Hsub-raw **by** (*simp add: galois-subgroups-iff*)
have $\text{Hle}: \text{field-auto } K \ E \subseteq \text{field-auto } K \ F$
using Hsub **by** (*rule galois-subgroups-subset*)
have $\text{finG}: \text{finite } (\text{field-auto } K \ F)$
by (*rule finite-normal-separable-field-auto[OF finite normal separable]*)
have $\text{finH}: \text{finite } (\text{field-auto } K \ E)$
by (*rule finite-subset[OF Hle finG]*)
have $\text{cardH}: \text{card } (\text{field-auto } K \ E) = \text{TEK.extension-degree}$
by (*rule finite-normal-separable-galois-degree[OF TEK-raw normalEK sepEK]*)

have $\text{towerFix}: \text{subfield-tower } (\text{fixed-field } K \ (\text{field-auto } K \ E)) \ K$
by (*rule fixed-field-tower[OF T.ext.Subfield-axioms Hle]*)
then interpret $\text{FixK}: \text{subfield-tower fixed-field } K \ (\text{field-auto } K \ E) \ K$.
obtain B **where** $\text{basis}: \text{FixK.vs.basis } B$
and $\text{cardB}: \text{card } B = \text{card } (\text{field-auto } K \ E)$
using *artin-fixed-field-degree-data* [**where** $K=K$ **and** $F=F$ **and** $H=\text{field-auto } K \ E$,
 $\text{OF } T.\text{ext.Subfield-axioms } T.\text{base.Subfield-axioms } T.\text{base-subset } \text{Hsub } \text{finH}$]
by *blast*
interpret $\text{TFixK}: \text{finite-subfield-tower fixed-field } K \ (\text{field-auto } K \ E) \ K$
by (*fast intro!: towerFix finite-subfield-tower.intro finite-subfield-tower-axioms.intro basis*)
have $\text{degreeFix}: \text{TFixK.extension-degree} = \text{card } (\text{field-auto } K \ E)$
using $\text{TFixK.extension-degree-eq-card-basis}$ [*OF basis*] cardB **by** *simp*

have $\text{EFix}: E \subseteq \text{fixed-field } K \ (\text{field-auto } K \ E)$
by (*rule GE.le-fixed-field-galois-group[OF E]*)
have $\text{sfFix}: \text{Subfield } (\text{fixed-field } K \ (\text{field-auto } K \ E))$

```

    by (rule fixed-field-subfield[OF T.ext.Subfield-axioms Hle])
  have TEFix-raw: finite-subfield-tower E (fixed-field K (field-auto K E))
    by (rule finite-subfield-tower-intermediate-left[OF TEK-raw sfFix EFix fixed-field-subset])
  then interpret TEFix: finite-subfield-tower E fixed-field K (field-auto K E) .
  interpret C: finite-subfield-tower-chain E fixed-field K (field-auto K E) K
    by (simp add: TEFix-raw TFixK.finite-subfield-tower-axioms finite-subfield-tower-chain-def)
  have cancel-eq: TFixK.extension-degree * 1 = TFixK.extension-degree * TEFix.extension-degree
    using C.extension-degree-tower-law-exists cardH degreeFix by (simp add: mult.commute)
  have degree-one: TEFix.extension-degree = 1
    using cancel-eq TFixK.extension-degree-pos nat-mult-eq-cancel1 by simp

  have sepEFix: separable-extension (fixed-field K (field-auto K E)) E
    using sepEK by (meson fixed-field-memD separable-extension-def)
  obtain theta where prim: primitive-element E (fixed-field K (field-auto K E)) theta
    and algTheta: algebraic-over E theta
    using TEFix.finite-extension-algebraic TEFix-raw finite-separable-extension-is-simple
  sepEFix
    by blast
  have ext-one: ext-degree E theta = 1
    using TEFix.primitive-element-degree[OF prim] degree-one by simp
  have thetaE: theta ∈ E
    using ext-one Subfield.ext-degree-eq-1-iff[OF sfE algTheta] by blast
  have eval-img E theta = generate-field (E ∪ {theta})
    by (rule Subfield.eval-img-eq-generate-field[OF sfE algTheta])
  also have ... = E
    by (simp add: TFE.ext.generate-field-self insert-absorb thetaE)
  finally have eval-img E theta = E .
  then show ?thesis using primitive-elementD[OF prim] by simp
qed

```

Conversely, a subgroup of the finite Galois group is recovered from its fixed field by Artin's theorem.

```

theorem finite-galois-fixed-group:
  assumes H: H ∈ galois-subgroups K F
  shows field-auto K (fixed-field K H) = H
proof -
  have finG: finite (field-auto K F)
    by (rule finite-normal-separable-field-auto[OF finite normal separable])
  have finH: finite H
    using assms finG by (metis finite-subset galois-subgroups-subset)
  show ?thesis
    by (rule artin-theorem[OF T.ext.Subfield-axioms T.base.Subfield-axioms
      T.base-subset H finH])
qed

```

The two antitone maps are therefore mutually inverse bijections.

```

theorem finite-galois-field-group-bij:
  bij-betw (field-auto K) (inter-fields K F) (galois-subgroups K F)
proof (rule bij-betw-byWitness[where f'=fixed-field K])

```

```

show  $\forall E \in \text{inter-fields } K \ F. \text{ fixed-field } K \ (\text{field-auto } K \ E) = E$ 
  using finite-galois-fixed-field by blast
show  $\forall H \in \text{galois-subgroups } K \ F. \text{ field-auto } K \ (\text{fixed-field } K \ H) = H$ 
  using finite-galois-fixed-group by blast
show  $\text{field-auto } K \ ' \text{ inter-fields } K \ F \subseteq \text{galois-subgroups } K \ F$ 
  using field-auto-subgroup-of-Galois-group[OF T.ext.Subfield-axioms T.base-subset]
  by (auto simp: galois-subgroups-iff)
show  $\text{fixed-field } K \ ' \text{ galois-subgroups } K \ F \subseteq \text{inter-fields } K \ F$ 
  using GE.fixed-field-in-inter-fields by blast
qed

```

theorem *finite-galois-group-field-bij*:

```

bij-betw (fixed-field K) (galois-subgroups K F) (inter-fields K F)
proof (rule bij-betw-byWitness[where  $f' = \text{field-auto } K$ ])
  show  $\forall H \in \text{galois-subgroups } K \ F. \text{ field-auto } K \ (\text{fixed-field } K \ H) = H$ 
    using finite-galois-fixed-group by blast
  show  $\forall E \in \text{inter-fields } K \ F. \text{ fixed-field } K \ (\text{field-auto } K \ E) = E$ 
    using finite-galois-fixed-field by blast
  show  $\text{fixed-field } K \ ' \text{ galois-subgroups } K \ F \subseteq \text{inter-fields } K \ F$ 
  using finite-galois-fixed-group
    using GE.fixed-field-in-inter-fields by blast
  show  $\text{field-auto } K \ ' \text{ inter-fields } K \ F \subseteq \text{galois-subgroups } K \ F$ 
    using field-auto-subgroup-of-Galois-group[OF T.ext.Subfield-axioms T.base-subset]
    by (auto simp: galois-subgroups-iff)
qed

```

theorem *finite-galois-correspondence*:

```

bij-betw (field-auto K) (inter-fields K F) (galois-subgroups K F)  $\wedge$ 
  bij-betw (fixed-field K) (galois-subgroups K F) (inter-fields K F)
  using finite-galois-field-group-bij finite-galois-group-field-bij by blast

```

106.2 Degree and index formulas

context

fixes $E :: 'a \text{ set}$

assumes *intermediate*: $E \in \text{inter-fields } K \ F$

begin

lemma *intermediate-data*:

```

Subfield  $E \wedge F \subseteq E \wedge E \subseteq K$ 
  using intermediate by (simp add: inter-fields-iff)

```

lemma *TFE-raw: finite-subfield-tower* $F \ E$

```

  using intermediate-data
  by (blast intro: finite-subfield-tower-intermediate-left[OF finite])

```

lemma *TEK-raw: finite-subfield-tower* $E \ K$

```

  using intermediate-data
  by (blast intro: finite-subfield-tower-intermediate-right[OF finite])

```

interpretation *TFE*: *finite-subfield-tower* $F E$ **by** (rule *TFE-raw*)
interpretation *TEK*: *finite-subfield-tower* $E K$ **by** (rule *TEK-raw*)

lemma *normalEK*: *normal-extension* $K E$
using *intermediate-data*
by (blast intro: *finite-normal-extension-base-change*[*OF* *finite normal*])

lemma *separableEK*: *separable-extension* $K E$
using *intermediate-data*
by (blast intro: *finite-separable-extension-base-change*[*OF* *finite separable*])

lemma *relative-subgroup*:
Subgroup (*field-auto* $K E$) (*field-auto* $K F$) (*compose* K) (*identity* K)
by (rule *field-auto-subgroup-of-Galois-group*[*OF* *T.ext.Subfield-axioms* *T.base-subset* *intermediate*])

interpretation *S*: *subgroup-of-group* *field-auto* $K E$ *field-auto* $K F$
compose K *identity* K
by (*simp add*: *T.base-subset* *T.ext.Subfield-axioms* *field-auto-group* *relative-subgroup* *subgroup-of-group.intro*)

The relative Galois group has order equal to the degree of the upper extension.

theorem *finite-galois-relative-group-order*:
card (*field-auto* $K E$) = *TEK.extension-degree*
by (rule *finite-normal-separable-galois-degree*[*OF* *TEK-raw* *normalEK* *separableEK*])

Its index in the full Galois group is the degree of the lower extension.

theorem *finite-galois-relative-group-index*:
S.index = *TFE.extension-degree*
proof –
have *finG*: *finite* (*field-auto* $K F$)
by (rule *finite-normal-separable-field-auto*[*OF* *finite normal separable*])
have *global-order*: *card* (*field-auto* $K F$) = *T.extension-degree*
by (rule *finite-normal-separable-galois-degree*[*OF* *finite normal separable*])
interpret *C*: *finite-subfield-tower-chain* $F E K$
by (rule *finite-subfield-tower-chain.intro*[*OF* *TFE-raw* *TEK-raw*])
have *cancel-eq*: *TEK.extension-degree* * *TFE.extension-degree* = *TEK.extension-degree*
* *S.index*
using *C.extension-degree-tower-law-exists* *S.lagrange* *finG* *finite-galois-relative-group-order*
global-order **by** *force*
show *?thesis*
using *cancel-eq* *TEK.extension-degree-pos* *nat-mult-eq-cancel1* **by** *simp*
qed

theorem *finite-galois-relative-degree-index*:
card (*field-auto* $K E$) = *TEK.extension-degree* $\wedge$
S.index = *TFE.extension-degree*
using *finite-galois-relative-group-order* *finite-galois-relative-group-index* **by** *blast*

