The Isabelle/HOL Algebra Library

Clemens Ballarin (Editor)

With contributions by Jesús Aransay, Clemens Ballarin, Martin Baillon, Paulo Emílio de Vilhena, Stephan Hohe, Florian Kammüller and Lawrence C Paulson

October 24, 2026

Contents

1 Objects **13**
- 1.1 Structure with Carrier Set. . . . 13
- 1.2 Structure with Carrier and Equivalence Relation `eq` . . . 13

2 Orders **20**
- 2.1 Partial Orders . . . 20
  - 2.1.1 The order relation . . . 21
  - 2.1.2 Upper and lower bounds of a set . . . 22
  - 2.1.3 Least and greatest, as predicate . . . 25
  - 2.1.4 Intervals . . . 27
  - 2.1.5 Isotone functions . . . 28
  - 2.1.6 Idempotent functions . . . 29
  - 2.1.7 Order embeddings . . . 29
  - 2.1.8 Commuting functions . . . 29
- 2.2 Partial orders where `eq` is the Equality . . . 29
- 2.3 Bounded Orders . . . 30
- 2.4 Total Orders . . . 31
- 2.5 Total orders where `eq` is the Equality . . . 31

3 Lattices **32**
- 3.1 Supremum and infimum . . . 32
- 3.2 Dual operators . . . 33
- 3.3 Lattices . . . 34
  - 3.3.1 Supremum . . . 34
  - 3.3.2 Infimum . . . 36
- 3.4 Weak Bounded Lattices . . . 38
- 3.5 Lattices where `eq` is the Equality . . . 39
- 3.6 Bounded Lattices . . . 40

4 Complete Lattices **41**
4.1 Infimum Laws . . . 43
4.2 Supremum Laws . . . 43
4.3 Fixed points of a lattice . . . 44
4.3.1 Least fixed points . . . 45
4.3.2 Greatest fixed points . . . 46
4.4 Complete lattices where `eq` is the Equality . . . 47
4.5 Fixed points . . . 47
4.6 Interval complete lattices . . . 48
4.7 Knaster-Tarski theorem and variants . . . 48
4.8 Examples . . . 49
4.8.1 The Powerset of a Set is a Complete Lattice . . . 49
4.9 Limit preserving functions . . . 50

5 Galois connections **51**
5.1 Definition and basic properties . . . 51
5.2 Well-typed connections . . . 51
5.3 Galois connections . . . 51
5.4 Composition of Galois connections . . . 54
5.5 Retracts . . . 54
5.6 Coretracts . . . 55
5.7 Galois Bijections . . . 55

6 Monoids and Groups **56**
6.1 Definitions . . . 56
6.2 Groups . . . 58
6.3 Cancellation Laws and Basic Properties . . . 59
6.4 Power . . . 60
6.5 Submonoids . . . 64
6.6 Subgroups . . . 65
6.7 Direct Products . . . 67
6.8 Homomorphisms (mono and epi) and Isomorphisms . . . 68
6.8.1 HOL Light's concept of an isomorphism pair . . . 71
6.8.2 Involving direct products . . . 72
6.9 The locale for a homomorphism between two groups . . . 73
6.10 Commutative Structures . . . 75
6.11 The Lattice of Subgroups of a Group . . . 77
6.12 The units in any monoid give rise to a group . . . 78
6.13 Product Operator for Commutative Monoids . . . 79
6.13.1 Inductive Definition of a Relation for Products over Sets 79
6.13.2 Left-Commutative Operations . . . 80
6.13.3 Products over Finite Sets . . . 83

7 Cosets and Quotient Groups **87**
7.1 Stable Operations for Subgroups . . . 88
7.2 Basic Properties of set multiplication . . . 88
7.3 Basic Properties of Cosets . . . 89
7.4 Normal subgroups . . . 91
7.5 More Properties of Left Cosets . . . 93
7.5.1 Set of Inverses of an `r_coset`. . . . 93
7.5.2 Theorems for `<#>` with `#>` or `<#`. . . . 94
7.5.3 An Equivalence Relation . . . 94
7.5.4 Two Distinct Right Cosets are Disjoint . . . 95
7.6 Further lemmas for `r_congruent` . . . 95
7.7 Order of a Group and Lagrange's Theorem . . . 95
7.8 Quotient Groups: Factorization of a Group . . . 97
7.9 The First Isomorphism Theorem . . . 99
7.9.1 Trivial homomorphisms . . . 101
7.10 Image kernel theorems . . . 101
7.11 Factor Groups and Direct product . . . 102
7.11.1 More Lemmas about set multiplication . . . 103
7.11.2 Lemmas about intersection and normal subgroups . . . 103

8 Flattening the type of group carriers **105**

9 Sylow's Theorem **106**
9.1 Main Part of the Proof . . . 107
9.2 Discharging the Assumptions of `sylow_central` . . . 108
9.2.1 Introduction and Destruct Rules for `H` . . . 109
9.3 Equal Cardinalities of `M` and the Set of Cosets . . . 110
9.3.1 The Opposite Injection . . . 110
9.4 Sylow's Theorem . . . 111

10 Bijections of a Set, Permutation and Automorphism Groups **111**
10.1 Bijections Form a Group . . . 112
10.2 Automorphisms Form a Group . . . 112

11 The Algebraic Hierarchy of Rings **113**
11.1 Abelian Groups . . . 113
11.2 Basic Properties . . . 114
11.3 Rings: Basic Definitions . . . 118
11.4 Rings . . . 118
11.4.1 Normaliser for Rings . . . 120
11.4.2 Sums over Finite Sets . . . 122
11.5 Integral Domains . . . 123
11.6 Fields . . . 123
11.7 Morphisms . . . 124

11.8 Jeremy Avigad's `More_Finite_Product` material 126
11.9 Jeremy Avigad's `More_Ring` material 126

12 Modules over an Abelian Group **127**
12.1 Definitions 127
12.2 Basic Properties of Modules 129
12.3 Submodules 129
12.4 More Lifting from Groups to Abelian Groups 130
12.4.1 Definitions 130
12.4.2 Cosets 132
12.4.3 Subgroups 134
12.4.4 Additive subgroups are normal 134
12.4.5 Congruence Relation 137
12.4.6 Factorization 138
12.4.7 The First Isomorphism Theorem 139
12.4.8 Homomorphisms 139
12.4.9 Cosets 141
12.4.10 Addition of Subgroups 142

13 Ideals **143**
13.1 Definitions 143
13.1.1 General definition 143
13.1.2 Ideals Generated by a Subset of `carrier R` 143
13.1.3 Principal Ideals 143
13.1.4 Maximal Ideals 144
13.1.5 Prime Ideals 144
13.2 Special Ideals 145
13.3 General Ideal Properties 145
13.4 Intersection of Ideals 146
13.5 Addition of Ideals 146
13.6 Ideals generated by a subset of `carrier R` 146
13.7 Union of Ideals 148
13.8 Properties of Principal Ideals 148
13.9 Prime Ideals 149
13.10 Maximal Ideals 149
13.11 Derived Theorems 149

14 Homomorphisms of Non-Commutative Rings **150**
14.1 The Kernel of a Ring Homomorphism 151
14.2 Cosets 152

15 Univariate Polynomials **152**
15.1 The Constructor for Univariate Polynomials . . . 153
15.2 Effect of Operations on Coefficients . . . 154
15.3 Polynomials Form a Ring. . . . 155
15.4 Polynomials Form a Commutative Ring. . . . 157
15.5 Polynomials over a commutative ring for a commutative ring 157
15.6 Polynomials Form an Algebra . . . 158
15.7 Further Lemmas Involving Monomials . . . 159
15.8 The Degree Function . . . 160
15.9 Polynomials over Integral Domains . . . 163
15.10 The Evaluation Homomorphism and Universal Property . . . 163
15.11 The long division algorithm: some previous facts. . . . 167
15.12 The long division proof for commutative rings . . . 168
15.13 Sample Application of Evaluation Homomorphism . . . 170

16 Generated Groups **171**
16.1 Generated Groups . . . 171
16.1.1 Basic Properties . . . 171
16.2 Derived Subgroup . . . 172
16.2.1 Definitions . . . 172
16.2.2 Basic Properties . . . 173
16.2.3 Generated subgroup of a group . . . 175
16.3 And homomorphisms . . . 177

17 Elementary Group Constructions **178**
17.1 Direct sum/product lemmas . . . 178
17.2 The one-element group on a given object . . . 180
17.3 Similarly, trivial groups . . . 181
17.4 The additive group of integers . . . 182
17.5 Additive group of integers modulo n (n = 0 gives just the integers) . . . 183
17.6 Cyclic groups . . . 184

18 Simplification Rules for Polynomials **185**

19 Properties of the Euler φ-function **186**

20 Order of an Element of a Group **188**

21 Number of Roots of a Polynomial **192**

22 The Multiplicative Group of a Field **193**

23 Group Actions **194**
23.1 Prelimineries . . . 195
23.2 Orbits . . . 195
23.2.1 Transitive Actions . . . 196
23.3 Stabilizers . . . 197
23.4 The Orbit-Stabilizer Theorem . . . 197
23.4.1 Rcosets - Supporting Lemmas . . . 197
23.4.2 Bijection Between Rcosets and an Orbit - Definition and Supporting Lemmas . . . 198
23.4.3 The Theorem . . . 198
23.5 The Burnside's Lemma . . . 199
23.5.1 Sums and Cardinals . . . 199
23.5.2 The Lemma . . . 199
23.6 Action by Conjugation . . . 199
23.6.1 Action Over Itself . . . 199
23.6.2 Action Over The Set of Subgroups . . . 200
23.6.3 Action Over The Power Set . . . 201
23.7 Subgroup of an Acting Group . . . 201

24 The Zassenhaus Lemma **202**
24.1 Lemmas about normalizer . . . 202
24.2 Second Isomorphism Theorem . . . 202
24.3 The Zassenhaus Lemma . . . 203

25 Divisibility in monoids and rings **204**

26 Factorial Monoids **204**
26.1 Monoids with Cancellation Law . . . 204
26.2 Products of Units in Monoids . . . 205
26.3 Divisibility and Association . . . 205
26.3.1 Function definitions . . . 205
26.3.2 Divisibility . . . 206
26.3.3 Association . . . 208
26.3.4 Division and associativity . . . 209
26.3.5 Multiplication and associativity . . . 210
26.3.6 Units . . . 210
26.3.7 Proper factors . . . 211
26.4 Irreducible Elements and Primes . . . 213
26.4.1 Irreducible elements . . . 213
26.4.2 Prime elements . . . 214
26.5 Factorization and Factorial Monoids . . . 215
26.5.1 Function definitions . . . 215
26.5.2 Comparing lists of elements . . . 216
26.5.3 Properties of lists of elements . . . 218

26.5.4 Factorization in irreducible elements . . . 218
26.5.5 Essentially equal factorizations . . . 220
26.5.6 Factorial monoids and wfactors . . . 223
26.6 Factorizations as Multisets . . . 223
26.6.1 Comparing multisets . . . 224
26.6.2 Interpreting multisets as factorizations . . . 225
26.6.3 Multiplication on multisets . . . 225
26.6.4 Divisibility on multisets . . . 225
26.7 Irreducible Elements are Prime . . . 227
26.8 Greatest Common Divisors and Lowest Common Multiples . 227
26.8.1 Definitions . . . 227
26.8.2 Connections to `Lattice.thy` . . . 228
26.8.3 Existence of gcd and lcm . . . 228
26.9 Conditions for Factoriality . . . 229
26.9.1 Gcd condition . . . 229
26.9.2 Divisor chain condition . . . 231
26.9.3 Primeness condition . . . 231
26.9.4 Application to factorial monoids . . . 231
26.10Factoriality Theorems . . . 233

27 Quotient Rings **233**
27.1 Multiplication on Cosets . . . 233
27.2 Quotient Ring Definition . . . 233
27.3 Factorization over General Ideals . . . 234
27.4 Factorization over Prime Ideals . . . 234
27.5 Factorization over Maximal Ideals . . . 234
27.6 Isomorphism . . . 237

28 The Ring of Integers **241**
28.1 Some properties of `int` . . . 241
28.2 $\mathcal{Z}$: The Set of Integers as Algebraic Structure . . . 242
28.3 Interpretations . . . 242
28.4 Generated Ideals of $\mathcal{Z}$ . . . 243
28.5 Ideals and Divisibility . . . 244
28.6 Ideals and the Modulus . . . 244
28.7 Factorization . . . 244

29 Weak Morphisms **245**
29.1 Definitions . . . 245
29.2 Weak Group Morphisms . . . 246
29.3 Weak Ring Morphisms . . . 247
29.4 Injective Functions . . . 249

30 Examples **249**
30.1 Direct Product . . . 250
30.1.1 Basic Properties . . . 250

31 The Arithmetic of Rings **251**
31.1 Definitions . . . 251
31.2 The cancellative monoid of a domain. . . . 252
31.3 Passing from `R` to `Ring_Divisibility.mult_of R` and vice-versa. 252
31.4 Irreducible . . . 254
31.5 Primes . . . 255
31.6 Basic Properties . . . 255
31.7 Noetherian Rings . . . 256
31.8 Principal Domains . . . 256
31.9 Euclidean Domains . . . 257

32 Subrings **258**
32.1 Definitions . . . 258
32.2 Basic Properties . . . 258
32.2.1 Subrings . . . 258
32.2.2 Subcrings . . . 259
32.2.3 Subdomains . . . 260
32.2.4 Subfields . . . 261
32.3 Subring Homomorphisms . . . 262

33 Polynomials **263**
33.1 Definitions . . . 263
33.2 Basic Properties . . . 264
33.3 Polynomial Addition . . . 267
33.4 Dense Representation . . . 270
33.5 Polynomial Multiplication . . . 270
33.6 Properties Within a Domain . . . 271
33.7 Algebraic Structure of Polynomials . . . 274
33.8 Long Division Theorem . . . 275
33.9 Consistency Rules . . . 276
33.9.1 Corollaries . . . 277
33.10 The Evaluation Homomorphism . . . 277
33.11 Homomorphisms . . . 279
33.12 The X Variable . . . 279
33.13 The Constant Term . . . 281
33.14 The Canonical Embedding of K in K[X] . . . 282

34 Definitions **283**
34.0.1 Syntactic Definitions . . . 283
34.1 Basic Properties - First Part . . . 284
34.2 Some Basic Properties of Linear Independence . . . 286
34.3 Basic Properties - Second Part . . . 286
34.4 Span as Linear Combinations . . . 287
34.4.1 Corollaries . . . 287
34.5 Span as the minimal subgroup that contains `K <#> set Us` . . . 288
34.5.1 Corollaries . . . 288
34.6 Characterisation of Linearly Independent "Sets" . . . 289
34.7 Replacement Theorem . . . 290
34.8 Dimension . . . 291
34.9 Finite Dimension . . . 294
34.9.1 Basic Properties . . . 294
34.9.2 Reformulation of some lemmas in this new language. . . 296

35 Divisibility of Polynomials **297**
35.1 Definitions . . . 297
35.2 Basic Properties . . . 297
35.3 Division . . . 300
35.4 Polynomial Power . . . 303
35.5 Ideals . . . 304
35.6 Roots and Multiplicity . . . 305
35.7 Link between `pmod` and `rupture_surj` . . . 309
35.8 Dimension . . . 310

36 Indexed Polynomials **311**
36.1 Definitions . . . 311
36.2 Basic Properties . . . 312
36.3 Indexed Eval . . . 313
36.4 Link with Weak Morphisms . . . 315

37 Finite Extensions **317**
37.1 Definitions . . . 317
37.2 Basic Properties . . . 317
37.3 Minimal Polynomial . . . 319
37.4 Simple Extensions . . . 319
37.5 Link between dimension of K-algebras and algebraic extensions 321
37.6 Finite Extensions . . . 322
37.7 Arithmetic of algebraic numbers . . . 324

38 Algebraic Closure **324**
38.1 Definitions . . . 324
38.2 Basic Properties . . . 325
38.3 Partial Order . . . 325
38.4 Extensions Non Empty . . . 326
38.5 Chains . . . 326
38.6 Zorn . . . 328
38.7 Existence of roots . . . 328
38.8 Existence of Algebraic Closure . . . 328
38.9 Definition . . . 330
38.10 The algebraic closure is algebraically closed . . . 331
38.11 Converting between the base field and the closure . . . 332
38.12 The algebraic closure is an algebraic extension . . . 334

39 The Residue Ring as a HOL-Algebra Structure **335**
39.1 A locale for residue rings . . . 336
39.2 Prime residues . . . 338
39.3 Euler's theorem . . . 339
39.4 Wilson's theorem . . . 339

40 Product of Ideals **340**
40.1 Basic Properties . . . 340
40.2 Structure of the Set of Ideals . . . 342
40.3 Another Characterization of Prime Ideals . . . 342

41 Direct Product of Rings **343**
41.1 Definitions . . . 343
41.2 Basic Properties . . . 343
41.3 Direct Product of a List of Rings . . . 345

42 Chinese Remainder Theorem **346**
42.1 Definitions . . . 346
42.2 Chinese Remainder Simple . . . 347
42.3 Chinese Remainder Generalized . . . 347

43 Generated Rings **348**
43.1 Basic Properties of Generated Rings - First Part . . . 349
43.2 Basic Properties of Generated Rings - First Part . . . 350

44 Product and Sum Groups **352**
44.1 Product of a Family of Groups . . . 352
44.2 Sum of a Family of Groups . . . 353

45 Free Abelian Groups **355**
45.1 Generalised finite product 355
45.2 Free Abelian groups on a set, using the "frag" type constructor. 357

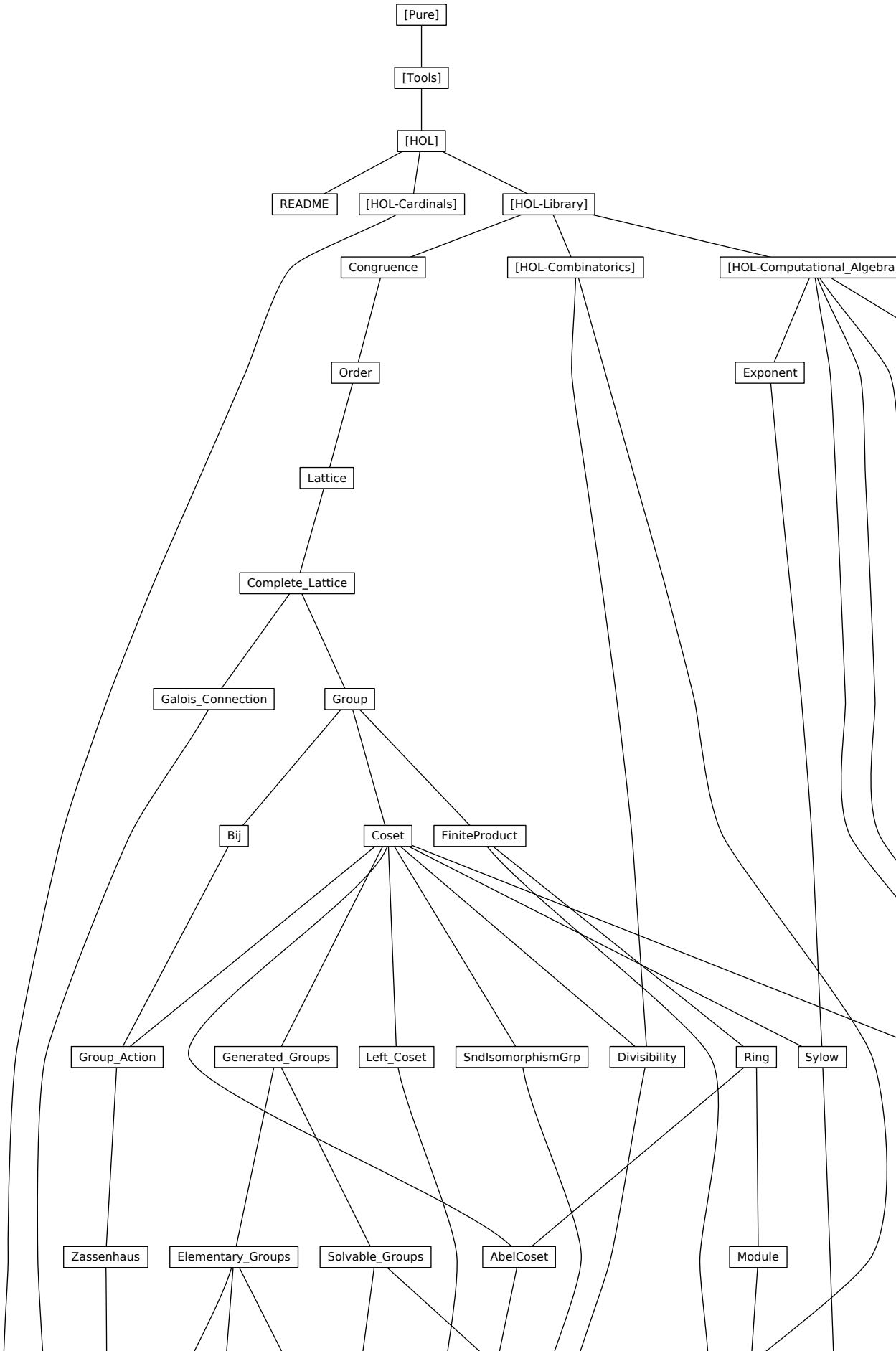
46 Solvable Groups **359**
46.1 Definitions 359
46.2 Solvable Groups and Derived Subgroups 360
46.3 Short Exact Sequences 360

47 Symmetric Groups **361**
47.1 Definitions 361
47.2 Basic Properties 362
47.3 Transposition Sequences 363
47.4 Unsolvability of Symmetric Groups 365

48 Exact Sequences **366**
48.1 Definitions 366
48.2 Basic Properties 366
48.3 Link Between Exact Sequences and Solvable Conditions . . . 367
48.4 Splitting lemmas and Short exact sequences 368

49 Simple Groups **372**

50 The Second Isomorphism Theorem for Groups **373**


[Pure]
[Tools]
[HOL]
README
[HOL-Cardinals]
[HOL-Library]
Congruence
[HOL-Combinatorics]
[HOL-Computational_Algebra
Order
Exponent
Lattice
Complete_Lattice
Galois_Connection
Group
Bij
Coset
FiniteProduct
Group_Action
Generated_Groups
Left_Coset
SndIsomorphismGrp
Divisibility
Ring
Sylow
Zassenhaus
Elementary_Groups
Solvable_Groups
AbelCoset
Module

```
theory Congruence
  imports
    Main
    "HOL-Library.FuncSet"
begin
```

1 Objects

1.1 Structure with Carrier Set.

```
record 'a partial_object =
  carrier :: "'a set"

lemma funcset_carrier:
  "⟦ f ∈ carrier X → carrier Y; x ∈ carrier X ⟧ ⟹ f x ∈ carrier Y"
  ⟨proof⟩

lemma funcset_carrier':
  "⟦ f ∈ carrier A → carrier A; x ∈ carrier A ⟧ ⟹ f x ∈ carrier A"
  ⟨proof⟩
```

1.2 Structure with Carrier and Equivalence Relation `eq`

```
record 'a eq_object = "'a partial_object" +
  eq :: "'a ⇒ 'a ⇒ bool" (infixl <.=ı> 50)

definition
  elem :: "_ ⇒ 'a ⇒ 'a set ⇒ bool" (infixl <.∈ı> 50)
  where "x .∈S A ⟷ (∃y ∈ A. x .=S y)"

definition
  set_eq :: "_ ⇒ 'a set ⇒ 'a set ⇒ bool" (infixl <{.=}ı> 50)
  where "A {.=}S B ⟷ ((∀x ∈ A. x .∈S B) ∧ (∀x ∈ B. x .∈S A))"

definition
  eq_class_of :: "_ ⇒ 'a ⇒ 'a set" (<class'_ofı>)
  where "class_ofS x = {y ∈ carrier S. x .=S y}"

definition
  eq_classes :: "_ ⇒ ('a set) set" (<classesı>)
  where "classesS = {class_ofS x | x. x ∈ carrier S}"

definition
  eq_closure_of :: "_ ⇒ 'a set ⇒ 'a set" (<closure'_ofı>)
  where "closure_ofS A = {y ∈ carrier S. y .∈S A}"

definition
```

```
  eq_is_closed :: "_ ⇒ 'a set ⇒ bool" (‹is'_closed› )
  where "is_closed_S A ⟷ A ⊆ carrier S ∧ closure_of_S A = A"

abbreviation
  not_eq :: "_ ⇒ 'a ⇒ 'a ⇒ bool" (infixl ‹.≠› 50)
  where "x .≠_S y ≡ ¬(x .=_S y)"

abbreviation
  not_elem :: "_ ⇒ 'a ⇒ 'a set ⇒ bool" (infixl ‹.∉› 50)
  where "x .∉_S A ≡ ¬(x .∈_S A)"

abbreviation
  set_not_eq :: "_ ⇒ 'a set ⇒ 'a set ⇒ bool" (infixl ‹{.≠}› 50)
  where "A {.≠}_S B ≡ ¬(A {.=}_S B)"

locale equivalence =
  fixes S (structure)
  assumes refl [simp, intro]: "x ∈ carrier S ⟹ x .= x"
    and sym [sym]: "⟦ x .= y; x ∈ carrier S; y ∈ carrier S ⟧ ⟹ y .=
x"
    and trans [trans]:
      "⟦ x .= y; y .= z; x ∈ carrier S; y ∈ carrier S; z ∈ carrier S ⟧
⟹ x .= z"

lemma equivalenceI:
  fixes P :: "'a ⇒ 'a ⇒ bool" and E :: "'a set"
  assumes refl: "⋀x.     ⟦ x ∈ E ⟧ ⟹ P x x"
    and    sym: "⋀x y.   ⟦ x ∈ E; y ∈ E ⟧ ⟹ P x y ⟹ P y x"
    and  trans: "⋀x y z. ⟦ x ∈ E; y ∈ E; z ∈ E ⟧ ⟹ P x y ⟹ P y z
⟹ P x z"
  shows "equivalence (| carrier = E, eq = P |)"
  ⟨proof⟩

locale partition =
  fixes A :: "'a set" and B :: "('a set) set"
  assumes unique_class: "⋀a. a ∈ A ⟹ ∃!b ∈ B. a ∈ b"
    and incl: "⋀b. b ∈ B ⟹ b ⊆ A"

lemma equivalence_subset:
  assumes "equivalence L" "A ⊆ carrier L"
  shows "equivalence (L(| carrier := A |))"
⟨proof⟩



lemma elemI:
  fixes R (structure)
  assumes "a' ∈ A" "a .= a'"
```

shows "a .∈ A"
⟨*proof*⟩

lemma (**in** equivalence) elem_exact:
assumes "a ∈ carrier S" "a ∈ A"
shows "a .∈ A"
⟨*proof*⟩

lemma elemE:
fixes S (**structure**)
assumes "a .∈ A"
and "⋀a'. ⟦a' ∈ A; a .= a'⟧ ⟹ P"
shows "P"
⟨*proof*⟩

lemma (**in** equivalence) elem_cong_l [trans]:
assumes "a ∈ carrier S" "a' ∈ carrier S" "A ⊆ carrier S"
and "a' .= a" "a .∈ A"
shows "a' .∈ A"
⟨*proof*⟩

lemma (**in** equivalence) elem_subsetD:
assumes "A ⊆ B" "a .∈ A"
shows "a .∈ B"
⟨*proof*⟩

lemma (**in** equivalence) mem_imp_elem [simp, intro]:
assumes "x ∈ carrier S"
shows "x ∈ A ⟹ x .∈ A"
⟨*proof*⟩

lemma set_eqI:
fixes R (**structure**)
assumes "⋀a. a ∈ A ⟹ a .∈ B"
and "⋀b. b ∈ B ⟹ b .∈ A"
shows "A {.=} B"
⟨*proof*⟩

lemma set_eqI2:
fixes R (**structure**)
assumes "⋀a. a ∈ A ⟹ ∃b ∈ B. a .= b"
and "⋀b. b ∈ B ⟹ ∃a ∈ A. b .= a"
shows "A {.=} B"
⟨*proof*⟩

lemma set_eqD1:
fixes R (**structure**)
assumes "A {.=} A'" **and** "a ∈ A"
shows "∃a'∈A'. a .= a'"

⟨*proof*⟩

lemma set_eqD2:
fixes R **(structure)**
assumes "A {.=} A'" **and** "a' ∈ A'"
shows "∃a∈A. a' .= a"
⟨*proof*⟩

lemma set_eqE:
fixes R **(structure)**
assumes "A {.=} B"
and "⟦ ∀a ∈ A. a .∈ B; ∀b ∈ B. b .∈ A ⟧ ⟹ P"
shows "P"
⟨*proof*⟩

lemma set_eqE2:
fixes R **(structure)**
assumes "A {.=} B"
and "⟦ ∀a ∈ A. ∃b ∈ B. a .= b; ∀b ∈ B. ∃a ∈ A. b .= a ⟧ ⟹ P"
shows "P"
⟨*proof*⟩

lemma set_eqE':
fixes R **(structure)**
assumes "A {.=} B" "a ∈ A" "b ∈ B"
and "⋀a' b'. ⟦ a' ∈ A; b' ∈ B ⟧ ⟹ b .= a' ⟹ a .= b' ⟹ P"
shows "P"
⟨*proof*⟩

lemma **(in** equivalence**)** eq_elem_cong_r [trans]:
assumes "A ⊆ carrier S" "A' ⊆ carrier S" "A {.=} A'"
shows "⟦ a ∈ carrier S ⟧ ⟹ a .∈ A ⟹ a .∈ A'"
⟨*proof*⟩

lemma **(in** equivalence**)** set_eq_sym [sym]:
assumes "A ⊆ carrier S" "B ⊆ carrier S"
shows "A {.=} B ⟹ B {.=} A"
⟨*proof*⟩

lemma **(in** equivalence**)** equal_set_eq_trans [trans]:
"⟦ A = B; B {.=} C ⟧ ⟹ A {.=} C"
⟨*proof*⟩

lemma **(in** equivalence**)** set_eq_equal_trans [trans]:
"⟦ A {.=} B; B = C ⟧ ⟹ A {.=} C"
⟨*proof*⟩

lemma **(in** equivalence**)** set_eq_trans_aux:
assumes "A ⊆ carrier S" "B ⊆ carrier S" "C ⊆ carrier S"

and "A {.=} B" "B {.=} C"
shows "⋀a. a ∈ A ⟹ a .∈ C"
⟨*proof*⟩

corollary (**in** equivalence) set_eq_trans [trans]:
assumes "A ⊆ carrier S" "B ⊆ carrier S" "C ⊆ carrier S"
and "A {.=} B" " B {.=} C"
shows "A {.=} C"
⟨*proof*⟩

lemma (**in** equivalence) is_closedI:
assumes closed: "⋀x y. ⟦x .= y; x ∈ A; y ∈ carrier S⟧ ⟹ y ∈ A"
and S: "A ⊆ carrier S"
shows "is_closed A"
⟨*proof*⟩

lemma (**in** equivalence) closure_of_eq:
assumes "A ⊆ carrier S" "x ∈ closure_of A"
shows "⟦ x' ∈ carrier S; x .= x' ⟧ ⟹ x' ∈ closure_of A"
⟨*proof*⟩

lemma (**in** equivalence) is_closed_eq [dest]:
assumes "is_closed A" "x ∈ A"
shows "⟦ x .= x'; x' ∈ carrier S ⟧ ⟹ x' ∈ A"
⟨*proof*⟩

corollary (**in** equivalence) is_closed_eq_rev [dest]:
assumes "is_closed A" "x' ∈ A"
shows "⟦ x .= x'; x ∈ carrier S ⟧ ⟹ x ∈ A"
⟨*proof*⟩

lemma closure_of_closed [simp, intro]:
fixes S (**structure**)
shows "closure_of A ⊆ carrier S"
⟨*proof*⟩

lemma closure_of_memI:
fixes S (**structure**)
assumes "a .∈ A" "a ∈ carrier S"
shows "a ∈ closure_of A"
⟨*proof*⟩

lemma closure_ofI2:
fixes S (**structure**)
assumes "a .= a'" **and** "a' ∈ A" **and** "a ∈ carrier S"
shows "a ∈ closure_of A"
⟨*proof*⟩

lemma closure_of_memE:

fixes S **(structure)**
assumes "a ∈ closure_of A"
and "⟦a ∈ carrier S; a .∈ A⟧ ⟹ P"
shows "P"
⟨*proof*⟩

lemma closure_ofE2:
fixes S **(structure)**
assumes "a ∈ closure_of A"
and "⋀a'. ⟦a ∈ carrier S; a' ∈ A; a .= a'⟧ ⟹ P"
shows "P"
⟨*proof*⟩

lemma (in partition**)** equivalence_from_partition:
"equivalence (| carrier = A, eq = (λx y. y ∈ (THE b. b ∈ B ∧ x ∈ b)) |)"
⟨*proof*⟩

lemma (in partition**)** partition_coverture: "⋃B = A"
⟨*proof*⟩

lemma (in partition**)** disjoint_union:
assumes "b1 ∈ B" "b2 ∈ B"
and "b1 ∩ b2 ≠ {}"
shows "b1 = b2"
⟨*proof*⟩

lemma partitionI:
fixes A :: "'a set" **and** B :: "('a set) set"
assumes "⋃B = A"
and "⋀b1 b2. ⟦ b1 ∈ B; b2 ∈ B ⟧ ⟹ b1 ∩ b2 ≠ {} ⟹ b1 = b2"
shows "partition A B"
⟨*proof*⟩

lemma (in partition**)** remove_elem:
assumes "b ∈ B"
shows "partition (A - b) (B - {b})"
⟨*proof*⟩

lemma disjoint_sum:
"⟦ finite B; finite A; partition A B⟧ ⟹ (∑b∈B. ∑a∈b. f a) = (∑a∈A. f a)"
⟨*proof*⟩

lemma (in partition**)** disjoint_sum:
assumes "finite A"
shows "(∑b∈B. ∑a∈b. f a) = (∑a∈A. f a)"
⟨*proof*⟩

lemma (in equivalence**)** set_eq_insert_aux:
assumes "A ⊆ carrier S"
and "x ∈ carrier S" "x' ∈ carrier S" "x .= x'"
and "y ∈ insert x A"
shows "y .∈ insert x' A"
⟨*proof*⟩

corollary (in equivalence**)** set_eq_insert:
assumes "A ⊆ carrier S"
and "x ∈ carrier S" "x' ∈ carrier S" "x .= x'"
shows "insert x A {.=} insert x' A"
⟨*proof*⟩

lemma (in equivalence**)** set_eq_pairI:
assumes xx': "x .= x'"
and carr: "x ∈ carrier S" "x' ∈ carrier S" "y ∈ carrier S"
shows "{x, y} {.=} {x', y}"
⟨*proof*⟩

lemma (in equivalence**)** closure_inclusion:
assumes "A ⊆ B"
shows "closure_of A ⊆ closure_of B"
⟨*proof*⟩

lemma (in equivalence**)** classes_small:
assumes "is_closed B"
and "A ⊆ B"
shows "closure_of A ⊆ B"
⟨*proof*⟩

lemma (in equivalence**)** classes_eq:
assumes "A ⊆ carrier S"
shows "A {.=} closure_of A"
⟨*proof*⟩

lemma (in equivalence**)** complete_classes:
assumes "is_closed A"
shows "A = closure_of A"
⟨*proof*⟩

lemma (in equivalence**)** closure_idem_weak:
"closure_of (closure_of A) {.=} closure_of A"
⟨*proof*⟩

lemma (in equivalence**)** closure_idem_strong:
assumes "A ⊆ carrier S"
shows "closure_of (closure_of A) = closure_of A"
⟨*proof*⟩

lemma (in equivalence**)** classes_consistent:
  assumes "A ⊆ carrier S"
  shows "is_closed (closure_of A)"
  ⟨*proof*⟩

lemma (in equivalence**)** classes_coverture:
  "⋃classes = carrier S"
⟨*proof*⟩

lemma (in equivalence**)** disjoint_union:
  assumes "class1 ∈ classes" "class2 ∈ classes"
    and "class1 ∩ class2 ≠ {}"
    shows "class1 = class2"
⟨*proof*⟩

lemma (in equivalence**)** partition_from_equivalence:
  "partition (carrier S) classes"
⟨*proof*⟩

lemma (in equivalence**)** disjoint_sum:
  assumes "finite (carrier S)"
  shows "(∑c∈classes. ∑x∈c. f x) = (∑x∈(carrier S). f x)"
⟨*proof*⟩

end

theory Order
  imports
    Congruence
begin

2 Orders

2.1 Partial Orders

record 'a gorder = "'a eq_object" +
  le :: "['a, 'a] => bool" (**infixl** ‹⊑ı› 50)

abbreviation inv_gorder :: "_ ⇒ 'a gorder" **where**
  "inv_gorder L ≡
   (| carrier = carrier L,
     eq = (.=$_L$),
     le = (λ x y. y ⊑$_L$ x) |)"

lemma inv_gorder_inv:
  "inv_gorder (inv_gorder L) = L"
  ⟨*proof*⟩

locale weak_partial_order = equivalence L **for** L (**structure**) +
  assumes le_refl [intro, simp]:
      "x ∈ carrier L ⟹ x ⊑ x"
    and weak_le_antisym [intro]:
      "⟦x ⊑ y; y ⊑ x; x ∈ carrier L; y ∈ carrier L⟧ ⟹ x .= y"
    and le_trans [trans]:
      "⟦x ⊑ y; y ⊑ z; x ∈ carrier L; y ∈ carrier L; z ∈ carrier L⟧ ⟹
x ⊑ z"
    and le_cong:
      "⟦x .= y; z .= w; x ∈ carrier L; y ∈ carrier L; z ∈ carrier L; w
∈ carrier L⟧ ⟹
      x ⊑ z ⟷ y ⊑ w"

definition
  lless :: "[_, 'a, 'a] => bool" (**infixl** ‹⊏ı› 50)
  where "x ⊏L y ⟷ x ⊑L y ∧ x .≠L y"

2.1.1 The order relation

context weak_partial_order
begin

lemma le_cong_l [intro, trans]:
  "⟦x .= y; y ⊑ z; x ∈ carrier L; y ∈ carrier L; z ∈ carrier L⟧ ⟹ x
⊑ z"
  ⟨*proof*⟩

lemma le_cong_r [intro, trans]:
  "⟦x ⊑ y; y .= z; x ∈ carrier L; y ∈ carrier L; z ∈ carrier L⟧ ⟹ x
⊑ z"
  ⟨*proof*⟩

lemma weak_refl [intro, simp]: "⟦x .= y; x ∈ carrier L; y ∈ carrier
L⟧ ⟹ x ⊑ y"
  ⟨*proof*⟩

end

lemma weak_llessI:
  fixes R (**structure**)
  assumes "x ⊑ y" **and** "¬(x .= y)"
  shows "x ⊏ y"
  ⟨*proof*⟩

lemma lless_imp_le:
  fixes R (**structure**)
  assumes "x ⊏ y"
  shows "x ⊑ y"
  ⟨*proof*⟩

lemma weak_lless_imp_not_eq:
fixes R **(structure)**
assumes "x ⊏ y"
shows "¬ (x .= y)"
⟨*proof*⟩

lemma weak_llessE:
fixes R **(structure)**
assumes p: "x ⊏ y" **and** e: "⟦x ⊑ y; ¬ (x .= y)⟧ ⟹ P"
shows "P"
⟨*proof*⟩

lemma (in weak_partial_order**)** lless_cong_l [trans]:
assumes xx': "x .= x'"
and xy: "x' ⊏ y"
and carr: "x ∈ carrier L" "x' ∈ carrier L" "y ∈ carrier L"
shows "x ⊏ y"
⟨*proof*⟩

lemma (in weak_partial_order**)** lless_cong_r [trans]:
assumes xy: "x ⊏ y"
and yy': "y .= y'"
and carr: "x ∈ carrier L" "y ∈ carrier L" "y' ∈ carrier L"
shows "x ⊏ y'"
⟨*proof*⟩

lemma (in weak_partial_order**)** lless_antisym:
assumes "a ∈ carrier L" "b ∈ carrier L"
and "a ⊏ b" "b ⊏ a"
shows "P"
⟨*proof*⟩

lemma (in weak_partial_order**)** lless_trans [trans]:
assumes "a ⊏ b" "b ⊏ c"
and carr[simp]: "a ∈ carrier L" "b ∈ carrier L" "c ∈ carrier L"
shows "a ⊏ c"
⟨*proof*⟩

lemma weak_partial_order_subset:
assumes "weak_partial_order L" "A ⊆ carrier L"
shows "weak_partial_order (L⦇ carrier := A ⦈)"
⟨*proof*⟩

2.1.2 Upper and lower bounds of a set

definition
Upper :: "[_, 'a set] => 'a set"

where "Upper L A = {u. (∀x. x ∈ A ∩ carrier L ⟶ x ⊑$_L$ u)} ∩ carrier L"

definition
Lower :: "[_, 'a set] => 'a set"
where "Lower L A = {l. (∀x. x ∈ A ∩ carrier L ⟶ l ⊑$_L$ x)} ∩ carrier L"

lemma Lower_dual [simp]:
"Lower (inv_gorder L) A = Upper L A"
⟨*proof*⟩

lemma Upper_dual [simp]:
"Upper (inv_gorder L) A = Lower L A"
⟨*proof*⟩

lemma (**in** weak_partial_order) equivalence_dual: "equivalence (inv_gorder L)"
⟨*proof*⟩

lemma (**in** weak_partial_order) dual_weak_order: "weak_partial_order (inv_gorder L)"
⟨*proof*⟩

lemma (**in** weak_partial_order) dual_eq_iff [simp]: "A {.=}$_{\text{inv_gorder L}}$ A' ⟷ A {.=} A'"
⟨*proof*⟩

lemma dual_weak_order_iff:
"weak_partial_order (inv_gorder A) ⟷ weak_partial_order A"
⟨*proof*⟩

lemma Upper_closed [iff]:
"Upper L A ⊆ carrier L"
⟨*proof*⟩

lemma Upper_memD [dest]:
fixes L (**structure**)
shows "⟦u ∈ Upper L A; x ∈ A; A ⊆ carrier L⟧ ⟹ x ⊑ u ∧ u ∈ carrier L"
⟨*proof*⟩

lemma (**in** weak_partial_order) Upper_elemD [dest]:
"⟦u .∈ Upper L A; u ∈ carrier L; x ∈ A; A ⊆ carrier L⟧ ⟹ x ⊑ u"
⟨*proof*⟩

lemma Upper_memI:
fixes L (**structure**)
shows "⟦!! y. y ∈ A ⟹ y ⊑ x; x ∈ carrier L⟧ ⟹ x ∈ Upper L A"

⟨*proof*⟩

lemma (in weak_partial_order**)** Upper_elemI:
"⟦!! y. y ∈ A ⟹ y ⊑ x; x ∈ carrier L⟧ ⟹ x .∈ Upper L A"
⟨*proof*⟩

lemma Upper_antimono:
"A ⊆ B ⟹ Upper L B ⊆ Upper L A"
⟨*proof*⟩

lemma (in weak_partial_order**)** Upper_is_closed [simp]:
"A ⊆ carrier L ⟹ is_closed (Upper L A)"
⟨*proof*⟩

lemma (in weak_partial_order**)** Upper_mem_cong:
assumes "a' ∈ carrier L" "A ⊆ carrier L" "a .= a'" "a ∈ Upper L A"
shows "a' ∈ Upper L A"
⟨*proof*⟩

lemma (in weak_partial_order**)** Upper_semi_cong:
assumes "A ⊆ carrier L" "A {.=} A'"
shows "Upper L A ⊆ Upper L A'"
⟨*proof*⟩

lemma (in weak_partial_order**)** Upper_cong:
assumes "A ⊆ carrier L" "A' ⊆ carrier L" "A {.=} A'"
shows "Upper L A = Upper L A'"
⟨*proof*⟩

lemma Lower_closed [intro!, simp]:
"Lower L A ⊆ carrier L"
⟨*proof*⟩

lemma Lower_memD [dest]:
fixes L **(structure)**
shows "⟦l ∈ Lower L A; x ∈ A; A ⊆ carrier L⟧ ⟹ l ⊑ x ∧ l ∈ carrier
L"
⟨*proof*⟩

lemma Lower_memI:
fixes L **(structure)**
shows "⟦!! y. y ∈ A ⟹ x ⊑ y; x ∈ carrier L⟧ ⟹ x ∈ Lower L A"
⟨*proof*⟩

lemma Lower_antimono:
"A ⊆ B ⟹ Lower L B ⊆ Lower L A"
⟨*proof*⟩

lemma (in weak_partial_order**)** Lower_is_closed [simp]:

"A ⊆ carrier L ⟹ is_closed (Lower L A)"
⟨*proof*⟩

lemma (in weak_partial_order**)** Lower_mem_cong:
assumes "a' ∈ carrier L" "A ⊆ carrier L" "a .= a'" "a ∈ Lower L A"
shows "a' ∈ Lower L A"
⟨*proof*⟩

lemma (in weak_partial_order**)** Lower_cong:
assumes "A ⊆ carrier L" "A' ⊆ carrier L" "A {.=} A'"
shows "Lower L A = Lower L A'"
⟨*proof*⟩

Jacobson: Theorem 8.1

lemma Lower_empty [simp]:
"Lower L {} = carrier L"
⟨*proof*⟩

lemma Upper_empty [simp]:
"Upper L {} = carrier L"
⟨*proof*⟩

2.1.3 Least and greatest, as predicate

definition
least :: "[_, 'a, 'a set] => bool"
where "least L l A ⟷ A ⊆ carrier L ∧ l ∈ A ∧ (∀x∈A. l ⊑$_L$ x)"

definition
greatest :: "[_, 'a, 'a set] => bool"
where "greatest L g A ⟷ A ⊆ carrier L ∧ g ∈ A ∧ (∀x∈A. x ⊑$_L$ g)"

Could weaken these to l ∈ carrier L ∧ l .∈ A and g ∈ carrier L ∧ g .∈ A.

lemma least_dual [simp]:
"least (inv_gorder L) x A = greatest L x A"
⟨*proof*⟩

lemma greatest_dual [simp]:
"greatest (inv_gorder L) x A = least L x A"
⟨*proof*⟩

lemma least_closed [intro, simp]:
"least L l A ⟹ l ∈ carrier L"
⟨*proof*⟩

lemma least_mem:
"least L l A ⟹ l ∈ A"
⟨*proof*⟩

lemma (in weak_partial_order**)** weak_least_unique:
"⟦least L x A; least L y A⟧ ⟹ x .= y"
⟨*proof*⟩

lemma least_le:
fixes L **(structure)**
shows "⟦least L x A; a ∈ A⟧ ⟹ x ⊑ a"
⟨*proof*⟩

lemma (in weak_partial_order**)** least_cong:
"⟦x .= x'; x ∈ carrier L; x' ∈ carrier L; is_closed A⟧ ⟹ least L x A = least L x' A"
⟨*proof*⟩

abbreviation is_lub :: "[_, 'a, 'a set] => bool"
where "is_lub L x A ≡ least L x (Upper L A)"

least is not congruent in the second parameter for A {.=} A'

lemma (in weak_partial_order**)** least_Upper_cong_l:
assumes "x .= x'"
and "x ∈ carrier L" "x' ∈ carrier L"
and "A ⊆ carrier L"
shows "least L x (Upper L A) = least L x' (Upper L A)"
⟨*proof*⟩

lemma (in weak_partial_order**)** least_Upper_cong_r:
assumes "A ⊆ carrier L" "A' ⊆ carrier L" "A {.=} A'"
shows "least L x (Upper L A) = least L x (Upper L A')"
⟨*proof*⟩

lemma least_UpperI:
fixes L **(structure)**
assumes above: "!! x. x ∈ A ⟹ x ⊑ s"
and below: "!! y. y ∈ Upper L A ⟹ s ⊑ y"
and L: "A ⊆ carrier L" "s ∈ carrier L"
shows "least L s (Upper L A)"
⟨*proof*⟩

lemma least_Upper_above:
fixes L **(structure)**
shows "⟦least L s (Upper L A); x ∈ A; A ⊆ carrier L⟧ ⟹ x ⊑ s"
⟨*proof*⟩

lemma greatest_closed [intro, simp]:
"greatest L l A ⟹ l ∈ carrier L"
⟨*proof*⟩

lemma greatest_mem:

"greatest L l A ⟹ l ∈ A"
⟨*proof*⟩

lemma (**in** weak_partial_order) weak_greatest_unique:
"⟦greatest L x A; greatest L y A⟧ ⟹ x .= y"
⟨*proof*⟩

lemma greatest_le:
fixes L (**structure**)
shows "⟦greatest L x A; a ∈ A⟧ ⟹ a ⊑ x"
⟨*proof*⟩

lemma (**in** weak_partial_order) greatest_cong:
"⟦x .= x'; x ∈ carrier L; x' ∈ carrier L; is_closed A⟧ ⟹
greatest L x A = greatest L x' A"
⟨*proof*⟩

abbreviation is_glb :: "[_, 'a, 'a set] => bool"
where "is_glb L x A ≡ greatest L x (Lower L A)"

greatest is not congruent in the second parameter for A {.=} A'

lemma (**in** weak_partial_order) greatest_Lower_cong_l:
assumes "x .= x'"
and "x ∈ carrier L" "x' ∈ carrier L"
shows "greatest L x (Lower L A) = greatest L x' (Lower L A)"
⟨*proof*⟩

lemma (**in** weak_partial_order) greatest_Lower_cong_r:
assumes "A ⊆ carrier L" "A' ⊆ carrier L" "A {.=} A'"
shows "greatest L x (Lower L A) = greatest L x (Lower L A')"
⟨*proof*⟩

lemma greatest_LowerI:
fixes L (**structure**)
assumes below: "!! x. x ∈ A ⟹ i ⊑ x"
and above: "!! y. y ∈ Lower L A ⟹ y ⊑ i"
and L: "A ⊆ carrier L" "i ∈ carrier L"
shows "greatest L i (Lower L A)"
⟨*proof*⟩

lemma greatest_Lower_below:
fixes L (**structure**)
shows "⟦greatest L i (Lower L A); x ∈ A; A ⊆ carrier L⟧ ⟹ i ⊑ x"
⟨*proof*⟩

2.1.4 Intervals

definition
at_least_at_most :: "('a, 'c) gorder_scheme ⇒ 'a => 'a => 'a set"

```
    (<(<indent=1 notation=<mixfix interval>>⦃_.._⦄>)>)
  where "⦃l..u⦄_A = {x ∈ carrier A. l ⊑_A x ∧ x ⊑_A u}"

context weak_partial_order
begin

  lemma at_least_at_most_upper [dest]:
    "x ∈ ⦃a..b⦄ ⟹ x ⊑ b"
    ⟨proof⟩

  lemma at_least_at_most_lower [dest]:
    "x ∈ ⦃a..b⦄ ⟹ a ⊑ x"
    ⟨proof⟩

  lemma at_least_at_most_closed: "⦃a..b⦄ ⊆ carrier L"
    ⟨proof⟩

  lemma at_least_at_most_member [intro]:
    "⟦x ∈ carrier L; a ⊑ x; x ⊑ b⟧ ⟹ x ∈ ⦃a..b⦄"
    ⟨proof⟩

end
```

2.1.5 Isotone functions

```
definition isotone :: "('a, 'c) gorder_scheme ⇒ ('b, 'd) gorder_scheme
⇒ ('a ⇒ 'b) ⇒ bool"
  where
  "isotone A B f ≡
   weak_partial_order A ∧ weak_partial_order B ∧
   (∀x∈carrier A. ∀y∈carrier A. x ⊑_A y ⟶ f x ⊑_B f y)"

lemma isotoneI [intro?]:
  fixes f :: "'a ⇒ 'b"
  assumes "weak_partial_order L1"
          "weak_partial_order L2"
          "(⋀x y. ⟦x ∈ carrier L1; y ∈ carrier L1; x ⊑_L1 y⟧
                  ⟹ f x ⊑_L2 f y)"
  shows "isotone L1 L2 f"
  ⟨proof⟩

abbreviation Monotone :: "('a, 'b) gorder_scheme ⇒ ('a ⇒ 'a) ⇒ bool"
    (<(<open_block notation=<prefix Mono>>Mono_)>)
  where "Mono_L f ≡ isotone L L f"

lemma use_iso1:
  "⟦isotone A A f; x ∈ carrier A; y ∈ carrier A; x ⊑_A y⟧ ⟹
   f x ⊑_A f y"
  ⟨proof⟩
```

lemma use_iso2:
"⟦isotone A B f; x ∈ carrier A; y ∈ carrier A; x ⊑$_A$ y⟧ ⟹
f x ⊑$_B$ f y"
⟨*proof*⟩

lemma iso_compose:
"⟦f ∈ carrier A → carrier B; isotone A B f; g ∈ carrier B → carrier C; isotone B C g⟧ ⟹
isotone A C (g ∘ f)"
⟨*proof*⟩

lemma (**in** weak_partial_order) inv_isotone [simp]:
"isotone (inv_gorder A) (inv_gorder B) f = isotone A B f"
⟨*proof*⟩

2.1.6 Idempotent functions

definition idempotent ::
"('a, 'b) gorder_scheme ⇒ ('a ⇒ 'a) ⇒ bool"
(<(<open_block notation=<prefix Idem>>Idem₂)>)
where "Idem$_L$ f ≡ ∀x∈carrier L. f (f x) .=$_L$ f x"

lemma (**in** weak_partial_order) idempotent:
"⟦Idem f; x ∈ carrier L⟧ ⟹ f (f x) .= f x"
⟨*proof*⟩

2.1.7 Order embeddings

definition order_emb :: "('a, 'c) gorder_scheme ⇒ ('b, 'd) gorder_scheme ⇒ ('a ⇒ 'b) ⇒ bool"
where
"order_emb A B f ≡ weak_partial_order A
∧ weak_partial_order B
∧ (∀x∈carrier A. ∀y∈carrier A. f x ⊑$_B$ f y ⟷ x ⊑$_A$ y)"

lemma order_emb_isotone: "order_emb A B f ⟹ isotone A B f"
⟨*proof*⟩

2.1.8 Commuting functions

definition commuting :: "('a, 'c) gorder_scheme ⇒ ('a ⇒ 'a) ⇒ ('a ⇒ 'a) ⇒ bool" **where**
"commuting A f g = (∀x∈carrier A. (f ∘ g) x .=$_A$ (g ∘ f) x)"

2.2 Partial orders where eq is the Equality

locale partial_order = weak_partial_order +
assumes eq_is_equal: "(.=) = (=)"

begin

declare weak_le_antisym [rule del]

lemma le_antisym [intro]:
"⟦x ⊑ y; y ⊑ x; x ∈ carrier L; y ∈ carrier L⟧ ⟹ x = y"
⟨*proof*⟩

lemma lless_eq:
"x ⊏ y ⟷ x ⊑ y ∧ x ≠ y"
⟨*proof*⟩

lemma set_eq_is_eq: "A {.=} B ⟷ A = B"
⟨*proof*⟩

end

lemma (**in** partial_order) dual_order:
"partial_order (inv_gorder L)"
⟨*proof*⟩

lemma dual_order_iff:
"partial_order (inv_gorder A) ⟷ partial_order A"
⟨*proof*⟩

Least and greatest, as predicate

lemma (**in** partial_order) least_unique:
"⟦least L x A; least L y A⟧ ⟹ x = y"
⟨*proof*⟩

lemma (**in** partial_order) greatest_unique:
"⟦greatest L x A; greatest L y A⟧ ⟹ x = y"
⟨*proof*⟩

2.3 Bounded Orders

definition
top :: "_ => 'a" (‹⊤ı›) **where**
"⊤$_L$ = (SOME x. greatest L x (carrier L))"

definition
bottom :: "_ => 'a" (‹⊥ı›) **where**
"⊥$_L$ = (SOME x. least L x (carrier L))"

locale weak_partial_order_bottom = weak_partial_order L **for** L (**structure**)
+
assumes bottom_exists: "∃ x. least L x (carrier L)"
begin

lemma bottom_least: "least L ⊥ (carrier L)"
⟨*proof*⟩

lemma bottom_closed [simp, intro]:
"⊥ ∈ carrier L"
⟨*proof*⟩

lemma bottom_lower [simp, intro]:
"x ∈ carrier L ⟹ ⊥ ⊑ x"
⟨*proof*⟩

end

locale weak_partial_order_top = weak_partial_order L **for** L (**structure**)
+
assumes top_exists: "∃ x. greatest L x (carrier L)"
begin

lemma top_greatest: "greatest L ⊤ (carrier L)"
⟨*proof*⟩

lemma top_closed [simp, intro]:
"⊤ ∈ carrier L"
⟨*proof*⟩

lemma top_higher [simp, intro]:
"x ∈ carrier L ⟹ x ⊑ ⊤"
⟨*proof*⟩

end

2.4 Total Orders

locale weak_total_order = weak_partial_order +
assumes total: "⟦x ∈ carrier L; y ∈ carrier L⟧ ⟹ x ⊑ y ∨ y ⊑ x"

Introduction rule: the usual definition of total order

lemma (**in** weak_partial_order) weak_total_orderI:
assumes total: "!!x y. ⟦x ∈ carrier L; y ∈ carrier L⟧ ⟹ x ⊑ y ∨ y
⊑ x"
shows "weak_total_order L"
⟨*proof*⟩

2.5 Total orders where eq is the Equality

locale total_order = partial_order +
assumes total_order_total: "⟦x ∈ carrier L; y ∈ carrier L⟧ ⟹ x ⊑
y ∨ y ⊑ x"

sublocale total_order < weak?: weak_total_order
⟨*proof*⟩

Introduction rule: the usual definition of total order

lemma (in partial_order**)** total_orderI:
assumes total: "!!x y. ⟦x ∈ carrier L; y ∈ carrier L⟧ ⟹ x ⊑ y ∨ y ⊑ x"
shows "total_order L"
⟨*proof*⟩

end

theory Lattice
imports Order
begin

3 Lattices

3.1 Supremum and infimum

definition
sup :: "[_, 'a set] => 'a" (‹(‹open_block notation=‹prefix ⨆››⨆ı _)› [90] 90)
where "⨆$_L$A = (SOME x. least L x (Upper L A))"

definition
inf :: "[_, 'a set] => 'a" (‹(‹open_block notation=‹prefix ⨅››⨅ı _)› [90] 90)
where "⨅$_L$A = (SOME x. greatest L x (Lower L A))"

definition supr ::
"('a, 'b) gorder_scheme ⇒ 'c set ⇒ ('c ⇒ 'a) ⇒ 'a "
where "supr L A f = ⨆$_L$(f ‘ A)"

definition infi ::
"('a, 'b) gorder_scheme ⇒ 'c set ⇒ ('c ⇒ 'a) ⇒ 'a "
where "infi L A f = ⨅$_L$(f ‘ A)"

syntax
"_inf1" :: "('a, 'b) gorder_scheme ⇒ pttrns ⇒ 'a ⇒ 'a"
(‹(‹indent=3 notation=‹binder IINF››IINFı _./ _)› [0, 10] 10)
"_inf" :: "('a, 'b) gorder_scheme ⇒ pttrn ⇒ 'c set ⇒ 'a ⇒ 'a"
(‹(‹indent=3 notation=‹binder IINF››IINFı _:_./ _)› [0, 0, 10] 10)
"_sup1" :: "('a, 'b) gorder_scheme ⇒ pttrns ⇒ 'a ⇒ 'a"
(‹(‹indent=3 notation=‹binder SSUP››SSUPı _./ _)› [0, 10] 10)
"_sup" :: "('a, 'b) gorder_scheme ⇒ pttrn ⇒ 'c set ⇒ 'a ⇒ 'a"
(‹(‹indent=3 notation=‹binder SSUP››SSUPı _:_./ _)› [0, 0, 10] 10)
syntax_consts

"_inf1" "_inf" == infi **and**
"_sup1" "_sup" == supr
translations
"IINF$_L$ x. B" == "CONST infi L CONST UNIV (%x. B)"
"IINF$_L$ x:A. B" == "CONST infi L A (%x. B)"
"SSUP$_L$ x. B" == "CONST supr L CONST UNIV (%x. B)"
"SSUP$_L$ x:A. B" == "CONST supr L A (%x. B)"

definition
join :: "[_, 'a, 'a] => 'a" (**infixl** <⊔ı> 65)
where "x ⊔$_L$ y = ⨆$_L${x, y}"

definition
meet :: "[_, 'a, 'a] => 'a" (**infixl** <⊓ı> 70)
where "x ⊓$_L$ y = ⨅$_L${x, y}"

definition
LEAST_FP :: "('a, 'b) gorder_scheme ⇒ ('a ⇒ 'a) ⇒ 'a" (<LFPı>) **where**
"LEAST_FP L f = ⨅$_L$ {u ∈ carrier L. f u ⊑$_L$ u}" — least fixed point

definition
GREATEST_FP:: "('a, 'b) gorder_scheme ⇒ ('a ⇒ 'a) ⇒ 'a" (<GFPı>)
where
"GREATEST_FP L f = ⨆$_L$ {u ∈ carrier L. u ⊑$_L$ f u}" — greatest fixed point

3.2 Dual operators

lemma sup_dual [simp]:
"⨆$_{\text{inv_gorder } L}$A = ⨅$_L$A"
⟨*proof*⟩

lemma inf_dual [simp]:
"⨅$_{\text{inv_gorder } L}$A = ⨆$_L$A"
⟨*proof*⟩

lemma join_dual [simp]:
"p ⊔$_{\text{inv_gorder } L}$ q = p ⊓$_L$ q"
⟨*proof*⟩

lemma meet_dual [simp]:
"p ⊓$_{\text{inv_gorder } L}$ q = p ⊔$_L$ q"
⟨*proof*⟩

lemma top_dual [simp]:
"⊤$_{\text{inv_gorder } L}$ = ⊥$_L$"
⟨*proof*⟩

lemma bottom_dual [simp]:

```
  "⊥inv_gorder L = ⊤L"
```
⟨*proof*⟩

lemma LFP_dual [simp]:
```
  "LEAST_FP (inv_gorder L) f = GREATEST_FP L f"
```
⟨*proof*⟩

lemma GFP_dual [simp]:
```
  "GREATEST_FP (inv_gorder L) f = LEAST_FP L f"
```
⟨*proof*⟩

3.3 Lattices

locale weak_upper_semilattice = weak_partial_order +
```
  assumes sup_of_two_exists:
    "[| x ∈ carrier L; y ∈ carrier L |] ==> ∃s. least L s (Upper L {x,
y})"
```

locale weak_lower_semilattice = weak_partial_order +
```
  assumes inf_of_two_exists:
    "[| x ∈ carrier L; y ∈ carrier L |] ==> ∃s. greatest L s (Lower L
{x, y})"
```

locale weak_lattice = weak_upper_semilattice + weak_lower_semilattice

lemma (**in** weak_lattice) dual_weak_lattice:
```
  "weak_lattice (inv_gorder L)"
```
⟨*proof*⟩

3.3.1 Supremum

lemma (**in** weak_upper_semilattice) joinI:
```
  "[| !!l. least L l (Upper L {x, y}) ==> P l; x ∈ carrier L; y ∈ carrier
L |]
  ==> P (x ⊔ y)"
```
⟨*proof*⟩

lemma (**in** weak_upper_semilattice) join_closed [simp]:
```
  "[| x ∈ carrier L; y ∈ carrier L |] ==> x ⊔ y ∈ carrier L"
```
⟨*proof*⟩

lemma (**in** weak_upper_semilattice) join_cong_l:
```
  assumes carr: "x ∈ carrier L" "x' ∈ carrier L" "y ∈ carrier L"
    and xx': "x .= x'"
  shows "x ⊔ y .= x' ⊔ y"
```
⟨*proof*⟩

lemma (**in** weak_upper_semilattice) join_cong_r:
```
  assumes carr: "x ∈ carrier L" "y ∈ carrier L" "y' ∈ carrier L"
    and yy': "y .= y'"
```

shows "x ⊔ y .= x ⊔ y'"
⟨*proof*⟩

lemma (in weak_partial_order**)** sup_of_singletonI:
"x ∈ carrier L ==> least L x (Upper L {x})"
⟨*proof*⟩

lemma (in weak_partial_order**)** weak_sup_of_singleton [simp]:
"x ∈ carrier L ==> ⨆{x} .= x"
⟨*proof*⟩

lemma (in weak_partial_order**)** sup_of_singleton_closed [simp]:
"x ∈ carrier L ⟹ ⨆{x} ∈ carrier L"
⟨*proof*⟩

Condition on A: supremum exists.

lemma (in weak_upper_semilattice**)** sup_insertI:
"[| !!s. least L s (Upper L (insert x A)) ==> P s;
least L a (Upper L A); x ∈ carrier L; A ⊆ carrier L |]
==> P (⨆(insert x A))"
⟨*proof*⟩

lemma (in weak_upper_semilattice**)** finite_sup_least:
"[| finite A; A ⊆ carrier L; A ≠ {} |] ==> least L (⨆A) (Upper L A)"
⟨*proof*⟩

lemma (in weak_upper_semilattice**)** finite_sup_insertI:
assumes P: "!!l. least L l (Upper L (insert x A)) ==> P l"
and xA: "finite A" "x ∈ carrier L" "A ⊆ carrier L"
shows "P (⨆ (insert x A))"
⟨*proof*⟩

lemma (in weak_upper_semilattice**)** finite_sup_closed [simp]:
"[| finite A; A ⊆ carrier L; A ≠ {} |] ==> ⨆A ∈ carrier L"
⟨*proof*⟩

lemma (in weak_upper_semilattice**)** join_left:
"[| x ∈ carrier L; y ∈ carrier L |] ==> x ⊑ x ⊔ y"
⟨*proof*⟩

lemma (in weak_upper_semilattice**)** join_right:
"[| x ∈ carrier L; y ∈ carrier L |] ==> y ⊑ x ⊔ y"
⟨*proof*⟩

lemma (in weak_upper_semilattice**)** sup_of_two_least:
"[| x ∈ carrier L; y ∈ carrier L |] ==> least L (⨆{x, y}) (Upper L
{x, y})"
⟨*proof*⟩

lemma (in weak_upper_semilattice**)** join_le:
assumes sub: "x ⊑ z" "y ⊑ z"
and x: "x ∈ carrier L" **and** y: "y ∈ carrier L" **and** z: "z ∈ carrier L"
shows "x ⊔ y ⊑ z"
⟨*proof*⟩

lemma (in weak_lattice**)** weak_le_iff_meet:
assumes "x ∈ carrier L" "y ∈ carrier L"
shows "x ⊑ y ⟷ (x ⊔ y) .= y"
⟨*proof*⟩

lemma (in weak_upper_semilattice**)** weak_join_assoc_lemma:
assumes L: "x ∈ carrier L" "y ∈ carrier L" "z ∈ carrier L"
shows "x ⊔ (y ⊔ z) .= ⨆{x, y, z}"
⟨*proof*⟩

Commutativity holds for =.

lemma join_comm:
fixes L **(structure)**
shows "x ⊔ y = y ⊔ x"
⟨*proof*⟩

lemma (in weak_upper_semilattice**)** weak_join_assoc:
assumes L: "x ∈ carrier L" "y ∈ carrier L" "z ∈ carrier L"
shows "(x ⊔ y) ⊔ z .= x ⊔ (y ⊔ z)"
⟨*proof*⟩

3.3.2 Infimum

lemma (in weak_lower_semilattice**)** meetI:
"[| !!i. greatest L i (Lower L {x, y}) ==> P i;
x ∈ carrier L; y ∈ carrier L |]
==> P (x ⊓ y)"
⟨*proof*⟩

lemma (in weak_lower_semilattice**)** meet_closed [simp]:
"[| x ∈ carrier L; y ∈ carrier L |] ==> x ⊓ y ∈ carrier L"
⟨*proof*⟩

lemma (in weak_lower_semilattice**)** meet_cong_l:
assumes carr: "x ∈ carrier L" "x' ∈ carrier L" "y ∈ carrier L"
and xx': "x .= x'"
shows "x ⊓ y .= x' ⊓ y"
⟨*proof*⟩

lemma (in weak_lower_semilattice**)** meet_cong_r:
assumes carr: "x ∈ carrier L" "y ∈ carrier L" "y' ∈ carrier L"
and yy': "y .= y'"

shows "x ⊓ y .= x ⊓ y'"
⟨*proof*⟩

lemma (in weak_partial_order**)** inf_of_singletonI:
"x ∈ carrier L ==> greatest L x (Lower L {x})"
⟨*proof*⟩

lemma (in weak_partial_order**)** weak_inf_of_singleton [simp]:
"x ∈ carrier L ==> ⨅{x} .= x"
⟨*proof*⟩

lemma (in weak_partial_order**)** inf_of_singleton_closed:
"x ∈ carrier L ==> ⨅{x} ∈ carrier L"
⟨*proof*⟩

Condition on A: infimum exists.

lemma (in weak_lower_semilattice**)** inf_insertI:
"[| !!i. greatest L i (Lower L (insert x A)) ==> P i;
greatest L a (Lower L A); x ∈ carrier L; A ⊆ carrier L |]
==> P (⨅(insert x A))"
⟨*proof*⟩

lemma (in weak_lower_semilattice**)** finite_inf_greatest:
"[| finite A; A ⊆ carrier L; A ≠ {} |] ==> greatest L (⨅A) (Lower L A)"
⟨*proof*⟩

lemma (in weak_lower_semilattice**)** finite_inf_insertI:
assumes P: "!!i. greatest L i (Lower L (insert x A)) ==> P i"
and xA: "finite A" "x ∈ carrier L" "A ⊆ carrier L"
shows "P (⨅ (insert x A))"
⟨*proof*⟩

lemma (in weak_lower_semilattice**)** finite_inf_closed [simp]:
"[| finite A; A ⊆ carrier L; A ≠ {} |] ==> ⨅A ∈ carrier L"
⟨*proof*⟩

lemma (in weak_lower_semilattice**)** meet_left:
"[| x ∈ carrier L; y ∈ carrier L |] ==> x ⊓ y ⊑ x"
⟨*proof*⟩

lemma (in weak_lower_semilattice**)** meet_right:
"[| x ∈ carrier L; y ∈ carrier L |] ==> x ⊓ y ⊑ y"
⟨*proof*⟩

lemma (in weak_lower_semilattice**)** inf_of_two_greatest:
"[| x ∈ carrier L; y ∈ carrier L |] ==>
greatest L (⨅{x, y}) (Lower L {x, y})"
⟨*proof*⟩

lemma (in weak_lower_semilattice**)** meet_le:
assumes sub: "z ⊑ x" "z ⊑ y"
and x: "x ∈ carrier L" **and** y: "y ∈ carrier L" **and** z: "z ∈ carrier L"
shows "z ⊑ x ⊓ y"
⟨*proof*⟩

lemma (in weak_lattice**)** weak_le_iff_join:
assumes "x ∈ carrier L" "y ∈ carrier L"
shows "x ⊑ y ⟷ x .= (x ⊓ y)"
⟨*proof*⟩

lemma (in weak_lower_semilattice**)** weak_meet_assoc_lemma:
assumes L: "x ∈ carrier L" "y ∈ carrier L" "z ∈ carrier L"
shows "x ⊓ (y ⊓ z) .= ⨅{x, y, z}"
⟨*proof*⟩

lemma meet_comm:
fixes L **(structure)**
shows "x ⊓ y = y ⊓ x"
⟨*proof*⟩

lemma (in weak_lower_semilattice**)** weak_meet_assoc:
assumes L: "x ∈ carrier L" "y ∈ carrier L" "z ∈ carrier L"
shows "(x ⊓ y) ⊓ z .= x ⊓ (y ⊓ z)"
⟨*proof*⟩