106.3 Normal intermediate fields

An intermediate field is normal over the base exactly when every automorphism of the ambient finite Galois extension preserves it setwise. Some duplication with a later proof.

theorem *finite-galois-normal-iff-stable*:

normal-extension $E F \longleftrightarrow$
 $(\forall \sigma \in \text{field-auto } K F. \sigma \text{ ' } E = E)$

proof (*intro iffI strip*)

assume *normalE*: *normal-extension* $E F$

have *image-subset*: $\sigma \text{ ' } E \subseteq E$ **if** *sigma*: $\sigma \in \text{field-auto } K F$ **for** σ

proof

fix y **assume** $y \in \sigma \text{ ' } E$

then obtain x **where** $x \in E$ **and** $y = \sigma x$ **by** *blast*

have xK : $x \in K$ **using** *intermediate-data* $x \in E$ **by** *blast*

have $\text{alg}x$: *algebraic-over* $F x$

by (*rule* *TFE.finite-extension-algebraic*[*OF* $x \in E$])

have $\text{roots}E$: *poly-root-set* (*minpoly* $F x$) $\subseteq E$

using *GE.Fsub* $T.\text{base.minpoly-over } T.\text{base.minpoly-root } \text{alg}x$ *irreducible-over-minpoly*

normalE

normal-extensionD $x \in E$ **by** *blast*

have hom : *field-hom-on* $K \sigma$

by (*rule* *field-auto-as-hom*[*OF* $T.\text{ext.Subfield-axioms } \sigma$])

have $\text{fix}F$: $\bigwedge z. z \in F \implies \sigma z = z$

using *sigma* **by** (*auto simp*: *field-auto-mem-iff*)

have *image-root*: *poly* (*minpoly* $F x$) (σx) = 0

using *GE.Fsub* $T.\text{base.minpoly-over } T.\text{base.minpoly-root } T.\text{base-subset } \text{alg}x$ $\text{fix}F$

hom xK

by (*meson* *hom-preserves-roots*)

have $\sigma x \in \text{poly-root-set } (\text{minpoly } F x)$

using *image-root* **unfolding** *poly-root-set-def* **by** *simp*

then show $y \in E$ **using** $\text{roots}E y$ **by** *blast*

qed

show $\sigma \text{ ' } E = E$ **if** *sigma*: $\sigma \in \text{field-auto } K F$ **for** σ

proof –

define τ **where** $\tau \equiv \text{restrict } (\text{inv-into } K \sigma) K$

have τG : $\tau \in \text{field-auto } K F$

by (*simp add*: $T.\text{base-subset } T.\text{ext.Subfield-axioms } \text{field-auto-inverse } \sigma$

tau-def)

show $\sigma \text{ ' } E = E$

proof (*intro antisym subsetI*)

fix y **assume** $y \in E$

have yK : $y \in K$ **using** *intermediate-data* $y \in E$ **by** *blast*

have $y \in \sigma \text{ ' } K$

using *sigma* yK **by** (*metis* *bij-betw-imp-surj-on* *field-auto-mem-iff*)

then have *sigma-tau*: $\sigma (\tau y) = y$

unfolding *tau-def* **using** yK **by** (*simp add*: *f-inv-into-f*)

then show $y \in \sigma \text{ ' } E$

unfolding *image-iff* **by** (*metis* *image-subset* $\tauG y \in \text{image-subset-iff}$)

```

    qed (use image-subset sigma in auto)
  qed
next
  assume stable:  $\forall \sigma \in \text{field-auto } K F. \sigma \cdot E = E$ 
  show normal-extension E F
  proof (rule normal-extensionI)
    fix p a
    assume pF:  $p \in \text{poly-over } F$  and irr:  $\text{irreducible-over } F p$ 
    and aE:  $a \in E$  and pa:  $\text{poly } p a = 0$ 
    have aK:  $a \in K$  using intermediate-data aE by blast
    show poly-root-set  $p \subseteq E$ 
    proof
      fix b assume bR:  $b \in \text{poly-root-set } p$ 
      have min-a:  $\text{is-minpoly } F a$  (minpoly F a)
      using T.base.is-minpoly-minpoly T.finite-extension-algebraic aK by blast
      have min-b:  $\text{poly } (\text{minpoly } F a) b = 0$ 
      using T.base.irreducible-over-root-is-minpoly-root bR irr min-a pF pa by auto
      then show  $b \in E$ 
      using aE stable finite-galois-root-transfer[OF aK min-b] by fastforce
    qed
  qed
qed

```

Setwise stability is also exactly normality of the corresponding subgroup. A lot of duplication with the similar theorem below.

```

theorem finite-galois-normal-subgroup-iff-stable:
  normal-subgroup (field-auto K E) (field-auto K F) (compose K) (identity K)  $\longleftrightarrow$ 
  ( $\forall \sigma \in \text{field-auto } K F. \sigma \cdot E = E$ )
proof (intro iffI strip)
  assume normalH:
    normal-subgroup (field-auto K E) (field-auto K F) (compose K) (identity K)
  interpret N: normal-subgroup field-auto K E field-auto K F
  compose K identity K by (rule normalH)
  have fixedE: fixed-field K (field-auto K E) = E
  by (rule finite-galois-fixed-field[OF intermediate])
  have image-subset:  $\sigma \cdot E \subseteq E$  if sigma:  $\sigma \in \text{field-auto } K F$  for  $\sigma$ 
  proof
    fix y assume y  $\in \sigma \cdot E$ 
    then obtain x where xE:  $x \in E$  and y:  $y = \sigma x$  by blast
    have xFix:  $x \in \text{fixed-field } K$  (field-auto K E) using xE fixedE by simp
    have xK:  $x \in K$  using fixed-field-memD[OF xFix] by blast
    have sigxK:  $\sigma x \in K$ 
    using sigma xK by (auto simp: field-auto-mem-iff PiE-iff)
    have sigxFix:  $\sigma x \in \text{fixed-field } K$  (field-auto K E)
  proof (rule fixed-field-memI[OF sigxK])
    fix h assume hH:  $h \in \text{field-auto } K E$ 
    have cosets:
      coset-notation.Left-Coset (compose K)  $\sigma$  (field-auto K E) =
      coset-notation.Right-Coset (compose K) (field-auto K E)  $\sigma$ 

```

```

    by (rule N.Left-equals-Right-coset[OF sigma])
  have compose K h sigma ∈ coset-notation.Right-Coset (compose K) (field-auto K E)
σ
    using hH by blast
  then obtain k where kH: k ∈ field-auto K E and commute: compose K sigma k =
compose K h sigma
    unfolding coset-notation.Left-Coset-def
    using S.Left-Coset-memE cosets by metis
  then show h (sigma x) = sigma x
    by (metis fixedE xE compose-eq fixed-field-memD)
  qed
  show y ∈ E using sigxFix fixedE y by simp
  qed
  show sigma ' E = E if sigma: sigma ∈ field-auto K F for sigma
  proof -
    define tau where tau ≡ restrict (inv-into K sigma) K
    have tauG: tau ∈ field-auto K F
      by (simp add: T.base-subset T.ext.Subfield-axioms field-auto-inverse sigma
tau-def)
    show sigma ' E = E
    proof
      show E ⊆ sigma ' E
      proof
        fix y assume yE: y ∈ E
        have yK: y ∈ K using intermediate-data yE by blast
        have y ∈ sigma ' K
          using sigma yK by (metis bij-betw-imp-surj-on field-auto-mem-iff)
        then have sigma-tau: sigma (tau y) = y
          unfolding tau-def using yK by (simp add: f-inv-into-f)
        then show y ∈ sigma ' E
          unfolding image-iff by (metis image-subset tauG yE image-subset-iff)
      qed
      show sigma ' E ⊆ E
        by (simp add: image-subset sigma)
    qed
  qed
next
  assume stable: ∀ sigma ∈ field-auto K F. sigma ' E = E
  interpret R: group-homomorphism galois-restriction K E F
    field-auto K F compose K identity K
    field-auto E F compose E identity E
  by (simp add: T.ext.Subfield-axioms galois-restriction-def galois-restriction-hom
intermediate-data stable)
  have ker: R.Ker = field-auto K E
    unfolding galois-restriction-def
  by (simp add: T.ext.Subfield-axioms galois-restriction-Ker intermediate-data stable)
  then show normal-subgroup (field-auto K E) (field-auto K F) (compose K) (identity
K)
    using R.kernel.normal-subgroup-axioms by presburger

```


qed

This is the normal-field/normal-subgroup clause of the fundamental theorem.

theorem *finite-galois-normal-iff-normal-subgroup*:

normal-extension $E F \longleftrightarrow$

normal-subgroup (*field-auto* $K E$) (*field-auto* $K F$) (*compose* K) (*identity* K)

using *finite-galois-normal-iff-stable* *finite-galois-normal-subgroup-iff-stable* **by** *blast*

106.4 Restriction and the quotient theorem

Restriction onto a stable intermediate field is surjective. A primitive element for the full extension supplies a finite root-generated presentation over the intermediate field, so the generic homomorphism-extension theorem applies.