Total orders are lattices.

sublocale weak_total_order ⊆ weak?: weak_lattice
⟨*proof*⟩

3.4 Weak Bounded Lattices

locale weak_bounded_lattice =
weak_lattice +
weak_partial_order_bottom +
weak_partial_order_top
begin

lemma bottom_meet: "x ∈ carrier L ⟹ ⊥ ⊓ x .= ⊥"
⟨*proof*⟩

lemma bottom_join: "x ∈ carrier L ⟹ ⊥ ⊔ x .= x"
⟨*proof*⟩

lemma bottom_weak_eq:
"⟦ b ∈ carrier L; ⋀ x. x ∈ carrier L ⟹ b ⊑ x ⟧ ⟹ b .= ⊥"
⟨*proof*⟩

lemma top_join: "x ∈ carrier L ⟹ ⊤ ⊔ x .= ⊤"
⟨*proof*⟩

lemma top_meet: "x ∈ carrier L ⟹ ⊤ ⊓ x .= x"
⟨*proof*⟩

lemma top_weak_eq: "⟦ t ∈ carrier L; ⋀ x. x ∈ carrier L ⟹ x ⊑ t ⟧ ⟹ t .= ⊤"
⟨*proof*⟩

end

sublocale weak_bounded_lattice ⊆ weak_partial_order ⟨*proof*⟩

3.5 Lattices where eq is the Equality

locale upper_semilattice = partial_order +
assumes sup_of_two_exists:
"[| x ∈ carrier L; y ∈ carrier L |] ==> ∃s. least L s (Upper L {x, y})"

sublocale upper_semilattice ⊆ weak?: weak_upper_semilattice
⟨*proof*⟩

locale lower_semilattice = partial_order +
assumes inf_of_two_exists:
"[| x ∈ carrier L; y ∈ carrier L |] ==> ∃s. greatest L s (Lower L {x, y})"

sublocale lower_semilattice ⊆ weak?: weak_lower_semilattice
⟨*proof*⟩

locale lattice = upper_semilattice + lower_semilattice

sublocale lattice ⊆ weak_lattice ⟨*proof*⟩

lemma (**in** lattice) dual_lattice:
"lattice (inv_gorder L)"
⟨*proof*⟩

lemma (**in** lattice) le_iff_join:
assumes "x ∈ carrier L" "y ∈ carrier L"
shows "x ⊑ y ⟷ x = (x ⊓ y)"
⟨*proof*⟩

lemma (**in** lattice) le_iff_meet:
assumes "x ∈ carrier L" "y ∈ carrier L"
shows "x ⊑ y ⟷ (x ⊔ y) = y"

⟨*proof*⟩

Total orders are lattices.

sublocale total_order ⊆ weak?: lattice
⟨*proof*⟩

Functions that preserve joins and meets

definition join_pres :: "('a, 'c) gorder_scheme ⇒ ('b, 'd) gorder_scheme
⇒ ('a ⇒ 'b) ⇒ bool" **where**
"join_pres X Y f ≡ lattice X ∧ lattice Y ∧ (∀ x ∈ carrier X. ∀ y ∈ carrier
X. f (x ⊔$_X$ y) = f x ⊔$_Y$ f y)"

definition meet_pres :: "('a, 'c) gorder_scheme ⇒ ('b, 'd) gorder_scheme
⇒ ('a ⇒ 'b) ⇒ bool" **where**
"meet_pres X Y f ≡ lattice X ∧ lattice Y ∧ (∀ x ∈ carrier X. ∀ y ∈ carrier
X. f (x ⊓$_X$ y) = f x ⊓$_Y$ f y)"

lemma join_pres_isotone:
assumes "f ∈ carrier X → carrier Y" "join_pres X Y f"
shows "isotone X Y f"
⟨*proof*⟩

lemma meet_pres_isotone:
assumes "f ∈ carrier X → carrier Y" "meet_pres X Y f"
shows "isotone X Y f"
⟨*proof*⟩

3.6 Bounded Lattices

locale bounded_lattice =
lattice +
weak_partial_order_bottom +
weak_partial_order_top

sublocale bounded_lattice ⊆ weak_bounded_lattice ⟨*proof*⟩

context bounded_lattice
begin

lemma bottom_eq:
"⟦ b ∈ carrier L; ⋀ x. x ∈ carrier L ⟹ b ⊑ x ⟧ ⟹ b = ⊥"
⟨*proof*⟩

lemma top_eq: "⟦ t ∈ carrier L; ⋀ x. x ∈ carrier L ⟹ x ⊑ t ⟧ ⟹
t = ⊤"
⟨*proof*⟩

end

hide_const (open) Lattice.inf
hide_const (open) Lattice.sup

end

theory Complete_Lattice
imports Lattice
begin

4 Complete Lattices

locale weak_complete_lattice = weak_partial_order +
  assumes sup_exists:
    "[| A ⊆ carrier L |] ==> ∃s. least L s (Upper L A)"
    and inf_exists:
    "[| A ⊆ carrier L |] ==> ∃i. greatest L i (Lower L A)"

sublocale weak_complete_lattice ⊆ weak_lattice
⟨*proof*⟩

Introduction rule: the usual definition of complete lattice

lemma (in weak_partial_order) weak_complete_latticeI:
  assumes sup_exists:
    "!!A. [| A ⊆ carrier L |] ==> ∃s. least L s (Upper L A)"
    and inf_exists:
    "!!A. [| A ⊆ carrier L |] ==> ∃i. greatest L i (Lower L A)"
  shows "weak_complete_lattice L"
  ⟨*proof*⟩

lemma (in weak_complete_lattice) dual_weak_complete_lattice:
  "weak_complete_lattice (inv_gorder L)"
⟨*proof*⟩

lemma (in weak_complete_lattice) supI:
  "[| !!l. least L l (Upper L A) ==> P l; A ⊆ carrier L |]
  ==> P (⨆A)"
⟨*proof*⟩

lemma (in weak_complete_lattice) sup_closed [simp]:
  "A ⊆ carrier L ==> ⨆A ∈ carrier L"
  ⟨*proof*⟩

lemma (in weak_complete_lattice) sup_cong:
  assumes "A ⊆ carrier L" "B ⊆ carrier L" "A {.=} B"
  shows "⨆ A .= ⨆ B"
⟨*proof*⟩

sublocale weak_complete_lattice ⊆ weak_bounded_lattice

⟨*proof*⟩

lemma (in weak_complete_lattice**)** infI:
"[| !!i. greatest L i (Lower L A) ==> P i; A ⊆ carrier L |]
==> P (⨅A)"
⟨*proof*⟩

lemma (in weak_complete_lattice**)** inf_closed [simp]:
"A ⊆ carrier L ==> ⨅A ∈ carrier L"
⟨*proof*⟩

lemma (in weak_complete_lattice**)** inf_cong:
assumes "A ⊆ carrier L" "B ⊆ carrier L" "A {.=} B"
shows "⨅ A .= ⨅ B"
⟨*proof*⟩

theorem (in weak_partial_order**)** weak_complete_lattice_criterion1:
assumes top_exists: "∃g. greatest L g (carrier L)"
and inf_exists:
"⋀A. [| A ⊆ carrier L; A ≠ {} |] ==> ∃i. greatest L i (Lower L
A)"
shows "weak_complete_lattice L"
⟨*proof*⟩

Supremum

declare (in partial_order**)** weak_sup_of_singleton [simp del]

lemma (in partial_order**)** sup_of_singleton [simp]:
"x ∈ carrier L ==> ⨆{x} = x"
⟨*proof*⟩

lemma (in upper_semilattice**)** join_assoc_lemma:
assumes L: "x ∈ carrier L" "y ∈ carrier L" "z ∈ carrier L"
shows "x ⊔ (y ⊔ z) = ⨆{x, y, z}"
⟨*proof*⟩

lemma (in upper_semilattice**)** join_assoc:
assumes L: "x ∈ carrier L" "y ∈ carrier L" "z ∈ carrier L"
shows "(x ⊔ y) ⊔ z = x ⊔ (y ⊔ z)"
⟨*proof*⟩

Infimum

declare (in partial_order**)** weak_inf_of_singleton [simp del]

lemma (in partial_order**)** inf_of_singleton [simp]:
"x ∈ carrier L ==> ⨅{x} = x"
⟨*proof*⟩

Condition on A: infimum exists.

lemma (in lower_semilattice**)** meet_assoc_lemma:
assumes L: "x ∈ carrier L" "y ∈ carrier L" "z ∈ carrier L"
shows "x ⊓ (y ⊓ z) = ⨅{x, y, z}"
⟨*proof*⟩

lemma (in lower_semilattice**)** meet_assoc:
assumes L: "x ∈ carrier L" "y ∈ carrier L" "z ∈ carrier L"
shows "(x ⊓ y) ⊓ z = x ⊓ (y ⊓ z)"
⟨*proof*⟩

4.1 Infimum Laws

context weak_complete_lattice
begin

lemma inf_glb:
assumes "A ⊆ carrier L"
shows "greatest L (⨅A) (Lower L A)"
⟨*proof*⟩

lemma inf_lower:
assumes "A ⊆ carrier L" "x ∈ A"
shows "⨅A ⊑ x"
⟨*proof*⟩

lemma inf_greatest:
assumes "A ⊆ carrier L" "z ∈ carrier L"
"(⋀x. x ∈ A ⟹ z ⊑ x)"
shows "z ⊑ ⨅A"
⟨*proof*⟩

lemma weak_inf_empty [simp]: "⨅{} .= ⊤"
⟨*proof*⟩

lemma weak_inf_carrier [simp]: "⨅carrier L .= ⊥"
⟨*proof*⟩

lemma weak_inf_insert [simp]:
assumes "a ∈ carrier L" "A ⊆ carrier L"
shows "⨅insert a A .= a ⊓ ⨅A"
⟨*proof*⟩

4.2 Supremum Laws

lemma sup_lub:
assumes "A ⊆ carrier L"
shows "least L (⨆A) (Upper L A)"
⟨*proof*⟩

lemma sup_upper:

assumes "A ⊆ carrier L" "x ∈ A"
shows "x ⊑ ⨆A"
⟨*proof*⟩

lemma sup_least:
assumes "A ⊆ carrier L" "z ∈ carrier L"
"(⋀x. x ∈ A ⟹ x ⊑ z)"
shows "⨆A ⊑ z"
⟨*proof*⟩

lemma weak_sup_empty [simp]: "⨆{} .= ⊥"
⟨*proof*⟩

lemma weak_sup_carrier [simp]: "⨆carrier L .= ⊤"
⟨*proof*⟩

lemma weak_sup_insert [simp]:
assumes "a ∈ carrier L" "A ⊆ carrier L"
shows "⨆insert a A .= a ⊔ ⨆A"
⟨*proof*⟩

end

4.3 Fixed points of a lattice

definition "fps L f = {x ∈ carrier L. f x .=$_L$ x}"

abbreviation "fpl L f ≡ L⦇carrier := fps L f⦈"

lemma (**in** weak_partial_order)
use_fps: "x ∈ fps L f ⟹ f x .= x"
⟨*proof*⟩

lemma fps_carrier [simp]:
"fps L f ⊆ carrier L"
⟨*proof*⟩

lemma (**in** weak_complete_lattice) fps_sup_image:
assumes "f ∈ carrier L → carrier L" "A ⊆ fps L f"
shows "⨆ (f ‘ A) .= ⨆ A"
⟨*proof*⟩

lemma (**in** weak_complete_lattice) fps_idem:
assumes "f ∈ carrier L → carrier L" "Idem f"
shows "fps L f {.=} f ‘ carrier L"
⟨*proof*⟩

context weak_complete_lattice
begin

lemma weak_sup_pre_fixed_point:
assumes "f ∈ carrier L → carrier L" "isotone L L f" "A ⊆ fps L f"
shows "(⨆L A) ⊑L f (⨆L A)"
⟨*proof*⟩

lemma weak_sup_post_fixed_point:
assumes "f ∈ carrier L → carrier L" "isotone L L f" "A ⊆ fps L f"
shows "f (⨅L A) ⊑L (⨅L A)"
⟨*proof*⟩

4.3.1 Least fixed points

lemma LFP_closed [intro, simp]:
"LFP f ∈ carrier L"
⟨*proof*⟩

lemma LFP_lowerbound:
assumes "x ∈ carrier L" "f x ⊑ x"
shows "LFP f ⊑ x"
⟨*proof*⟩

lemma LFP_greatest:
assumes "x ∈ carrier L"
"(⋀u. ⟦ u ∈ carrier L; f u ⊑ u ⟧ ⟹ x ⊑ u)"
shows "x ⊑ LFP f"
⟨*proof*⟩

lemma LFP_lemma2:
assumes "Mono f" "f ∈ carrier L → carrier L"
shows "f (LFP f) ⊑ LFP f"
⟨*proof*⟩

lemma LFP_lemma3:
assumes "Mono f" "f ∈ carrier L → carrier L"
shows "LFP f ⊑ f (LFP f)"
⟨*proof*⟩

lemma LFP_weak_unfold:
"⟦ Mono f; f ∈ carrier L → carrier L ⟧ ⟹ LFP f .= f (LFP f)"
⟨*proof*⟩

lemma LFP_fixed_point [intro]:
assumes "Mono f" "f ∈ carrier L → carrier L"
shows "LFP f ∈ fps L f"
⟨*proof*⟩

lemma LFP_least_fixed_point:
assumes "Mono f" "f ∈ carrier L → carrier L" "x ∈ fps L f"

shows "LFP f ⊑ x"
⟨*proof*⟩

lemma LFP_idem:
assumes "f ∈ carrier L → carrier L" "Mono f" "Idem f"
shows "LFP f .= (f ⊥)"
⟨*proof*⟩

4.3.2 Greatest fixed points

lemma GFP_closed [intro, simp]:
"GFP f ∈ carrier L"
⟨*proof*⟩

lemma GFP_upperbound:
assumes "x ∈ carrier L" "x ⊑ f x"
shows "x ⊑ GFP f"
⟨*proof*⟩

lemma GFP_least:
assumes "x ∈ carrier L"
"(⋀u. ⟦ u ∈ carrier L; u ⊑ f u ⟧ ⟹ u ⊑ x)"
shows "GFP f ⊑ x"
⟨*proof*⟩

lemma GFP_lemma2:
assumes "Mono f" "f ∈ carrier L → carrier L"
shows "GFP f ⊑ f (GFP f)"
⟨*proof*⟩

lemma GFP_lemma3:
assumes "Mono f" "f ∈ carrier L → carrier L"
shows "f (GFP f) ⊑ GFP f"
⟨*proof*⟩

lemma GFP_weak_unfold:
"⟦ Mono f; f ∈ carrier L → carrier L ⟧ ⟹ GFP f .= f (GFP f)"
⟨*proof*⟩

lemma (**in** weak_complete_lattice) GFP_fixed_point [intro]:
assumes "Mono f" "f ∈ carrier L → carrier L"
shows "GFP f ∈ fps L f"
⟨*proof*⟩

lemma GFP_greatest_fixed_point:
assumes "Mono f" "f ∈ carrier L → carrier L" "x ∈ fps L f"
shows "x ⊑ GFP f"
⟨*proof*⟩

lemma GFP_idem:
  assumes "f ∈ carrier L → carrier L" "Mono f" "Idem f"
  shows "GFP f .= (f ⊤)"
⟨*proof*⟩

end

4.4 Complete lattices where eq is the Equality

locale complete_lattice = partial_order +
  assumes sup_exists:
    "[| A ⊆ carrier L |] ==> ∃s. least L s (Upper L A)"
    and inf_exists:
    "[| A ⊆ carrier L |] ==> ∃i. greatest L i (Lower L A)"

sublocale complete_lattice ⊆ lattice
⟨*proof*⟩

sublocale complete_lattice ⊆ weak?: weak_complete_lattice
  ⟨*proof*⟩

lemma complete_lattice_lattice [simp]:
  assumes "complete_lattice X"
  shows "lattice X"
⟨*proof*⟩

Introduction rule: the usual definition of complete lattice

lemma (**in** partial_order) complete_latticeI:
  assumes sup_exists:
    "!!A. [| A ⊆ carrier L |] ==> ∃s. least L s (Upper L A)"
    and inf_exists:
    "!!A. [| A ⊆ carrier L |] ==> ∃i. greatest L i (Lower L A)"
  shows "complete_lattice L"
  ⟨*proof*⟩

theorem (**in** partial_order) complete_lattice_criterion1:
  assumes top_exists: "∃g. greatest L g (carrier L)"
    and inf_exists:
      "!!A. [| A ⊆ carrier L; A ≠ {} |] ==> ∃i. greatest L i (Lower L
A)"
  shows "complete_lattice L"
⟨*proof*⟩

4.5 Fixed points

context complete_lattice
begin

lemma LFP_unfold:

"⟦ Mono f; f ∈ carrier L → carrier L ⟧ ⟹ LFP f = f (LFP f)"
⟨*proof*⟩

lemma LFP_const:
"t ∈ carrier L ⟹ LFP (λ x. t) = t"
⟨*proof*⟩

lemma LFP_id:
"LFP id = ⊥"
⟨*proof*⟩

lemma GFP_unfold:
"⟦ Mono f; f ∈ carrier L → carrier L ⟧ ⟹ GFP f = f (GFP f)"
⟨*proof*⟩

lemma GFP_const:
"t ∈ carrier L ⟹ GFP (λ x. t) = t"
⟨*proof*⟩

lemma GFP_id:
"GFP id = ⊤"
⟨*proof*⟩

end

4.6 Interval complete lattices

context weak_complete_lattice
begin

lemma at_least_at_most_Sup: "⟦ a ∈ carrier L; b ∈ carrier L; a ⊑ b ⟧ ⟹ ⨆ {a..b} .= b"
⟨*proof*⟩

lemma at_least_at_most_Inf: "⟦ a ∈ carrier L; b ∈ carrier L; a ⊑ b ⟧ ⟹ ⨅ {a..b} .= a"
⟨*proof*⟩

end

lemma weak_complete_lattice_interval:
assumes "weak_complete_lattice L" "a ∈ carrier L" "b ∈ carrier L" "a ⊑$_L$ b"
shows "weak_complete_lattice (L (| carrier := {a..b}$_L$ |))"
⟨*proof*⟩

4.7 Knaster-Tarski theorem and variants

The set of fixed points of a complete lattice is itself a complete lattice

theorem Knaster_Tarski:
  assumes "weak_complete_lattice L" **and** f: "f ∈ carrier L → carrier L" **and** "isotone L L f"
  shows "weak_complete_lattice (fpl L f)" (**is** "weak_complete_lattice ?L'")
⟨*proof*⟩

theorem Knaster_Tarski_top:
  assumes "weak_complete_lattice L" "isotone L L f" "f ∈ carrier L → carrier L"
  shows "$\top_{\text{fpl L f}}$.=$_{\text{L}}$ GFP$_{\text{L}}$ f"
⟨*proof*⟩

theorem Knaster_Tarski_bottom:
  assumes "weak_complete_lattice L" "isotone L L f" "f ∈ carrier L → carrier L"
  shows "$\bot_{\text{fpl L f}}$.=$_{\text{L}}$ LFP$_{\text{L}}$ f"
⟨*proof*⟩

If a function is both idempotent and isotone then the image of the function forms a complete lattice

theorem Knaster_Tarski_idem:
  assumes "complete_lattice L" "f ∈ carrier L → carrier L" "isotone L L f" "idempotent L f"
  shows "complete_lattice (L⦇carrier := f ‘ carrier L⦈)"
⟨*proof*⟩

theorem Knaster_Tarski_idem_extremes:
  assumes "weak_complete_lattice L" "isotone L L f" "idempotent L f" "f ∈ carrier L → carrier L"
  shows "$\top_{\text{fpl L f}}$.=$_{\text{L}}$ f ($\top_{\text{L}}$)" "$\bot_{\text{fpl L f}}$.=$_{\text{L}}$ f ($\bot_{\text{L}}$)"
⟨*proof*⟩

theorem Knaster_Tarski_idem_inf_eq:
  assumes "weak_complete_lattice L" "isotone L L f" "idempotent L f" "f ∈ carrier L → carrier L"
          "A ⊆ fps L f"
  shows "$\bigsqcap_{\text{fpl L f}}$ A .=$_{\text{L}}$ f ($\bigsqcap_{\text{L}}$ A)"
⟨*proof*⟩

4.8 Examples

4.8.1 The Powerset of a Set is a Complete Lattice

theorem powerset_is_complete_lattice:
  "complete_lattice ⦇carrier = Pow A, eq = (=), le = (⊆)⦈"
  (**is** "complete_lattice ?L")
⟨*proof*⟩

Another example, that of the lattice of subgroups of a group, can be found in Group theory (Section 6.11).

4.9 Limit preserving functions

definition weak_sup_pres :: "('a, 'c) gorder_scheme ⇒ ('b, 'd) gorder_scheme ⇒ ('a ⇒ 'b) ⇒ bool" **where**
"weak_sup_pres X Y f ≡ complete_lattice X ∧ complete_lattice Y ∧ (∀ A ⊆ carrier X. A ≠ {} ⟶ f (⨆X A) = (⨆Y (f ‘ A)))"

definition sup_pres :: "('a, 'c) gorder_scheme ⇒ ('b, 'd) gorder_scheme ⇒ ('a ⇒ 'b) ⇒ bool" **where**
"sup_pres X Y f ≡ complete_lattice X ∧ complete_lattice Y ∧ (∀ A ⊆ carrier X. f (⨆X A) = (⨆Y (f ‘ A)))"

definition weak_inf_pres :: "('a, 'c) gorder_scheme ⇒ ('b, 'd) gorder_scheme ⇒ ('a ⇒ 'b) ⇒ bool" **where**
"weak_inf_pres X Y f ≡ complete_lattice X ∧ complete_lattice Y ∧ (∀ A ⊆ carrier X. A ≠ {} ⟶ f (⨅X A) = (⨅Y (f ‘ A)))"

definition inf_pres :: "('a, 'c) gorder_scheme ⇒ ('b, 'd) gorder_scheme ⇒ ('a ⇒ 'b) ⇒ bool" **where**
"inf_pres X Y f ≡ complete_lattice X ∧ complete_lattice Y ∧ (∀ A ⊆ carrier X. f (⨅X A) = (⨅Y (f ‘ A)))"

lemma weak_sup_pres:
"sup_pres X Y f ⟹ weak_sup_pres X Y f"
⟨*proof*⟩

lemma weak_inf_pres:
"inf_pres X Y f ⟹ weak_inf_pres X Y f"
⟨*proof*⟩

lemma sup_pres_is_join_pres:
assumes "weak_sup_pres X Y f"
shows "join_pres X Y f"
⟨*proof*⟩

lemma inf_pres_is_meet_pres:
assumes "weak_inf_pres X Y f"
shows "meet_pres X Y f"
⟨*proof*⟩

end

theory Galois_Connection
imports Complete_Lattice
begin

5 Galois connections

5.1 Definition and basic properties

record ('a, 'b, 'c, 'd) galcon =
  orderA :: "('a, 'c) gorder_scheme" (‹$\mathcal{X}_\imath$›)
  orderB :: "('b, 'd) gorder_scheme" (‹$\mathcal{Y}_\imath$›)
  lower :: "'a ⇒ 'b" (‹$\pi^*_\imath$›)
  upper :: "'b ⇒ 'a" (‹$\pi_{*\imath}$›)

type_synonym ('a, 'b) galois = "('a, 'b, unit, unit) galcon"

abbreviation "inv_galcon G ≡ (| orderA = inv_gorder $\mathcal{Y}_G$, orderB = inv_gorder $\mathcal{X}_G$, lower = upper G, upper = lower G |)"

definition comp_galcon :: "('b, 'c) galois ⇒ ('a, 'b) galois ⇒ ('a, 'c) galois" (**infixr** ‹$\circ_g$› 85)
  where "G $\circ_g$ F = (| orderA = orderA F, orderB = orderB G, lower = lower G ∘ lower F, upper = upper F ∘ upper G |)"

definition id_galcon :: "'a gorder ⇒ ('a, 'a) galois" (‹I_g›) **where**
"I_g(A) = (| orderA = A, orderB = A, lower = id, upper = id |)"

5.2 Well-typed connections

locale connection =
  fixes G (**structure**)
  assumes is_order_A: "partial_order $\mathcal{X}$"
  and is_order_B: "partial_order $\mathcal{Y}$"
  and lower_closure: "π^* ∈ carrier $\mathcal{X}$ → carrier $\mathcal{Y}$"
  and upper_closure: "π_* ∈ carrier $\mathcal{Y}$ → carrier $\mathcal{X}$"
begin

  lemma lower_closed: "x ∈ carrier $\mathcal{X}$ ⟹ π^* x ∈ carrier $\mathcal{Y}$"
    ⟨*proof*⟩

  lemma upper_closed: "y ∈ carrier $\mathcal{Y}$ ⟹ π_* y ∈ carrier $\mathcal{X}$"
    ⟨*proof*⟩

end

5.3 Galois connections

locale galois_connection = connection +
  assumes galois_property: "⟦x ∈ carrier $\mathcal{X}$; y ∈ carrier $\mathcal{Y}$⟧ ⟹ π^* x $\sqsubseteq_{\mathcal{Y}}$ y ⟷ x $\sqsubseteq_{\mathcal{X}}$ π_* y"
begin

  lemma is_weak_order_A: "weak_partial_order $\mathcal{X}$"
  ⟨*proof*⟩

lemma is_weak_order_B: "weak_partial_order $\mathcal{Y}$"
⟨*proof*⟩

lemma right: "⟦x ∈ carrier $\mathcal{X}$; y ∈ carrier $\mathcal{Y}$; π^* x $\sqsubseteq_{\mathcal{Y}}$ y⟧ ⟹ x $\sqsubseteq_{\mathcal{X}}$ π_* y"
⟨*proof*⟩

lemma left: "⟦x ∈ carrier $\mathcal{X}$; y ∈ carrier $\mathcal{Y}$; x $\sqsubseteq_{\mathcal{X}}$ π_* y⟧ ⟹ π^* x $\sqsubseteq_{\mathcal{Y}}$ y"
⟨*proof*⟩

lemma deflation: "y ∈ carrier $\mathcal{Y}$ ⟹ π^* (π_* y) $\sqsubseteq_{\mathcal{Y}}$ y"
⟨*proof*⟩

lemma inflation: "x ∈ carrier $\mathcal{X}$ ⟹ x $\sqsubseteq_{\mathcal{X}}$ π_* (π^* x)"
⟨*proof*⟩

lemma lower_iso: "isotone $\mathcal{X}$ $\mathcal{Y}$ π^*"
⟨*proof*⟩

lemma upper_iso: "isotone $\mathcal{Y}$ $\mathcal{X}$ π_*"
⟨*proof*⟩

lemma lower_comp: "x ∈ carrier $\mathcal{X}$ ⟹ π^* (π_* (π^* x)) = π^* x"
⟨*proof*⟩

lemma lower_comp': "x ∈ carrier $\mathcal{X}$ ⟹ (π^* ∘ π_* ∘ π^*) x = π^* x"
⟨*proof*⟩

lemma upper_comp: "y ∈ carrier $\mathcal{Y}$ ⟹ π_* (π^* (π_* y)) = π_* y"
⟨*proof*⟩

lemma upper_comp': "y ∈ carrier $\mathcal{Y}$ ⟹ (π_* ∘ π^* ∘ π_*) y = π_* y"
⟨*proof*⟩

lemma adjoint_idem1: "idempotent $\mathcal{Y}$ (π^* ∘ π_*)"
⟨*proof*⟩

lemma adjoint_idem2: "idempotent $\mathcal{X}$ (π_* ∘ π^*)"
⟨*proof*⟩

lemma fg_iso: "isotone $\mathcal{Y}$ $\mathcal{Y}$ (π^* ∘ π_*)"
⟨*proof*⟩

lemma gf_iso: "isotone $\mathcal{X}$ $\mathcal{X}$ (π_* ∘ π^*)"
⟨*proof*⟩

lemma semi_inverse1: "x ∈ carrier $\mathcal{X}$ ⟹ π^* x = π^* (π_* (π^* x))"

⟨*proof*⟩

lemma semi_inverse2: "x ∈ carrier $\mathcal{Y}$ ⟹ π_* x = π_* (π^* (π_* x))"
⟨*proof*⟩

theorem lower_by_complete_lattice:
assumes "complete_lattice $\mathcal{Y}$" "x ∈ carrier $\mathcal{X}$"
shows "π^*(x) = $\bigsqcap_{\mathcal{Y}}$ { y ∈ carrier $\mathcal{Y}$. x $\sqsubseteq_{\mathcal{X}}$ π_*(y) }"
⟨*proof*⟩

theorem upper_by_complete_lattice:
assumes "complete_lattice $\mathcal{X}$" "y ∈ carrier $\mathcal{Y}$"
shows "π_*(y) = $\bigsqcup_{\mathcal{X}}$ { x ∈ carrier $\mathcal{X}$. π^*(x) $\sqsubseteq_{\mathcal{Y}}$ y }"
⟨*proof*⟩

end

lemma dual_galois [simp]: " galois_connection (| orderA = inv_gorder B, orderB = inv_gorder A, lower = f, upper = g |)
= galois_connection (| orderA = A, orderB = B, lower = g, upper = f |)"
⟨*proof*⟩

definition lower_adjoint :: "('a, 'c) gorder_scheme ⇒ ('b, 'd) gorder_scheme ⇒ ('a ⇒ 'b) ⇒ bool" **where**
"lower_adjoint A B f ≡ ∃g. galois_connection (| orderA = A, orderB = B, lower = f, upper = g |)"

definition upper_adjoint :: "('a, 'c) gorder_scheme ⇒ ('b, 'd) gorder_scheme ⇒ ('b ⇒ 'a) ⇒ bool" **where**
"upper_adjoint A B g ≡ ∃f. galois_connection (| orderA = A, orderB = B, lower = f, upper = g |)"

lemma lower_adjoint_dual [simp]: "lower_adjoint (inv_gorder A) (inv_gorder B) f = upper_adjoint B A f"
⟨*proof*⟩

lemma upper_adjoint_dual [simp]: "upper_adjoint (inv_gorder A) (inv_gorder B) f = lower_adjoint B A f"
⟨*proof*⟩

lemma lower_type: "lower_adjoint A B f ⟹ f ∈ carrier A → carrier B"
⟨*proof*⟩

lemma upper_type: "upper_adjoint A B g ⟹ g ∈ carrier B → carrier A"
⟨*proof*⟩

5.4 Composition of Galois connections

lemma id_galois: "partial_order A ⟹ galois_connection (I$_g$(A))"
⟨*proof*⟩

lemma comp_galcon_closed:
assumes "galois_connection G" "galois_connection F" "$\mathcal{Y}_F$ = $\mathcal{X}_G$"
shows "galois_connection (G ∘$_g$ F)"
⟨*proof*⟩

lemma comp_galcon_right_unit [simp]: "F ∘$_g$ I$_g$($\mathcal{X}_F$) = F"
⟨*proof*⟩

lemma comp_galcon_left_unit [simp]: "I$_g$($\mathcal{Y}_F$) ∘$_g$ F = F"
⟨*proof*⟩

lemma galois_connectionI:
assumes
"partial_order A" "partial_order B"
"L ∈ carrier A → carrier B" "R ∈ carrier B → carrier A"
"isotone A B L" "isotone B A R"
"⋀ x y. ⟦ x ∈ carrier A; y ∈ carrier B ⟧ ⟹ L x ⊑$_B$ y ⟷ x ⊑$_A$ R y"
shows "galois_connection (| orderA = A, orderB = B, lower = L, upper = R |)"
⟨*proof*⟩

lemma galois_connectionI':
assumes
"partial_order A" "partial_order B"
"L ∈ carrier A → carrier B" "R ∈ carrier B → carrier A"
"isotone A B L" "isotone B A R"
"⋀ X. X ∈ carrier(B) ⟹ L(R(X)) ⊑$_B$ X"
"⋀ X. X ∈ carrier(A) ⟹ X ⊑$_A$ R(L(X))"
shows "galois_connection (| orderA = A, orderB = B, lower = L, upper = R |)"
⟨*proof*⟩

5.5 Retracts

locale retract = galois_connection +
assumes retract_property: "x ∈ carrier $\mathcal{X}$ ⟹ π_* (π^* x) ⊑$_{\mathcal{X}}$ x"
begin
lemma retract_inverse: "x ∈ carrier $\mathcal{X}$ ⟹ π_* (π^* x) = x"
⟨*proof*⟩

lemma retract_injective: "inj_on π^* (carrier $\mathcal{X}$)"
⟨*proof*⟩
end

theorem comp_retract_closed:
  assumes "retract G" "retract F" "$\mathcal{Y}_F$ = $\mathcal{X}_G$"
  shows "retract (G $\circ_g$ F)"
⟨*proof*⟩

5.6 Coretracts

locale coretract = galois_connection +
  assumes coretract_property: "y ∈ carrier $\mathcal{Y}$ ⟹ y $\sqsubseteq_{\mathcal{Y}}$ π^* (π_* y)"
begin
  lemma coretract_inverse: "y ∈ carrier $\mathcal{Y}$ ⟹ π^* (π_* y) = y"
    ⟨*proof*⟩

  lemma retract_injective: "inj_on π_* (carrier $\mathcal{Y}$)"
    ⟨*proof*⟩
end

theorem comp_coretract_closed:
  assumes "coretract G" "coretract F" "$\mathcal{Y}_F$ = $\mathcal{X}_G$"
  shows "coretract (G $\circ_g$ F)"
⟨*proof*⟩

5.7 Galois Bijections

locale galois_bijection = connection +
  assumes lower_iso: "isotone $\mathcal{X}$ $\mathcal{Y}$ π^*"
  and upper_iso: "isotone $\mathcal{Y}$ $\mathcal{X}$ π_*"
  and lower_inv_eq: "x ∈ carrier $\mathcal{X}$ ⟹ π_* (π^* x) = x"
  and upper_inv_eq: "y ∈ carrier $\mathcal{Y}$ ⟹ π^* (π_* y) = y"
begin

  lemma lower_bij: "bij_betw π^* (carrier $\mathcal{X}$) (carrier $\mathcal{Y}$)"
    ⟨*proof*⟩

  lemma upper_bij: "bij_betw π_* (carrier $\mathcal{Y}$) (carrier $\mathcal{X}$)"
    ⟨*proof*⟩

sublocale gal_bij_conn: galois_connection
  ⟨*proof*⟩

sublocale gal_bij_ret: retract
  ⟨*proof*⟩

sublocale gal_bij_coret: coretract
  ⟨*proof*⟩

end

theorem comp_galois_bijection_closed:
  assumes "galois_bijection G" "galois_bijection F" "$\mathcal{Y}_F$ = $\mathcal{X}_G$"

shows "galois_bijection (G $\circ_g$ F)"
⟨*proof*⟩

end

theory Group
imports Complete_Lattice "HOL-Library.FuncSet"
begin

6 Monoids and Groups

6.1 Definitions

Definitions follow [3].