theorem *finite-galois-restriction-image*:

assumes *stable*: $\bigwedge \sigma. \sigma \in \text{field-auto } K F \implies \sigma ' E = E$

shows *galois-restriction* $K E F ' \text{field-auto } K F = \text{field-auto } E F$

proof –

have *sfE*: *Subfield* E **and** *FE*: $F \subseteq E$ **and** *EK*: $E \subseteq K$

using *intermediate-data* **by** *blast+*

obtain *theta* **where** *thetaK*: $\text{theta} \in K$ **and** *prim*: *primitive-element* $F K \text{ theta}$

using *finite-separable-extension-is-simple*[*OF* *finite separable*] **by** *blast*

have *algheta*: *algebraic-over* $F \text{ theta}$

by (*rule* *T.finite-extension-algebraic*[*OF* *thetaK*])

define p **where** $p \equiv \text{minpoly } F \text{ theta}$

define R **where** $R \equiv \{r. \text{poly } p r = 0\}$

obtain *pF*: $p \in \text{poly-over } F$ **and** *p0*: $p \neq 0$

using *T.base.minpoly-nonzero* *T.base.minpoly-over* *algheta* *p-def* **by** *presburger*

have *p0*: $p \neq 0$

unfolding *p-def* **by** (*rule* *T.base.minpoly-nonzero*[*OF* *algheta*])

have *rootsK*: $R \subseteq K$

using *irreducible-over-minpoly* *normal-extensionD* [*OF* *normal* *pF*]

using *R-def* *T.base.Subfield-axioms* *T.base.minpoly-root* *algheta* *p-def* *thetaK* **by**

fastforce

have *thetaR*: $\text{theta} \in R$

unfolding *R-def* *p-def* **using** *T.base.minpoly-root*[*OF* *algheta*] **by** *simp*

have *Kgen*: $K = \text{generate-field } (E \cup R)$

proof (*rule* *subset-antisym*)

have $K = \text{generate-field } (F \cup \{\text{theta}\})$

using *T.base.eval-img-eq-generate-field* *algheta* *prim* *primitive-elementD* **by** *blast*

also have $\dots \subseteq \text{generate-field } (E \cup R)$

by (*rule* *generate-field-mono*) (*use* *FE* *thetaR* **in** *auto*)

finally show $K \subseteq \text{generate-field } (E \cup R)$.

show *generate-field* $(E \cup R) \subseteq K$

by (*simp* *add*: *T.ext.Subfield-axioms* *generate-field-least* *local.EK* *rootsK*)

qed

have *algR*: $\bigwedge r. \text{poly } p r = 0 \implies \text{algebraic-over } E r$

using *R-def* *TEK.finite-extension-algebraic* *rootsK* **by** *blast*

have *Kgen'*: $K = \text{generate-field } (E \cup \{r. \text{poly } p r = 0\})$

using *Kgen* **unfolding** *R-def* .

```

have finR': finite {r. poly p r = 0}
  by (simp add: p0 poly-roots-finite)
have image-subset: galois-restriction K E F ' field-auto K F  $\subseteq$  field-auto E F
  using T.ext.Subfield-axioms gal-res-mem intermediate-data stable by auto
show ?thesis
proof
  show field-auto E F  $\subseteq$  galois-restriction K E F ' field-auto K F
  proof
    fix tau assume tau: tau  $\in$  field-auto E F
    obtain sigma where sigma: sigma  $\in$  field-auto K F
    and agree:  $\forall x \in E. \text{sigma } x = \text{tau } x$ 
    using galois-restriction-surjective[OF sfE T.base.Subfield-axioms FE pF
      Kgen' finR' algR alg-closed-imp-poly-has-root tau]
    by blast
    have restrict-eq': restrict sigma E = restrict tau E
    by (rule restrict-ext) (use agree in simp)
    have tauPiE: tau  $\in$  E  $\rightarrow_E$  E
    using tau by (simp only: field-auto-mem-iff)
    have restrict-tau: restrict tau E = tau by (rule PiE-restrict[OF tauPiE])
    show tau  $\in$  galois-restriction K E F ' field-auto K F
    using sigma sigma restrict-eq' restrict-tau by auto
  qed
qed (use image-subset in blast)
qed

context
  assumes normal-intermediate: normal-extension E F
begin

lemma stable-intermediate:
   $\bigwedge \sigma. \sigma \in \text{field-auto } K F \implies \sigma ' E = E$ 
  using iffD1[OF finite-galois-normal-iff-stable normal-intermediate] by blast

lemma restriction-image:
  galois-restriction K E F ' field-auto K F = field-auto E F
  by (rule finite-galois-restriction-image[OF stable-intermediate])

interpretation restriction: group-epimorphism galois-restriction K E F
  field-auto K F compose K identity K
  field-auto E F compose E identity E
  using galois-restriction-epimorphism[OF T.ext.Subfield-axioms]
  using intermediate-data restriction-image stable-intermediate by blast

theorem finite-galois-restriction-kernel: restriction.Ker = field-auto K E
  using intermediate-data
  unfolding galois-restriction-def
  by (simp add: T.ext.Subfield-axioms galois-restriction-Ker stable-intermediate)

```

The first isomorphism theorem gives $\text{Gal}(K/F) / \text{Gal}(K/E) \cong \text{Gal}(E/F)$; the preceding kernel theorem identifies the factor-group kernel with the relative

Galois group literally.

theorem *finite-galois-quotient-isomorphism:*

(*restriction.kernel.Factor-Group*, *restriction.kernel.quotient-composition*, *restriction.kernel.Class*
(*identity K*))

$\cong_G$

(*field-auto E F*, *compose E*, *identity E*)

using *restriction.first-isomorphism restriction-image* **by** *argo*

end

end

end

end

107 Radical extensions and solvability by radicals

theory *Radical-Extension*

imports *Simple-Extension*

begin

This theory begins the *forward* Abel–Ruffini direction (phase R1: the definitions). It fixes the notion of *solvability by radicals* over a base field, on the same generic '*a/generate-field*' foundation used for the quintic. Later phases will connect it to solvability of the Galois group (via the Galois correspondence, cyclotomic and Kummer theory). Here we only make the target statement expressible.

107.1 Radical steps and towers

A single *radical step* from F to F' adjoins an element a some power of which already lies in F : $F' = \text{generate-field } (F \cup \{a\})$ with $a^n \in F$ for some $0 < n$.

definition *radical-step* :: ' a :: field set $\Rightarrow$ ' a set $\Rightarrow$ bool

where *radical-step* $F F' \equiv (\exists a n. n > 0 \wedge a \wedge n \in F \wedge F' = \text{generate-field } (F \cup \{a\}))$

A *radical tower* is a list of fields, each obtained from the previous by a radical step. *radical-tower* $F Fs$ says the tower starting at F and following Fs is valid (each consecutive pair is a radical step).

fun *radical-tower* :: ' a :: field set $\Rightarrow$ ' a set list $\Rightarrow$ bool **where**

radical-tower $F [] \longleftrightarrow \text{True}$

| *radical-tower* $F (G \# Gs) \longleftrightarrow \text{radical-step } F G \wedge \text{radical-tower } G Gs$

The top field of a tower (the last field, or the base if the tower is empty).

definition *tower-top* :: 'a :: field set $\Rightarrow$ 'a set list $\Rightarrow$ 'a set
where *tower-top* *F* *Fs* = last (*F* # *Fs*)

lemma *tower-top-Nil* [*simp*]: *tower-top* *F* [] = *F*
by (*simp* add: *tower-top-def*)

lemma *tower-top-Cons* [*simp*]: *tower-top* *F* (*G* # *Gs*) = *tower-top* *G* *Gs*
by (*simp* add: *tower-top-def*)

107.2 Solvability by radicals

A set *S* (typically the roots of a polynomial, together with the base field) is *solvable by radicals* over *F* when there is a radical tower over *F* whose top field contains *S*.

definition *solvable-by-radicals* :: 'a :: field set $\Rightarrow$ 'a set $\Rightarrow$ bool
where *solvable-by-radicals* *F* *S* $\longleftrightarrow (\exists F_s. \text{radical-tower } F F_s \wedge S \subseteq \text{tower-top } F F_s)$

The base field itself is trivially solvable by radicals (the empty tower).

lemma *solvable-by-radicals-base*:
assumes $S \subseteq F$ **shows** *solvable-by-radicals* *F* *S*
unfolding *solvable-by-radicals-def*
using *assms* **by** (*auto* intro!: *exI*[of - []])

A single radical step gives a one-element tower.

lemma *solvable-by-radicals-step*:
assumes $n > 0$ **and** $a \wedge n \in F$ **and** $S \subseteq \text{generate-field } (F \cup \{a\})$
shows *solvable-by-radicals* *F* *S*
unfolding *solvable-by-radicals-def*
proof (*intro* *exI*[of - [generate-field (*F* $\cup$ {*a*})]] *conjI*)
show *radical-tower* *F* [generate-field (*F* $\cup$ {*a*})]
using *assms* **by** (*auto* *simp*: *radical-step-def*)
show $S \subseteq \text{tower-top } F$ [generate-field (*F* $\cup$ {*a*})] **using** *assms*(3) **by** *simp*
qed

107.3 Concatenation of radical towers

The top of a concatenated tower is the top of the second part (or of the first, if the second is empty).

lemma *tower-top-append*:
tower-top *F* (*Fs* @ *Gs*) = *tower-top* (*tower-top* *F* *Fs*) *Gs*
by (*simp* add: *tower-top-def* *last-append*)

Radical towers concatenate. A radical tower over *F* followed by a radical tower over its top field *tower-top* *F* *Fs* yields a radical tower over *F* along the appended list. This is the splicing primitive used to build a compositum of radical extensions into a single tower.

lemma *radical-tower-append*:

radical-tower F $Fs \implies$ *radical-tower* (*tower-top* F Fs) $Gs \implies$ *radical-tower* F (Fs
@ Gs)
by (*induction* Fs *arbitrary*: F) *auto*

Iterated concatenation. Given a list of towers, each valid over the top reached so far, their concatenation is a single radical tower. We phrase validity of a list of towers recursively.

fun *radical-tower-list* :: 'a :: field set $\Rightarrow$ 'a set list list $\Rightarrow$ bool **where**
 radical-tower-list F [] $\longleftrightarrow$ True
| *radical-tower-list* F ($Ts \# Tss$) $\longleftrightarrow$
 radical-tower F $Ts \wedge$ *radical-tower-list* (*tower-top* F Ts) Tss

lemma *radical-tower-concat*:

radical-tower-list F $Tss \implies$ *radical-tower* F (*concat* Tss)
by (*induction* Tss *arbitrary*: F) (*auto simp*: *radical-tower-append*)

lemma *tower-top-concat*:

radical-tower-list F $Tss \implies$ *tower-top* F (*concat* Tss) = *foldl* *tower-top* F Tss
by (*induction* Tss *arbitrary*: F) (*auto simp*: *tower-top-append*)

107.4 Base change: lifting a radical tower to a larger base

Adjoining a fixed set E to every field of a radical tower over F yields a radical tower over *generate-field* ($E \cup F$): each step $H \rightsquigarrow$ *generate-field* ($H \cup \{a\}$) becomes *generate-field* ($E \cup H$) $\rightsquigarrow$ *generate-field* ($E \cup$ *generate-field* ($H \cup \{a\}$)), which by the collapse lemmas is *generate-field* (*generate-field* ($E \cup H$) $\cup \{a\}$) with the same radical a (its power $a^n \in H \subseteq$ *generate-field* ($E \cup H$)). This is what lets conjugate towers, each based at F , be spliced onto a growing compositum.

lemma *radical-step-basechange*:

assumes *radical-step* F G
shows *radical-step* (*generate-field* ($E \cup F$)) (*generate-field* ($E \cup G$))

proof –

obtain a n **where** $n: n > 0$ **and** $anF: a^n \in F$ **and** $Gdef: G =$ *generate-field* ($F \cup \{a\}$)

using *assms* **by** (*auto simp*: *radical-step-def*)

have $a^n \in$ *generate-field* ($E \cup F$) **using** anF *subset-generate-field* **by** *blast*

moreover have *generate-field* ($E \cup G$) = *generate-field* (*generate-field* ($E \cup F$) $\cup \{a\}$)

by (*metis* $Gdef$ *Un-commute generate-field-Un-collapse2 sup.assoc*)

ultimately show *?thesis* **unfolding** *radical-step-def* **using** n **by** *blast*

qed

Hence base change of a whole radical tower: mapping each field H to *generate-field* ($E \cup H$) sends a radical tower over F to a radical tower over *generate-field* ($E \cup F$).

lemma *radical-tower-basechange*:

```

    radical-tower  $F$   $Fs \implies$  radical-tower (generate-field ( $E \cup F$ )) (map ( $\lambda H$ . generate-field ( $E \cup H$ ))  $Fs$ )
  by (induction  $Fs$  arbitrary:  $F$ ) (auto simp: radical-step-basechange)
end

```

108 Stability of root-generated intermediate fields

```

theory Normal-Closure
  imports Galois-Restriction Radical-Extension
begin

```

The *solvable-tower* predicate of *Galois-Solvable-Tower* requires each intermediate field to be *stable* under the ambient Galois group. The reusable fact that supplies this is: a field *generate-field* ($F \cup X$) generated over F by the *full* root set X of an F -polynomial is stable under every F -automorphism of any ambient field K containing it.

108.1 An automorphism fixes the base field setwise

An F -automorphism of K maps F onto F : it fixes F pointwise, and every element of F is the image of itself.

```

lemma field-auto-image-base:
  assumes  $s$ :  $\sigma \in \text{field-auto } K F$ 
  shows  $\sigma \text{ ' } F = F$ 
proof -
  have base-fix:  $\forall x. x \in F \longrightarrow \sigma x = x$ 
    using  $s$  by (auto simp: field-auto-mem-iff)
  have image-subset:  $\sigma \text{ ' } F \subseteq F$ 
proof (rule subsetI)
  fix  $y$ 
  assume  $y$ -image:  $y \in \sigma \text{ ' } F$ 
  obtain  $x$  where  $xF$ :  $x \in F$  and  $y$ -eq:  $\sigma x = y$ 
    using  $y$ -image by blast
  have sigma-fix:  $\sigma x = x$  using base-fix  $xF$  by blast
  then show  $y \in F$  using  $xF$   $y$ -eq by simp
qed
have subset-image:  $F \subseteq \sigma \text{ ' } F$ 
proof (rule subsetI)
  fix  $x$ 
  assume  $xF$ :  $x \in F$ 
  have  $\sigma x \in \sigma \text{ ' } F$  using  $xF$  by (rule imageI)
  then show  $x \in \sigma \text{ ' } F$  using base-fix  $xF$  by simp
qed
show ?thesis by (rule subset-antisym[OF image-subset subset-image])
qed

```

108.2 Root-generated fields are stable

The stability lemma. Let X be the full set of roots (in the ambient K) of an F -polynomial Phi , and let $E = \text{generate-field } (F \cup X)$ be the field it generates over F . Then E is stable under every F -automorphism of K .

theorem *root-generated-stable:*

```

fixes  $K\ F :: \text{complex set}$  and  $\text{Phi} :: \text{complex poly}$ 
assumes  $K$ : Subfield  $K$  and  $\text{sf}F$ : Subfield  $F$  and  $FK$ :  $F \subseteq K$ 
and  $\text{Phi}$ :  $\text{Phi} \in \text{poly-over } F$ 
and  $X\text{def}$ :  $X = \{r. \text{poly } \text{Phi } r = 0\}$  and  $XK$ :  $X \subseteq K$  and  $\text{fin}X$ : finite  $X$ 
and  $s$ :  $\sigma \in \text{field-auto } K\ F$ 
shows  $\sigma \text{ ' generate-field } (F \cup X) = \text{generate-field } (F \cup X)$ 
proof –
  have  $\text{hom}$ : field-hom-on  $K\ \sigma$  using  $K\ s$  by (rule field-auto-imp-field-hom-on)
  —  $\sigma$  permutes the roots and fixes the base, so it fixes the generating set.
  have  $sX$ :  $\sigma \text{ ' } X = X$ 
  using field-auto-bij-on-root[ $OF\ K\ \text{sf}F\ FK\ \text{Phi}\ X\text{def}\ XK\ \text{fin}X\ s$ ] by (rule bij-betw-imp-surj-on)
  have  $\text{sf}F$ :  $\sigma \text{ ' } F = F$  by (rule field-auto-image-base[ $OF\ s$ ])
  have  $sUn$ :  $\sigma \text{ ' } (F \cup X) = F \cup X$ 
  using  $\text{sf}\ sX$  by (simp add: image-Un)
  — The generating set lies in  $K$ , and generation commutes with the homomorphism.
  have  $F \cup X \subseteq K$  using  $FK\ XK$  by simp
  then show ?thesis
  using field-hom-on.image-generate-field-eq[ $OF\ \text{hom}$ ]  $sUn$  by presburger
qed

```

108.3 A field homomorphism carries radical steps to radical steps

A conjugate of a radical is a radical

lemma *radical-step-image:*

```

fixes  $f :: \text{complex} \Rightarrow \text{complex}$ 
assumes  $\text{hom}$ : field-hom-on  $K\ f$  and  $\text{sf}F$ : Subfield  $F$  and  $FK$ :  $F \subseteq K$ 
and  $\text{fix}F$ :  $\bigwedge x. x \in F \implies f\ x = x$ 
and  $\text{step}$ : radical-step  $F\ G$  and  $GK$ :  $G \subseteq K$ 
shows radical-step  $F\ (f \text{ ' } G)$ 
proof –
  obtain  $a\ n$  where  $n$ :  $n > 0$  and  $\text{an}F$ :  $a \wedge^n \in F$  and  $G\text{def}$ :  $G = \text{generate-field } (F \cup \{a\})$ 
  using  $\text{step}$  by (auto simp: radical-step-def)
  —  $a$  lies in  $G \subseteq K$ , and  $F$  is fixed setwise.
  have  $aK$ :  $a \in K$  using  $G\text{def}\ GK\ \text{subset-generate-field}$  by fastforce
  — The image of the radical is again a radical:  $(f\ a)^n \in F$ .
  have  $\text{fan}F$ :  $(f\ a) \wedge^n \in F$ 
  using  $aK\ \text{field-hom-on.hom-power}$ [ $OF\ \text{hom}$ ]  $\text{an}F\ \text{fix}F$  by fastforce
  — The image field is generated over  $F$  by the image radical  $f\ a$ .
  have  $f \text{ ' } G = \text{generate-field } (F \cup \{f\ a\})$ 
  using  $\text{fix}F\ FK\ aK\ \text{field-hom-on.image-generate-field-eq}$ [ $OF\ \text{hom}$ ] by (simp add:

```

Gdef image-Un)

then show *?thesis unfolding radical-step-def using n fanF by blast*
qed

General form (no fixing hypothesis): a field homomorphism carries a radical step $F \rightsquigarrow G$ to the radical step $f \cdot F \rightsquigarrow f \cdot G$. This is the form that iterates along a tower, where after the first step the base is no longer fixed pointwise.

lemma *radical-step-image-gen:*

fixes $f :: \text{complex} \Rightarrow \text{complex}$

assumes $\text{hom}: \text{field-hom-on } K \text{ } f$ **and** $\text{FK}: F \subseteq K$

and $\text{step}: \text{radical-step } F \ G$ **and** $\text{GK}: G \subseteq K$

shows $\text{radical-step } (f \cdot F) \ (f \cdot G)$

proof —

obtain $a \ n$ **where** $n: n > 0$ **and** $\text{anF}: a \wedge n \in F$ **and** $\text{Gdef}: G = \text{generate-field } (F \cup \{a\})$

using step **by** $(\text{auto simp: radical-step-def})$

have $aK: a \in K$ **using** $\text{Gdef GK subset-generate-field by fastforce}$

— $f \ a$ is a radical over $f \cdot F: (f \ a)^n = f(a^n) \in f \cdot F$.

have $\text{fanF}: (f \ a) \wedge n \in f \cdot F$

using $aK \text{ field-hom-on.hom-power}[OF \ \text{hom}]$ **by** $(\text{metis anF imageI})$

have $f \cdot G = f \cdot \text{generate-field } (F \cup \{a\})$ **by** (simp add: Gdef)

also have $\dots = \text{generate-field } (f \cdot F \cup \{f \ a\})$

using $\text{field-hom-on.image-generate-field-eq}[OF \ \text{hom}] \ \text{FK } aK$ **by** $(\text{simp add: image-Un})$

finally show *?thesis unfolding radical-step-def using n fanF by blast*

qed

108.4 A field homomorphism carries radical towers to radical towers

Each field of a radical tower is a subfield (each is generated over the previous), and the tower is increasing. In particular every field is contained in any subfield K that contains the *top* field — but for the tower-image induction we only need that consecutive fields nest, which we package as: every field of the tower is a subfield of the last. We instead carry an explicit ambient K with *tower-top* F $Fs \subseteq K$.

lemma *radical-step-field:*

assumes $\text{radical-step } F \ G$ **shows** $\text{Subfield } G \wedge F \subseteq G$

using $\text{assms radical-step-def by auto}$

lemma *radical-tower-subset-ambient:*

$\text{Subfield } K \Longrightarrow \text{Subfield } F \Longrightarrow \text{radical-tower } F \ Fs \Longrightarrow \text{tower-top } F \ Fs \subseteq K$