record 'a monoid = "'a partial_object" +
  mult :: "['a, 'a] ⇒ 'a" (**infixl** ‹⊗›) 70)
  one :: 'a (‹**1**›)

definition m_inv :: "('a, 'b) monoid_scheme => 'a => 'a"
  where "m_inv G x = (THE y. y ∈ carrier G ∧ x ⊗$_G$ y = **1**$_G$ ∧ y ⊗$_G$ x = **1**$_G$)"

open_bundle m_inv_syntax
begin
notation m_inv (‹(‹open_block notation=‹prefix inv››inv› _)› [81] 80)
end

definition
  Units :: "_ => 'a set"
  — The set of invertible elements
  where "Units G = {y. y ∈ carrier G ∧ (∃x ∈ carrier G. x ⊗$_G$ y = **1**$_G$ ∧ y ⊗$_G$ x = **1**$_G$)}"

locale monoid =
  fixes G (**structure**)
  assumes m_closed [intro, simp]:
        "⟦x ∈ carrier G; y ∈ carrier G⟧ ⟹ x ⊗ y ∈ carrier G"
      and m_assoc:
        "⟦x ∈ carrier G; y ∈ carrier G; z ∈ carrier G⟧
         ⟹ (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
      and one_closed [intro, simp]: "**1** ∈ carrier G"
      and l_one [simp]: "x ∈ carrier G ⟹ **1** ⊗ x = x"
      and r_one [simp]: "x ∈ carrier G ⟹ x ⊗ **1** = x"

lemma monoidI:
  fixes G (**structure**)
  assumes m_closed:

```
      "!!x y. [| x ∈ carrier G; y ∈ carrier G |] ==> x ⊗ y ∈ carrier
G"
    and one_closed: "𝟭 ∈ carrier G"
    and m_assoc:
      "!!x y z. [| x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |] ==>
      (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
    and l_one: "!!x. x ∈ carrier G ==> 𝟭 ⊗ x = x"
    and r_one: "!!x. x ∈ carrier G ==> x ⊗ 𝟭 = x"
  shows "monoid G"
  ⟨proof⟩

lemma (in monoid) Units_closed [dest]:
  "x ∈ Units G ==> x ∈ carrier G"
  ⟨proof⟩

lemma (in monoid) one_unique:
  assumes "u ∈ carrier G"
    and "⋀x. x ∈ carrier G ⟹ u ⊗ x = x"
  shows "u = 𝟭"
  ⟨proof⟩

lemma (in monoid) inv_unique:
  assumes eq: "y ⊗ x = 𝟭"  "x ⊗ y' = 𝟭"
    and G: "x ∈ carrier G"  "y ∈ carrier G"  "y' ∈ carrier G"
  shows "y = y'"
⟨proof⟩

lemma (in monoid) Units_m_closed [simp, intro]:
  assumes x: "x ∈ Units G" and y: "y ∈ Units G"
  shows "x ⊗ y ∈ Units G"
⟨proof⟩

lemma (in monoid) Units_one_closed [intro, simp]:
  "𝟭 ∈ Units G"
  ⟨proof⟩

lemma (in monoid) Units_inv_closed [intro, simp]:
  "x ∈ Units G ==> inv x ∈ carrier G"
  ⟨proof⟩

lemma (in monoid) Units_l_inv_ex:
  "x ∈ Units G ==> ∃y ∈ carrier G. y ⊗ x = 𝟭"
  ⟨proof⟩

lemma (in monoid) Units_r_inv_ex:
  "x ∈ Units G ==> ∃y ∈ carrier G. x ⊗ y = 𝟭"
  ⟨proof⟩

lemma (in monoid) Units_l_inv [simp]:
```

```
  "x ∈ Units G ==> inv x ⊗ x = 1"
```
⟨*proof*⟩

lemma (in monoid**)** Units_r_inv [simp]:
```
  "x ∈ Units G ==> x ⊗ inv x = 1"
```
⟨*proof*⟩

lemma (in monoid**)** inv_one [simp]:
```
  "inv 1 = 1"
```
⟨*proof*⟩

lemma (in monoid**)** Units_inv_Units [intro, simp]:
```
  "x ∈ Units G ==> inv x ∈ Units G"
```
⟨*proof*⟩

lemma (in monoid**)** Units_l_cancel [simp]:
```
  "[| x ∈ Units G; y ∈ carrier G; z ∈ carrier G |] ==>
   (x ⊗ y = x ⊗ z) = (y = z)"
```
⟨*proof*⟩

lemma (in monoid**)** Units_inv_inv [simp]:
```
  "x ∈ Units G ==> inv (inv x) = x"
```
⟨*proof*⟩

lemma (in monoid**)** inv_inj_on_Units:
```
  "inj_on (m_inv G) (Units G)"
```
⟨*proof*⟩

lemma (in monoid**)** Units_inv_comm:
```
  assumes inv: "x ⊗ y = 1"
    and G: "x ∈ Units G"  "y ∈ Units G"
  shows "y ⊗ x = 1"
```
⟨*proof*⟩

lemma (in monoid**)** carrier_not_empty: "carrier G ≠ {}"
⟨*proof*⟩

6.2 Groups

A group is a monoid all of whose elements are invertible.

locale group = monoid +
```
  assumes Units: "carrier G <= Units G"
```

lemma (in group**)** is_group [iff]: "group G" ⟨*proof*⟩

lemma (in group**)** is_monoid [iff]: "monoid G"
⟨*proof*⟩

theorem groupI:

fixes G **(structure)**
assumes m_closed [simp]:
"!!x y. [| x ∈ carrier G; y ∈ carrier G |] ==> x ⊗ y ∈ carrier G"
and one_closed [simp]: "**1** ∈ carrier G"
and m_assoc:
"!!x y z. [| x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |] ==> (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
and l_one [simp]: "!!x. x ∈ carrier G ==> **1** ⊗ x = x"
and l_inv_ex: "!!x. x ∈ carrier G ==> ∃y ∈ carrier G. y ⊗ x = **1**"
shows "group G"
⟨*proof*⟩

lemma (in monoid**)** group_l_invI:
assumes l_inv_ex:
"!!x. x ∈ carrier G ==> ∃y ∈ carrier G. y ⊗ x = **1**"
shows "group G"
⟨*proof*⟩

lemma (in group**)** Units_eq [simp]:
"Units G = carrier G"
⟨*proof*⟩

lemma (in group**)** inv_closed [intro, simp]:
"x ∈ carrier G ==> inv x ∈ carrier G"
⟨*proof*⟩

lemma (in group**)** l_inv_ex [simp]:
"x ∈ carrier G ==> ∃y ∈ carrier G. y ⊗ x = **1**"
⟨*proof*⟩

lemma (in group**)** r_inv_ex [simp]:
"x ∈ carrier G ==> ∃y ∈ carrier G. x ⊗ y = **1**"
⟨*proof*⟩

lemma (in group**)** l_inv [simp]:
"x ∈ carrier G ==> inv x ⊗ x = **1**"
⟨*proof*⟩

6.3 Cancellation Laws and Basic Properties

lemma (in group**)** inv_eq_1_iff [simp]:
assumes "x ∈ carrier G" **shows** "inv$_G$ x = **1**$_G$ ⟷ x = **1**$_G$"
⟨*proof*⟩

lemma (in group**)** r_inv [simp]:
"x ∈ carrier G ==> x ⊗ inv x = **1**"
⟨*proof*⟩

lemma (in group**)** right_cancel [simp]:
"[| x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |] ==>
(y ⊗ x = z ⊗ x) = (y = z)"
⟨*proof*⟩

lemma (in group**)** inv_inv [simp]:
"x ∈ carrier G ==> inv (inv x) = x"
⟨*proof*⟩

lemma (in group**)** inv_inj:
"inj_on (m_inv G) (carrier G)"
⟨*proof*⟩

lemma (in group**)** inv_mult_group:
"[| x ∈ carrier G; y ∈ carrier G |] ==> inv (x ⊗ y) = inv y ⊗ inv x"
⟨*proof*⟩

lemma (in group**)** inv_comm:
"[| x ⊗ y = **1**; x ∈ carrier G; y ∈ carrier G |] ==> y ⊗ x = **1**"
⟨*proof*⟩

lemma (in group**)** inv_equality:
"[|y ⊗ x = **1**; x ∈ carrier G; y ∈ carrier G|] ==> inv x = y"
⟨*proof*⟩

lemma (in group**)** inv_solve_left:
"⟦ a ∈ carrier G; b ∈ carrier G; c ∈ carrier G ⟧ ⟹ a = inv b ⊗ c
⟷ c = b ⊗ a"
⟨*proof*⟩

lemma (in group**)** inv_solve_left':
"⟦ a ∈ carrier G; b ∈ carrier G; c ∈ carrier G ⟧ ⟹ inv b ⊗ c = a
⟷ c = b ⊗ a"
⟨*proof*⟩

lemma (in group**)** inv_solve_right:
"⟦ a ∈ carrier G; b ∈ carrier G; c ∈ carrier G ⟧ ⟹ a = b ⊗ inv c
⟷ b = a ⊗ c"
⟨*proof*⟩

lemma (in group**)** inv_solve_right':
"⟦a ∈ carrier G; b ∈ carrier G; c ∈ carrier G⟧ ⟹ b ⊗ inv c = a ⟷
b = a ⊗ c"
⟨*proof*⟩

6.4 Power

consts
pow :: "[('a, 'm) monoid_scheme, 'a, 'b::semiring_1] => 'a" (**infixr**

‹ [^] ı › 75)

overloading nat_pow == "pow :: [_, 'a, nat] => 'a"
begin
  definition "nat_pow G a n = rec_nat $\mathbf{1}_G$ (%u b. b $\otimes_G$ a) n"
end

lemma (**in** monoid) nat_pow_closed [intro, simp]:
  "x ∈ carrier G ==> x [^] (n::nat) ∈ carrier G"
  ⟨*proof*⟩

lemma (**in** monoid) nat_pow_0 [simp]:
  "x [^] (0::nat) = **1**"
  ⟨*proof*⟩

lemma (**in** monoid) nat_pow_Suc [simp]:
  "x [^] (Suc n) = x [^] n ⊗ x"
  ⟨*proof*⟩

lemma (**in** monoid) nat_pow_one [simp]:
  "**1** [^] (n::nat) = **1**"
  ⟨*proof*⟩

lemma (**in** monoid) nat_pow_mult:
  "x ∈ carrier G ==> x [^] (n::nat) ⊗ x [^] m = x [^] (n + m)"
  ⟨*proof*⟩

lemma (**in** monoid) nat_pow_comm:
  "x ∈ carrier G ⟹ (x [^] (n::nat)) ⊗ (x [^] (m :: nat)) = (x [^] m)
⊗ (x [^] n)"
  ⟨*proof*⟩

lemma (**in** monoid) nat_pow_Suc2:
  "x ∈ carrier G ⟹ x [^] (Suc n) = x ⊗ (x [^] n)"
  ⟨*proof*⟩

lemma (**in** monoid) nat_pow_pow:
  "x ∈ carrier G ==> (x [^] n) [^] m = x [^] (n * m::nat)"
  ⟨*proof*⟩

lemma (**in** monoid) nat_pow_consistent:
  "x [^] (n :: nat) = x [^]$_{(G ⦇ carrier := H ⦈)}$ n"
  ⟨*proof*⟩

lemma nat_pow_0 [simp]: "x [^]$_G$ (0::nat) = $\mathbf{1}_G$"
  ⟨*proof*⟩

lemma nat_pow_Suc [simp]: "x [^]$_G$ (Suc n) = (x [^]$_G$ n)$\otimes_G$ x"
  ⟨*proof*⟩

lemma (in group**)** nat_pow_inv:
assumes "x ∈ carrier G" **shows** "(inv x) [^] (i :: nat) = inv (x [^] i)"
⟨*proof*⟩

overloading int_pow == "pow :: [_, 'a, int] => 'a"
begin
definition "int_pow G a z =
(let p = rec_nat $\mathbf{1}_G$ (%u b. b $\otimes_G$ a)
in if z < 0 then inv_G (p (nat (-z))) else p (nat z))"
end

lemma int_pow_int: "x [^]$_G$ (int n) = x [^]$_G$ n"
⟨*proof*⟩

lemma pow_nat:
assumes "i≥0"
shows "x [^]$_G$ nat i = x [^]$_G$ i"
⟨*proof*⟩

lemma int_pow_0 [simp]: "x [^]$_G$ (0::int) = $\mathbf{1}_G$"
⟨*proof*⟩

lemma int_pow_def2: "a [^]$_G$ z =
(if z < 0 then inv_G (a [^]$_G$ (nat (-z))) else a [^]$_G$ (nat z))"
⟨*proof*⟩

lemma (in group**)** int_pow_one [simp]:
"**1** [^] (z::int) = **1**"
⟨*proof*⟩

lemma (in group**)** int_pow_closed [intro, simp]:
"x ∈ carrier G ==> x [^] (i::int) ∈ carrier G"
⟨*proof*⟩

lemma (in group**)** int_pow_1 [simp]:
"x ∈ carrier G ⟹ x [^] (1::int) = x"
⟨*proof*⟩

lemma (in group**)** int_pow_neg:
"x ∈ carrier G ⟹ x [^] (-i::int) = inv (x [^] i)"
⟨*proof*⟩

lemma (in group**)** int_pow_neg_int: "x ∈ carrier G ⟹ x [^] -(int n) = inv (x [^] n)"
⟨*proof*⟩

lemma (in group**)** int_pow_mult:

assumes "x ∈ carrier G" **shows** "x [^] (i + j::int) = x [^] i ⊗ x [^] j"
⟨*proof*⟩

lemma (in group**)** int_pow_inv:
assumes "x ∈ carrier G ⟹ (inv x) [^] (i :: int) = inv (x [^] i)"
⟨*proof*⟩

lemma (in group**)** int_pow_pow:
assumes "x ∈ carrier G"
shows "(x [^] (n :: int)) [^] (m :: int) = x [^] (n * m :: int)"
⟨*proof*⟩

lemma (in group**)** int_pow_diff:
"x ∈ carrier G ⟹ x [^] (n - m :: int) = x [^] n ⊗ inv (x [^] m)"
⟨*proof*⟩

lemma (in group**)** inj_on_multc: "c ∈ carrier G ⟹ inj_on (λx. x ⊗ c) (carrier G)"
⟨*proof*⟩

lemma (in group**)** inj_on_cmult: "c ∈ carrier G ⟹ inj_on (λx. c ⊗ x) (carrier G)"
⟨*proof*⟩

lemma (in monoid**)** group_commutes_pow:
fixes n::nat
shows "⟦x ⊗ y = y ⊗ x; x ∈ carrier G; y ∈ carrier G⟧ ⟹ x [^] n ⊗ y = y ⊗ x [^] n"
⟨*proof*⟩

lemma (in monoid**)** pow_mult_distrib:
assumes eq: "x ⊗ y = y ⊗ x" **and** xy: "x ∈ carrier G" "y ∈ carrier G"
shows "(x ⊗ y) [^] (n::nat) = x [^] n ⊗ y [^] n"
⟨*proof*⟩

lemma (in group**)** int_pow_mult_distrib:
assumes eq: "x ⊗ y = y ⊗ x" **and** xy: "x ∈ carrier G" "y ∈ carrier G"
shows "(x ⊗ y) [^] (i::int) = x [^] i ⊗ y [^] i"
⟨*proof*⟩

lemma (in group**)** pow_eq_div2:
fixes m n :: nat
assumes x_car: "x ∈ carrier G"
assumes pow_eq: "x [^] m = x [^] n"
shows "x [^] (m - n) = **1**"

⟨*proof*⟩

6.5 Submonoids

locale submonoid =
  fixes H **and** G **(structure)**
  assumes subset: "H ⊆ carrier G"
    and m_closed [intro, simp]: "⟦x ∈ H; y ∈ H⟧ ⟹ x ⊗ y ∈ H"
    and one_closed [simp]: "**1** ∈ H"

lemma (in submonoid) is_submonoid:
  "submonoid H G" ⟨*proof*⟩

lemma (in submonoid) mem_carrier [simp]:
  "x ∈ H ⟹ x ∈ carrier G"
  ⟨*proof*⟩

lemma (in submonoid) submonoid_is_monoid [intro]:
  assumes "monoid G"
  shows "monoid (G⦇carrier := H⦈)"
⟨*proof*⟩

lemma submonoid_nonempty:
  "~ submonoid {} G"
  ⟨*proof*⟩

lemma (in submonoid) finite_monoid_imp_card_positive:
  "finite (carrier G) ==> 0 < card H"
⟨*proof*⟩

lemma (in monoid) monoid_incl_imp_submonoid :
  assumes "H ⊆ carrier G"
and "monoid (G⦇carrier := H⦈)"
shows "submonoid H G"
⟨*proof*⟩

lemma (in monoid) inv_unique':
  assumes "x ∈ carrier G" "y ∈ carrier G"
  shows "⟦ x ⊗ y = **1**; y ⊗ x = **1** ⟧ ⟹ y = inv x"
⟨*proof*⟩

lemma (in monoid) m_inv_monoid_consistent:
  assumes "x ∈ Units (G ⦇ carrier := H ⦈)" **and** "submonoid H G"
  shows "inv$_{(G \langle\!\langle \text{carrier} := H \rangle\!\rangle)}$ x = inv x"
⟨*proof*⟩

6.6 Subgroups

locale subgroup =
  fixes H **and** G (**structure**)
  assumes subset: "H ⊆ carrier G"
    and m_closed [intro, simp]: "⟦x ∈ H; y ∈ H⟧ ⟹ x ⊗ y ∈ H"
    and one_closed [simp]: "**1** ∈ H"
    and m_inv_closed [intro,simp]: "x ∈ H ⟹ inv x ∈ H"

lemma (**in** subgroup) is_subgroup:
  "subgroup H G" ⟨*proof*⟩

declare (**in** subgroup) group.intro [intro]

lemma (**in** subgroup) mem_carrier [simp]:
  "x ∈ H ⟹ x ∈ carrier G"
  ⟨*proof*⟩

lemma (**in** subgroup) subgroup_is_group [intro]:
  assumes "group G"
  shows "group (G(|carrier := H|))"
⟨*proof*⟩

lemma (**in** group) triv_subgroup: "subgroup {**1**} G"
  ⟨*proof*⟩

lemma subgroup_is_submonoid:
  assumes "subgroup H G" **shows** "submonoid H G"
  ⟨*proof*⟩

lemma (**in** group) subgroup_Units:
  assumes "subgroup H G" **shows** "H ⊆ Units (G (| carrier := H |))"
  ⟨*proof*⟩

lemma (**in** group) m_inv_consistent [simp]:
  assumes "subgroup H G" "x ∈ H"
  shows "inv$_{(G (| carrier := H |))}$ x = inv x"
  ⟨*proof*⟩

lemma (**in** group) int_pow_consistent:
  assumes "subgroup H G" "x ∈ H"
  shows "x [^] (n :: int) = x [^]$_{(G (| carrier := H |))}$ n"
⟨*proof*⟩

Since H is nonempty, it contains some element x. Since it is closed under inverse, it contains inv x. Since it is closed under product, it contains x ⊗ inv x = **1**.

lemma (**in** group) one_in_subset:
  "⟦H ⊆ carrier G; H ≠ {}; ∀a ∈ H. inv a ∈ H; ∀a∈H. ∀b∈H. a ⊗ b ∈

```
H⟧
   ⟹ 1 ∈ H"
```
⟨*proof*⟩

A characterization of subgroups: closed, non-empty subset.

```
lemma (in group) subgroupI:
  assumes subset: "H ⊆ carrier G" and non_empty: "H ≠ {}"
    and inv: "!!a. a ∈ H ⟹ inv a ∈ H"
    and mult: "!!a b. ⟦a ∈ H; b ∈ H⟧ ⟹ a ⊗ b ∈ H"
  shows "subgroup H G"
```
⟨*proof*⟩

```
lemma (in group) subgroupE:
  assumes "subgroup H G"
  shows "H ⊆ carrier G"
    and "H ≠ {}"
    and "⋀a. a ∈ H ⟹ inv a ∈ H"
    and "⋀a b. ⟦ a ∈ H; b ∈ H ⟧ ⟹ a ⊗ b ∈ H"
```
⟨*proof*⟩

```
declare monoid.one_closed [iff] group.inv_closed [simp]
  monoid.l_one [simp] monoid.r_one [simp] group.inv_inv [simp]
```

```
lemma subgroup_nonempty:
  "¬ subgroup {} G"
```
⟨*proof*⟩

```
lemma (in subgroup) finite_imp_card_positive: "finite (carrier G) ⟹
0 < card H"
```
⟨*proof*⟩

```
lemma (in subgroup) subgroup_is_submonoid :
  "submonoid H G"
```
⟨*proof*⟩

```
lemma (in group) submonoid_subgroupI :
  assumes "submonoid H G"
    and "⋀a. a ∈ H ⟹ inv a ∈ H"
  shows "subgroup H G"
```
⟨*proof*⟩

```
lemma (in group) group_incl_imp_subgroup:
  assumes "H ⊆ carrier G"
    and "group (G⦇carrier := H⦈)"
  shows "subgroup H G"
```
⟨*proof*⟩

6.7 Direct Products

definition
```
  DirProd :: "_ ⇒ _ ⇒ ('a × 'b) monoid" (infixr ‹××› 80) where
  "G ×× H =
    (|carrier = carrier G × carrier H,
     mult = (λ(g, h) (g', h'). (g ⊗G g', h ⊗H h')),
     one = (1G, 1H)|)"
```

lemma `DirProd_monoid:`
  assumes `"monoid G"` **and** `"monoid H"`
  shows `"monoid (G ×× H)"`
⟨*proof*⟩

Does not use the previous result because it's easier just to use auto.

lemma `DirProd_group:`
  assumes `"group G"` **and** `"group H"`
  shows `"group (G ×× H)"`
⟨*proof*⟩

lemma `carrier_DirProd [simp]: "carrier (G ×× H) = carrier G × carrier H"`
  ⟨*proof*⟩

lemma `one_DirProd [simp]: "1G ×× H = (1G, 1H)"`
  ⟨*proof*⟩

lemma `mult_DirProd [simp]: "(g, h) ⊗(G ×× H) (g', h') = (g ⊗G g', h ⊗H h')"`
  ⟨*proof*⟩

lemma `mult_DirProd': "x ⊗(G ×× H) y = (fst x ⊗G fst y, snd x ⊗H snd y)"`
  ⟨*proof*⟩

lemma `DirProd_assoc: "(G ×× H ×× I) = (G ×× (H ×× I))"`
  ⟨*proof*⟩

lemma `inv_DirProd [simp]:`
  assumes `"group G"` **and** `"group H"`
  assumes `g: "g ∈ carrier G"`
      and `h: "h ∈ carrier H"`
  shows `"m_inv (G ×× H) (g, h) = (invG g, invH h)"`
⟨*proof*⟩

lemma `DirProd_subgroups :`
  assumes `"group G"`
    and `"subgroup H G"`
    and `"group K"`
    and `"subgroup I K"`

shows "subgroup (H × I) (G ×× K)"
⟨*proof*⟩

6.8 Homomorphisms (mono and epi) and Isomorphisms

definition
hom :: "_ => _ => ('a => 'b) set" **where**
"hom G H =
{h. h ∈ carrier G → carrier H ∧
(∀x ∈ carrier G. ∀y ∈ carrier G. h (x ⊗$_G$ y) = h x ⊗$_H$ h y)}"

lemma homI:
"⟦⋀x. x ∈ carrier G ⟹ h x ∈ carrier H;
⋀x y. ⟦x ∈ carrier G; y ∈ carrier G⟧ ⟹ h (x ⊗$_G$ y) = h x ⊗$_H$ h y⟧
⟹ h ∈ hom G H"
⟨*proof*⟩

lemma hom_carrier: "h ∈ hom G H ⟹ h ‘ carrier G ⊆ carrier H"
⟨*proof*⟩

lemma hom_in_carrier: "⟦h ∈ hom G H; x ∈ carrier G⟧ ⟹ h x ∈ carrier H"
⟨*proof*⟩

lemma hom_compose:
"⟦ f ∈ hom G H; g ∈ hom H I ⟧ ⟹ g ∘ f ∈ hom G I"
⟨*proof*⟩

lemma (**in** group) hom_restrict:
assumes "h ∈ hom G H" **and** "⋀g. g ∈ carrier G ⟹ h g = t g" **shows** "t ∈ hom G H"
⟨*proof*⟩

lemma (**in** group) hom_compose:
"[|h ∈ hom G H; i ∈ hom H I|] ==> compose (carrier G) i h ∈ hom G I"
⟨*proof*⟩

lemma (**in** group) restrict_hom_iff [simp]:
"(λx. if x ∈ carrier G then f x else g x) ∈ hom G H ⟷ f ∈ hom G H"
⟨*proof*⟩

definition iso :: "_ => _ => ('a => 'b) set"
where "iso G H = {h. h ∈ hom G H ∧ bij_betw h (carrier G) (carrier H)}"

definition is_iso :: "_ ⇒ _ ⇒ bool" (**infixr** ‹≅› 60)
where "G ≅ H = (iso G H ≠ {})"

definition mon **where** "mon G H = {f ∈ hom G H. inj_on f (carrier G)}"

definition epi **where** "epi G H = {f ∈ hom G H. f ‘ (carrier G) = carrier H}"

lemma isoI:
"⟦h ∈ hom G H; bij_betw h (carrier G) (carrier H)⟧ ⟹ h ∈ iso G H"
⟨*proof*⟩

lemma is_isoI: "h ∈ iso G H ⟹ G ≅ H"
⟨*proof*⟩

lemma epi_iff_subset:
"f ∈ epi G G’ ⟷ f ∈ hom G G’ ∧ carrier G’ ⊆ f ‘ carrier G"
⟨*proof*⟩

lemma iso_iff_mon_epi: "f ∈ iso G H ⟷ f ∈ mon G H ∧ f ∈ epi G H"
⟨*proof*⟩

lemma iso_set_refl: "(λx. x) ∈ iso G G"
⟨*proof*⟩

lemma id_iso: "id ∈ iso G G"
⟨*proof*⟩

corollary iso_refl [simp]: "G ≅ G"
⟨*proof*⟩

lemma iso_iff:
"h ∈ iso G H ⟷ h ∈ hom G H ∧ h ‘ (carrier G) = carrier H ∧ inj_on h (carrier G)"
⟨*proof*⟩

lemma iso_imp_homomorphism:
"h ∈ iso G H ⟹ h ∈ hom G H"
⟨*proof*⟩

lemma trivial_hom:
"group H ⟹ (λx. one H) ∈ hom G H"
⟨*proof*⟩

lemma (**in** group) hom_eq:
assumes "f ∈ hom G H" "⋀x. x ∈ carrier G ⟹ f’ x = f x"
shows "f’ ∈ hom G H"
⟨*proof*⟩

lemma (**in** group) iso_eq:
assumes "f ∈ iso G H" "⋀x. x ∈ carrier G ⟹ f’ x = f x"
shows "f’ ∈ iso G H"
⟨*proof*⟩

lemma (in group**)** iso_set_sym:
assumes "h ∈ iso G H"
shows "inv_into (carrier G) h ∈ iso H G"
⟨*proof*⟩

corollary (in group**)** iso_sym: "G ≅ H ⟹ H ≅ G"
⟨*proof*⟩

lemma iso_set_trans:
"⟦h ∈ Group.iso G H; i ∈ Group.iso H I⟧ ⟹ i ∘ h ∈ Group.iso G I"
⟨*proof*⟩

corollary iso_trans [trans]: "⟦G ≅ H ; H ≅ I⟧ ⟹ G ≅ I"
⟨*proof*⟩

lemma iso_same_card: "G ≅ H ⟹ card (carrier G) = card (carrier H)"
⟨*proof*⟩

lemma iso_finite: "G ≅ H ⟹ finite(carrier G) ⟷ finite(carrier H)"
⟨*proof*⟩

lemma mon_compose:
"⟦f ∈ mon G H; g ∈ mon H K⟧ ⟹ (g ∘ f) ∈ mon G K"
⟨*proof*⟩

lemma mon_compose_rev:
"⟦f ∈ hom G H; g ∈ hom H K; (g ∘ f) ∈ mon G K⟧ ⟹ f ∈ mon G H"
⟨*proof*⟩

lemma epi_compose:
"⟦f ∈ epi G H; g ∈ epi H K⟧ ⟹ (g ∘ f) ∈ epi G K"
⟨*proof*⟩

lemma epi_compose_rev:
"⟦f ∈ hom G H; g ∈ hom H K; (g ∘ f) ∈ epi G K⟧ ⟹ g ∈ epi H K"
⟨*proof*⟩

lemma iso_compose_rev:
"⟦f ∈ hom G H; g ∈ hom H K; (g ∘ f) ∈ iso G K⟧ ⟹ f ∈ mon G H ∧ g
∈ epi H K"
⟨*proof*⟩

lemma epi_iso_compose_rev:
assumes "f ∈ epi G H" "g ∈ hom H K" "(g ∘ f) ∈ iso G K"
shows "f ∈ iso G H ∧ g ∈ iso H K"
⟨*proof*⟩

lemma mon_left_invertible:

"⟦f ∈ hom G H; ⋀x. x ∈ carrier G ⟹ g(f x) = x⟧ ⟹ f ∈ mon G H"
⟨*proof*⟩

lemma epi_right_invertible:
"⟦g ∈ hom H G; f ∈ carrier G → carrier H; ⋀x. x ∈ carrier G ⟹ g(f x) = x⟧ ⟹ g ∈ epi H G"
⟨*proof*⟩

lemma (**in** monoid) hom_imp_img_monoid:
assumes "h ∈ hom G H"
shows "monoid (H ⦇ carrier := h ‘ (carrier G), one := h $\mathbf{1}_G$ ⦈)" (**is** "monoid ?h_img")
⟨*proof*⟩

lemma (**in** group) hom_imp_img_group:
assumes "h ∈ hom G H"
shows "group (H ⦇ carrier := h ‘ (carrier G), one := h $\mathbf{1}_G$ ⦈)" (**is** "group ?h_img")
⟨*proof*⟩

lemma (**in** group) iso_imp_group:
assumes "G ≅ H" **and** "monoid H"
shows "group H"
⟨*proof*⟩

corollary (**in** group) iso_imp_img_group:
assumes "h ∈ iso G H"
shows "group (H ⦇ one := h **1** ⦈)"
⟨*proof*⟩

6.8.1 HOL Light's concept of an isomorphism pair

definition group_isomorphisms
where
"group_isomorphisms G H f g ≡
f ∈ hom G H ∧ g ∈ hom H G ∧
(∀x ∈ carrier G. g(f x) = x) ∧
(∀y ∈ carrier H. f(g y) = y)"

lemma group_isomorphisms_sym: "group_isomorphisms G H f g ⟹ group_isomorphisms H G g f"
⟨*proof*⟩

lemma group_isomorphisms_imp_iso: "group_isomorphisms G H f g ⟹ f ∈ iso G H"
⟨*proof*⟩

lemma (**in** group) iso_iff_group_isomorphisms:
"f ∈ iso G H ⟷ (∃g. group_isomorphisms G H f g)"

⟨*proof*⟩

6.8.2 Involving direct products

lemma DirProd_commute_iso_set:
shows "(λ(x,y). (y,x)) ∈ iso (G ×× H) (H ×× G)"
⟨*proof*⟩

corollary DirProd_commute_iso :
"(G ×× H) ≅ (H ×× G)"
⟨*proof*⟩

lemma DirProd_assoc_iso_set:
shows "(λ(x,y,z). (x,(y,z))) ∈ iso (G ×× H ×× I) (G ×× (H ×× I))"
⟨*proof*⟩

lemma (**in** group) DirProd_iso_set_trans:
assumes "g ∈ iso G G2"
and "h ∈ iso H I"
shows "(λ(x,y). (g x, h y)) ∈ iso (G ×× H) (G2 ×× I)"
⟨*proof*⟩

corollary (**in** group) DirProd_iso_trans :
assumes "G ≅ G2" **and** "H ≅ I"
shows "G ×× H ≅ G2 ×× I"
⟨*proof*⟩

lemma hom_pairwise: "f ∈ hom G (DirProd H K) ⟷ (fst ∘ f) ∈ hom G H
∧ (snd ∘ f) ∈ hom G K"
⟨*proof*⟩

lemma hom_paired:
"(λx. (f x,g x)) ∈ hom G (DirProd H K) ⟷ f ∈ hom G H ∧ g ∈ hom
G K"
⟨*proof*⟩

lemma hom_paired2:
assumes "group G" "group H"
shows "(λ(x,y). (f x,g y)) ∈ hom (DirProd G H) (DirProd G' H') ⟷
f ∈ hom G G' ∧ g ∈ hom H H'"
⟨*proof*⟩

lemma iso_paired2:
assumes "group G" "group H"
shows "(λ(x,y). (f x,g y)) ∈ iso (DirProd G H) (DirProd G' H') ⟷
f ∈ iso G G' ∧ g ∈ iso H H'"
⟨*proof*⟩

lemma hom_of_fst:

assumes "group H"
shows "(f ∘ fst) ∈ hom (DirProd G H) K ⟷ f ∈ hom G K"
⟨*proof*⟩

lemma hom_of_snd:
assumes "group G"
shows "(f ∘ snd) ∈ hom (DirProd G H) K ⟷ f ∈ hom H K"
⟨*proof*⟩

6.9 The locale for a homomorphism between two groups

Basis for homomorphism proofs: we assume two groups G and H, with a homomorphism h between them

locale group_hom = G?: group G + H?: group H **for** G **(structure) and** H **(structure)**
+
fixes h
assumes homh [simp]: "h ∈ hom G H"

declare group_hom.homh [simp]

lemma (in group_hom) hom_mult [simp]:
"[| x ∈ carrier G; y ∈ carrier G |] ==> h (x ⊗$_G$ y) = h x ⊗$_H$ h y"
⟨*proof*⟩

lemma (in group_hom) hom_closed [simp]:
"x ∈ carrier G ==> h x ∈ carrier H"
⟨*proof*⟩

lemma (in group_hom) one_closed: "h **1** ∈ carrier H"
⟨*proof*⟩

lemma (in group_hom) hom_one [simp]: "h **1** = **1**$_H$"
⟨*proof*⟩

lemma hom_one:
assumes "h ∈ hom G H" "group G" "group H"
shows "h (one G) = one H"
⟨*proof*⟩

lemma hom_mult:
"⟦h ∈ hom G H; x ∈ carrier G; y ∈ carrier G⟧ ⟹ h (x ⊗$_G$ y) = h x ⊗$_H$
h y"
⟨*proof*⟩

lemma (in group_hom) inv_closed [simp]:
"x ∈ carrier G ==> h (inv x) ∈ carrier H"
⟨*proof*⟩

lemma (in group_hom) hom_inv [simp]:

assumes "x ∈ carrier G" **shows** "h (inv x) = inv_H (h x)"
⟨*proof*⟩

lemma (in group**)** int_pow_is_hom:
"x ∈ carrier G ⟹ (([^]) x) ∈ hom ⦇ carrier = UNIV, mult = (+), one = 0::int ⦈ G "
⟨*proof*⟩

lemma (in group_hom**)** img_is_subgroup: "subgroup (h ‘ (carrier G)) H"

⟨*proof*⟩

lemma (in group_hom**)** subgroup_img_is_subgroup:
assumes "subgroup I G"
shows "subgroup (h ‘ I) H"
⟨*proof*⟩

lemma (in subgroup**)** iso_subgroup:
assumes "group G" "group F"
assumes "φ ∈ iso G F"
shows "subgroup (φ ‘ H) F"
⟨*proof*⟩

lemma (in group_hom**)** induced_group_hom:
assumes "subgroup I G"
shows "group_hom (G ⦇ carrier := I ⦈) (H ⦇ carrier := h ‘ I ⦈) h"
⟨*proof*⟩

An isomorphism restricts to an isomorphism of subgroups.

lemma iso_restrict:
assumes "φ ∈ iso G F"
assumes groups: "group G" "group F"
assumes HG: "subgroup H G"
shows "(restrict φ H) ∈ iso (G⦇carrier := H⦈) (F⦇carrier := φ ‘ H⦈)"
⟨*proof*⟩

lemma (in group**)** canonical_inj_is_hom:
assumes "subgroup H G"
shows "group_hom (G ⦇ carrier := H ⦈) G id"
⟨*proof*⟩

lemma (in group_hom**)** hom_nat_pow:
"x ∈ carrier G ⟹ h (x [^] (n :: nat)) = (h x) $[^]_H$ n"
⟨*proof*⟩

lemma (in group_hom**)** hom_int_pow:
"x ∈ carrier G ⟹ h (x [^] (n :: int)) = (h x) $[^]_H$ n"
⟨*proof*⟩

lemma hom_nat_pow:
"⟦h ∈ hom G H; x ∈ carrier G; group G; group H⟧ ⟹ h (x [^]$_G$ (n :: nat)) = (h x) [^]$_H$ n"
⟨*proof*⟩

lemma hom_int_pow:
"⟦h ∈ hom G H; x ∈ carrier G; group G; group H⟧ ⟹ h (x [^]$_G$ (n :: int)) = (h x) [^]$_H$ n"
⟨*proof*⟩

6.10 Commutative Structures

Naming convention: multiplicative structures that are commutative are called *commutative*, additive structures are called *Abelian*.

locale comm_monoid = monoid +
assumes m_comm: "⟦x ∈ carrier G; y ∈ carrier G⟧ ⟹ x ⊗ y = y ⊗ x"

lemma (in comm_monoid**)** m_lcomm:
"⟦x ∈ carrier G; y ∈ carrier G; z ∈ carrier G⟧ ⟹
x ⊗ (y ⊗ z) = y ⊗ (x ⊗ z)"
⟨*proof*⟩

lemmas (in comm_monoid**)** m_ac = m_assoc m_comm m_lcomm

lemma comm_monoidI:
fixes G **(structure)**
assumes m_closed:
"!!x y. [| x ∈ carrier G; y ∈ carrier G |] ==> x ⊗ y ∈ carrier G"
and one_closed: "**1** ∈ carrier G"
and m_assoc:
"!!x y z. [| x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |] ==>
(x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
and l_one: "!!x. x ∈ carrier G ==> **1** ⊗ x = x"
and m_comm:
"!!x y. [| x ∈ carrier G; y ∈ carrier G |] ==> x ⊗ y = y ⊗ x"
shows "comm_monoid G"
⟨*proof*⟩

lemma (in monoid**)** monoid_comm_monoidI:
assumes m_comm:
"!!x y. [| x ∈ carrier G; y ∈ carrier G |] ==> x ⊗ y = y ⊗ x"
shows "comm_monoid G"
⟨*proof*⟩

lemma (in comm_monoid**)** submonoid_is_comm_monoid :
assumes "submonoid H G"
shows "comm_monoid (G⦇carrier := H⦈)"
⟨*proof*⟩

locale comm_group = comm_monoid + group

lemma (**in** group) group_comm_groupI:
  assumes m_comm: "!!x y. [| x ∈ carrier G; y ∈ carrier G |] ==> x ⊗ y = y ⊗ x"
  shows "comm_group G"
  ⟨*proof*⟩

lemma comm_groupI:
  fixes G (**structure**)
  assumes m_closed:
      "!!x y. [| x ∈ carrier G; y ∈ carrier G |] ==> x ⊗ y ∈ carrier G"
    and one_closed: "**1** ∈ carrier G"
    and m_assoc:
      "!!x y z. [| x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |] ==>
      (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
    and m_comm:
      "!!x y. [| x ∈ carrier G; y ∈ carrier G |] ==> x ⊗ y = y ⊗ x"
    and l_one: "!!x. x ∈ carrier G ==> **1** ⊗ x = x"
    and l_inv_ex: "!!x. x ∈ carrier G ==> ∃y ∈ carrier G. y ⊗ x = **1**"
  shows "comm_group G"
  ⟨*proof*⟩

lemma comm_groupE:
  fixes G (**structure**)
  assumes "comm_group G"
  shows "⋀x y. ⟦ x ∈ carrier G; y ∈ carrier G ⟧ ⟹ x ⊗ y ∈ carrier G"
    and "**1** ∈ carrier G"
    and "⋀x y z. ⟦ x ∈ carrier G; y ∈ carrier G; z ∈ carrier G ⟧ ⟹ (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
    and "⋀x y. ⟦ x ∈ carrier G; y ∈ carrier G ⟧ ⟹ x ⊗ y = y ⊗ x"
    and "⋀x. x ∈ carrier G ⟹ **1** ⊗ x = x"
    and "⋀x. x ∈ carrier G ⟹ ∃y ∈ carrier G. y ⊗ x = **1**"
  ⟨*proof*⟩

lemma (**in** comm_group) inv_mult:
  "[| x ∈ carrier G; y ∈ carrier G |] ==> inv (x ⊗ y) = inv x ⊗ inv y"
  ⟨*proof*⟩

lemma (**in** comm_monoid) nat_pow_distrib:
  fixes n::nat
  assumes "x ∈ carrier G" "y ∈ carrier G"
  shows "(x ⊗ y) [^] n = x [^] n ⊗ y [^] n"
  ⟨*proof*⟩

lemma (**in** comm_group) int_pow_distrib:

assumes "x ∈ carrier G" "y ∈ carrier G"
shows "(x ⊗ y) [^] (i::int) = x [^] i ⊗ y [^] i"
⟨*proof*⟩

lemma (**in** comm_monoid) hom_imp_img_comm_monoid:
assumes "h ∈ hom G H"
shows "comm_monoid (H (| carrier := h ‘ (carrier G), one := h $\mathbf{1}_G$ |))"
(**is** "comm_monoid ?h_img")
⟨*proof*⟩

lemma (**in** comm_group) hom_group_mult:
assumes "f ∈ hom H G" "g ∈ hom H G"
shows "(λx. f x $\otimes_G$ g x) ∈ hom H G"
⟨*proof*⟩

lemma (**in** comm_group) hom_imp_img_comm_group:
assumes "h ∈ hom G H"
shows "comm_group (H (| carrier := h ‘ (carrier G), one := h $\mathbf{1}_G$ |))"
⟨*proof*⟩

lemma (**in** comm_group) iso_imp_img_comm_group:
assumes "h ∈ iso G H"
shows "comm_group (H (| one := h $\mathbf{1}_G$ |))"
⟨*proof*⟩

lemma (**in** comm_group) iso_imp_comm_group:
assumes "G ≅ H" "monoid H"
shows "comm_group H"
⟨*proof*⟩

lemma (**in** group) incl_subgroup:
assumes "subgroup J G"
and "subgroup I (G(|carrier:=J|))"
shows "subgroup I G" ⟨*proof*⟩

lemma (**in** group) subgroup_incl:
assumes "subgroup I G" **and** "subgroup J G" **and** "I ⊆ J"
shows "subgroup I (G (| carrier := J |))"
⟨*proof*⟩

6.11 The Lattice of Subgroups of a Group

theorem (**in** group) subgroups_partial_order:
"partial_order (|carrier = {H. subgroup H G}, eq = (=), le = (⊆)|)"
⟨*proof*⟩

lemma (**in** group) subgroup_self:

```
  "subgroup (carrier G) G"
```
⟨*proof*⟩

lemma (**in** group) subgroup_imp_group:
```
  "subgroup H G ==> group (G(|carrier := H|))"
```
⟨*proof*⟩

lemma (**in** group) subgroup_mult_equality:
```
  "[| subgroup H G; h1 ∈ H; h2 ∈ H |] ==>  h1 ⊗G (| carrier := H |) h2 = h1
⊗ h2"
```
⟨*proof*⟩

theorem (**in** group) subgroups_Inter:
```
  assumes subgr: "(⋀H. H ∈ A ⟹ subgroup H G)"
    and not_empty: "A ≠ {}"
  shows "subgroup (⋂A) G"
```
⟨*proof*⟩

lemma (**in** group) subgroups_Inter_pair :
```
  assumes "subgroup I G" "subgroup J G" shows "subgroup (I∩J) G"
```
⟨*proof*⟩

theorem (**in** group) subgroups_complete_lattice:
```
  "complete_lattice (|carrier = {H. subgroup H G}, eq = (=), le = (⊆)|)"
    (is "complete_lattice ?L")
```
⟨*proof*⟩

6.12 The units in any monoid give rise to a group

Thanks to Jeremy Avigad. The file Residues.thy provides some infrastructure to use facts about the unit group within the ring locale.

definition units_of :: "('a, 'b) monoid_scheme ⇒ 'a monoid"
```
  where "units_of G =
    (|carrier = Units G, Group.monoid.mult = Group.monoid.mult G, one  =
one G|)"
```

lemma (**in** monoid) units_group: "group (units_of G)"
⟨*proof*⟩

lemma (**in** comm_monoid) units_comm_group: "comm_group (units_of G)"
⟨*proof*⟩

lemma units_of_carrier: "carrier (units_of G) = Units G"
```
  ⟨proof⟩
```

lemma units_of_mult: "mult (units_of G) = mult G"
```
  ⟨proof⟩
```

lemma units_of_one: "one (units_of G) = one G"

⟨*proof*⟩

lemma (in monoid**)** units_of_inv:
assumes "x ∈ Units G"
shows "m_inv (units_of G) x = m_inv G x"
⟨*proof*⟩

lemma units_of_units [simp] : "Units (units_of G) = Units G"
⟨*proof*⟩

lemma (in group**)** surj_const_mult: "a ∈ carrier G ⟹ (λx. a ⊗ x) ‘ carrier G = carrier G"
⟨*proof*⟩

lemma (in group**)** l_cancel_one [simp]: "x ∈ carrier G ⟹ a ∈ carrier G ⟹ x ⊗ a = x ⟷ a = one G"
⟨*proof*⟩

lemma (in group**)** r_cancel_one [simp]: "x ∈ carrier G ⟹ a ∈ carrier G ⟹ a ⊗ x = x ⟷ a = one G"
⟨*proof*⟩

lemma (in group**)** l_cancel_one’ [simp]: "x ∈ carrier G ⟹ a ∈ carrier G ⟹ x = x ⊗ a ⟷ a = one G"
⟨*proof*⟩

lemma (in group**)** r_cancel_one’ [simp]: "x ∈ carrier G ⟹ a ∈ carrier G ⟹ x = a ⊗ x ⟷ a = one G"
⟨*proof*⟩

declare pow_nat [simp]

end

theory FiniteProduct
imports Group
begin

6.13 Product Operator for Commutative Monoids

6.13.1 Inductive Definition of a Relation for Products over Sets

Instantiation of locale LC of theory Finite_Set is not possible, because here we have explicit typing rules like x ∈ carrier G. We introduce an explicit argument for the domain D.

inductive_set
foldSetD :: "[’a set, ’b ⇒ ’a ⇒ ’a, ’a] ⇒ (’b set * ’a) set"
for D :: "’a set" **and** f :: "’b ⇒ ’a ⇒ ’a" **and** e :: ’a

where
  emptyI [intro]: "e ∈ D ⟹ ({}, e) ∈ foldSetD D f e"
| insertI [intro]: "⟦x ∉ A; f x y ∈ D; (A, y) ∈ foldSetD D f e⟧ ⟹
  (insert x A, f x y) ∈ foldSetD D f e"

inductive_cases empty_foldSetDE [elim!]: "({}, x) ∈ foldSetD D f e"

definition
  foldD :: "['a set, 'b ⇒ 'a ⇒ 'a, 'a, 'b set] ⇒ 'a"
  where "foldD D f e A = (THE x. (A, x) ∈ foldSetD D f e)"

lemma foldSetD_closed: "(A, z) ∈ foldSetD D f e ⟹ z ∈ D"
  ⟨*proof*⟩

lemma Diff1_foldSetD:
  "⟦(A - {x}, y) ∈ foldSetD D f e; x ∈ A; f x y ∈ D⟧ ⟹
   (A, f x y) ∈ foldSetD D f e"
  ⟨*proof*⟩

lemma foldSetD_imp_finite [simp]: "(A, x) ∈ foldSetD D f e ⟹ finite A"
  ⟨*proof*⟩

lemma finite_imp_foldSetD:
  "⟦finite A; e ∈ D; ⋀x y. ⟦x ∈ A; y ∈ D⟧ ⟹ f x y ∈ D⟧
    ⟹ ∃x. (A, x) ∈ foldSetD D f e"
⟨*proof*⟩

lemma foldSetD_backwards:
  assumes "A ≠ {}" "(A, z) ∈ foldSetD D f e"
  shows "∃x y. x ∈ A ∧ (A - { x }, y) ∈ foldSetD D f e ∧ z = f x y"
  ⟨*proof*⟩

6.13.2 Left-Commutative Operations

locale LCD =
  fixes B :: "'b set"
  and D :: "'a set"
  and f :: "'b ⇒ 'a ⇒ 'a" (**infixl** ‹·› 70)
  assumes left_commute:
    "⟦x ∈ B; y ∈ B; z ∈ D⟧ ⟹ x · (y · z) = y · (x · z)"
  and f_closed [simp, intro!]: "!!x y. ⟦x ∈ B; y ∈ D⟧ ⟹ f x y ∈ D"

lemma (**in** LCD) foldSetD_closed [dest]: "(A, z) ∈ foldSetD D f e ⟹ z ∈ D"
  ⟨*proof*⟩

lemma (**in** LCD) Diff1_foldSetD:
  "⟦(A - {x}, y) ∈ foldSetD D f e; x ∈ A; A ⊆ B⟧ ⟹

(A, f x y) ∈ foldSetD D f e"
⟨*proof*⟩

lemma (in LCD**)** finite_imp_foldSetD:
"⟦finite A; A ⊆ B; e ∈ D⟧ ⟹ ∃x. (A, x) ∈ foldSetD D f e"
⟨*proof*⟩

lemma (in LCD**)** foldSetD_determ_aux:
assumes "e ∈ D" **and** A: "card A < n" "A ⊆ B" "(A, x) ∈ foldSetD D f e" "(A, y) ∈ foldSetD D f e"
shows "y = x"
⟨*proof*⟩

lemma (in LCD**)** foldSetD_determ:
"⟦(A, x) ∈ foldSetD D f e; (A, y) ∈ foldSetD D f e; e ∈ D; A ⊆ B⟧
⟹ y = x"
⟨*proof*⟩

lemma (in LCD**)** foldD_equality:
"⟦(A, y) ∈ foldSetD D f e; e ∈ D; A ⊆ B⟧ ⟹ foldD D f e A = y"
⟨*proof*⟩

lemma foldD_empty [simp]:
"e ∈ D ⟹ foldD D f e {} = e"
⟨*proof*⟩

lemma (in LCD**)** foldD_insert_aux:
"⟦x ∉ A; x ∈ B; e ∈ D; A ⊆ B⟧
⟹ ((insert x A, v) ∈ foldSetD D f e) ⟷ (∃y. (A, y) ∈ foldSetD D f e ∧ v = f x y)"
⟨*proof*⟩

lemma (in LCD**)** foldD_insert:
assumes "finite A" "x ∉ A" "x ∈ B" "e ∈ D" "A ⊆ B"
shows "foldD D f e (insert x A) = f x (foldD D f e A)"
⟨*proof*⟩

lemma (in LCD**)** foldD_closed [simp]:
"⟦finite A; e ∈ D; A ⊆ B⟧ ⟹ foldD D f e A ∈ D"
⟨*proof*⟩

lemma (in LCD**)** foldD_commute:
"⟦finite A; x ∈ B; e ∈ D; A ⊆ B⟧ ⟹
f x (foldD D f e A) = foldD D f (f x e) A"
⟨*proof*⟩

lemma Int_mono2:
"⟦A ⊆ C; B ⊆ C⟧ ⟹ A Int B ⊆ C"

⟨*proof*⟩

lemma (in LCD**)** foldD_nest_Un_Int:
"⟦finite A; finite C; e ∈ D; A ⊆ B; C ⊆ B⟧ ⟹
foldD D f (foldD D f e C) A = foldD D f (foldD D f e (A Int C)) (A Un C)"
⟨*proof*⟩

lemma (in LCD**)** foldD_nest_Un_disjoint:
"⟦finite A; finite B; A Int B = {}; e ∈ D; A ⊆ B; C ⊆ B⟧
⟹ foldD D f e (A Un B) = foldD D f (foldD D f e B) A"
⟨*proof*⟩

declare foldSetD_imp_finite [simp del]
empty_foldSetDE [rule del]
foldSetD.intros [rule del]
declare (in LCD**)**
foldSetD_closed [rule del]

Commutative Monoids

We enter a more restrictive context, with `f :: 'a ⇒ 'a ⇒ 'a` instead of `'b ⇒ 'a ⇒ 'a`.

locale ACeD =
fixes D :: "'a set"
and f :: "'a ⇒ 'a ⇒ 'a" (**infixl** ‹·› 70)
and e :: 'a
assumes ident [simp]: "x ∈ D ⟹ x · e = x"
and commute: "⟦x ∈ D; y ∈ D⟧ ⟹ x · y = y · x"
and assoc: "⟦x ∈ D; y ∈ D; z ∈ D⟧ ⟹ (x · y) · z = x · (y · z)"
and e_closed [simp]: "e ∈ D"
and f_closed [simp]: "⟦x ∈ D; y ∈ D⟧ ⟹ x · y ∈ D"

lemma (in ACeD**)** left_commute:
"⟦x ∈ D; y ∈ D; z ∈ D⟧ ⟹ x · (y · z) = y · (x · z)"
⟨*proof*⟩

lemmas (in ACeD**)** AC = assoc commute left_commute

lemma (in ACeD**)** left_ident [simp]: "x ∈ D ⟹ e · x = x"
⟨*proof*⟩

lemma (in ACeD**)** foldD_Un_Int:
"⟦finite A; finite B; A ⊆ D; B ⊆ D⟧ ⟹
foldD D f e A · foldD D f e B =
foldD D f e (A Un B) · foldD D f e (A Int B)"
⟨*proof*⟩

lemma (in ACeD**)** foldD_Un_disjoint:

"⟦finite A; finite B; A Int B = {}; A ⊆ D; B ⊆ D⟧ ⟹
  foldD D f e (A Un B) = foldD D f e A · foldD D f e B"
⟨*proof*⟩

6.13.3 Products over Finite Sets

definition
finprod :: "[('b, 'm) monoid_scheme, 'a ⇒ 'b, 'a set] ⇒ 'b"
where "finprod G f A =
 (if finite A
  then foldD (carrier G) (mult G ∘ f) $\mathbf{1}_G$ A
  else $\mathbf{1}_G$)"

syntax
"_finprod" :: "index ⇒ idt ⇒ 'a set ⇒ 'b ⇒ 'b"
  (<(<indent=3 notation=<binder ⨂ > >⨂ __∈_. _)> [1000, 0, 51, 10] 10)
syntax_consts
"_finprod" ⇌ finprod
translations
"⨂$_G$i∈A. b" ⇌ "CONST finprod G (%i. b) A"
— Beware of argument permutation!

lemma (in comm_monoid) finprod_empty [simp]:
"finprod G f {} = **1**"
⟨*proof*⟩

lemma (in comm_monoid) finprod_infinite[simp]:
"¬ finite A ⟹ finprod G f A = **1**"
⟨*proof*⟩

declare funcsetI [intro]
funcset_mem [dest]

context comm_monoid **begin**

lemma finprod_insert [simp]:
assumes "finite F" "a ∉ F" "f ∈ F → carrier G" "f a ∈ carrier G"
shows "finprod G f (insert a F) = f a ⊗ finprod G f F"
⟨*proof*⟩

lemma finprod_one_eqI: "(⋀x. x ∈ A ⟹ f x = **1**) ⟹ finprod G f A = **1**"
⟨*proof*⟩

lemma finprod_one [simp]: "(⨂i∈A. **1**) = **1**"
⟨*proof*⟩

lemma finprod_closed [simp]:

fixes A
assumes f: "f ∈ A → carrier G"
shows "finprod G f A ∈ carrier G"
⟨*proof*⟩

lemma funcset_Int_left [simp, intro]:
"⟦f ∈ A → C; f ∈ B → C⟧ ⟹ f ∈ A Int B → C"
⟨*proof*⟩

lemma funcset_Un_left [iff]:
"(f ∈ A Un B → C) = (f ∈ A → C ∧ f ∈ B → C)"
⟨*proof*⟩

lemma finprod_Un_Int:
"⟦finite A; finite B; g ∈ A → carrier G; g ∈ B → carrier G⟧ ⟹
finprod G g (A Un B) ⊗ finprod G g (A Int B) =
finprod G g A ⊗ finprod G g B"
— The reversed orientation looks more natural, but LOOPS as a simprule!
⟨*proof*⟩

lemma finprod_Un_disjoint:
"⟦finite A; finite B; A Int B = {};
g ∈ A → carrier G; g ∈ B → carrier G⟧
⟹ finprod G g (A Un B) = finprod G g A ⊗ finprod G g B"
⟨*proof*⟩

lemma finprod_multf [simp]:
"⟦f ∈ A → carrier G; g ∈ A → carrier G⟧ ⟹
finprod G (λx. f x ⊗ g x) A = (finprod G f A ⊗ finprod G g A)"
⟨*proof*⟩

lemma finprod_cong':
"⟦A = B; g ∈ B → carrier G;
!!i. i ∈ B ⟹ f i = g i⟧ ⟹ finprod G f A = finprod G g B"
⟨*proof*⟩

lemma finprod_cong:
"⟦A = B; f ∈ B → carrier G = True;
⋀i. i ∈ B =simp=> f i = g i⟧ ⟹ finprod G f A = finprod G g B"

⟨*proof*⟩

Usually, if this rule causes a failed congruence proof error, the reason is that the premise g ∈ B → carrier G cannot be shown. Adding Pi_def to the simpset is often useful. For this reason, finprod_cong is not added to the simpset by default.

end

declare funcsetI [rule del]

```
  funcset_mem [rule del]

context comm_monoid begin

lemma finprod_0 [simp]:
  "f ∈ {0::nat} → carrier G ⟹ finprod G f {..0} = f 0"
  ⟨proof⟩

lemma finprod_0':
  "f ∈ {..n} → carrier G ⟹ (f 0) ⊗ finprod G f {Suc 0..n} = finprod
G f {..n}"
⟨proof⟩

lemma finprod_Suc [simp]:
  "f ∈ {..Suc n} → carrier G ⟹
   finprod G f {..Suc n} = (f (Suc n) ⊗ finprod G f {..n})"
⟨proof⟩

lemma finprod_Suc2:
  "f ∈ {..Suc n} → carrier G ⟹
   finprod G f {..Suc n} = (finprod G (%i. f (Suc i)) {..n} ⊗ f 0)"
⟨proof⟩

lemma finprod_Suc3:
  assumes "f ∈ {..n :: nat} → carrier G"
  shows "finprod G f {.. n} = (f n) ⊗ finprod G f {..< n}"
⟨proof⟩

lemma finprod_reindex:
  "f ∈ (h ‘ A) → carrier G ⟹
        inj_on h A ⟹ finprod G f (h ‘ A) = finprod G (λx. f (h x)) A"
⟨proof⟩

lemma finprod_const:
  assumes a [simp]: "a ∈ carrier G"
    shows "finprod G (λx. a) A = a [^] card A"
⟨proof⟩

lemma finprod_singleton:
  assumes i_in_A: "i ∈ A" and fin_A: "finite A" and f_Pi: "f ∈ A →
carrier G"
  shows "(⨂j∈A. if i = j then f j else 𝟭) = f i"
  ⟨proof⟩

lemma finprod_singleton_swap:
  assumes i_in_A: "i ∈ A" and fin_A: "finite A" and f_Pi: "f ∈ A →
carrier G"
  shows "(⨂j∈A. if j = i then f j else 𝟭) = f i"
  ⟨proof⟩
```

lemma finprod_mono_neutral_cong_left:
  assumes "finite B"
    and "A ⊆ B"
    and 1: "⋀i. i ∈ B - A ⟹ h i = **1**"
    and gh: "⋀x. x ∈ A ⟹ g x = h x"
    and h: "h ∈ B → carrier G"
  shows "finprod G g A = finprod G h B"
⟨*proof*⟩

lemma finprod_mono_neutral_cong_right:
  assumes "finite B"
    and "A ⊆ B" "⋀i. i ∈ B - A ⟹ g i = **1**" "⋀x. x ∈ A ⟹ g x = h x" "g ∈ B → carrier G"
  shows "finprod G g B = finprod G h A"
  ⟨*proof*⟩

lemma finprod_mono_neutral_cong:
  assumes [simp]: "finite B" "finite A"
    and *: "⋀i. i ∈ B - A ⟹ h i = **1**" "⋀i. i ∈ A - B ⟹ g i = **1**"
    and gh: "⋀x. x ∈ A ∩ B ⟹ g x = h x"
    and g: "g ∈ A → carrier G"
    and h: "h ∈ B → carrier G"
 shows "finprod G g A = finprod G h B"
⟨*proof*⟩

end

lemma (**in** comm_group) power_order_eq_one:
  assumes fin [simp]: "finite (carrier G)"
    and a [simp]: "a ∈ carrier G"
  shows "a [^] card(carrier G) = one G"
⟨*proof*⟩

lemma (**in** comm_monoid) finprod_UN_disjoint:
  assumes
    "finite I" "⋀i. i ∈ I ⟹ finite (A i)" "pairwise (λi j. disjnt (A i) (A j)) I"
    "⋀i x. i ∈ I ⟹ x ∈ A i ⟹ g x ∈ carrier G"
shows "finprod G g (⋃(A ‘ I)) = finprod G (λi. finprod G g (A i)) I"
  ⟨*proof*⟩

lemma (**in** comm_monoid) finprod_Union_disjoint:
  "⟦finite C; ⋀A. A ∈ C ⟹ finite A ∧ (∀x∈A. f x ∈ carrier G); pairwise disjnt C⟧ ⟹
    finprod G f (⋃C) = finprod G (finprod G f) C"
  ⟨*proof*⟩

end

theory Coset
imports Group
begin

7 Cosets and Quotient Groups

definition
  r_coset :: "[_, 'a set, 'a] ⇒ 'a set" (**infixl** <#>ı 60)
  where "H #>$_G$ a = (⋃h∈H. {h ⊗$_G$ a})"

definition
  l_coset :: "[_, 'a, 'a set] ⇒ 'a set" (**infixl** <<#ı 60)
  where "a <#$_G$ H = (⋃h∈H. {a ⊗$_G$ h})"

definition
  RCOSETS :: "[_, 'a set] ⇒ ('a set)set"
    (<(<open_block notation=<prefix rcosets>>rcosetsı _)> [81] 80)
  where "rcosets$_G$ H = (⋃a∈carrier G. {H #>$_G$ a})"

definition
  set_mult :: "[_, 'a set ,'a set] ⇒ 'a set" (**infixl** <<#>ı 60)
  where "H <#>$_G$ K = (⋃h∈H. ⋃k∈K. {h ⊗$_G$ k})"

definition
  SET_INV :: "[_,'a set] ⇒ 'a set"
    (<(<open_block notation=<prefix set_inv>>set'_invı _)> [81] 80)
  where "set_inv$_G$ H = (⋃h∈H. {inv$_G$ h})"

locale normal = subgroup + group +
  assumes coset_eq: "(∀x ∈ carrier G. H #> x = x <# H)"

abbreviation
  normal_rel :: "['a set, ('a, 'b) monoid_scheme] ⇒ bool" (**infixl** <◁>
60) **where**
  "H ◁ G ≡ normal H G"

lemma (**in** comm_group) subgroup_imp_normal: "subgroup A G ⟹ A ◁ G"
  ⟨*proof*⟩

lemma l_coset_eq_set_mult:
  fixes G (**structure**)
  shows "x <# H = {x} <#> H"
  ⟨*proof*⟩

lemma r_coset_eq_set_mult:
fixes G **(structure)**
shows "H #> x = H <#> {x}"
⟨*proof*⟩

lemma (in subgroup**)** rcosets_non_empty:
assumes "R ∈ rcosets H"
shows "R ≠ {}"
⟨*proof*⟩

lemma (in group**)** diff_neutralizes:
assumes "subgroup H G" "R ∈ rcosets H"
shows "⋀r1 r2. ⟦ r1 ∈ R; r2 ∈ R ⟧ ⟹ r1 ⊗ (inv r2) ∈ H"
⟨*proof*⟩

lemma mono_set_mult: "⟦ H ⊆ H'; K ⊆ K' ⟧ ⟹ H <#>$_G$ K ⊆ H' <#>$_G$ K'"

⟨*proof*⟩

7.1 Stable Operations for Subgroups

lemma set_mult_consistent [simp]:
"N <#>$_{(G \lparen\!\lparen \text{carrier} := H \rparen\!\rparen)}$ K = N <#>$_G$ K"
⟨*proof*⟩

lemma r_coset_consistent [simp]:
"I #>$_G$ $\lparen\!\lparen$ carrier := H $\rparen\!\rparen$ h = I #>$_G$ h"
⟨*proof*⟩

lemma l_coset_consistent [simp]:
"h <#$_G$ $\lparen\!\lparen$ carrier := H $\rparen\!\rparen$ I = h <#$_G$ I"
⟨*proof*⟩

7.2 Basic Properties of set multiplication

lemma (in group**)** setmult_subset_G:
assumes "H ⊆ carrier G" "K ⊆ carrier G"
shows "H <#> K ⊆ carrier G" ⟨*proof*⟩

lemma (in monoid**)** set_mult_closed:
assumes "H ⊆ carrier G" "K ⊆ carrier G"
shows "H <#> K ⊆ carrier G"
⟨*proof*⟩

lemma (in group**)** set_mult_assoc:
assumes "M ⊆ carrier G" "H ⊆ carrier G" "K ⊆ carrier G"
shows "(M <#> H) <#> K = M <#> (H <#> K)"
⟨*proof*⟩

7.3 Basic Properties of Cosets

lemma (in group) coset_mult_assoc:
assumes "M ⊆ carrier G" "g ∈ carrier G" "h ∈ carrier G"
shows "(M #> g) #> h = M #> (g ⊗ h)"
⟨*proof*⟩

lemma (in group) coset_assoc:
assumes "x ∈ carrier G" "y ∈ carrier G" "H ⊆ carrier G"
shows "x <# (H #> y) = (x <# H) #> y"
⟨*proof*⟩

lemma (in group) coset_mult_one [simp]: "M ⊆ carrier G ==> M #> 1 = M"
⟨*proof*⟩

lemma (in group) coset_mult_inv1:
assumes "M #> (x ⊗ (inv y)) = M"
and "x ∈ carrier G" "y ∈ carrier G" "M ⊆ carrier G"
shows "M #> x = M #> y" ⟨*proof*⟩

lemma (in group) coset_mult_inv2:
assumes "M #> x = M #> y"
and "x ∈ carrier G" "y ∈ carrier G" "M ⊆ carrier G"
shows "M #> (x ⊗ (inv y)) = M " ⟨*proof*⟩

lemma (in group) coset_join1:
assumes "H #> x = H"
and "x ∈ carrier G" "subgroup H G"
shows "x ∈ H"
⟨*proof*⟩

lemma (in group) solve_equation:
assumes "subgroup H G" "x ∈ H" "y ∈ H"
shows "∃h ∈ H. y = h ⊗ x"
⟨*proof*⟩

lemma (in group_hom) inj_on_one_iff:
"inj_on h (carrier G) ⟷ (∀x. x ∈ carrier G ⟶ h x = one H ⟶ x = one G)"
⟨*proof*⟩

lemma inj_on_one_iff':
"⟦h ∈ hom G H; group G; group H⟧ ⟹ inj_on h (carrier G) ⟷ (∀x. x ∈ carrier G ⟶ h x = one H ⟶ x = one G)"
⟨*proof*⟩

lemma mon_iff_hom_one:
"⟦group G; group H⟧ ⟹ f ∈ mon G H ⟷ f ∈ hom G H ∧ (∀x. x ∈ carrier G ∧ f x = 1_H ⟶ x = 1_G)"

⟨*proof*⟩

lemma (in group_hom**)** iso_iff: "h ∈ iso G H ⟷ carrier H ⊆ h ‘ carrier G ∧ (∀x∈carrier G. h x = $\mathbf{1}_H$ ⟶ x = **1**)"
⟨*proof*⟩

lemma (in group**)** repr_independence:
assumes "y ∈ H #> x" "x ∈ carrier G" "subgroup H G"
shows "H #> x = H #> y" ⟨*proof*⟩

lemma (in group**)** coset_join2:
assumes "x ∈ carrier G" "subgroup H G" "x ∈ H"
shows "H #> x = H" ⟨*proof*⟩

lemma (in group**)** coset_join3:
assumes "x ∈ carrier G" "subgroup H G" "x ∈ H"
shows "x <# H = H"
⟨*proof*⟩

lemma (in monoid**)** r_coset_subset_G:
"⟦ H ⊆ carrier G; x ∈ carrier G ⟧ ⟹ H #> x ⊆ carrier G"
⟨*proof*⟩

lemma (in group**)** rcosI:
"⟦ h ∈ H; H ⊆ carrier G; x ∈ carrier G ⟧ ⟹ h ⊗ x ∈ H #> x"
⟨*proof*⟩

lemma (in group**)** rcosetsI:
"⟦H ⊆ carrier G; x ∈ carrier G⟧ ⟹ H #> x ∈ rcosets H"
⟨*proof*⟩

lemma (in group**)** rcos_self:
"⟦ x ∈ carrier G; subgroup H G ⟧ ⟹ x ∈ H #> x"
⟨*proof*⟩

Opposite of "repr_independence"

lemma (in group**)** repr_independenceD:
assumes "subgroup H G" "y ∈ carrier G"
and "H #> x = H #> y"
shows "y ∈ H #> x"
⟨*proof*⟩

Elements of a right coset are in the carrier

lemma (in subgroup**)** elemrcos_carrier:
assumes "group G" "a ∈ carrier G"
and "a’ ∈ H #> a"
shows "a’ ∈ carrier G"
⟨*proof*⟩

lemma (in subgroup**)** rcos_const:
  assumes "group G" "h ∈ H"
  shows "H #> h = H"
  ⟨*proof*⟩

lemma (in subgroup**)** rcos_module_imp:
  assumes "group G" "x ∈ carrier G"
    and "x' ∈ H #> x"
  shows "(x' ⊗ inv x) ∈ H"
⟨*proof*⟩

lemma (in subgroup**)** rcos_module_rev:
  assumes "group G" "x ∈ carrier G" "x' ∈ carrier G"
    and "(x' ⊗ inv x) ∈ H"
  shows "x' ∈ H #> x"
⟨*proof*⟩

Module property of right cosets

lemma (in subgroup**)** rcos_module:
  assumes "group G" "x ∈ carrier G" "x' ∈ carrier G"
  shows "(x' ∈ H #> x) = (x' ⊗ inv x ∈ H)"
  ⟨*proof*⟩

Right cosets are subsets of the carrier.

lemma (in subgroup**)** rcosets_carrier:
  assumes "group G" "X ∈ rcosets H"
  shows "X ⊆ carrier G"
  ⟨*proof*⟩

Multiplication of general subsets

lemma (in comm_group**)** mult_subgroups:
  assumes HG: "subgroup H G" **and** KG: "subgroup K G"
  shows "subgroup (H <#> K) G"
⟨*proof*⟩

lemma (in subgroup**)** lcos_module_rev:
  assumes "group G" "x ∈ carrier G" "x' ∈ carrier G"
    and "(inv x ⊗ x') ∈ H"
  shows "x' ∈ x <# H"
⟨*proof*⟩

7.4 Normal subgroups

lemma normal_imp_subgroup: "H ◁ G ⟹ subgroup H G"
  ⟨*proof*⟩

lemma (in group**)** normalI:
  "subgroup H G ⟹ (∀x ∈ carrier G. H #> x = x <# H) ⟹ H ◁ G"
  ⟨*proof*⟩

lemma (in normal**)** inv_op_closed1:
assumes "x ∈ carrier G" **and** "h ∈ H"
shows "(inv x) ⊗ h ⊗ x ∈ H"
⟨*proof*⟩

lemma (in normal**)** inv_op_closed2:
assumes "x ∈ carrier G" **and** "h ∈ H"
shows "x ⊗ h ⊗ (inv x) ∈ H"
⟨*proof*⟩

lemma (in comm_group**)** normal_iff_subgroup:
"N ◁ G ⟷ subgroup N G"
⟨*proof*⟩

Alternative characterization of normal subgroups

lemma (in group**)** normal_inv_iff:
"(N ◁ G) =
(subgroup N G ∧ (∀x ∈ carrier G. ∀h ∈ N. x ⊗ h ⊗ (inv x) ∈ N))"
(is "_ = ?rhs"**)**
⟨*proof*⟩

corollary (in group**)** normal_invI:
assumes "subgroup N G" **and** "⋀x h. ⟦ x ∈ carrier G; h ∈ N ⟧ ⟹ x ⊗ h ⊗ inv x ∈ N"
shows "N ◁ G"
⟨*proof*⟩

corollary (in group**)** normal_invE:
assumes "N ◁ G"
shows "subgroup N G" **and** "⋀x h. ⟦ x ∈ carrier G; h ∈ N ⟧ ⟹ x ⊗ h ⊗ inv x ∈ N"
⟨*proof*⟩

lemma (in group**)** one_is_normal: "{**1**} ◁ G"
⟨*proof*⟩

The intersection of two normal subgroups is, again, a normal subgroup.