$\Longrightarrow F \subseteq K \wedge (\forall G \in \text{set } Fs. G \subseteq K)$

proof $(\text{induction } Fs \text{ arbitrary: } F)$

case Nil

then show *?case by simp*

next

case $(\text{Cons } G \ Gs)$


```

  have radical-step F G and tower': radical-tower G Gs using Cons.prem(3) by
  auto
  then have sfG: Subfield G and FG: F ⊆ G
    using radical-step-field by auto
  have topGs: tower-top G Gs ⊆ K using Cons.prem(4) by simp
  with Cons show ?case
    by (metis FG order-trans set-ConsD sfG tower')
qed

```

A conjugate of a radical tower is a radical tower. Applying a field homomorphism f to every field of a radical tower over F yields a radical tower over $f \text{ ` } F$: each step maps to a step by *radical-step-image-gen*.

```

theorem radical-tower-image:
  fixes f :: complex ⇒ complex
  assumes K: Subfield K and hom: field-hom-on K f
  shows Subfield F ⇒ radical-tower F Fs ⇒ tower-top F Fs ⊆ K
    ⇒ radical-tower (f ` F) (List.map ((` ) f) Fs)
proof (induction Fs arbitrary: F)
  case Nil
  then show ?case by simp
next
  case (Cons G Gs)
  have step: radical-step F G and tower': radical-tower G Gs using Cons.prem(2)
  by auto
  have sfG: Subfield G
    using Cons.prem(1) local.step radical-step-field by blast
  — F and G live in K.
  have FK: F ⊆ K and GK: G ⊆ K
    using radical-tower-subset-ambient[OF K] Cons.prem sfG tower' tower-top-Cons
    by blast+
  show ?case
    using Cons.prem radical-step-image-gen[OF hom FK step GK] Cons.IH[OF sfG
    tower'] by simp
qed

```

108.5 The top of a radical tower is a subfield containing the base

```

lemma subfield-tower-top:
  Subfield F ⇒ radical-tower F Fs ⇒ Subfield (tower-top F Fs) ∧ F ⊆ tower-top
  F Fs
  by (induction Fs arbitrary: F) (use radical-step-field in force)+

```

108.6 Splicing a list of towers over a common base into one radical tower

Base change of a tower onto a subfield containing the base. If F is a subfield of the subfield B , a radical tower over F base-changes to a radical tower over

B (not merely over *generate-field* $(B \cup F)$): since B is a subfield containing F , *generate-field* $(B \cup F) = B$.

lemma *radical-tower-basechange-subfield*:

assumes B : Subfield B **and** FB : $F \subseteq B$ **and** $tower$: radical-tower F Fs
shows radical-tower B ($List.map (\lambda H. \text{generate-field } (B \cup H)) Fs$)
using B radical-tower-basechange[OF $tower$, of B]
by (*simp add*: FB *Un-absorb2* Subfield.generate-field-self)

Validity of a family of towers each based at F .

fun *all-towers-over* :: 'a :: field set $\Rightarrow$ 'a set list list $\Rightarrow$ bool **where**
all-towers-over F [] $\longleftrightarrow$ True
| *all-towers-over* F ($Ts \# Tss$) $\longleftrightarrow$ radical-tower F $Ts \wedge$ *all-towers-over* F Tss

Splice a family of towers (all based at F) into one list, base-changing each successive tower onto the running top B . The accumulator B is the subfield reached so far.

fun *spliced* :: 'a :: field set $\Rightarrow$ 'a set list list $\Rightarrow$ 'a set list **where**
spliced B [] = []
| *spliced* B ($Ts \# Tss$) =
(*let* $Ts' = List.map (\lambda H. \text{generate-field } (B \cup H)) Ts$
in $Ts' @ \text{spliced } (tower-top B Ts') Tss$)

Splicing a family of towers-over- F onto a subfield B containing F yields a radical tower over B .

lemma *radical-tower-spliced*:

Subfield $B \implies$ Subfield $F \implies F \subseteq B \implies$ *all-towers-over* F Tss
 $\implies$ radical-tower B (*spliced* B Tss)

proof (*induction* Tss arbitrary: B)

case *Nil*

then show ?case **by** *simp*

next

case (*Cons* Ts Tss)

define Ts' **where** $Ts' = List.map (\lambda H. \text{generate-field } (B \cup H)) Ts$

— The first tower, base-changed onto B , is a radical tower over B .

have $rtTs'$: radical-tower B Ts'

unfolding Ts' -def

using *Cons.prem*s radical-tower-basechange-subfield **by** *auto*

— Its top is a subfield containing B (hence F).

define B' **where** $B' = tower-top B Ts'$

have FB' : $F \subseteq B'$

using B' -def *Cons.prem*s(1,3) $rtTs'$ subfield-tower-top **by** *blast*

— The remaining family splices onto B' , by the induction hypothesis.

have radical-tower B' (*spliced* B' Tss)

using B' -def *Cons.IH* *Cons.prem*s FB' $rtTs'$ subfield-tower-top **by** *force*

then have radical-tower B ($Ts' @ \text{spliced } B' Tss$)

using $rtTs'$ **by** (*simp add*: B' -def radical-tower-append)

then show ?case **by** (*simp add*: Ts' -def B' -def *Let*-def)

qed

108.7 A radical tower is increasing; every field lies in the top

Each field of a radical tower over F is contained in the top field: the tower is increasing (each step is a field extension, *radical-step-field*), so F and every listed field embed into *tower-top* F F s.

lemma *field-subset-tower-top*:

Subfield $F \implies \text{radical-tower } F \text{ } F\text{s} \implies F \subseteq \text{tower-top } F \text{ } F\text{s} \wedge (\forall H \in \text{set } F\text{s}. H \subseteq \text{tower-top } F \text{ } F\text{s})$

proof (*induction* F s arbitrary: F)

case *Nil*

then show ?*case* **by** *simp*

next

case (*Cons* G G s)

then show ?*case*

by (*meson equalityE radical-tower-subset-ambient subfield-tower-top*)

qed

end

109 Finite field extensions

theory *Finite-Field-Extensions*

imports *Finite-Field-Frobenius Extension-Properties Algebraic-Transitivity*

begin

The polynomial $X^{\wedge \text{card } K} - X$ packages the two decisive properties of a finite subfield K : every element of the carrier is a root, and there can be no further roots. Its derivative is -1 , so minimal polynomials of elements of finite extensions divide a root-squarefree polynomial. This yields separability and normality directly in the native carrier-set representation.

definition *finite-field-vanishing-poly* :: ' a :: field set $\Rightarrow$ ' a poly

where *finite-field-vanishing-poly* $K \equiv \text{monom } 1 (\text{card } K) - \text{monom } 1 1$

lemma *poly-finite-field-vanishing-poly* [*simp*]:

poly (*finite-field-vanishing-poly* K) $x = x^{\wedge \text{card } K} - x$

by (*simp add: finite-field-vanishing-poly-def poly-monom*)

lemma *finite-field-vanishing-poly-nonzero*:

assumes *card* $K > 1$

shows *finite-field-vanishing-poly* $K \neq (0 :: 'a :: \text{field poly})$

proof

assume *zero: finite-field-vanishing-poly* $K = (0 :: 'a \text{ poly})$

have *one-coeff: coeff* (*finite-field-vanishing-poly* K) (*card* K) = 1

using *assms* **by** (*simp add: finite-field-vanishing-poly-def coeff-monom*)

with zero **show** *False* **by** *auto*

qed

```

lemma degree-finite-field-vanishing-poly:
  assumes card K > 1
  shows degree (finite-field-vanishing-poly K :: 'a :: field poly) = card K
proof -
  have upper: degree (finite-field-vanishing-poly K :: 'a poly) ≤ card K
    unfolding finite-field-vanishing-poly-def
  proof (rule degree-diff-le)
    show degree (monom (1 :: 'a) (card K)) ≤ card K
      by (rule degree-monom-le)
    with assms show degree (monom (1 :: 'a) 1) ≤ card K
      using degree-monom-le dual-order.strict-trans2 less-or-eq-imp-le by blast
  qed
  have leading: coeff (finite-field-vanishing-poly K :: 'a poly) (card K) = 1
    using assms by (simp add: finite-field-vanishing-poly-def coeff-monom)
  have lower: card K ≤ degree (finite-field-vanishing-poly K :: 'a poly)
    by (simp add: le-degree leading)
  show ?thesis by (rule antisym[OF upper lower])
qed

```

Over any field, at most m elements are fixed by the m -power map when $m > 1$. This root bound later distinguishes the proper iterates of Frobenius.

```

lemma card-power-fixed-points-le:
  fixes m :: nat
  assumes m: m > 1
  shows finite {x :: 'a :: field. x ^ m = x} ∧ card {x :: 'a. x ^ m = x} ≤ m
proof -
  define p where p = (monom 1 m - monom 1 1 :: 'a poly)
  have p0: p ≠ 0
  proof
    assume zero: p = 0
    have zero-coeff: coeff p m = 0 using zero by simp
    have one-coeff: coeff p m = 1
      using m by (simp add: p-def coeff-monom)
    show False using zero-coeff one-coeff by simp
  qed
  have upper: degree p ≤ m
    unfolding p-def
    by (metis degree-diff-le degree-monom-eq degree-monom-le monom-eq-0-iff order-less-imp-le)
  have lower: m ≤ degree p
    using m by (simp add: le-degree p-def coeff-monom)
  have degree: degree p = m by (rule antisym[OF upper lower])
  have roots: {x. poly p x = 0} = {x. x ^ m = x}
    by (auto simp: p-def poly-monom)
  have finite-roots: finite {x. poly p x = 0}
    by (rule poly-roots-finite[OF p0])
  have bound: card {x. poly p x = 0} ≤ m
    using degree p0 poly-roots-degree by auto
  show ?thesis using finite-roots bound roots by simp

```

qed

A root of multiplicity at least two is also a root of the formal derivative. Only this direction is characteristic-independent; it is the direction needed below.