lemma (in group**)** normal_subgroup_intersect:
assumes "M ◁ G" **and** "N ◁ G" **shows** "M ∩ N ◁ G"
⟨*proof*⟩

Being a normal subgroup is preserved by surjective homomorphisms.

lemma (in normal**)** surj_hom_normal_subgroup:
assumes φ: "group_hom G F φ"
assumes φsurj: "φ ‘ (carrier G) = carrier F"
shows "(φ ‘ H) ◁ F"
⟨*proof*⟩

Being a normal subgroup is preserved by group isomorphisms.

lemma iso_normal_subgroup:
assumes φ: "$\varphi \in$ iso G F" "group G" "group F" "H $\lhd$ G"
shows "(φ ‘ H) $\lhd$ F"
⟨*proof*⟩

The set product of two normal subgroups is a normal subgroup.

lemma (in group) setmult_lcos_assoc:
"⟦H ⊆ carrier G; K ⊆ carrier G; x ∈ carrier G⟧
$\Longrightarrow$ (x <# H) <#> K = x <# (H <#> K)"
⟨*proof*⟩

7.5 More Properties of Left Cosets

lemma (in group) l_repr_independence:
assumes "y ∈ x <# H" "x ∈ carrier G" **and** HG: "subgroup H G"
shows "x <# H = y <# H"
⟨*proof*⟩

lemma (in group) lcos_m_assoc:
"⟦ M ⊆ carrier G; g ∈ carrier G; h ∈ carrier G ⟧ $\Longrightarrow$ g <# (h <# M) =
(g ⊗ h) <# M"
⟨*proof*⟩

lemma (in group) lcos_mult_one: "M ⊆ carrier G $\Longrightarrow$ **1** <# M = M"
⟨*proof*⟩

lemma (in group) l_coset_subset_G:
"⟦ H ⊆ carrier G; x ∈ carrier G ⟧ $\Longrightarrow$ x <# H ⊆ carrier G"
⟨*proof*⟩

lemma (in group) l_coset_carrier:
"⟦ y ∈ x <# H; x ∈ carrier G; subgroup H G ⟧ $\Longrightarrow$ y ∈ carrier G"
⟨*proof*⟩

lemma (in group) l_coset_swap:
assumes "y ∈ x <# H" "x ∈ carrier G" "subgroup H G"
shows "x ∈ y <# H"
⟨*proof*⟩

lemma (in group) subgroup_mult_id:
assumes "subgroup H G"
shows "H <#> H = H"
⟨*proof*⟩

7.5.1 Set of Inverses of an r_coset.

lemma (in normal) rcos_inv:
assumes x: "x ∈ carrier G"

shows "set_inv (H #> x) = H #> (inv x)"
⟨*proof*⟩

7.5.2 Theorems for <#> with #> or <#.

lemma (in group) setmult_rcos_assoc:
"⟦H ⊆ carrier G; K ⊆ carrier G; x ∈ carrier G⟧ ⟹
H <#> (K #> x) = (H <#> K) #> x"
⟨*proof*⟩

lemma (in group) rcos_assoc_lcos:
"⟦H ⊆ carrier G; K ⊆ carrier G; x ∈ carrier G⟧ ⟹
(H #> x) <#> K = H <#> (x <# K)"
⟨*proof*⟩

lemma (in normal) rcos_mult_step1:
"⟦x ∈ carrier G; y ∈ carrier G⟧ ⟹
(H #> x) <#> (H #> y) = (H <#> (x <# H)) #> y"
⟨*proof*⟩

lemma (in normal) rcos_mult_step2:
"⟦x ∈ carrier G; y ∈ carrier G⟧
⟹ (H <#> (x <# H)) #> y = (H <#> (H #> x)) #> y"
⟨*proof*⟩

lemma (in normal) rcos_mult_step3:
"⟦x ∈ carrier G; y ∈ carrier G⟧
⟹ (H <#> (H #> x)) #> y = H #> (x ⊗ y)"
⟨*proof*⟩

lemma (in normal) rcos_sum:
"⟦x ∈ carrier G; y ∈ carrier G⟧
⟹ (H #> x) <#> (H #> y) = H #> (x ⊗ y)"
⟨*proof*⟩

lemma (in normal) rcosets_mult_eq: "M ∈ rcosets H ⟹ H <#> M = M"
— generalizes subgroup_mult_id
⟨*proof*⟩

7.5.3 An Equivalence Relation

definition
r_congruent :: "[('a,'b)monoid_scheme, 'a set] ⇒ ('a*'a)set"
(‹(‹open_block notation=‹prefix rcong››rcong› _)›)
where "rcong$_G$ H = {(x,y). x ∈ carrier G ∧ y ∈ carrier G ∧ inv$_G$ x ⊗$_G$ y ∈ H}"

lemma (in subgroup) equiv_rcong:
assumes "group G"

shows "equiv (carrier G) (rcong H)"
⟨*proof*⟩

Equivalence classes of rcong correspond to left cosets. Was there a mistake in the definitions? I'd have expected them to correspond to right cosets.

lemma (in subgroup) l_coset_eq_rcong:
assumes "group G"
assumes a: "a ∈ carrier G"
shows "a <# H = (rcong H) `` {a}"
⟨*proof*⟩

7.5.4 Two Distinct Right Cosets are Disjoint

lemma (in group) rcos_equation:
assumes "subgroup H G"
assumes p: "ha ⊗ a = h ⊗ b" "a ∈ carrier G" "b ∈ carrier G" "h ∈ H" "ha ∈ H" "hb ∈ H"
shows "hb ⊗ a ∈ (⋃h∈H. {h ⊗ b})"
⟨*proof*⟩

lemma (in group) rcos_disjoint:
assumes "subgroup H G"
shows "pairwise disjnt (rcosets H)"
⟨*proof*⟩

7.6 Further lemmas for r_congruent

The relation is a congruence

lemma (in normal) congruent_rcong:
shows "congruent2 (rcong H) (rcong H) (λa b. a ⊗ b <# H)"
⟨*proof*⟩

7.7 Order of a Group and Lagrange's Theorem

definition
order :: "('a, 'b) monoid_scheme ⇒ nat"
where "order S = card (carrier S)"

lemma iso_same_order:
assumes "φ ∈ iso G H"
shows "order G = order H"
⟨*proof*⟩

lemma (in monoid) order_gt_0_iff_finite: "0 < order G ⟷ finite (carrier G)"
⟨*proof*⟩

lemma (in group) order_one_triv_iff:
shows "(order G = 1) = (carrier G = {1})"

⟨*proof*⟩

lemma (in group**)** rcosets_part_G:
assumes "subgroup H G"
shows "⋃(rcosets H) = carrier G"
⟨*proof*⟩

lemma (in group**)** cosets_finite:
"⟦c ∈ rcosets H; H ⊆ carrier G; finite (carrier G)⟧ ⟹ finite c"
⟨*proof*⟩

The next two lemmas support the proof of card_cosets_equal.

lemma (in group**)** inj_on_f:
assumes "H ⊆ carrier G" **and** a: "a ∈ carrier G"
shows "inj_on (λy. y ⊗ inv a) (H #> a)"
⟨*proof*⟩

lemma (in group**)** inj_on_g:
"⟦H ⊆ carrier G; a ∈ carrier G⟧ ⟹ inj_on (λy. y ⊗ a) H"
⟨*proof*⟩

lemma (in group**)** card_cosets_equal:
assumes "R ∈ rcosets H" "H ⊆ carrier G"
shows "∃f. bij_betw f H R"
⟨*proof*⟩

corollary (in group**)** card_rcosets_equal:
assumes "R ∈ rcosets H" "H ⊆ carrier G"
shows "card H = card R"
⟨*proof*⟩

corollary (in group**)** rcosets_finite:
assumes "R ∈ rcosets H" "H ⊆ carrier G" "finite H"
shows "finite R"
⟨*proof*⟩

lemma (in group**)** rcosets_subset_PowG:
"subgroup H G ⟹ rcosets H ⊆ Pow(carrier G)"
⟨*proof*⟩

proposition (in group**)** lagrange_finite:
assumes "finite(carrier G)" **and** HG: "subgroup H G"
shows "card(rcosets H) * card(H) = order(G)"
⟨*proof*⟩

theorem (**in** group) lagrange:
  assumes "subgroup H G"
  shows "card (rcosets H) * card H = order G"
⟨*proof*⟩

The cardinality of the right cosets of the trivial subgroup is the cardinality of the group itself:

corollary (**in** group) card_rcosets_triv:
  assumes "finite (carrier G)"
  shows "card (rcosets {**1**}) = order G"
  ⟨*proof*⟩

7.8 Quotient Groups: Factorization of a Group

definition
  FactGroup :: "[('a,'b) monoid_scheme, 'a set] ⇒ ('a set) monoid" (**infixl** <Mod> 65)
    — Actually defined for groups rather than monoids
   where "FactGroup G H = (|carrier = rcosets$_G$ H, mult = set_mult G, one = H|)"

lemma (**in** normal) setmult_closed:
    "⟦K1 ∈ rcosets H; K2 ∈ rcosets H⟧ ⟹ K1 <#> K2 ∈ rcosets H"
⟨*proof*⟩

lemma (**in** normal) setinv_closed:
    "K ∈ rcosets H ⟹ set_inv K ∈ rcosets H"
⟨*proof*⟩

lemma (**in** normal) rcosets_assoc:
    "⟦M1 ∈ rcosets H; M2 ∈ rcosets H; M3 ∈ rcosets H⟧
     ⟹ M1 <#> M2 <#> M3 = M1 <#> (M2 <#> M3)"
  ⟨*proof*⟩

lemma (**in** subgroup) subgroup_in_rcosets:
  assumes "group G"
  shows "H ∈ rcosets H"
⟨*proof*⟩

lemma (**in** normal) rcosets_inv_mult_group_eq:
    "M ∈ rcosets H ⟹ set_inv M <#> M = H"
⟨*proof*⟩

theorem (**in** normal) factorgroup_is_group: "group (G Mod H)"
⟨*proof*⟩

lemma carrier_FactGroup: "carrier(G Mod N) = (λx. r_coset G N x) ‘ carrier G"

⟨*proof*⟩

lemma one_FactGroup [simp]: "one(G Mod N) = N"
⟨*proof*⟩

lemma mult_FactGroup [simp]: "monoid.mult (G Mod N) = set_mult G"
⟨*proof*⟩

lemma (in normal**)** inv_FactGroup:
assumes "X ∈ carrier (G Mod H)"
shows "inv$_{\text{G Mod H}}$ X = set_inv X"
⟨*proof*⟩

The coset map is a homomorphism from G to the quotient group G Mod H

lemma (in normal**)** r_coset_hom_Mod:
"(λa. H #> a) ∈ hom G (G Mod H)"
⟨*proof*⟩

lemma (in comm_group**)** set_mult_commute:
assumes "N ⊆ carrier G" "x ∈ rcosets N" "y ∈ rcosets N"
shows "x <#> y = y <#> x"
⟨*proof*⟩

lemma (in comm_group**)** abelian_FactGroup:
assumes "subgroup N G" **shows** "comm_group(G Mod N)"
⟨*proof*⟩

lemma FactGroup_universal:
assumes "h ∈ hom G H" "N ◁ G"
and h: "⋀x y. ⟦x ∈ carrier G; y ∈ carrier G; r_coset G N x = r_coset G N y⟧ ⟹ h x = h y"
obtains g
where "g ∈ hom (G Mod N) H" "⋀x. x ∈ carrier G ⟹ g(r_coset G N x) = h x"
⟨*proof*⟩

lemma (in normal**)** FactGroup_pow:
fixes k::nat
assumes "a ∈ carrier G"
shows "pow (FactGroup G H) (r_coset G H a) k = r_coset G H (pow G a k)"
⟨*proof*⟩

lemma (in normal**)** FactGroup_int_pow:
fixes k::int
assumes "a ∈ carrier G"

shows "pow (FactGroup G H) (r_coset G H a) k = r_coset G H (pow G a k)"
⟨*proof*⟩

7.9 The First Isomorphism Theorem

The quotient by the kernel of a homomorphism is isomorphic to the range of that homomorphism.

definition
kernel :: "('a, 'm) monoid_scheme ⇒ ('b, 'n) monoid_scheme ⇒ ('a ⇒ 'b) ⇒ 'a set"
— the kernel of a homomorphism
where "kernel G H h = {x. x ∈ carrier G ∧ h x = $\mathbf{1}_H$}"

lemma (in group_hom**)** subgroup_kernel: "subgroup (kernel G H h) G"
⟨*proof*⟩

The kernel of a homomorphism is a normal subgroup

lemma (in group_hom**)** normal_kernel: "(kernel G H h) ◁ G"
⟨*proof*⟩

lemma iso_kernel_image:
assumes "group G" "group H"
shows "f ∈ iso G H ⟷ f ∈ hom G H ∧ kernel G H f = {$\mathbf{1}_G$} ∧ f ‘ carrier G = carrier H"
(is "?lhs = ?rhs")
⟨*proof*⟩

lemma (in group_hom**)** FactGroup_nonempty:
assumes "X ∈ carrier (G Mod kernel G H h)"
shows "X ≠ {}"
⟨*proof*⟩

lemma (in group_hom**)** FactGroup_universal_kernel:
assumes "N ◁ G" **and** h: "N ⊆ kernel G H h"
obtains g **where** "g ∈ hom (G Mod N) H" "⋀x. x ∈ carrier G ⟹ g(r_coset G N x) = h x"
⟨*proof*⟩

lemma (in group_hom**)** FactGroup_the_elem_mem:
assumes X: "X ∈ carrier (G Mod (kernel G H h))"
shows "the_elem (h‘X) ∈ carrier H"
⟨*proof*⟩

lemma (in group_hom**)** FactGroup_hom:
"(λX. the_elem (h‘X)) ∈ hom (G Mod (kernel G H h)) H"

⟨*proof*⟩

Lemma for the following injectivity result

lemma (in group_hom**)** FactGroup_subset:
  assumes "g ∈ carrier G" "g' ∈ carrier G" "h g = h g'"
  shows "kernel G H h #> g ⊆ kernel G H h #> g'"
  ⟨*proof*⟩

lemma (in group_hom**)** FactGroup_inj_on:
    "inj_on (λX. the_elem (h ‘ X)) (carrier (G Mod kernel G H h))"
⟨*proof*⟩

If the homomorphism `h` is onto `H`, then so is the homomorphism from the quotient group

lemma (in group_hom**)** FactGroup_onto:
  assumes h: "h ‘ carrier G = carrier H"
  shows "(λX. the_elem (h ‘ X)) ‘ carrier (G Mod kernel G H h) = carrier H"
⟨*proof*⟩

If `h` is a homomorphism from `G` onto `H`, then the quotient group `G Mod Coset.kernel G H h` is isomorphic to `H`.

theorem (in group_hom**)** FactGroup_iso_set:
  "h ‘ carrier G = carrier H
   ⟹ (λX. the_elem (h‘X)) ∈ iso (G Mod (kernel G H h)) H"
⟨*proof*⟩

corollary (in group_hom**)** FactGroup_iso :
  "h ‘ carrier G = carrier H
   ⟹ (G Mod (kernel G H h))≅ H"
  ⟨*proof*⟩

lemma (in group_hom**)** trivial_hom_iff:
  "h ‘ (carrier G) = { $\mathbf{1}_H$ } ⟷ kernel G H h = carrier G"
  ⟨*proof*⟩

lemma (in group_hom**)** trivial_ker_imp_inj:
  assumes "kernel G H h = { **1** }"
  shows "inj_on h (carrier G)"
⟨*proof*⟩

lemma (in group_hom**)** inj_iff_trivial_ker:
  shows "inj_on h (carrier G) ⟷ kernel G H h = { **1** }"
⟨*proof*⟩

lemma (in group_hom**)** induced_group_hom':
  assumes "subgroup I G" **shows** "group_hom (G (| carrier := I |)) H h"

⟨*proof*⟩

lemma (in group_hom**)** inj_on_subgroup_iff_trivial_ker:
assumes "subgroup I G"
shows "inj_on h I ⟷ kernel (G ⦇ carrier := I ⦈) H h = { **1** }"
⟨*proof*⟩

lemma set_mult_hom:
assumes "h ∈ hom G H" "I ⊆ carrier G" **and** "J ⊆ carrier G"
shows "h ‘ (I <#>$_G$ J) = (h ‘ I) <#>$_H$ (h ‘ J)"
⟨*proof*⟩

corollary coset_hom:
assumes "h ∈ hom G H" "I ⊆ carrier G" "a ∈ carrier G"
shows "h ‘ (a <#$_G$ I) = h a <#$_H$ (h ‘ I)" **and** "h ‘ (I #>$_G$ a) = (h ‘ I) #>$_H$ h a"
⟨*proof*⟩

corollary (in group_hom**)** set_mult_ker_hom:
assumes "I ⊆ carrier G"
shows "h ‘ (I <#> (kernel G H h)) = h ‘ I" **and** "h ‘ ((kernel G H h) <#> I) = h ‘ I"
⟨*proof*⟩

7.9.1 Trivial homomorphisms

definition trivial_homomorphism **where**
"trivial_homomorphism G H f ≡ f ∈ hom G H ∧ (∀x ∈ carrier G. f x = one H)"

lemma trivial_homomorphism_kernel:
"trivial_homomorphism G H f ⟷ f ∈ hom G H ∧ kernel G H f = carrier G"
⟨*proof*⟩

lemma (in group**)** trivial_homomorphism_image:
"trivial_homomorphism G H f ⟷ f ∈ hom G H ∧ f ‘ carrier G = {one H}"
⟨*proof*⟩

7.10 Image kernel theorems

lemma group_Int_image_ker:
assumes f: "f ∈ hom G H" **and** g: "g ∈ hom H K"
and "inj_on (g ∘ f) (carrier G)" "group G" "group H" "group K"
shows "(f ‘ carrier G) ∩ (kernel H K g) = {$\mathbf{1}_H$}"
⟨*proof*⟩

lemma group_sum_image_ker:

assumes f: "f ∈ hom G H" **and** g: "g ∈ hom H K" **and** eq: "(g ∘ f) ‘ (carrier G) = carrier K"
and "group G" "group H" "group K"
shows "set_mult H (f ‘ carrier G) (kernel H K g) = carrier H" (**is** "?lhs = ?rhs")
⟨*proof*⟩

lemma group_sum_ker_image:
assumes f: "f ∈ hom G H" **and** g: "g ∈ hom H K" **and** eq: "(g ∘ f) ‘ (carrier G) = carrier K"
and "group G" "group H" "group K"
shows "set_mult H (kernel H K g) (f ‘ carrier G) = carrier H" (**is** "?lhs = ?rhs")
⟨*proof*⟩

lemma group_semidirect_sum_ker_image:
assumes "(g ∘ f) ∈ iso G K" "f ∈ hom G H" "g ∈ hom H K" "group G" "group H" "group K"
shows "(kernel H K g) ∩ (f ‘ carrier G) = {$\mathbf{1}_H$}"
"kernel H K g <#>$_H$ (f ‘ carrier G) = carrier H"
⟨*proof*⟩

lemma group_semidirect_sum_image_ker:
assumes f: "f ∈ hom G H" **and** g: "g ∈ hom H K" **and** iso: "(g ∘ f) ∈ iso G K"
and "group G" "group H" "group K"
shows "(f ‘ carrier G) ∩ (kernel H K g) = {$\mathbf{1}_H$}"
"f ‘ carrier G <#>$_H$ (kernel H K g) = carrier H"
⟨*proof*⟩

7.11 Factor Groups and Direct product

lemma (**in** group) DirProd_normal :
assumes "group K"
and "H ◁ G"
and "N ◁ K"
shows "H × N ◁ G ×× K"
⟨*proof*⟩

lemma (**in** group) FactGroup_DirProd_multiplication_iso_set :
assumes "group K"
and "H ◁ G"
and "N ◁ K"
shows "(λ (X, Y). X × Y) ∈ iso ((G Mod H) ×× (K Mod N)) (G ×× K Mod H × N)"
⟨*proof*⟩

corollary (**in** group) FactGroup_DirProd_multiplication_iso_1 :

assumes "group K"
and "H ◁ G"
and "N ◁ K"
shows " ((G Mod H) ×× (K Mod N)) ≅ (G ×× K Mod H × N)"
⟨*proof*⟩

corollary (in group**)** FactGroup_DirProd_multiplication_iso_2 :
assumes "group K"
and "H ◁ G"
and "N ◁ K"
shows "(G ×× K Mod H × N) ≅ ((G Mod H) ×× (K Mod N))"
⟨*proof*⟩

7.11.1 More Lemmas about set multiplication

A group multiplied by a subgroup stays the same

lemma (in group**)** set_mult_carrier_idem:
assumes "subgroup H G"
shows "(carrier G) <#> H = carrier G"
⟨*proof*⟩

Same lemma as above, but everything is included in a subgroup

lemma (in group**)** set_mult_subgroup_idem:
assumes HG: "subgroup H G" **and** NG: "subgroup N (G (| carrier := H |))"
shows "H <#> N = H"
⟨*proof*⟩

A normal subgroup is commutative with set multiplication

lemma (in group**)** commut_normal:
assumes "subgroup H G" **and** "N◁G"
shows "H<#>N = N<#>H"
⟨*proof*⟩

Same lemma as above, but everything is included in a subgroup

lemma (in group**)** commut_normal_subgroup:
assumes "subgroup H G" **and** "N ◁ (G(| carrier := H |))"
and "subgroup K (G (| carrier := H |))"
shows "K <#> N = N <#> K"
⟨*proof*⟩

7.11.2 Lemmas about intersection and normal subgroups

Mostly by Jakob von Raumer

lemma (in group**)** normal_inter:
assumes "subgroup H G"
and "subgroup K G"
and "H1◁G(|carrier := H|)"

shows "(H1∩K)◁(G(|carrier:= (H∩K)|))"
⟨*proof*⟩

lemma (in group**)** normal_Int_subgroup:
assumes "subgroup H G"
and "N ◁ G"
shows "(N∩H) ◁ (G(|carrier := H|))"
⟨*proof*⟩

lemma (in group**)** normal_restrict_supergroup:
assumes "subgroup S G" "N ◁ G" "N ⊆ S"
shows "N ◁ (G(|carrier := S|))"
⟨*proof*⟩

A subgroup relation survives factoring by a normal subgroup.

lemma (in group**)** normal_subgroup_factorize:
assumes "N ◁ G" **and** "N ⊆ H" **and** "subgroup H G"
shows "subgroup (rcosets$_{G(|carrier := H|)}$ N) (G Mod N)"
⟨*proof*⟩

A normality relation survives factoring by a normal subgroup.

lemma (in group**)** normality_factorization:
assumes NG: "N ◁ G" **and** NH: "N ⊆ H" **and** HG: "H ◁ G"
shows "(rcosets$_{G(|carrier := H|)}$ N) ◁ (G Mod N)"
⟨*proof*⟩

Factorizing by the trivial subgroup is an isomorphism.

lemma (in group**)** trivial_factor_iso:
shows "the_elem ∈ iso (G Mod {**1**}) G"
⟨*proof*⟩

And the dual theorem to the previous one: Factorizing by the group itself gives the trivial group

lemma (in group**)** self_factor_iso:
shows "(λX. the_elem ((λx. **1**) ‘ X)) ∈ iso (G Mod (carrier G)) (G(| carrier := {**1**} |))"
⟨*proof*⟩

Factoring by a normal subgroups yields the trivial group iff the subgroup is the whole group.

lemma (in normal**)** fact_group_trivial_iff:
assumes "finite (carrier G)"
shows "(carrier (G Mod H) = {**1**$_{G\ Mod\ H}$}) ⟷ (H = carrier G)"
⟨*proof*⟩

The union of all the cosets contained in a subgroup of a quotient group acts as a represenation for that subgroup.

lemma (in normal**)** factgroup_subgroup_union_char:
assumes "subgroup A (G Mod H)"
shows "(⋃A) = {x ∈ carrier G. H #> x ∈ A}"
⟨*proof*⟩

lemma (in normal**)** factgroup_subgroup_union_subgroup:
assumes "subgroup A (G Mod H)"
shows "subgroup (⋃A) G"
⟨*proof*⟩

lemma (in normal**)** factgroup_subgroup_union_normal:
assumes "A ◁ (G Mod H)"
shows "⋃A ◁ G"
⟨*proof*⟩

lemma (in normal**)** factgroup_subgroup_union_factor:
assumes "subgroup A (G Mod H)"
shows "A = rcosets$_{G(\!|\text{carrier} := \bigcup A|\!)}$ H"
⟨*proof*⟩

8 Flattening the type of group carriers

Flattening here means to convert the type of group elements from 'a set to 'a. This is possible whenever the empty set is not an element of the group.
By Jakob von Raumer

definition flatten **where**
"flatten (G::('a set, 'b) monoid_scheme) rep = (|carrier=(rep ‘ (carrier G)),
monoid.mult=(λ x y. rep ((the_inv_into (carrier G) rep x) ⊗$_G$ (the_inv_into (carrier G) rep y))),
one=rep $\mathbf{1}_G$ |)"

lemma flatten_set_group_hom:
assumes group: "group G"
assumes inj: "inj_on rep (carrier G)"
shows "rep ∈ hom G (flatten G rep)"
⟨*proof*⟩

lemma flatten_set_group:
assumes "group G" "inj_on rep (carrier G)"
shows "group (flatten G rep)"
⟨*proof*⟩

lemma (in normal**)** flatten_set_group_mod_inj:
shows "inj_on (λU. SOME g. g ∈ U) (carrier (G Mod H))"
⟨*proof*⟩

lemma (in normal**)** flatten_set_group_mod:

shows "group (flatten (G Mod H) (λU. SOME g. g ∈ U))"
⟨*proof*⟩

lemma (in normal**)** flatten_set_group_mod_iso:
shows "(λU. SOME g. g ∈ U) ∈ iso (G Mod H) (flatten (G Mod H) (λU. SOME g. g ∈ U))"
⟨*proof*⟩

end

theory Exponent
imports Main "HOL-Computational_Algebra.Primes"
begin

9 Sylow's Theorem

The Combinatorial Argument Underlying the First Sylow Theorem

needed in this form to prove Sylow's theorem

corollary (in algebraic_semidom**)** div_combine:
"⟦prime_elem p; ¬ p ^ Suc r dvd n; p ^ (a + r) dvd n * k⟧ ⟹ p ^ a dvd k"
⟨*proof*⟩

lemma exponent_p_a_m_k_equation:
fixes p :: nat
assumes "0 < m" "0 < k" "p ≠ 0" "k < p^a"
shows "multiplicity p (p^a * m - k) = multiplicity p (p^a - k)"
⟨*proof*⟩

lemma p_not_div_choose_lemma:
fixes p :: nat
assumes eeq: "⋀i. Suc i < K ⟹ multiplicity p (Suc i) = multiplicity p (Suc (j + i))"
and "k < K" **and** p: "prime p"
shows "multiplicity p (j + k choose k) = 0"
⟨*proof*⟩

The lemma above, with two changes of variables

lemma p_not_div_choose:
assumes "k < K" **and** "k ≤ n"
and eeq: "⋀j. ⟦0<j; j<K⟧ ⟹ multiplicity p (n - k + (K - j)) = multiplicity p (K - j)" "prime p"
shows "multiplicity p (n choose k) = 0"
⟨*proof*⟩

proposition const_p_fac:

assumes "m>0" **and** prime: "prime p"
shows "multiplicity p (p^a * m choose p^a) = multiplicity p m"
⟨*proof*⟩

end

theory Sylow
imports Coset Exponent
begin

See also [4].

The combinatorial argument is in theory Exponent.

lemma le_extend_mult: "⟦0 < c; a ≤ b⟧ ⟹ a ≤ b * c" **for** c :: nat
⟨*proof*⟩

locale sylow = group +
fixes p **and** a **and** m **and** calM **and** RelM
assumes prime_p: "prime p"
and order_G: "order G = (p^a) * m"
and finite_G[iff]: "finite (carrier G)"
defines "calM ≡ {s. s ⊆ carrier G ∧ card s = p^a}"
and "RelM ≡ {(N1, N2). N1 ∈ calM ∧ N2 ∈ calM ∧ (∃g ∈ carrier G.
N1 = N2 #> g)}"
begin

lemma RelM_subset: "RelM ⊆ calM × calM"
⟨*proof*⟩

lemma RelM_refl_on: "refl_on calM RelM"
⟨*proof*⟩

lemma RelM_sym: "sym RelM"
⟨*proof*⟩

lemma RelM_trans: "trans RelM"
⟨*proof*⟩

lemma RelM_equiv: "equiv calM RelM"
⟨*proof*⟩

lemma M_subset_calM_prep: "M' ∈ calM // RelM ⟹ M' ⊆ calM"
⟨*proof*⟩

end

9.1 Main Part of the Proof

locale sylow_central = sylow +

```
  fixes H and M1 and M
  assumes M_in_quot: "M ∈ calM // RelM"
    and not_dvd_M: "¬ (p ^ Suc (multiplicity p m) dvd card M)"
    and M1_in_M: "M1 ∈ M"
  defines "H ≡ {g. g ∈ carrier G ∧ M1 #> g = M1}"
begin

lemma M_subset_calM: "M ⊆ calM"
  ⟨proof⟩

lemma card_M1: "card M1 = p^a"
  ⟨proof⟩

lemma exists_x_in_M1: "∃x. x ∈ M1"
  ⟨proof⟩

lemma M1_subset_G [simp]: "M1 ⊆ carrier G"
  ⟨proof⟩

lemma M1_inj_H: "∃f ∈ H→M1. inj_on f H"
⟨proof⟩

end
```

9.2 Discharging the Assumptions of `sylow_central`

```
context sylow
begin

lemma EmptyNotInEquivSet: "{} ∉ calM // RelM"
  ⟨proof⟩

lemma existsM1inM: "M ∈ calM // RelM ⟹ ∃M1. M1 ∈ M"
  ⟨proof⟩

lemma zero_less_o_G: "0 < order G"
  ⟨proof⟩

lemma zero_less_m: "m > 0"
  ⟨proof⟩

lemma card_calM: "card calM = (p^a) * m choose p^a"
  ⟨proof⟩

lemma zero_less_card_calM: "card calM > 0"
  ⟨proof⟩

lemma max_p_div_calM: "¬ (p ^ Suc (multiplicity p m) dvd card calM)"
⟨proof⟩
```

lemma finite_calM: "finite calM"
⟨*proof*⟩

lemma lemma_A1: "∃M ∈ calM // RelM. ¬ (p ^ Suc (multiplicity p m) dvd card M)"
⟨*proof*⟩

end

9.2.1 Introduction and Destruct Rules for H

context sylow_central
begin

lemma H_I: "⟦g ∈ carrier G; M1 #> g = M1⟧ ⟹ g ∈ H"
⟨*proof*⟩

lemma H_into_carrier_G: "x ∈ H ⟹ x ∈ carrier G"
⟨*proof*⟩

lemma in_H_imp_eq: "g ∈ H ⟹ M1 #> g = M1"
⟨*proof*⟩

lemma H_m_closed: "⟦x ∈ H; y ∈ H⟧ ⟹ x ⊗ y ∈ H"
⟨*proof*⟩

lemma H_not_empty: "H ≠ {}"
⟨*proof*⟩

lemma H_is_subgroup: "subgroup H G"
⟨*proof*⟩

lemma rcosetGM1g_subset_G: "⟦g ∈ carrier G; x ∈ M1 #> g⟧ ⟹ x ∈ carrier G"
⟨*proof*⟩

lemma finite_M1: "finite M1"
⟨*proof*⟩

lemma finite_rcosetGM1g: "g ∈ carrier G ⟹ finite (M1 #> g)"
⟨*proof*⟩

lemma M1_cardeq_rcosetGM1g: "g ∈ carrier G ⟹ card (M1 #> g) = card M1"
⟨*proof*⟩

lemma M1_RelM_rcosetGM1g:
assumes "g ∈ carrier G"

shows "(M1, M1 #> g) ∈ RelM"
⟨*proof*⟩

end

9.3 Equal Cardinalities of M and the Set of Cosets

Injections between M and rcosets$_G$ H show that their cardinalities are equal.

lemma ElemClassEquiv: "⟦equiv A r; C ∈ A // r⟧ ⟹ ∀x ∈ C. ∀y ∈ C. (x, y) ∈ r"
⟨*proof*⟩

context sylow_central
begin

lemma M_elem_map: "M2 ∈ M ⟹ ∃g. g ∈ carrier G ∧ M1 #> g = M2"
⟨*proof*⟩

lemmas M_elem_map_carrier = M_elem_map [THEN someI_ex, THEN conjunct1]

lemmas M_elem_map_eq = M_elem_map [THEN someI_ex, THEN conjunct2]

lemma M_funcset_rcosets_H:
"(λx∈M. H #> (SOME g. g ∈ carrier G ∧ M1 #> g = x)) ∈ M → rcosets H"
⟨*proof*⟩

lemma inj_M_GmodH: "∃f ∈ M → rcosets H. inj_on f M"
⟨*proof*⟩

end

9.3.1 The Opposite Injection

context sylow_central
begin

lemma H_elem_map: "H1 ∈ rcosets H ⟹ ∃g. g ∈ carrier G ∧ H #> g = H1"
⟨*proof*⟩

lemmas H_elem_map_carrier = H_elem_map [THEN someI_ex, THEN conjunct1]

lemmas H_elem_map_eq = H_elem_map [THEN someI_ex, THEN conjunct2]

lemma rcosets_H_funcset_M:
"(λC ∈ rcosets H. M1 #> (SOME g. g ∈ carrier G ∧ H #> g = C)) ∈ rcosets H → M"
⟨*proof*⟩

lemma inj_GmodH_M: "∃g ∈ rcosets H→M. inj_on g (rcosets H)"
⟨*proof*⟩

lemma calM_subset_PowG: "calM ⊆ Pow (carrier G)"
⟨*proof*⟩

lemma finite_M: "finite M"
⟨*proof*⟩

lemma cardMeqIndexH: "card M = card (rcosets H)"
⟨*proof*⟩

lemma index_lem: "card M * card H = order G"
⟨*proof*⟩

lemma card_H_eq: "card H = p^a"
⟨*proof*⟩

end

lemma (**in** sylow) sylow_thm: "∃H. subgroup H G ∧ card H = p^a"
⟨*proof*⟩

Needed because the locale's automatic definition refers to semigroup G and Group.group_axioms G rather than simply to Group.group G.