lemma *order-ge-two-imp-pderiv-root*:

fixes $p :: 'a :: \text{field poly}$
assumes $p0: p \neq 0$ **and** $two: 2 \leq \text{order } a \ p$
shows $\text{poly } (\text{pderiv } p) \ a = 0$

proof –

have $\text{Suc } (\text{Suc } 0) \leq \text{order } a \ p$ **using** two **by** simp
then obtain r **where** $p = [:-a, 1:] \wedge \text{Suc } (\text{Suc } 0) * r$
using order-divides **by** blast
have $\text{poly } (\text{pderiv } p) \ a = \text{poly } ([:-a, 1:] \wedge \text{Suc } (\text{Suc } 0)) \ a * \text{poly } (\text{pderiv } r) \ a +$
 $\text{poly } r \ a * \text{poly } (\text{pderiv } ([:-a, 1:] \wedge \text{Suc } (\text{Suc } 0))) \ a$
by $(\text{metis } (\text{no-types}) \ p \ \text{pderiv-mult} \ \text{poly-add} \ \text{poly-mult})$
also have $\dots = 0$
by $(\text{force simp: pderiv-pCons})$
finally show $?thesis$.

qed

context *Subfield*

begin

lemma *finite-field-vanishing-poly-over*:

finite-field-vanishing-poly $K \in \text{poly-over } K$
unfolding *finite-field-vanishing-poly-def*
by $(\text{intro poly-over-diff poly-over-monom one-closed})$

lemma *finite-field-vanishing-poly-roots*:

assumes $\text{fin: finite } K$
shows $\{x. \text{poly } (\text{finite-field-vanishing-poly } K) \ x = 0\} = K$

proof –

have $K\text{-roots: } K \subseteq \{x. \text{poly } (\text{finite-field-vanishing-poly } K) \ x = 0\}$
using *finite-field-frobenius-identity* $[OF \ \text{fin}]$
by $(\text{auto simp: frobenius-power-def})$
have $p0: \text{finite-field-vanishing-poly } K \neq 0$
using $\text{cardK-gt1 fin finite-field-vanishing-poly-nonzero}$ **by** blast
have $\text{fin-roots: finite } \{x. \text{poly } (\text{finite-field-vanishing-poly } K) \ x = 0\}$
by $(\text{rule poly-roots-finite} [OF \ p0])$
have $\text{card-roots-le: card } \{x. \text{poly } (\text{finite-field-vanishing-poly } K) \ x = 0\} \leq \text{card } K$
using $\text{poly-roots-degree} [OF \ p0] \ \text{cardK-gt1 assms}$
by $(\text{simp add: degree-finite-field-vanishing-poly})$
have cardK-le:
 $\text{card } K \leq \text{card } \{x. \text{poly } (\text{finite-field-vanishing-poly } K) \ x = 0\}$
by $(\text{rule card-mono} [OF \ \text{fin-roots } K\text{-roots}])$
then show $?thesis$
using $K\text{-roots card-roots-le fin-roots}$ **by** $(\text{metis card-seteq})$

qed

```

lemma finite-field-vanishing-poly-rsquarefree:
  assumes fin: finite K
  shows rsquarefree (finite-field-vanishing-poly K)
proof -
  obtain n where n: n > 0 card K = CHAR('a) ^ n
    using finite-subfield-cardinality-char-power[OF fin] by blast
  have deriv: pderiv (finite-field-vanishing-poly K) = -1
    unfolding finite-field-vanishing-poly-def
    by (simp add: pderiv-diff pderiv-monom n)
  have pair: {0, 1} ⊆ K by auto
  have p0: finite-field-vanishing-poly K ≠ 0
    using deriv by force
  show ?thesis
    unfolding rsquarefree-def
  proof (intro conjI allI p0)
    fix x
    show order x (finite-field-vanishing-poly K) = 0 ∨ order x (finite-field-vanishing-poly
K) = 1
    proof (cases poly (finite-field-vanishing-poly K) x = 0)
      case False
      then show ?thesis
        using False by (simp add: order-eq-0-iff[OF p0])
    next
      case True
      have opos: order x (finite-field-vanishing-poly K) > 0
        using True by (simp add: order-gt-0-iff[OF p0])
      have not-two: ¬ 2 ≤ order x (finite-field-vanishing-poly K)
        using deriv order-1-eq-0 order-ge-two-imp-pderiv-root p0 by fastforce
      with opos show ?thesis by auto
    qed
  qed
qed
end

```

```

lemma finite-field-vanishing-poly-over-subfield:
  fixes F K :: 'a :: field set
  assumes sfF: Subfield F
  shows finite-field-vanishing-poly K ∈ poly-over F
  by (simp add: Subfield.one-closed Subfield.poly-over-diff Subfield.poly-over-monom
finite-field-vanishing-poly-def sfF)

```

Every finite carrier extension is separable over each of its subfields. The minimal polynomial of an element of the top field divides the top field's vanishing polynomial, whose roots are all simple.

```

theorem finite-subfield-extension-separable:
  fixes F K :: 'a :: field set
  assumes sfF: Subfield F and sfK: Subfield K and FK: F ⊆ K
  and finK: finite K

```

```

shows separable-extension K F
proof (rule separable-extensionI)
interpret T: subfield-tower F K
  by (intro subfield-tower.intro subfield-tower-axioms.intro sfF sfK FK)
interpret F: Subfield F by (rule sfF)
interpret K: Subfield K by (rule sfK)
fix a
assume aK: a ∈ K and alg: algebraic-over F a
have qF: finite-field-vanishing-poly K ∈ poly-over F
  by (rule finite-field-vanishing-poly-over-subfield[OF sfF])
have qa: poly (finite-field-vanishing-poly K) a = 0
  using Subfield.finite-field-frobenius-identity[OF sfK finK aK]
  by (simp add: frobenius-power-def)
have minpoly F a dvd finite-field-vanishing-poly K
  using T.base.is-minpoly-minpoly T.base.minpoly-dvd alg qF qa by blast
then show rsquarefree (minpoly F a)
  using T.ext.finite-field-vanishing-poly-rsquarefree finK rsquarefree-dvd by blast
qed

```

Likewise every finite carrier extension is normal over each of its subfields. Any irreducible polynomial with one root in the top field has its minimal polynomial as an associate; that minimal polynomial divides the same vanishing polynomial, whose entire root set is the top carrier.

```

theorem finite-subfield-extension-normal:
fixes F K :: 'a :: field set
assumes sfF: Subfield F and sfK: Subfield K and FK: F ⊆ K
and finK: finite K
shows normal-extension K F
proof (rule normal-extensionI)
interpret T: subfield-tower F K
  by (intro subfield-tower.intro subfield-tower-axioms.intro sfF sfK FK)
interpret F: Subfield F by (rule sfF)
interpret K: Subfield K by (rule sfK)
fix p a
assume pF: p ∈ poly-over F and irr: irreducible-over F p
and aK: a ∈ K and pa: poly p a = 0
have p0: p ≠ 0
  using irr by (auto simp: irreducible-over-def)
have min: is-minpoly F a (minpoly F a)
  using T.base.is-minpoly-minpoly algebraic-over-def p0 pF pa by auto
have qF: finite-field-vanishing-poly K ∈ poly-over F
  by (rule finite-field-vanishing-poly-over-subfield[OF sfF])
have qa: poly (finite-field-vanishing-poly K) a = 0
  using Subfield.finite-field-frobenius-identity[OF sfK finK aK]
  by (simp add: frobenius-power-def)
obtain c where qeq: finite-field-vanishing-poly K = minpoly F a * c
  using Subfield.minpoly-dvd[OF sfF min qF qa] by (auto simp: dvd-def)
show poly-root-set p ⊆ K
proof

```

```

    fix b assume bp: b ∈ poly-root-set p
    then have minb: poly (minpoly F a) b = 0
      using T.base.irreducible-over-root-is-minpoly-root irr min pF pa poly-root-set-iff
  by blast
  have roots: {x. poly (finite-field-vanishing-poly K) x = 0} = K
    by (rule Subfield.finite-field-vanishing-poly-roots[OF sfK finK])
  show b ∈ K
    using minb qeq roots by auto
qed
qed

```

```

theorem finite-field-extension-cardinality:
  fixes F K :: 'a :: field set
  assumes T: finite-subfield-tower F K and finF: finite F
  shows card K = card F ^ finite-subfield-tower.extension-degree F K
proof -
  interpret T: finite-subfield-tower F K by (rule T)
  have finK: finite K by (rule T.finite-carrier-of-finite-base[OF finF])
  show ?thesis by (rule T.finite-extension-cardinality[OF finK])
qed

```

```

theorem finite-field-extension-separable:
  fixes F K :: 'a :: field set
  assumes T: finite-subfield-tower F K and finF: finite F
  shows separable-extension K F
proof -
  interpret T: finite-subfield-tower F K by (rule T)
  have finK: finite K by (rule T.finite-carrier-of-finite-base[OF finF])
  show ?thesis
    by (simp add: T.base.Subfield-axioms T.base-subset T.ext.Subfield-axioms finK
        finite-subfield-extension-separable)
qed

```

```

theorem finite-field-extension-normal:
  fixes F K :: 'a :: field set
  assumes T: finite-subfield-tower F K and finF: finite F
  shows normal-extension K F
proof -
  interpret T: finite-subfield-tower F K by (rule T)
  have finK: finite K by (rule T.finite-carrier-of-finite-base[OF finF])
  then show ?thesis
    by (simp add: T.base.Subfield-axioms T.base-subset T.ext.Subfield-axioms
        finite-subfield-extension-normal)
qed

```

end

110 Galois groups of finite fields

```

theory Finite-Field-Galois
  imports Finite-Field-Extensions Galois-Finite-Correspondence
begin

```

The restricted power map is a genuine element of the carrier-set automorphism group: outside the top carrier it has the canonical undefined value required by *PiE*.

```

definition finite-field-frobenius ::
  'a :: field set  $\Rightarrow$  nat  $\Rightarrow$  ('a  $\Rightarrow$  'a)
  where finite-field-frobenius K q  $\equiv$  restrict (frobenius-power q) K

```

```

lemma finite-field-frobenius-apply [simp]:
  x  $\in$  K  $\implies$  finite-field-frobenius K q x = x  $\wedge$  q
  by (simp add: finite-field-frobenius-def frobenius-power-def restrict-apply')

```

```

theorem finite-field-frobenius-in-field-auto:

```

```

  fixes F K :: 'a :: field set
  assumes T: finite-subfield-tower F K and finF: finite F
  shows finite-field-frobenius K (card F)  $\in$  field-auto K F
proof -
  interpret T: finite-subfield-tower F K by (rule T)
  have finK: finite K by (rule T.finite-carrier-of-finite-base[OF finF])
  obtain e where e: e > 0 card F = CHAR('a)  $\wedge$  e
    using T.base.finite-subfield-cardinality-char-power[OF finF] by blast
  have char: prime CHAR('a)
    by (rule T.base.finite-subfield-CHAR-prime[OF finF])
  let ?sigma = finite-field-frobenius K (card F)
  have frob-bij: bij-betw (frobenius-power (card F)) K K
    by (rule T.ext.finite-frobenius-bij-betw[OF finK char e(2)])
  have sigma-bij: bij-betw ?sigma K K
    by (simp add: finite-field-frobenius-def frob-bij)
  show ?thesis
    unfolding field-auto-mem-iff
  proof (intro conjI ballI)
    show ?sigma  $\in$  K  $\rightarrow_E$  K
      using T.ext.frobenius-power-closed
      by (auto simp: finite-field-frobenius-def PiE-iff extensional-def restrict-apply')
    show bij-betw ?sigma K K by (rule sigma-bij)
    show  $\bigwedge x y. x \in K \implies y \in K \implies ?sigma (x + y) = ?sigma x + ?sigma y$ 
      using frobenius-power-add-char-power[OF char e(2)]
      by (simp add: finite-field-frobenius-def restrict-apply' T.ext.add-closed)
    show  $\bigwedge x y. x \in K \implies y \in K \implies ?sigma (x * y) = ?sigma x * ?sigma y$ 
      by (simp add: finite-field-frobenius-def restrict-apply' T.ext.mult-closed frobenius-power-mult)
    show ?sigma 1 = 1
      by (simp add: finite-field-frobenius-def restrict-apply')
    show ?sigma x = x if xF: x  $\in$  F for x
  proof -

```

```

    have fixed: frobenius-power (card F) x = x
      by (rule T.base.finite-field-frobenius-identity[OF finF xF])
    show ?sigma x = x
      using T.vs.scale-closed xF fixed by (fastforce simp add: finite-field-frobenius-def
restrict-apply')
  qed
qed
qed

```

theorem *finite-field-frobenius-power-apply*:

```

  fixes F K :: 'a :: field set
  assumes T: finite-subfield-tower F K and finF: finite F and xK: x ∈ K
  shows Monoid.power (compose K) (identity K)
    (finite-field-frobenius K (card F)) i x = x ^ (card F ^ i)
proof -
  interpret T: finite-subfield-tower F K by (rule T)
  interpret G: Group field-auto K F compose K identity K
    by (rule field-auto-group[OF T.ext.Subfield-axioms T.base-subset])
  let ?sigma = finite-field-frobenius K (card F)
  have sigma: ?sigma ∈ field-auto K F
    by (rule finite-field-frobenius-in-field-auto[OF assms(1) finF])
  show ?thesis
  proof (induction i)
    case 0 then show ?case
      by (metis power-0 power-one-right restrict-apply' G.power-0 xK)
  next
    case (Suc i)
    have power-maps: G.power ?sigma i x ∈ K
      using sigma xK by (simp add: Suc T.ext.power-closed)
    have G.power ?sigma (Suc i) x = (G.power ?sigma i x) ^ card F
      by (simp add: compose-eq power-maps xK)
    also have ... = x ^ (card F ^ Suc i)
      by (metis Suc power-Suc2 power-mult)
    finally show ?case .
  qed
qed

```

theorem *finite-field-frobenius-power-degree*:

```

  fixes F K :: 'a :: alg-closed-field set
  assumes T: finite-subfield-tower F K and finF: finite F
    and n: n > 0 card K = card F ^ n
  shows Monoid.power (compose K) (identity K)
    (finite-field-frobenius K (card F)) n = identity K
proof -
  interpret T: finite-subfield-tower F K by (rule T)
  interpret G: Group field-auto K F compose K identity K
    by (rule field-auto-group[OF T.ext.Subfield-axioms T.base-subset])
  let ?sigma = finite-field-frobenius K (card F)
  have power-mem: G.power ?sigma n ∈ field-auto K F
    using finite-field-frobenius-in-field-auto[OF T finF] by simp

```

```

have power-maps:  $G.\text{power } ?\sigma n \in K \rightarrow_E K$ 
  using power-mem by (simp add: field-auto-mem-iff)
have id-maps:  $\text{identity } K \in K \rightarrow_E K$ 
  by (auto simp: identity-apply PiE-iff)
show ?thesis
proof (rule PiE-ext[OF power-maps id-maps])
  fix x assume xK:  $x \in K$ 
  have iter:  $G.\text{power } ?\sigma n x = x \wedge (\text{card } F \wedge n)$ 
    by (rule finite-field-frobenius-power-apply[OF T finF xK])
  have top-fixed-frob:  $\text{frobenius-power } (\text{card } K) x = x$ 
  using Subfield.finite-field-frobenius-identity T.ext.Subfield-axioms T.finite-carrier-of-finite-base
    finF xK by blast
  have top-fixed:  $x \wedge \text{card } K = x$ 
  using top-fixed-frob by (simp add: frobenius-power-def)
  have id-x:  $\text{identity } K x = x$  by (rule identity-apply[OF xK])
  show  $G.\text{power } ?\sigma n x = \text{identity } K x$ 
    using id-x iter n(2) top-fixed by argo
qed
qed

```

The Frobenius automorphism has exact order the degree of a finite extension. The upper bound is the finite-field identity on the top carrier; a smaller order would make every top-field element a root of a polynomial whose degree is too small.

```

theorem finite-field-frobenius-order:
  fixes F K :: 'a :: alg-closed-field set
  assumes T: finite-subfield-tower F K and finF: finite F
  shows Monoid.element-order (compose K) (identity K) (finite-field-frobenius K
    (card F)) =
    finite-subfield-tower.extension-degree F K
proof -
  interpret T: finite-subfield-tower F K by (rule T)
  interpret G: Group field-auto K F compose K identity K
    by (rule field-auto-group[OF T.ext.Subfield-axioms T.base-subset])
  let ?sigma = finite-field-frobenius K (card F)
  define X where X  $\equiv \{x :: 'a. x \wedge (\text{card } F \wedge G.\text{element-order } ?\sigma) = x\}$ 
  have sigma:  $?sigma \in \text{field-auto } K F$ 
    by (rule finite-field-frobenius-in-field-auto[OF T finF])
  have finK: finite K by (rule T.finite-carrier-of-finite-base[OF finF])
  have normal: normal-extension K F
    by (rule finite-field-extension-normal[OF T finF])
  have sep: separable-extension K F
    by (rule finite-field-extension-separable[OF T finF])
  have finG: finite (field-auto K F)
    by (rule finite-normal-separable-field-auto[OF T normal sep])
  define n where n  $\equiv \text{finite-subfield-tower.extension-degree } F K$ 
  have n-pos:  $n > 0$ 
    unfolding n-def by (rule T.extension-degree-pos)
  have card-eq:  $\text{card } K = \text{card } F \wedge n$ 

```

```

unfolding n-def by (rule finite-field-extension-cardinality[OF T finF])
have power-n: G.power ?sigma n = identity K
  by (rule finite-field-frobenius-power-degree[OF T finF n-pos card-eq])
have ord-le: G.element-order ?sigma ≤ n
  by (rule G.element-order-minimal[OF finG sigma n-pos power-n])
have q-gt1: 1 < card F
  using T.base.cardK-gt1 finF by blast
have ord-not-less:  $\neg G.element-order\ ?sigma < n$ 
proof
  assume ord-less: G.element-order ?sigma < n
  have ord-pos: G.element-order ?sigma > 0
    by (rule G.element-order-pos[OF finG sigma])
  have qpow-gt1: 1 < card F ^ G.element-order ?sigma
    by (rule one-less-power[OF q-gt1 ord-pos])
  have X: finite X ∧ card X ≤ card F ^ G.element-order ?sigma
    unfolding X-def
    by (rule card-power-fixed-points-le[OF qpow-gt1])
  have top-subset: K ⊆ X
proof
  fix x assume xK: x ∈ K
  have G.power ?sigma (G.element-order ?sigma) x = x ^ (card F ^ G.element-order ?sigma)
    by (rule finite-field-frobenius-power-apply[OF T finF xK])
  then show x ∈ X
    unfolding X-def
    using G.power-element-order[OF finG sigma] xK by simp
qed
with X power-strict-increasing[OF ord-less q-gt1] card-eq show False
  by (smt (verit, best) binomial-eq-0-iff card-seteq nat-less-le order-trans zero-less-binomial-iff)
qed
show ?thesis
  using ord-le ord-not-less n-def by linarith
qed

```

Every finite-field Galois group is generated by the relative Frobenius. The equality is proved at the carrier-set level, so downstream users can use the native automorphism group directly without introducing a second cyclic-group representation.

```

theorem finite-field-galois-group-cyclic:
  fixes F K :: 'a :: alg-closed-field set
  assumes T: finite-subfield-tower F K and finF: finite F
  shows Group.cyclic-subgroup (compose K) (identity K) (finite-field-frobenius K (card F)) = field-auto K F
proof –
  interpret T: finite-subfield-tower F K by (rule T)
  interpret G: Group field-auto K F compose K identity K
    by (rule field-auto-group[OF T.ext.Subfield-axioms T.base-subset])
  let ?sigma = finite-field-frobenius K (card F)
  obtain sigma: ?sigma ∈ field-auto K F and normal: normal-extension K F and

```

```

sep: separable-extension K F
  using T finF
  by (auto simp: finite-field-frobenius-in-field-auto finite-field-extension-normal fi-
nite-field-extension-separable)
  have finG: finite (field-auto K F)
    by (rule finite-normal-separable-field-auto[OF T normal sep])
  have ord: G.element-order ?sigma = finite-subfield-tower.extension-degree F K
    by (rule finite-field-frobenius-order[OF T finF])
  have group-card: card (field-auto K F) = finite-subfield-tower.extension-degree F K
    by (rule finite-normal-separable-galois-degree[OF T normal sep])
  have cyc-subset: G.cyclic-subgroup ?sigma ⊆ field-auto K F
    by (rule G.cyclic-subgroup-subset[OF sigma])
  have cyc-card: card (G.cyclic-subgroup ?sigma) = finite-subfield-tower.extension-degree
F K
    using G.card-cyclic-subgroup[OF finG sigma] ord by simp
  have card-eq: card (G.cyclic-subgroup ?sigma) = card (field-auto K F)
    using cyc-card group-card by simp
  show ?thesis
    by (rule card-subset-eq[OF finG cyc-subset card-eq])
qed