lemma sylow_eq: "sylow G p a m ⟷ group G ∧ sylow_axioms G p a m"
⟨*proof*⟩

9.4 Sylow's Theorem

theorem sylow_thm:
"⟦prime p; group G; order G = (p^a) * m; finite (carrier G)⟧
⟹ ∃H. subgroup H G ∧ card H = p^a"
⟨*proof*⟩

end

theory Bij
imports Group
begin

10 Bijections of a Set, Permutation and Automorphism Groups

definition

```
  Bij :: "'a set ⇒ ('a ⇒ 'a) set"
```
— Only extensional functions, since otherwise we get too many.
```
   where "Bij S = extensional S ∩ {f. bij_betw f S S}"
```

definition
```
  BijGroup :: "'a set ⇒ ('a ⇒ 'a) monoid"
  where "BijGroup S =
    (|carrier = Bij S,
     mult = λg ∈ Bij S. λf ∈ Bij S. compose S g f,
     one = λx ∈ S. x|)"
```

declare `Id_compose [simp] compose_Id [simp]`

lemma `Bij_imp_extensional: "f ∈ Bij S ⟹ f ∈ extensional S"`
⟨*proof*⟩

lemma `Bij_imp_funcset: "f ∈ Bij S ⟹ f ∈ S → S"`
⟨*proof*⟩

10.1 Bijections Form a Group

lemma `restrict_inv_into_Bij: "f ∈ Bij S ⟹ (λx ∈ S. (inv_into S f) x) ∈ Bij S"`
⟨*proof*⟩

lemma `id_Bij: "(λx∈S. x) ∈ Bij S "`
⟨*proof*⟩

lemma `compose_Bij: "⟦x ∈ Bij S; y ∈ Bij S⟧ ⟹ compose S x y ∈ Bij S"`
⟨*proof*⟩

lemma `Bij_compose_restrict_eq:`
`"f ∈ Bij S ⟹ compose S (restrict (inv_into S f) S) f = (λx∈S. x)"`
⟨*proof*⟩

theorem `group_BijGroup: "group (BijGroup S)"`
⟨*proof*⟩

10.2 Automorphisms Form a Group

lemma `Bij_inv_into_mem: "⟦ f ∈ Bij S;  x ∈ S⟧ ⟹ inv_into S f x ∈ S"`
⟨*proof*⟩

lemma `Bij_inv_into_lemma:`
assumes `eq: "⋀x y. ⟦x ∈ S; y ∈ S⟧ ⟹ h(g x y) = g (h x) (h y)"`
and `hg: "h ∈ Bij S" "g ∈ S → S → S"` **and** `"x ∈ S" "y ∈ S"`
shows `"inv_into S h (g x y) = g (inv_into S h x) (inv_into S h y)"`
⟨*proof*⟩

definition
  auto :: "('a, 'b) monoid_scheme ⇒ ('a ⇒ 'a) set"
  where "auto G = hom G G ∩ Bij (carrier G)"

definition
  AutoGroup :: "('a, 'c) monoid_scheme ⇒ ('a ⇒ 'a) monoid"
  where "AutoGroup G = BijGroup (carrier G) (|carrier := auto G|)"

lemma (**in** group) id_in_auto: "(λx ∈ carrier G. x) ∈ auto G"
  ⟨*proof*⟩

lemma (**in** group) mult_funcset: "mult G ∈ carrier G → carrier G → carrier G"
  ⟨*proof*⟩

lemma (**in** group) restrict_inv_into_hom:
      "⟦h ∈ hom G G; h ∈ Bij (carrier G)⟧
       ⟹ restrict (inv_into (carrier G) h) (carrier G) ∈ hom G G"
  ⟨*proof*⟩

lemma inv_BijGroup:
     "f ∈ Bij S ⟹ m_inv (BijGroup S) f = (λx ∈ S. (inv_into S f) x)"
⟨*proof*⟩

lemma (**in** group) subgroup_auto:
      "subgroup (auto G) (BijGroup (carrier G))"
⟨*proof*⟩

theorem (**in** group) AutoGroup: "group (AutoGroup G)"
⟨*proof*⟩

end

theory Ring
imports FiniteProduct
begin

11 The Algebraic Hierarchy of Rings

11.1 Abelian Groups

record 'a ring = "'a monoid" +
  zero :: 'a (‹𝟬›)
  add :: "['a, 'a] ⇒ 'a" (**infixl** ‹⊕› 65)

abbreviation

```
  add_monoid :: "('a, 'm) ring_scheme ⇒ ('a, 'm) monoid_scheme"
  where "add_monoid R ≡ ⦇ carrier = carrier R, mult = add R, one = zero
R, ... = (undefined :: 'm) ⦈"
```

Derived operations.

definition

```
  a_inv :: "['a, 'm) ring_scheme, 'a ] ⇒ 'a" (‹(‹open_block notation=‹prefix
⊖››⊖› _)› [81] 80)
  where "a_inv R = m_inv (add_monoid R)"
```

definition

```
  a_minus :: "['a, 'm) ring_scheme, 'a, 'a] => 'a" (‹(‹notation=‹infix
⊖›› _ ⊖› _)› [65,66] 65)
  where "x ⊖R y = x ⊕R (⊖R y)"
```

definition

```
  add_pow :: "[_, ('b :: semiring_1), 'a] ⇒ 'a"
    (‹(‹open_block notation=‹mixfix ·››[_] ·› _)› [81, 81] 80)
  where "[k] ·R a = pow (add_monoid R) a k"
```

locale abelian_monoid =

```
  fixes G (structure)
  assumes a_comm_monoid:
     "comm_monoid (add_monoid G)"
```

definition

```
  finsum :: "[('b, 'm) ring_scheme, 'a ⇒ 'b, 'a set] ⇒ 'b" where
  "finsum G = finprod (add_monoid G)"
```

syntax

```
  "_finsum" :: "index ⇒ idt ⇒ 'a set ⇒ 'b ⇒ 'b"
    (‹(‹indent=3 notation=‹binder ⨁ ››⨁ __∈_. _)› [1000, 0, 51, 10]
10)
```

syntax_consts

```
  "_finsum" ⇌ finsum
```

translations

```
  "⨁G i∈A. b" ⇌ "CONST finsum G (λi. b) A"
```

— Beware of argument permutation!

locale abelian_group = abelian_monoid +

```
  assumes a_comm_group:
     "comm_group (add_monoid G)"
```

11.2 Basic Properties

lemma abelian_monoidI:

```
  fixes R (structure)
  assumes "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y ∈ carrier
```

```
R"
      and "0 ∈ carrier R"
      and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)"
      and "⋀x. x ∈ carrier R ⟹ 0 ⊕ x = x"
      and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y = y ⊕ x"
  shows "abelian_monoid R"
  ⟨proof⟩

lemma abelian_monoidE:
  fixes R (structure)
  assumes "abelian_monoid R"
  shows "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y ∈ carrier
R"
    and "0 ∈ carrier R"
    and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)"
    and "⋀x. x ∈ carrier R ⟹ 0 ⊕ x = x"
    and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y = y ⊕ x"
  ⟨proof⟩

lemma abelian_groupI:
  fixes R (structure)
  assumes "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y ∈ carrier
R"
      and "0 ∈ carrier R"
      and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)"
      and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y = y ⊕ x"
      and "⋀x. x ∈ carrier R ⟹ 0 ⊕ x = x"
      and "⋀x. x ∈ carrier R ⟹ ∃y ∈ carrier R. y ⊕ x = 0"
  shows "abelian_group R"
  ⟨proof⟩

lemma abelian_groupE:
  fixes R (structure)
  assumes "abelian_group R"
  shows "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y ∈ carrier
R"
    and "0 ∈ carrier R"
    and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)"
    and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y = y ⊕ x"
    and "⋀x. x ∈ carrier R ⟹ 0 ⊕ x = x"
    and "⋀x. x ∈ carrier R ⟹ ∃y ∈ carrier R. y ⊕ x = 0"
  ⟨proof⟩

lemma (in abelian_monoid) a_monoid:
  "monoid (add_monoid G)"
```

⟨*proof*⟩

lemma (in abelian_group) a_group:
  "group (add_monoid G)"
  ⟨*proof*⟩

lemmas monoid_record_simps = partial_object.simps monoid.simps

Transfer facts from multiplicative structures via interpretation.

sublocale abelian_monoid <
      add: monoid "(add_monoid G)"
  rewrites "carrier (add_monoid G) = carrier G"
      and "mult   (add_monoid G) = add G"
      and "one   (add_monoid G) = zero G"
      and "(λa k. pow (add_monoid G) a k) = (λa k. add_pow G k a)"
  ⟨*proof*⟩

context abelian_monoid
begin

lemmas a_closed = add.m_closed
lemmas zero_closed = add.one_closed
lemmas a_assoc = add.m_assoc
lemmas l_zero = add.l_one
lemmas r_zero = add.r_one
lemmas minus_unique = add.inv_unique

end

sublocale abelian_monoid <
  add: comm_monoid "(add_monoid G)"
  rewrites "carrier (add_monoid G) = carrier G"
      and "mult   (add_monoid G) = add G"
      and "one   (add_monoid G) = zero G"
      and "finprod (add_monoid G) = finsum G"
      and "pow   (add_monoid G) = (λa k. add_pow G k a)"
  ⟨*proof*⟩

context abelian_monoid **begin**

lemmas a_comm = add.m_comm
lemmas a_lcomm = add.m_lcomm
lemmas a_ac = a_assoc a_comm a_lcomm

lemmas finsum_empty = add.finprod_empty
lemmas finsum_insert = add.finprod_insert
lemmas finsum_zero = add.finprod_one
lemmas finsum_closed = add.finprod_closed
lemmas finsum_Un_Int = add.finprod_Un_Int

```
lemmas finsum_Un_disjoint = add.finprod_Un_disjoint
lemmas finsum_addf = add.finprod_multf
lemmas finsum_cong' = add.finprod_cong'
lemmas finsum_0 = add.finprod_0
lemmas finsum_Suc = add.finprod_Suc
lemmas finsum_Suc2 = add.finprod_Suc2
lemmas finsum_infinite = add.finprod_infinite

lemmas finsum_cong = add.finprod_cong
```

Usually, if this rule causes a failed congruence proof error, the reason is that the premise `g ∈ B → carrier G` cannot be shown. Adding `Pi_def` to the simpset is often useful.

```
lemmas finsum_reindex = add.finprod_reindex



lemmas finsum_singleton = add.finprod_singleton

end

sublocale abelian_group <
        add: group "(add_monoid G)"
  rewrites "carrier (add_monoid G) = carrier G"
       and "mult    (add_monoid G) = add G"
       and "one     (add_monoid G) = zero G"
       and "m_inv   (add_monoid G) = a_inv G"
       and "pow     (add_monoid G) = (λa k. add_pow G k a)"
  ⟨proof⟩

context abelian_group
begin

lemmas a_inv_closed = add.inv_closed

lemma minus_closed [intro, simp]:
  "[| x ∈ carrier G; y ∈ carrier G |] ==> x ⊖ y ∈ carrier G"
  ⟨proof⟩

lemmas l_neg = add.l_inv [simp del]
lemmas r_neg = add.r_inv [simp del]
lemmas minus_minus = add.inv_inv
lemmas a_inv_inj = add.inv_inj
lemmas minus_equality = add.inv_equality

end

sublocale abelian_group <
   add: comm_group "(add_monoid G)"
```

```
  rewrites "carrier (add_monoid G) = carrier G"
       and "mult    (add_monoid G) = add G"
       and "one     (add_monoid G) = zero G"
       and "m_inv   (add_monoid G) = a_inv G"
       and "finprod (add_monoid G) = finsum G"
       and "pow     (add_monoid G) = (λa k. add_pow G k a)"
```
⟨*proof*⟩

lemmas (in abelian_group) minus_add = add.inv_mult

Derive an abelian_group from a comm_group

lemma comm_group_abelian_groupI:
```
  fixes G (structure)
  assumes cg: "comm_group (add_monoid G)"
  shows "abelian_group G"
```
⟨*proof*⟩

11.3 Rings: Basic Definitions

```
locale semiring = abelian_monoid  R + monoid  R for R (structure) +
  assumes l_distr: "⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
      and r_distr: "⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
z ⊗ (x ⊕ y) = z ⊗ x ⊕ z ⊗ y"
      and l_null[simp]: "x ∈ carrier R ⟹ 0 ⊗ x = 0"
      and r_null[simp]: "x ∈ carrier R ⟹ x ⊗ 0 = 0"

locale ring = abelian_group  R + monoid  R for R (structure) +
  assumes "⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹ (x ⊕ y)
⊗ z = x ⊗ z ⊕ y ⊗ z"
      and "⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹ z ⊗ (x
⊕ y) = z ⊗ x ⊕ z ⊗ y"

locale cring = ring + comm_monoid  R

locale "domain" = cring +
  assumes one_not_zero [simp]: "1 ≠ 0"
      and integral: "⟦ a ⊗ b = 0; a ∈ carrier R; b ∈ carrier R ⟧ ⟹
a = 0 ∨ b = 0"

locale field = "domain" +
  assumes field_Units: "Units R = carrier R - {0}"
```

11.4 Rings

lemma ringI:
```
  fixes R (structure)
  assumes "abelian_group R"
      and "monoid R"
```

```
      and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
      and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
z ⊗ (x ⊕ y) = z ⊗ x ⊕ z ⊗ y"
  shows "ring R"
  ⟨proof⟩

lemma ringE:
  fixes R (structure)
  assumes "ring R"
  shows "abelian_group R"
    and "monoid R"
    and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
    and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
z ⊗ (x ⊕ y) = z ⊗ x ⊕ z ⊗ y"
  ⟨proof⟩

context ring begin

lemma is_abelian_group: "abelian_group R" ⟨proof⟩

lemma is_monoid: "monoid R"
  ⟨proof⟩

end

thm monoid_record_simps
lemmas ring_record_simps = monoid_record_simps ring.simps

lemma cringI:
  fixes R (structure)
  assumes abelian_group: "abelian_group R"
    and comm_monoid: "comm_monoid R"
    and l_distr: "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier
R ⟧ ⟹
                              (x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
  shows "cring R"
⟨proof⟩

lemma cringE:
  fixes R (structure)
  assumes "cring R"
  shows "comm_monoid R"
    and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
  ⟨proof⟩

lemma (in cring) is_cring:
```

"cring R" ⟨*proof*⟩

lemma (in ring**)** minus_zero [simp]: "⊖ **0** = **0**"
⟨*proof*⟩

11.4.1 Normaliser for Rings

lemma (in abelian_group**)** r_neg1:
"⟦ x ∈ carrier G; y ∈ carrier G ⟧ ⟹ (⊖ x) ⊕ (x ⊕ y) = y"
⟨*proof*⟩

lemma (in abelian_group**)** r_neg2:
"⟦ x ∈ carrier G; y ∈ carrier G ⟧ ⟹ x ⊕ ((⊖ x) ⊕ y) = y"
⟨*proof*⟩

context ring **begin**

The following proofs are from Jacobson, Basic Algebra I, pp. 88–89.

sublocale semiring
⟨*proof*⟩

lemma l_minus:
"⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ (⊖ x) ⊗ y = ⊖ (x ⊗ y)"
⟨*proof*⟩

lemma r_minus:
"⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊗ (⊖ y) = ⊖ (x ⊗ y)"
⟨*proof*⟩

end

lemma (in abelian_group**)** minus_eq: "x ⊖ y = x ⊕ (⊖ y)"
⟨*proof*⟩

Setup algebra method: compute distributive normal form in locale contexts

⟨*ML*⟩

lemmas (in semiring**)** semiring_simprules
[algebra ring "zero R" "add R" "a_inv R" "a_minus R" "one R" "mult R"]
=
a_closed zero_closed m_closed one_closed
a_assoc l_zero a_comm m_assoc l_one l_distr r_zero
a_lcomm r_distr l_null r_null

lemmas (in ring**)** ring_simprules
[algebra ring "zero R" "add R" "a_inv R" "a_minus R" "one R" "mult R"]
=
a_closed zero_closed a_inv_closed minus_closed m_closed one_closed
a_assoc l_zero l_neg a_comm m_assoc l_one l_distr minus_eq

```
  r_zero r_neg r_neg2 r_neg1 minus_add minus_minus minus_zero
  a_lcomm r_distr l_null r_null l_minus r_minus

lemmas (in cring)
  [algebra del: ring "zero R" "add R" "a_inv R" "a_minus R" "one R" "mult
R"] =
  -

lemmas (in cring) cring_simprules
  [algebra add: cring "zero R" "add R" "a_inv R" "a_minus R" "one R" "mult
R"] =
  a_closed zero_closed a_inv_closed minus_closed m_closed one_closed
  a_assoc l_zero l_neg a_comm m_assoc l_one l_distr m_comm minus_eq
  r_zero r_neg r_neg2 r_neg1 minus_add minus_minus minus_zero
  a_lcomm m_lcomm r_distr l_null r_null l_minus r_minus

lemma (in semiring) nat_pow_zero:
  "(n::nat) ≠ 0 ⟹ 0 [^] n = 0"
  ⟨proof⟩

context semiring begin

lemma one_zeroD:
  assumes onezero: "1 = 0"
  shows "carrier R = {0}"
⟨proof⟩

lemma one_zeroI:
  assumes carrzero: "carrier R = {0}"
  shows "1 = 0"
⟨proof⟩

lemma carrier_one_zero: "(carrier R = {0}) = (1 = 0)"
  ⟨proof⟩

lemma carrier_one_not_zero: "(carrier R ≠ {0}) = (1 ≠ 0)"
  ⟨proof⟩

end
```

Two examples for use of method algebra

```
lemma
  fixes R (structure) and S (structure)
  assumes "ring R" "cring S"
  assumes RS: "a ∈ carrier R" "b ∈ carrier R" "c ∈ carrier S" "d ∈ carrier
S"
  shows "a ⊕ (⊖ (a ⊕ (⊖ b))) = b ∧ c ⊗S d = d ⊗S c"
⟨proof⟩
```

lemma
fixes R **(structure)**
assumes "ring R"
assumes R: "a ∈ carrier R" "b ∈ carrier R"
shows "a ⊖ (a ⊖ b) = b"
⟨*proof*⟩

11.4.2 Sums over Finite Sets

lemma (in semiring**)** finsum_ldistr:
"⟦ finite A; a ∈ carrier R; f: A → carrier R ⟧ ⟹
(⨁ i ∈ A. (f i)) ⊗ a = (⨁ i ∈ A. ((f i) ⊗ a))"
⟨*proof*⟩

lemma (in semiring**)** finsum_rdistr:
"⟦ finite A; a ∈ carrier R; f: A → carrier R ⟧ ⟹
a ⊗ (⨁ i ∈ A. (f i)) = (⨁ i ∈ A. (a ⊗ (f i)))"
⟨*proof*⟩

A quick detour

lemma add_pow_int_ge: "(k :: int) ≥ 0 ⟹ [k] ·$_R$ a = [nat k] ·$_R$ a"

⟨*proof*⟩

lemma add_pow_int_lt: "(k :: int) < 0 ⟹ [k] ·$_R$ a = ⊖$_R$ ([nat (- k)] ·$_R$ a)"
⟨*proof*⟩

corollary (in semiring**)** add_pow_ldistr:
assumes "a ∈ carrier R" "b ∈ carrier R"
shows "([(k :: nat)] · a) ⊗ b = [k] · (a ⊗ b)"
⟨*proof*⟩

corollary (in semiring**)** add_pow_rdistr:
assumes "a ∈ carrier R" "b ∈ carrier R"
shows "a ⊗ ([(k :: nat)] · b) = [k] · (a ⊗ b)"
⟨*proof*⟩

lemma (in ring**)** add_pow_ldistr_int:
assumes "a ∈ carrier R" "b ∈ carrier R"
shows "([(k :: int)] · a) ⊗ b = [k] · (a ⊗ b)"
⟨*proof*⟩

lemma (in ring**)** add_pow_rdistr_int:
assumes "a ∈ carrier R" "b ∈ carrier R"
shows "a ⊗ ([(k :: int)] · b) = [k] · (a ⊗ b)"
⟨*proof*⟩

11.5 Integral Domains

context "domain" **begin**

lemma zero_not_one [simp]: "**0** ≠ **1**"
⟨*proof*⟩

lemma integral_iff:
"⟦ a ∈ carrier R; b ∈ carrier R ⟧ ⟹ (a ⊗ b = **0**) = (a = **0** ∨ b = **0**)"
⟨*proof*⟩

lemma m_lcancel:
assumes prem: "a ≠ **0**"
and R: "a ∈ carrier R" "b ∈ carrier R" "c ∈ carrier R"
shows "(a ⊗ b = a ⊗ c) = (b = c)"
⟨*proof*⟩

lemma m_rcancel:
assumes prem: "a ≠ **0**"
and R: "a ∈ carrier R" "b ∈ carrier R" "c ∈ carrier R"
shows conc: "(b ⊗ a = c ⊗ a) = (b = c)"
⟨*proof*⟩

end

11.6 Fields

Field would not need to be derived from domain, the properties for domain follow from the assumptions of field

lemma (**in** field) is_ring: "ring R"
⟨*proof*⟩

lemma fieldE :
fixes R (**structure**)
assumes "field R"
shows "cring R"
and one_not_zero : "**1** ≠ **0**"
and integral: "⋀a b. ⟦ a ⊗ b = **0**; a ∈ carrier R; b ∈ carrier R ⟧
⟹ a = **0** ∨ b = **0**"
and field_Units: "Units R = carrier R - {**0**}"
⟨*proof*⟩

lemma (**in** cring) cring_fieldI:
assumes field_Units: "Units R = carrier R - {**0**}"
shows "field R"
⟨*proof*⟩

Another variant to show that something is a field

lemma (**in** cring) cring_fieldI2:

assumes notzero: "**0** $\neq$ **1**"
and invex: "$\bigwedge$a. ⟦a ∈ carrier R; a $\neq$ **0**⟧ $\Longrightarrow$ ∃b∈carrier R. a ⊗ b = **1**"
shows "field R"
⟨*proof*⟩

11.7 Morphisms

definition
ring_hom :: "[('a, 'm) ring_scheme, ('b, 'n) ring_scheme] => ('a => 'b) set"
where "ring_hom R S =
{h. h ∈ carrier R → carrier S ∧
(∀x y. x ∈ carrier R ∧ y ∈ carrier R ⟶
h (x ⊗$_R$ y) = h x ⊗$_S$ h y ∧ h (x ⊕$_R$ y) = h x ⊕$_S$ h y) ∧
h **1**$_R$ = **1**$_S$}"

lemma ring_hom_memI:
fixes R **(structure)** **and** S **(structure)**
assumes "$\bigwedge$x. x ∈ carrier R $\Longrightarrow$ h x ∈ carrier S"
and "$\bigwedge$x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ $\Longrightarrow$ h (x ⊗ y) = h x ⊗$_S$ h y"
and "$\bigwedge$x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ $\Longrightarrow$ h (x ⊕ y) = h x ⊕$_S$ h y"
and "h **1** = **1**$_S$"
shows "h ∈ ring_hom R S"
⟨*proof*⟩

lemma ring_hom_memE:
fixes R **(structure)** **and** S **(structure)**
assumes "h ∈ ring_hom R S"
shows "$\bigwedge$x. x ∈ carrier R $\Longrightarrow$ h x ∈ carrier S"
and "$\bigwedge$x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ $\Longrightarrow$ h (x ⊗ y) = h x ⊗$_S$ h y"
and "$\bigwedge$x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ $\Longrightarrow$ h (x ⊕ y) = h x ⊕$_S$ h y"
and "h **1** = **1**$_S$"
⟨*proof*⟩

lemma ring_hom_closed:
"⟦ h ∈ ring_hom R S; x ∈ carrier R ⟧ $\Longrightarrow$ h x ∈ carrier S"
⟨*proof*⟩

lemma ring_hom_mult:
fixes R **(structure)** **and** S **(structure)**
shows "⟦ h ∈ ring_hom R S; x ∈ carrier R; y ∈ carrier R ⟧ $\Longrightarrow$ h (x ⊗ y) = h x ⊗$_S$ h y"
⟨*proof*⟩

lemma ring_hom_add:
fixes R **(structure) and** S **(structure)**
shows "⟦ h ∈ ring_hom R S; x ∈ carrier R; y ∈ carrier R ⟧ ⟹ h (x ⊕ y) = h x ⊕$_S$ h y"
⟨*proof*⟩

lemma ring_hom_one:
fixes R **(structure) and** S **(structure)**
shows "h ∈ ring_hom R S ⟹ h **1** = **1**$_S$"
⟨*proof*⟩

lemma ring_hom_zero:
fixes R **(structure) and** S **(structure)**
assumes "h ∈ ring_hom R S" "ring R" "ring S"
shows "h **0** = **0**$_S$"
⟨*proof*⟩

locale ring_hom_cring =
R?: cring R + S?: cring S **for** R **(structure) and** S **(structure) + fixes** h
assumes homh [simp, intro]: "h ∈ ring_hom R S"
notes hom_closed [simp, intro] = ring_hom_closed [OF homh]
and hom_mult [simp] = ring_hom_mult [OF homh]
and hom_add [simp] = ring_hom_add [OF homh]
and hom_one [simp] = ring_hom_one [OF homh]

lemma (in ring_hom_cring**)** hom_zero [simp]: "h **0** = **0**$_S$"
⟨*proof*⟩

lemma (in ring_hom_cring**)** hom_a_inv [simp]:
"x ∈ carrier R ⟹ h (⊖ x) = ⊖$_S$ h x"
⟨*proof*⟩

lemma (in ring_hom_cring**)** hom_finsum [simp]:
assumes "f: A → carrier R"
shows "h (⨁ i ∈ A. f i) = (⨁$_S$ i ∈ A. (h o f) i)"
⟨*proof*⟩

lemma (in ring_hom_cring**)** hom_finprod:
assumes "f: A → carrier R"
shows "h (⨂ i ∈ A. f i) = (⨂$_S$ i ∈ A. (h o f) i)"
⟨*proof*⟩

declare ring_hom_cring.hom_finprod [simp]

lemma id_ring_hom [simp]: "id ∈ ring_hom R R"
⟨*proof*⟩

lemma ring_hom_trans:

"⟦ f ∈ ring_hom R S; g ∈ ring_hom S T ⟧ ⟹ g ∘ f ∈ ring_hom R T"
⟨*proof*⟩

11.8 Jeremy Avigad's More_Finite_Product material

lemma (in cring**)** sum_zero_eq_neg: "x ∈ carrier R ⟹ y ∈ carrier R ⟹ x ⊕ y = **0** ⟹ x = ⊖ y"
⟨*proof*⟩

lemma (in domain**)** square_eq_one:
fixes x
assumes [simp]: "x ∈ carrier R"
and "x ⊗ x = **1**"
shows "x = **1** ∨ x = ⊖**1**"
⟨*proof*⟩

lemma (in domain**)** inv_eq_self: "x ∈ Units R ⟹ x = inv x ⟹ x = **1** ∨ x = ⊖**1**"
⟨*proof*⟩

The following translates theorems about groups to the facts about the units of a ring. (The list should be expanded as more things are needed.)

lemma (in ring**)** finite_ring_finite_units [intro]: "finite (carrier R) ⟹ finite (Units R)"
⟨*proof*⟩

lemma (in monoid**)** units_of_pow:
fixes n :: nat
shows "x ∈ Units G ⟹ x [^]$_{\text{units_of G}}$ n = x [^]$_{\text{G}}$ n"
⟨*proof*⟩

lemma (in cring**)** units_power_order_eq_one:
"finite (Units R) ⟹ a ∈ Units R ⟹ a [^] card(Units R) = **1**"
⟨*proof*⟩

11.9 Jeremy Avigad's More_Ring material

lemma (in cring**)** field_intro2:
assumes "$\mathbf{0}_{\text{R}}$ ≠ $\mathbf{1}_{\text{R}}$" **and** un: "⋀x. x ∈ carrier R - {$\mathbf{0}_{\text{R}}$} ⟹ x ∈ Units R"
shows "field R"
⟨*proof*⟩

lemma (in monoid**)** inv_char:
assumes "x ∈ carrier G" "y ∈ carrier G" "x ⊗ y = **1**" "y ⊗ x = **1**"
shows "inv x = y"
⟨*proof*⟩

lemma (in comm_monoid**)** comm_inv_char: "x ∈ carrier G ⟹ y ∈ carrier G ⟹ x ⊗ y = **1** ⟹ inv x = y"

⟨*proof*⟩

lemma (in ring**)** inv_neg_one [simp]: "inv (⊖ **1**) = ⊖ **1**"
⟨*proof*⟩

lemma (in monoid**)** inv_eq_imp_eq [dest!]: "inv x = inv y ⟹ x ∈ Units G ⟹ y ∈ Units G ⟹ x = y"
⟨*proof*⟩

lemma (in ring**)** Units_minus_one_closed [intro]: "⊖ **1** ∈ Units R"
⟨*proof*⟩

lemma (in ring**)** inv_eq_neg_one_eq: "x ∈ Units R ⟹ inv x = ⊖ **1** ⟷ x = ⊖ **1**"
⟨*proof*⟩

lemma (in monoid**)** inv_eq_one_eq: "x ∈ Units G ⟹ inv x = **1** ⟷ x = **1**"
⟨*proof*⟩

end

theory Module
imports Ring
begin

12 Modules over an Abelian Group

12.1 Definitions

record ('a, 'b) module = "'b ring" +
  smult :: "['a, 'b] => 'b" (**infixl** ‹⊙› 70)

locale module = R?: cring + M?: abelian_group M **for** M (**structure**) +
  assumes smult_closed [simp, intro]:
      "[| a ∈ carrier R; x ∈ carrier M |] ==> a ⊙$_M$ x ∈ carrier M"
    and smult_l_distr:
      "[| a ∈ carrier R; b ∈ carrier R; x ∈ carrier M |] ==>
      (a ⊕ b) ⊙$_M$ x = a ⊙$_M$ x ⊕$_M$ b ⊙$_M$ x"
    and smult_r_distr:
      "[| a ∈ carrier R; x ∈ carrier M; y ∈ carrier M |] ==>
      a ⊙$_M$ (x ⊕$_M$ y) = a ⊙$_M$ x ⊕$_M$ a ⊙$_M$ y"
    and smult_assoc1:
      "[| a ∈ carrier R; b ∈ carrier R; x ∈ carrier M |] ==>
      (a ⊗ b) ⊙$_M$ x = a ⊙$_M$ (b ⊙$_M$ x)"
    and smult_one [simp]:
      "x ∈ carrier M ==> **1** ⊙$_M$ x = x"

locale algebra = module + cring M +
  assumes smult_assoc2:
      "[| a ∈ carrier R; x ∈ carrier M; y ∈ carrier M |] ==>
      (a ⊙$_M$ x) ⊗$_M$ y = a ⊙$_M$ (x ⊗$_M$ y)"

lemma moduleI:
  fixes R (**structure**) **and** M (**structure**)
  assumes cring: "cring R"
    and abelian_group: "abelian_group M"
    and smult_closed:
      "!!a x. [| a ∈ carrier R; x ∈ carrier M |] ==> a ⊙$_M$ x ∈ carrier
M"
    and smult_l_distr:
      "!!a b x. [| a ∈ carrier R; b ∈ carrier R; x ∈ carrier M |] ==>
      (a ⊕ b) ⊙$_M$ x = (a ⊙$_M$ x) ⊕$_M$ (b ⊙$_M$ x)"
    and smult_r_distr:
      "!!a x y. [| a ∈ carrier R; x ∈ carrier M; y ∈ carrier M |] ==>
      a ⊙$_M$ (x ⊕$_M$ y) = (a ⊙$_M$ x) ⊕$_M$ (a ⊙$_M$ y)"
    and smult_assoc1:
      "!!a b x. [| a ∈ carrier R; b ∈ carrier R; x ∈ carrier M |] ==>
      (a ⊗ b) ⊙$_M$ x = a ⊙$_M$ (b ⊙$_M$ x)"
    and smult_one:
      "!!x. x ∈ carrier M ==> **1** ⊙$_M$ x = x"
  shows "module R M"
  ⟨*proof*⟩

lemma algebraI:
  fixes R (**structure**) **and** M (**structure**)
  assumes R_cring: "cring R"
    and M_cring: "cring M"
    and smult_closed:
      "!!a x. [| a ∈ carrier R; x ∈ carrier M |] ==> a ⊙$_M$ x ∈ carrier
M"
    and smult_l_distr:
      "!!a b x. [| a ∈ carrier R; b ∈ carrier R; x ∈ carrier M |] ==>
      (a ⊕ b) ⊙$_M$ x = (a ⊙$_M$ x) ⊕$_M$ (b ⊙$_M$ x)"
    and smult_r_distr:
      "!!a x y. [| a ∈ carrier R; x ∈ carrier M; y ∈ carrier M |] ==>
      a ⊙$_M$ (x ⊕$_M$ y) = (a ⊙$_M$ x) ⊕$_M$ (a ⊙$_M$ y)"
    and smult_assoc1:
      "!!a b x. [| a ∈ carrier R; b ∈ carrier R; x ∈ carrier M |] ==>
      (a ⊗ b) ⊙$_M$ x = a ⊙$_M$ (b ⊙$_M$ x)"
    and smult_one:
      "!!x. x ∈ carrier M ==> (one R) ⊙$_M$ x = x"
    and smult_assoc2:
      "!!a x y. [| a ∈ carrier R; x ∈ carrier M; y ∈ carrier M |] ==>
      (a ⊙$_M$ x) ⊗$_M$ y = a ⊙$_M$ (x ⊗$_M$ y)"
  shows "algebra R M"
  ⟨*proof*⟩

lemma (in algebra) R_cring: "cring R" ⟨*proof*⟩

lemma (in algebra) M_cring: "cring M" ⟨*proof*⟩

lemma (in algebra) module: "module R M"
⟨*proof*⟩

12.2 Basic Properties of Modules

lemma (in module) smult_l_null [simp]:
"x ∈ carrier M ==> **0** ⊙$_M$ x = **0**$_M$"
⟨*proof*⟩

lemma (in module) smult_r_null [simp]:
"a ∈ carrier R ==> a ⊙$_M$ **0**$_M$ = **0**$_M$"
⟨*proof*⟩

lemma (in module) smult_l_minus:
"⟦ a ∈ carrier R; x ∈ carrier M ⟧ ⟹ (⊖a) ⊙$_M$ x = ⊖$_M$ (a ⊙$_M$ x)"
⟨*proof*⟩

lemma (in module) smult_r_minus:
"[| a ∈ carrier R; x ∈ carrier M |] ==> a ⊙$_M$ (⊖$_M$x) = ⊖$_M$ (a ⊙$_M$ x)"
⟨*proof*⟩

lemma (in module) finsum_smult_ldistr:
"⟦ finite A; a ∈ carrier R; f: A → carrier M ⟧ ⟹
a ⊙$_M$ (⨁$_M$ i ∈ A. (f i)) = (⨁$_M$ i ∈ A. (a ⊙$_M$ (f i)))"
⟨*proof*⟩