```

The canonical degree- d root set inside a finite extension. It is written as a set of ambient-field elements lying in the top carrier, which keeps the statement useful for the native set-based subfield representation.

definition *finite-field-power-roots* ::
 'a :: field set $\Rightarrow$ 'a set $\Rightarrow$ nat $\Rightarrow$ 'a set
where *finite-field-power-roots* K F d $\equiv \{x \in K. x^{\wedge}(\text{card } F \wedge d) = x\}$

lemma *finite-field-power-roots-iff* [simp]:
 $x \in \text{finite-field-power-roots } K F d \iff$
 $x \in K \wedge x^{\wedge}(\text{card } F \wedge d) = x$
by (simp add: *finite-field-power-roots-def*)

Every intermediate field is exactly the root set of the appropriate Frobenius power. The root bound supplies the reverse inclusion: the intermediate field already has the full degree-many roots, so no additional ambient roots can occur.

theorem *finite-field-intermediate-power-roots*:
fixes F K E :: 'a :: alg-closed-field set
assumes T: *finite-subfield-tower* F K **and** finF: *finite* F
and E: $E \in \text{inter-fields } K F$
shows $E = \text{finite-field-power-roots } K F (\text{finite-subfield-tower.extension-degree } F E)$
proof –
interpret T: *finite-subfield-tower* F K **by** (rule T)
have sfE: *Subfield* E **and** FE: $F \subseteq E$ **and** EK: $E \subseteq K$
using E **by** (auto simp: *inter-fields-iff*)
have TFE-raw: *finite-subfield-tower* F E
by (rule *finite-subfield-tower-intermediate-left*[OF T sfE FE EK])
interpret TFE: *finite-subfield-tower* F E **by** (rule TFE-raw)
have finE: *finite* E **by** (rule TFE.*finite-carrier-of-finite-base*[OF finF])

```

have cardE: card E = card F ^ TFE.extension-degree
  by (rule finite-field-extension-cardinality[OF TFE-raw finF])
have q-gt1: 1 < card F
  using T.base.cardK-gt1 finF by blast
have m-gt1: 1 < card F ^ TFE.extension-degree
  by (rule one-less-power[OF q-gt1 TFE.extension-degree-pos])
have roots:
  finite {x :: 'a. x ^ (card F ^ TFE.extension-degree) = x} ^
  card {x :: 'a. x ^ (card F ^ TFE.extension-degree) = x} ≤ card F ^ TFE.extension-degree
  by (rule card-power-fixed-points-le[OF m-gt1])
let ?R = finite-field-power-roots K F TFE.extension-degree
have R-subset: ?R ⊆ {x :: 'a. x ^ (card F ^ TFE.extension-degree) = x}
  by (auto simp: finite-field-power-roots-def)
have R-card-le: card ?R ≤ card F ^ TFE.extension-degree
  using roots order-trans[OF card-mono] using R-subset cardE by metis
show ?thesis
proof (intro card-subset-eq)
  show E-subset-R: E ⊆ ?R
    using local.EK TFE.ext.finite-field-frobenius-identity cardE finE
    by (auto simp: finite-field-power-roots-def frobenius-power-def)
  show finite ?R
    by (metis R-subset roots finite-subset)
  then show card E = card ?R
    by (metis E-subset-R R-card-le cardE card-seteq)
qed
qed

```

Consequently, the degree is a complete invariant for intermediate fields of a finite field extension.

```

theorem finite-field-intermediate-degree-injective:
  fixes F K :: 'a :: alg-closed-field set
  assumes T: finite-subfield-tower F K and finF: finite F
    and E: E ∈ inter-fields K F and E': E' ∈ inter-fields K F
    and degree-eq: finite-subfield-tower.extension-degree F E = finite-subfield-tower.extension-degree
      F E'
  shows E = E'
  using assms by (metis finite-field-intermediate-power-roots)

```

The lower degree of every intermediate field divides the total extension degree, as expected from the finite-field divisor classification.

```

theorem finite-field-intermediate-degree-dvd:
  fixes F K E :: 'a :: alg-closed-field set
  assumes T: finite-subfield-tower F K and finF: finite F
    and E: E ∈ inter-fields K F
  shows finite-subfield-tower.extension-degree F E dvd finite-subfield-tower.extension-degree
    F K
proof -
  interpret T: finite-subfield-tower F K by (rule T)
  have sE: Subfield E and FE: F ⊆ E and EK: E ⊆ K

```

```

    using E by (auto simp: inter-fields-iff)
  interpret C: finite-subfield-tower-chain F E K
  proof (intro finite-subfield-tower-chain.intro)
    show finite-subfield-tower F E
      using FE T local.EK sfE by (metis finite-subfield-tower-intermediate-left)
    show finite-subfield-tower E K
      using FE T local.EK sfE by (metis finite-subfield-tower-intermediate-right)
  qed
  show ?thesis
    by (simp add: C.extension-degree-tower-law-exists)
qed

```

Every divisor of the total degree is realised by a unique intermediate field. The subgroup generated by the corresponding power of Frobenius has the complementary order; finite Galois correspondence and the tower law then compute the degree of its fixed field.

```

theorem finite-field-intermediate-exists:
  fixes F K :: 'a :: alg-closed-field set
  assumes T: finite-subfield-tower F K and finF: finite F
    and d: d dvd finite-subfield-tower.extension-degree F K
  shows  $\exists E. E \in \text{inter-fields } K F \wedge \text{finite-subfield-tower.extension-degree } F E = d$ 
     $\wedge E = \text{finite-field-power-roots } K F d$ 
proof –
  interpret T: finite-subfield-tower F K by (rule T)
  interpret G: Group field-auto K F compose K identity K
    by (rule field-auto-group[OF T.ext.Subfield-axioms T.base-subset])
  interpret GE: galois-extension K F
    by (rule galois-extension.intro[OF T.ext.Subfield-axioms T.base.Subfield-axioms
      T.base-subset])
  let ?sigma = finite-field-frobenius K (card F)
  have sigma: ?sigma  $\in$  field-auto K F
    by (rule finite-field-frobenius-in-field-auto[OF T finF])
  have normal: normal-extension K F
    by (rule finite-field-extension-normal[OF T finF])
  have sep: separable-extension K F
    by (rule finite-field-extension-separable[OF T finF])
  have finG: finite (field-auto K F)
    by (rule finite-normal-separable-field-auto[OF T normal sep])
  define n where n  $\equiv$  finite-subfield-tower.extension-degree F K
  have n-pos: n > 0
    unfolding n-def by (rule T.extension-degree-pos)
  have ord-sigma: G.element-order ?sigma = n
    unfolding n-def by (rule finite-field-frobenius-order[OF T finF])
  let ?hgen = G.power ?sigma d
  have hgen: ?hgen  $\in$  field-auto K F
    by (rule G.power-closed[OF sigma])
  let ?H = G.cyclic-subgroup ?hgen
  have Hsub: Subgroup ?H (field-auto K F) (compose K) (identity K)
    by (rule G.cyclic-subgroup-is-subgroup[OF finG hgen])

```

```

have H: ?H ∈ galois-subgroups K F
  using Hsub by (simp add: galois-subgroups-iff)
have H-card: card ?H = n div d
  using G.card-cyclic-subgroup G.element-order-power finG d hgen ord-sigma sigma
  by (auto simp: n-def)
let ?E = fixed-field K ?H
have E: ?E ∈ inter-fields K F
  by (rule GE.fixed-field-in-inter-fields[OF H])
have sfE: Subfield ?E and FE: F ⊆ ?E and EK: ?E ⊆ K
  using E by (auto simp add: inter-fields-iff)
have TEK-raw: finite-subfield-tower ?E K
  by (rule finite-subfield-tower-intermediate-right[OF T sfE FE EK])
interpret TFE: finite-subfield-tower F ?E
  by (rule finite-subfield-tower-intermediate-left[OF T sfE FE EK])
interpret TEK: finite-subfield-tower ?E K by (rule TEK-raw)
have normalEK: normal-extension K ?E
  by (rule finite-normal-extension-base-change[OF T normal sfE FE EK])
have sepEK: separable-extension K ?E
  by (rule finite-separable-extension-base-change[OF T sep sfE FE EK])
have fixed-group: field-auto K ?E = ?H
  by (rule finite-galois-fixed-group[OF T normal sep H])
have H-degree: n div d = TEK.extension-degree
  using H-card fixed-group finite-normal-separable-galois-degree[OF TEK-raw nor-
malEK sepEK] by simp
interpret C: finite-subfield-tower-chain F ?E K
  by (rule finite-subfield-tower-chain.intro[OF TFE.finite-subfield-tower-axioms TEK-raw])
have cancel-eq: TEK.extension-degree * TFE.extension-degree = TEK.extension-degree
* d
  using H-degree dvd-div-mult-self[of d n] d C.extension-degree-tower-law-exists un-
folding n-def
  by auto
have lower-degree: TFE.extension-degree = d
  using cancel-eq TEK.extension-degree-pos nat-mult-eq-cancel1 by simp
then show ?thesis
  by (metis E T finF finite-field-intermediate-power-roots)
qed

end

```

111 The New-Algebra Development

```

theory New-Algebra
imports

```

```

  Subgroup-Lattice
  Monoid-Iso-Classes
  Normal-Series
  Normal-Chain
  Group-Product

```


Group-Family-Product
Schreier-Refinement
Simple-Factor-Chain
Jordan-Hoelder-Uniqueness
Maximal-Normal-Subgroup
Finite-Composition-Series
Sn-Composition-Factors
Abel-Ruffini
P-Group

Subring-Generated
Finite-Field-Frobenius
Ideal-Extension
Chinese-Remainder-Rings
Field-Of-Fractions
Poly-Divisibility
Closure-Algebraic

Module-Complements
Module-Exact-Sequence
Module-Family-Product
Free-Module-Universal
Finite-Dimensional-Isomorphism
Finite-Dimensional-Splitting

Field-Extension
Field-Mult-Cyclic
Galois-Restriction-Splitting
Galois-Finite-Correspondence
Normal-Closure

Finite-Field-Galois
GF-p

begin

end

References

- [1] C. Ballarín. A case study in basic algebra. *Archive of Formal Proofs*, 2019. https://isa-afp.org/entries/Jacobson_Basic_Algebra.html, Formal proof development.
- [2] N. Jacobson. *Basic Algebra I*. W. H. Freeman, second edition, 1985.