12.3 Submodules

locale submodule = subgroup H "add_monoid M" **for** H **and** R :: "('a, 'b) ring_scheme" **and** M (**structure**)
+ **assumes** smult_closed [simp, intro]:
"⟦a ∈ carrier R; x ∈ H⟧ ⟹ a ⊙$_M$ x ∈ H"

lemma (in module) submoduleI :
assumes subset: "H ⊆ carrier M"
and zero: "**0**$_M$ ∈ H"
and a_inv: "!!a. a ∈ H ⟹ ⊖$_M$ a ∈ H"
and add : "⋀ a b. ⟦a ∈ H ; b ∈ H⟧ ⟹ a ⊕$_M$ b ∈ H"
and smult_closed : "⋀ a x. ⟦a ∈ carrier R; x ∈ H⟧ ⟹ a ⊙$_M$ x ∈ H"
shows "submodule H R M"
⟨*proof*⟩

lemma (in module) submoduleE :

assumes "submodule H R M"
shows "H ⊆ carrier M"
and "H ≠ {}"
and "⋀a. a ∈ H ⟹ ⊖$_M$ a ∈ H"
and "⋀a b. ⟦a ∈ carrier R; b ∈ H⟧ ⟹ a ⊙$_M$ b ∈ H"
and "⋀ a b. ⟦a ∈ H ; b ∈ H⟧ ⟹ a ⊕$_M$ b ∈ H"
and "⋀ x. x ∈ H ⟹ (a_inv M x) ∈ H"
⟨*proof*⟩

lemma (in module) carrier_is_submodule :
"submodule (carrier M) R M"
⟨*proof*⟩

lemma (in submodule) submodule_is_module :
assumes "module R M"
shows "module R (M⦇carrier := H⦈)"
⟨*proof*⟩

lemma (in module) module_incl_imp_submodule :
assumes "H ⊆ carrier M"
and "module R (M⦇carrier := H⦈)"
shows "submodule H R M"
⟨*proof*⟩

end

theory AbelCoset
imports Coset Ring
begin

12.4 More Lifting from Groups to Abelian Groups

12.4.1 Definitions

Hiding <+> from HOL.Sum_Type until I come up with better syntax here

no_notation Sum_Type.Plus (**infixr** <<+>> 65)

definition
a_r_coset :: "[_, 'a set, 'a] ⇒ 'a set" (**infixl** <+>$_r$> 60)
where "a_r_coset G = r_coset (add_monoid G)"

definition
a_l_coset :: "[_, 'a, 'a set] ⇒ 'a set" (**infixl** <<+$_r$> 60)
where "a_l_coset G = l_coset (add_monoid G)"

definition
A_RCOSETS :: "[_, 'a set] ⇒ ('a set)set"
(<(<open_block notation=<prefix a_rcosets>>a'_rcosets‸ _)> [81]
80)
where "A_RCOSETS G H = RCOSETS (add_monoid G) H"

definition
set_add :: "[_, 'a set ,'a set] ⇒ 'a set" (**infixl** <<+>‸> 60)
where "set_add G = set_mult (add_monoid G)"

definition
A_SET_INV :: "[_,'a set] ⇒ 'a set"
(<(<open_block notation=<prefix a_set_inv>>a'_set'_inv‸ _)> [81]
80)
where "A_SET_INV G H = SET_INV (add_monoid G) H"

definition
a_r_congruent :: "[('a,'b)ring_scheme, 'a set] ⇒ ('a*'a)set" (<racong‸>)
where "a_r_congruent G = r_congruent (add_monoid G)"

definition
A_FactGroup :: "[('a,'b) ring_scheme, 'a set] ⇒ ('a set) monoid" (**infixl**
<A'_Mod> 65)
— Actually defined for groups rather than monoids
where "A_FactGroup G H = FactGroup (add_monoid G) H"

definition
a_kernel :: "('a, 'm) ring_scheme ⇒ ('b, 'n) ring_scheme ⇒ ('a ⇒
'b) ⇒ 'a set"
— the kernel of a homomorphism (additive)
where "a_kernel G H h = kernel (add_monoid G) (add_monoid H) h"

locale abelian_group_hom = G?: abelian_group G + H?: abelian_group H
for G (**structure**) **and** H (**structure**) +
fixes h
assumes a_group_hom: "group_hom (add_monoid G) (add_monoid H) h"

lemmas a_r_coset_defs =
a_r_coset_def r_coset_def

lemma a_r_coset_def':
fixes G (**structure**)
shows "H +> a ≡ ⋃h∈H. {h ⊕ a}"
⟨*proof*⟩

lemmas a_l_coset_defs =
a_l_coset_def l_coset_def

lemma a_l_coset_def':

fixes G (**structure**)
shows "a <+ H ≡ ⋃h∈H. {a ⊕ h}"
⟨*proof*⟩

lemmas A_RCOSETS_defs =
A_RCOSETS_def RCOSETS_def

lemma A_RCOSETS_def':
fixes G (**structure**)
shows "a_rcosets H ≡ ⋃a∈carrier G. {H +> a}"
⟨*proof*⟩

lemmas set_add_defs =
set_add_def set_mult_def

lemma set_add_def':
fixes G (**structure**)
shows "H <+> K ≡ ⋃h∈H. ⋃k∈K. {h ⊕ k}"
⟨*proof*⟩

lemmas A_SET_INV_defs =
A_SET_INV_def SET_INV_def

lemma A_SET_INV_def':
fixes G (**structure**)
shows "a_set_inv H ≡ ⋃h∈H. {⊖ h}"
⟨*proof*⟩

12.4.2 Cosets

sublocale abelian_group <
add: group "(add_monoid G)"
rewrites "carrier (add_monoid G) = carrier G"
and " mult (add_monoid G) = add G"
and " one (add_monoid G) = zero G"
and " m_inv (add_monoid G) = a_inv G"
and "finprod (add_monoid G) = finsum G"
and "r_coset (add_monoid G) = a_r_coset G"
and "l_coset (add_monoid G) = a_l_coset G"
and "(λa k. pow (add_monoid G) a k) = (λa k. add_pow G k a)"
⟨*proof*⟩

context abelian_group
begin

thm add.coset_mult_assoc
lemmas a_repr_independence' = add.repr_independence

end

lemma (**in** abelian_group) a_coset_add_assoc:
"[| M ⊆ carrier G; g ∈ carrier G; h ∈ carrier G |]
==> (M +> g) +> h = M +> (g ⊕ h)"
⟨*proof*⟩

thm abelian_group.a_coset_add_assoc

lemma (**in** abelian_group) a_coset_add_zero [simp]:
"M ⊆ carrier G ==> M +> **0** = M"
⟨*proof*⟩

lemma (**in** abelian_group) a_coset_add_inv1:
"[| M +> (x ⊕ (⊖ y)) = M; x ∈ carrier G ; y ∈ carrier G;
M ⊆ carrier G |] ==> M +> x = M +> y"
⟨*proof*⟩

lemma (**in** abelian_group) a_coset_add_inv2:
"[| M +> x = M +> y; x ∈ carrier G; y ∈ carrier G; M ⊆ carrier
G |]
==> M +> (x ⊕ (⊖ y)) = M"
⟨*proof*⟩

lemma (**in** abelian_group) a_coset_join1:
"[| H +> x = H; x ∈ carrier G; subgroup H (add_monoid G) |] ==>
x ∈ H"
⟨*proof*⟩

lemma (**in** abelian_group) a_solve_equation:
"⟦subgroup H (add_monoid G); x ∈ H; y ∈ H⟧ ⟹ ∃h∈H. y = h ⊕ x"
⟨*proof*⟩

lemma (**in** abelian_group) a_repr_independence:
"⟦ y ∈ H +> x; x ∈ carrier G; subgroup H (add_monoid G) ⟧ ⟹
H +> x = H +> y"
⟨*proof*⟩

lemma (**in** abelian_group) a_coset_join2:
"⟦x ∈ carrier G; subgroup H (add_monoid G); x∈H⟧ ⟹ H +> x = H"
⟨*proof*⟩

lemma (**in** abelian_monoid) a_r_coset_subset_G:
"[| H ⊆ carrier G; x ∈ carrier G |] ==> H +> x ⊆ carrier G"
⟨*proof*⟩

lemma (**in** abelian_group) a_rcosI:
"[| h ∈ H; H ⊆ carrier G; x ∈ carrier G|] ==> h ⊕ x ∈ H +> x"

⟨*proof*⟩

lemma (in abelian_group**)** a_rcosetsI:
"⟦H ⊆ carrier G; x ∈ carrier G⟧ ⟹ H +> x ∈ a_rcosets H"
⟨*proof*⟩

Really needed?

lemma (in abelian_group**)** a_transpose_inv:
"[| x ⊕ y = z; x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |]
==> (⊖ x) ⊕ z = y"
⟨*proof*⟩

12.4.3 Subgroups

locale additive_subgroup =
fixes H **and** G **(structure)**
assumes a_subgroup: "subgroup H (add_monoid G)"

lemma (in additive_subgroup**)** is_additive_subgroup:
shows "additive_subgroup H G"
⟨*proof*⟩

lemma additive_subgroupI:
fixes G **(structure)**
assumes a_subgroup: "subgroup H (add_monoid G)"
shows "additive_subgroup H G"
⟨*proof*⟩

lemma (in additive_subgroup**)** a_subset:
"H ⊆ carrier G"
⟨*proof*⟩

lemma (in additive_subgroup**)** a_closed [intro, simp]:
"⟦x ∈ H; y ∈ H⟧ ⟹ x ⊕ y ∈ H"
⟨*proof*⟩

lemma (in additive_subgroup**)** zero_closed [simp]:
"**0** ∈ H"
⟨*proof*⟩

lemma (in additive_subgroup**)** a_inv_closed [intro,simp]:
"x ∈ H ⟹ ⊖ x ∈ H"
⟨*proof*⟩

12.4.4 Additive subgroups are normal

Every subgroup of an abelian_group is normal

locale abelian_subgroup = additive_subgroup + abelian_group G +
assumes a_normal: "normal H (add_monoid G)"

lemma (**in** abelian_subgroup) is_abelian_subgroup:
  shows "abelian_subgroup H G"
⟨*proof*⟩

lemma abelian_subgroupI:
  assumes a_normal: "normal H (add_monoid G)"
      and a_comm: "!!x y. [| x ∈ carrier G; y ∈ carrier G |] ==> x ⊕$_G$ y = y ⊕$_G$ x"
  shows "abelian_subgroup H G"
⟨*proof*⟩

lemma abelian_subgroupI2:
  fixes G (**structure**)
  assumes a_comm_group: "comm_group (add_monoid G)"
      and a_subgroup: "subgroup H (add_monoid G)"
  shows "abelian_subgroup H G"
⟨*proof*⟩

lemma abelian_subgroupI3:
  fixes G (**structure**)
  assumes "additive_subgroup H G"
    and "abelian_group G"
  shows "abelian_subgroup H G"
  ⟨*proof*⟩

lemma (**in** abelian_subgroup) a_coset_eq:
     "(∀x ∈ carrier G. H +> x = x <+ H)"
⟨*proof*⟩

lemma (**in** abelian_subgroup) a_inv_op_closed1:
  shows "⟦x ∈ carrier G; h ∈ H⟧ ⟹ (⊖ x) ⊕ h ⊕ x ∈ H"
⟨*proof*⟩

lemma (**in** abelian_subgroup) a_inv_op_closed2:
  shows "⟦x ∈ carrier G; h ∈ H⟧ ⟹ x ⊕ h ⊕ (⊖ x) ∈ H"
⟨*proof*⟩

lemma (**in** abelian_group) a_lcos_m_assoc:
  "⟦ M ⊆ carrier G; g ∈ carrier G; h ∈ carrier G ⟧ ⟹ g <+ (h <+ M) = (g ⊕ h) <+ M"
⟨*proof*⟩

lemma (**in** abelian_group) a_lcos_mult_one:
     "M ⊆ carrier G ==> **0** <+ M = M"
⟨*proof*⟩

lemma (**in** abelian_group) a_l_coset_subset_G:
  "⟦ H ⊆ carrier G; x ∈ carrier G ⟧ ⟹ x <+ H ⊆ carrier G"

⟨*proof*⟩

lemma (**in** abelian_group) a_l_coset_swap:
"⟦y ∈ x <+ H; x ∈ carrier G; subgroup H (add_monoid G)⟧ ⟹ x ∈ y <+ H"
⟨*proof*⟩

lemma (**in** abelian_group) a_l_coset_carrier:
"[| y ∈ x <+ H; x ∈ carrier G; subgroup H (add_monoid G) |] ==> y ∈ carrier G"
⟨*proof*⟩

lemma (**in** abelian_group) a_l_repr_imp_subset:
assumes "y ∈ x <+ H" "x ∈ carrier G" "subgroup H (add_monoid G)"
shows "y <+ H ⊆ x <+ H"
⟨*proof*⟩

lemma (**in** abelian_group) a_l_repr_independence:
assumes y: "y ∈ x <+ H" **and** x: "x ∈ carrier G" **and** sb: "subgroup H (add_monoid G)"
shows "x <+ H = y <+ H"
⟨*proof*⟩

lemma (**in** abelian_group) setadd_subset_G:
"⟦H ⊆ carrier G; K ⊆ carrier G⟧ ⟹ H <+> K ⊆ carrier G"
⟨*proof*⟩

lemma (**in** abelian_group) subgroup_add_id: "subgroup H (add_monoid G) ⟹ H <+> H = H"
⟨*proof*⟩

lemma (**in** abelian_subgroup) a_rcos_inv:
assumes x: "x ∈ carrier G"
shows "a_set_inv (H +> x) = H +> (⊖ x)"
⟨*proof*⟩

lemma (**in** abelian_group) a_setmult_rcos_assoc:
"⟦H ⊆ carrier G; K ⊆ carrier G; x ∈ carrier G⟧
⟹ H <+> (K +> x) = (H <+> K) +> x"
⟨*proof*⟩

lemma (**in** abelian_group) a_rcos_assoc_lcos:
"⟦H ⊆ carrier G; K ⊆ carrier G; x ∈ carrier G⟧
⟹ (H +> x) <+> K = H <+> (x <+ K)"
⟨*proof*⟩

lemma (**in** abelian_subgroup) a_rcos_sum:
"⟦x ∈ carrier G; y ∈ carrier G⟧
⟹ (H +> x) <+> (H +> y) = H +> (x ⊕ y)"

⟨*proof*⟩

lemma (in abelian_subgroup**)** rcosets_add_eq:
"M ∈ a_rcosets H ⟹ H <+> M = M"
— generalizes subgroup_mult_id
⟨*proof*⟩

12.4.5 Congruence Relation

lemma (in abelian_subgroup**)** a_equiv_rcong:
shows "equiv (carrier G) (racong H)"
⟨*proof*⟩

lemma (in abelian_subgroup**)** a_l_coset_eq_rcong:
assumes a: "a ∈ carrier G"
shows "a <+ H = racong H `` {a}"
⟨*proof*⟩

lemma (in abelian_subgroup**)** a_rcos_equation:
shows
"⟦ha ⊕ a = h ⊕ b; a ∈ carrier G; b ∈ carrier G;
h ∈ H; ha ∈ H; hb ∈ H⟧
⟹ hb ⊕ a ∈ (⋃h∈H. {h ⊕ b})"
⟨*proof*⟩

lemma (in abelian_subgroup**)** a_rcos_disjoint: "pairwise disjnt (a_rcosets H)"
⟨*proof*⟩

lemma (in abelian_subgroup**)** a_rcos_self:
shows "x ∈ carrier G ⟹ x ∈ H +> x"
⟨*proof*⟩

lemma (in abelian_subgroup**)** a_rcosets_part_G:
shows "⋃(a_rcosets H) = carrier G"
⟨*proof*⟩

lemma (in abelian_subgroup**)** a_cosets_finite:
"⟦c ∈ a_rcosets H; H ⊆ carrier G; finite (carrier G)⟧ ⟹ finite c"
⟨*proof*⟩

lemma (in abelian_group**)** a_card_cosets_equal:
"⟦c ∈ a_rcosets H; H ⊆ carrier G; finite(carrier G)⟧
⟹ card c = card H"
⟨*proof*⟩

lemma (in abelian_group**)** rcosets_subset_PowG:
"additive_subgroup H G ⟹ a_rcosets H ⊆ Pow(carrier G)"

⟨*proof*⟩

theorem (**in** abelian_group) a_lagrange:
"⟦finite(carrier G); additive_subgroup H G⟧
⟹ card(a_rcosets H) * card(H) = order(G)"
⟨*proof*⟩

12.4.6 Factorization

lemmas A_FactGroup_defs = A_FactGroup_def FactGroup_def

lemma A_FactGroup_def':
fixes G (**structure**)
shows "G A_Mod H ≡ ⦇carrier = a_rcosets$_G$ H, mult = set_add G, one = H⦈"
⟨*proof*⟩

lemma (**in** abelian_subgroup) a_setmult_closed:
"⟦K1 ∈ a_rcosets H; K2 ∈ a_rcosets H⟧ ⟹ K1 <+> K2 ∈ a_rcosets H"
⟨*proof*⟩

lemma (**in** abelian_subgroup) a_setinv_closed:
"K ∈ a_rcosets H ⟹ a_set_inv K ∈ a_rcosets H"
⟨*proof*⟩

lemma (**in** abelian_subgroup) a_rcosets_assoc:
"⟦M1 ∈ a_rcosets H; M2 ∈ a_rcosets H; M3 ∈ a_rcosets H⟧
⟹ M1 <+> M2 <+> M3 = M1 <+> (M2 <+> M3)"
⟨*proof*⟩

lemma (**in** abelian_subgroup) a_subgroup_in_rcosets:
"H ∈ a_rcosets H"
⟨*proof*⟩

lemma (**in** abelian_subgroup) a_rcosets_inv_mult_group_eq:
"M ∈ a_rcosets H ⟹ a_set_inv M <+> M = H"
⟨*proof*⟩

theorem (**in** abelian_subgroup) a_factorgroup_is_group:
"group (G A_Mod H)"
⟨*proof*⟩

Since the Factorization is based on an *abelian* subgroup, is results in a commutative group

theorem (**in** abelian_subgroup) a_factorgroup_is_comm_group: "comm_group (G A_Mod H)"
⟨*proof*⟩

lemma add_A_FactGroup [simp]: "X $\otimes_{(G\ A_Mod\ H)}$ X' = X <+>$_G$ X'"
⟨*proof*⟩

lemma (**in** abelian_subgroup) a_inv_FactGroup:
"X ∈ carrier (G A_Mod H) ⟹ $inv_{G\ A_Mod\ H}$ X = a_set_inv X"
⟨*proof*⟩

The coset map is a homomorphism from G to the quotient group G Mod H

lemma (**in** abelian_subgroup) a_r_coset_hom_A_Mod:
"(λa. H +> a) ∈ hom (add_monoid G) (G A_Mod H)"
⟨*proof*⟩

The isomorphism theorems have been omitted from lifting, at least for now

12.4.7 The First Isomorphism Theorem

The quotient by the kernel of a homomorphism is isomorphic to the range of that homomorphism.

lemmas a_kernel_defs =
a_kernel_def kernel_def

lemma a_kernel_def':
"a_kernel R S h = {x ∈ carrier R. h x = $\mathbf{0}_S$}"
⟨*proof*⟩

12.4.8 Homomorphisms

lemma abelian_group_homI:
assumes "abelian_group G"
assumes "abelian_group H"
assumes a_group_hom: "group_hom (add_monoid G)
(add_monoid H) h"
shows "abelian_group_hom G H h"
⟨*proof*⟩

lemma (**in** abelian_group_hom) is_abelian_group_hom:
"abelian_group_hom G H h"
⟨*proof*⟩

lemma (**in** abelian_group_hom) hom_add [simp]:
"[| x ∈ carrier G; y ∈ carrier G |]
==> h (x $\oplus_G$ y) = h x $\oplus_H$ h y"
⟨*proof*⟩

lemma (**in** abelian_group_hom) hom_closed [simp]:
"x ∈ carrier G ⟹ h x ∈ carrier H"
⟨*proof*⟩

lemma (in abelian_group_hom**)** zero_closed [simp]:
"h **0** ∈ carrier H"
⟨*proof*⟩

lemma (in abelian_group_hom**)** hom_zero [simp]:
"h **0** = **0**$_H$"
⟨*proof*⟩

lemma (in abelian_group_hom**)** a_inv_closed [simp]:
"x ∈ carrier G ==> h (⊖x) ∈ carrier H"
⟨*proof*⟩

lemma (in abelian_group_hom**)** hom_a_inv [simp]:
"x ∈ carrier G ==> h (⊖x) = ⊖$_H$ (h x)"
⟨*proof*⟩

lemma (in abelian_group_hom**)** additive_subgroup_a_kernel:
"additive_subgroup (a_kernel G H h) G"
⟨*proof*⟩

The kernel of a homomorphism is an abelian subgroup

lemma (in abelian_group_hom**)** abelian_subgroup_a_kernel:
"abelian_subgroup (a_kernel G H h) G"
⟨*proof*⟩

lemma (in abelian_group_hom**)** A_FactGroup_nonempty:
assumes X: "X ∈ carrier (G A_Mod a_kernel G H h)"
shows "X ≠ {}"
⟨*proof*⟩

lemma (in abelian_group_hom**)** FactGroup_the_elem_mem:
assumes X: "X ∈ carrier (G A_Mod (a_kernel G H h))"
shows "the_elem (h‘X) ∈ carrier H"
⟨*proof*⟩

lemma (in abelian_group_hom**)** A_FactGroup_hom:
"(λX. the_elem (h‘X)) ∈ hom (G A_Mod (a_kernel G H h))
(add_monoid H)"
⟨*proof*⟩

lemma (in abelian_group_hom**)** A_FactGroup_inj_on:
"inj_on (λX. the_elem (h ‘ X)) (carrier (G A_Mod a_kernel G H h))"
⟨*proof*⟩

If the homomorphism h is onto H, then so is the homomorphism from the quotient group

lemma (in abelian_group_hom**)** A_FactGroup_onto:
assumes h: "h ‘ carrier G = carrier H"

shows "(λX. the_elem (h ‘ X)) ‘ carrier (G A_Mod a_kernel G H h) = carrier H"
⟨*proof*⟩

If h is a homomorphism from G onto H, then the quotient group G Mod Coset.kernel G H h is isomorphic to H.

theorem (in abelian_group_hom) A_FactGroup_iso_set:
shows "h ‘ carrier G = carrier H
⟹ (λX. the_elem (h‘X)) ∈ iso (G A_Mod (a_kernel G H h)) (add_monoid H)"
⟨*proof*⟩

corollary (in abelian_group_hom) A_FactGroup_iso :
"h ‘ carrier G = carrier H
⟹ (G A_Mod (a_kernel G H h)) ≅ (add_monoid H)"
⟨*proof*⟩

12.4.9 Cosets

Not eveything from CosetExt.thy is lifted here.

lemma (in additive_subgroup) a_Hcarr [simp]:
assumes hH: "h ∈ H"
shows "h ∈ carrier G"
⟨*proof*⟩

lemma (in abelian_subgroup) a_elemrcos_carrier:
assumes acarr: "a ∈ carrier G"
and a': "a' ∈ H +> a"
shows "a' ∈ carrier G"
⟨*proof*⟩

lemma (in abelian_subgroup) a_rcos_const:
assumes hH: "h ∈ H"
shows "H +> h = H"
⟨*proof*⟩

lemma (in abelian_subgroup) a_rcos_module_imp:
assumes xcarr: "x ∈ carrier G"
and x'cos: "x' ∈ H +> x"
shows "(x' ⊕ ⊖x) ∈ H"
⟨*proof*⟩

lemma (in abelian_subgroup) a_rcos_module_rev:
assumes "x ∈ carrier G" "x' ∈ carrier G"
and "(x' ⊕ ⊖x) ∈ H"
shows "x' ∈ H +> x"
⟨*proof*⟩

lemma (in abelian_subgroup**)** a_rcos_module:
assumes "x ∈ carrier G" "x' ∈ carrier G"
shows "(x' ∈ H +> x) = (x' ⊕ ⊖x ∈ H)"
⟨*proof*⟩
lemma (in abelian_subgroup**)** a_rcos_module_minus:
assumes "ring G"
assumes carr: "x ∈ carrier G" "x' ∈ carrier G"
shows "(x' ∈ H +> x) = (x' ⊖ x ∈ H)"
⟨*proof*⟩

lemma (in abelian_subgroup**)** a_repr_independence':
assumes "y ∈ H +> x" "x ∈ carrier G"
shows "H +> x = H +> y"
⟨*proof*⟩

lemma (in abelian_subgroup**)** a_repr_independenceD:
assumes "y ∈ carrier G" "H +> x = H +> y"
shows "y ∈ H +> x"
⟨*proof*⟩

lemma (in abelian_subgroup**)** a_rcosets_carrier:
"X ∈ a_rcosets H ⟹ X ⊆ carrier G"
⟨*proof*⟩

12.4.10 Addition of Subgroups

lemma (in abelian_monoid**)** set_add_closed:
assumes "A ⊆ carrier G" "B ⊆ carrier G"
shows "A <+> B ⊆ carrier G"
⟨*proof*⟩

lemma (in abelian_group**)** add_additive_subgroups:
assumes subH: "additive_subgroup H G"
and subK: "additive_subgroup K G"
shows "additive_subgroup (H <+> K) G"
⟨*proof*⟩

end

theory Ideal
imports Ring AbelCoset
begin

13 Ideals

13.1 Definitions

13.1.1 General definition

locale ideal = additive_subgroup I R + ring R **for** I **and** R **(structure)** +
  assumes I_l_closed: "⟦a ∈ I; x ∈ carrier R⟧ ⟹ x ⊗ a ∈ I"
    and I_r_closed: "⟦a ∈ I; x ∈ carrier R⟧ ⟹ a ⊗ x ∈ I"

sublocale ideal ⊆ abelian_subgroup I R
⟨*proof*⟩

lemma (in ideal**)** is_ideal: "ideal I R"
  ⟨*proof*⟩

lemma idealI:
  fixes R **(structure)**
  assumes "ring R"
  assumes a_subgroup: "subgroup I (add_monoid R)"
    and I_l_closed: "⋀a x. ⟦a ∈ I; x ∈ carrier R⟧ ⟹ x ⊗ a ∈ I"
    and I_r_closed: "⋀a x. ⟦a ∈ I; x ∈ carrier R⟧ ⟹ a ⊗ x ∈ I"
  shows "ideal I R"
⟨*proof*⟩

13.1.2 Ideals Generated by a Subset of carrier R

definition genideal :: "_ ⇒ 'a set ⇒ 'a set"
    (‹(‹open_block notation=‹prefix Idl››Idl⇘ _)›› [80] 79)
  where "Idl$_R$ S = ⋂{I. ideal I R ∧ S ⊆ I}"

13.1.3 Principal Ideals

locale principalideal = ideal +
  assumes generate: "∃i ∈ carrier R. I = Idl {i}"

lemma (in principalideal**)** is_principalideal: "principalideal I R"
  ⟨*proof*⟩

lemma principalidealI:
  fixes R **(structure)**
  assumes "ideal I R"
    and generate: "∃i ∈ carrier R. I = Idl {i}"
  shows "principalideal I R"
⟨*proof*⟩

lemma (in ideal**)** rcos_const_imp_mem:
  assumes "i ∈ carrier R" **and** "I +> i = I" **shows** "i ∈ I"
  ⟨*proof*⟩

lemma (in ring**)** a_rcos_zero:
assumes "ideal I R" "i ∈ I" **shows** "I +> i = I"
⟨*proof*⟩

lemma (in ring**)** ideal_is_normal:
assumes "ideal I R" **shows** "I ◁ (add_monoid R)"
⟨*proof*⟩

lemma (in ideal**)** a_rcos_sum:
assumes "a ∈ carrier R" **and** "b ∈ carrier R" **shows** "(I +> a) <+> (I +> b) = I +> (a ⊕ b)"
⟨*proof*⟩

lemma (in ring**)** set_add_comm:
assumes "I ⊆ carrier R" "J ⊆ carrier R" **shows** "I <+> J = J <+> I"
⟨*proof*⟩

13.1.4 Maximal Ideals

locale maximalideal = ideal +
assumes I_notcarr: "carrier R ≠ I"
and I_maximal: "⟦ideal J R; I ⊆ J; J ⊆ carrier R⟧ ⟹ (J = I) ∨ (J = carrier R)"

lemma (in maximalideal**)** is_maximalideal: "maximalideal I R"
⟨*proof*⟩

lemma maximalidealI:
fixes R
assumes "ideal I R"
and I_notcarr: "carrier R ≠ I"
and I_maximal: "⋀J. ⟦ideal J R; I ⊆ J; J ⊆ carrier R⟧ ⟹ (J = I) ∨ (J = carrier R)"
shows "maximalideal I R"
⟨*proof*⟩

13.1.5 Prime Ideals

locale primeideal = ideal + cring +
assumes I_notcarr: "carrier R ≠ I"

and I_prime: "⟦a ∈ carrier R; b ∈ carrier R; a ⊗ b ∈ I⟧ ⟹ a ∈ I ∨ b ∈ I"

lemma (in primeideal**)** primeideal: "primeideal I R"
⟨*proof*⟩

lemma primeidealI:
fixes R **(structure)**
assumes "ideal I R"
and "cring R"
and I_notcarr: "carrier R ≠ I"
and I_prime: "⋀a b. ⟦a ∈ carrier R; b ∈ carrier R; a ⊗ b ∈ I⟧ ⟹ a ∈ I ∨ b ∈ I"
shows "primeideal I R"
⟨*proof*⟩

lemma primeidealI2:
fixes R **(structure)**
assumes "additive_subgroup I R"
and "cring R"
and I_l_closed: "⋀a x. ⟦a ∈ I; x ∈ carrier R⟧ ⟹ x ⊗ a ∈ I"
and I_r_closed: "⋀a x. ⟦a ∈ I; x ∈ carrier R⟧ ⟹ a ⊗ x ∈ I"
and I_notcarr: "carrier R ≠ I"
and I_prime: "⋀a b. ⟦a ∈ carrier R; b ∈ carrier R; a ⊗ b ∈ I⟧ ⟹ a ∈ I ∨ b ∈ I"
shows "primeideal I R"
⟨*proof*⟩

13.2 Special Ideals

lemma (in ring**)** zeroideal: "ideal {**0**} R"
⟨*proof*⟩

lemma (in ring**)** oneideal: "ideal (carrier R) R"
⟨*proof*⟩

lemma (in "domain"**)** zeroprimeideal: "primeideal {**0**} R"
⟨*proof*⟩

13.3 General Ideal Properties

lemma (in ideal**)** one_imp_carrier:
assumes I_one_closed: "**1** ∈ I"
shows "I = carrier R"
⟨*proof*⟩

lemma (in ideal**)** Icarr:
assumes iI: "i ∈ I"
shows "i ∈ carrier R"
⟨*proof*⟩

lemma (in ring**)** quotient_eq_iff_same_a_r_cos:
  assumes "ideal I R" **and** "a ∈ carrier R" **and** "b ∈ carrier R"
  shows "a ⊖ b ∈ I ⟷ I +> a = I +> b"
⟨*proof*⟩

13.4 Intersection of Ideals

Intersection of two ideals The intersection of any two ideals is again an ideal in R

lemma (in ring**)** i_intersect:
  assumes "ideal I R"
  assumes "ideal J R"
  shows "ideal (I ∩ J) R"
⟨*proof*⟩

The intersection of any Number of Ideals is again an Ideal in R

lemma (in ring**)** i_Intersect:
  assumes Sideals: "⋀I. I ∈ S ⟹ ideal I R" **and** notempty: "S ≠ {}"
  shows "ideal (⋂S) R"
⟨*proof*⟩

13.5 Addition of Ideals

lemma (in ring**)** add_ideals:
  assumes idealI: "ideal I R" **and** idealJ: "ideal J R"
  shows "ideal (I <+> J) R"
⟨*proof*⟩

13.6 Ideals generated by a subset of carrier R

genideal generates an ideal

lemma (in ring**)** genideal_ideal:
  assumes Scarr: "S ⊆ carrier R"
  shows "ideal (Idl S) R"
⟨*proof*⟩

lemma (in ring**)** genideal_self:
  assumes "S ⊆ carrier R"
  shows "S ⊆ Idl S"
  ⟨*proof*⟩

lemma (in ring**)** genideal_self':
  assumes carr: "i ∈ carrier R"
  shows "i ∈ Idl {i}"
  ⟨*proof*⟩

genideal generates the minimal ideal

lemma (in ring**)** genideal_minimal:
  assumes "ideal I R" "S ⊆ I"
  shows "Idl S ⊆ I"
  ⟨*proof*⟩

Generated ideals and subsets

lemma (in ring**)** Idl_subset_ideal:
  assumes Iideal: "ideal I R"
    and Hcarr: "H ⊆ carrier R"
  shows "(Idl H ⊆ I) = (H ⊆ I)"
⟨*proof*⟩

lemma (in ring**)** subset_Idl_subset:
  assumes Icarr: "I ⊆ carrier R"
    and HI: "H ⊆ I"
  shows "Idl H ⊆ Idl I"
⟨*proof*⟩

lemma (in ring**)** Idl_subset_ideal':
  assumes acarr: "a ∈ carrier R" **and** bcarr: "b ∈ carrier R"
  shows "Idl {a} ⊆ Idl {b} ⟷ a ∈ Idl {b}"
⟨*proof*⟩

lemma (in ring**)** genideal_zero: "Idl {**0**} = {**0**}"
⟨*proof*⟩

lemma (in ring**)** genideal_one: "Idl {**1**} = carrier R"
⟨*proof*⟩

Generation of Principal Ideals in Commutative Rings

definition cgenideal :: "_ ⇒ 'a ⇒ 'a set"
    (‹(‹open_block notation=‹prefix PIdl››PIdl⇘ _)› [80] 79)
  where "PIdl$_R$ a = {x ⊗$_R$ a | x. x ∈ carrier R}"

genhideal (?) really generates an ideal

lemma (in cring**)** cgenideal_ideal:
  assumes acarr: "a ∈ carrier R"
  shows "ideal (PIdl a) R"
  ⟨*proof*⟩

lemma (in ring**)** cgenideal_self:
  assumes icarr: "i ∈ carrier R"
  shows "i ∈ PIdl i"
  ⟨*proof*⟩

cgenideal is minimal

lemma (in ring**)** cgenideal_minimal:
  assumes "ideal J R"

assumes aJ: "a ∈ J"
shows "PIdl a ⊆ J"
⟨*proof*⟩

lemma (**in** cring) cgenideal_eq_genideal:
assumes icarr: "i ∈ carrier R"
shows "PIdl i = Idl {i}"
⟨*proof*⟩

lemma (**in** cring) cgenideal_eq_rcos: "PIdl i = carrier R #> i"
⟨*proof*⟩

lemma (**in** cring) cgenideal_is_principalideal:
assumes "i ∈ carrier R"
shows "principalideal (PIdl i) R"
⟨*proof*⟩

13.7 Union of Ideals

lemma (**in** ring) union_genideal:
assumes idealI: "ideal I R" **and** idealJ: "ideal J R"
shows "Idl (I ∪ J) = I <+> J"
⟨*proof*⟩

13.8 Properties of Principal Ideals

The zero ideal is a principal ideal

corollary (**in** ring) zeropideal: "principalideal {**0**} R"
⟨*proof*⟩

The unit ideal is a principal ideal

corollary (**in** ring) onepideal: "principalideal (carrier R) R"
⟨*proof*⟩

Every principal ideal is a right coset of the carrier

lemma (**in** principalideal) rcos_generate:
assumes "cring R"
shows "∃x∈I. I = carrier R #> x"
⟨*proof*⟩

This next lemma would be trivial if placed in a theory that imports QuotRing, but it makes more sense to have it here (easier to find and coherent with the previous developments).

lemma (**in** cring) cgenideal_prod:
assumes "a ∈ carrier R" "b ∈ carrier R"
shows "(PIdl a) <#> (PIdl b) = PIdl (a ⊗ b)"
⟨*proof*⟩

13.9 Prime Ideals

lemma (in ideal**)** primeidealCD:
assumes "cring R"
assumes notprime: "¬ primeideal I R"
shows "carrier R = I ∨ (∃a b. a ∈ carrier R ∧ b ∈ carrier R ∧ a ⊗ b ∈ I ∧ a ∉ I ∧ b ∉ I)"
⟨*proof*⟩

lemma (in ideal**)** primeidealCE:
assumes "cring R"
assumes notprime: "¬ primeideal I R"
obtains "carrier R = I"
| "∃a b. a ∈ carrier R ∧ b ∈ carrier R ∧ a ⊗ b ∈ I ∧ a ∉ I ∧ b ∉ I"
⟨*proof*⟩

If {**0**} is a prime ideal of a commutative ring, the ring is a domain

lemma (in cring**)** zeroprimeideal_domainI:
assumes pi: "primeideal {**0**} R"
shows "domain R"
⟨*proof*⟩

corollary (in cring**)** domain_eq_zeroprimeideal: "domain R = primeideal {**0**} R"
⟨*proof*⟩

13.10 Maximal Ideals

lemma (in ideal**)** helper_I_closed:
assumes carr: "a ∈ carrier R" "x ∈ carrier R" "y ∈ carrier R"
and axI: "a ⊗ x ∈ I"
shows "a ⊗ (x ⊗ y) ∈ I"
⟨*proof*⟩

lemma (in ideal**)** helper_max_prime:
assumes "cring R"
assumes acarr: "a ∈ carrier R"
shows "ideal {x∈carrier R. a ⊗ x ∈ I} R"
⟨*proof*⟩

In a cring every maximal ideal is prime

lemma (in cring**)** maximalideal_prime:
assumes "maximalideal I R"
shows "primeideal I R"
⟨*proof*⟩

13.11 Derived Theorems

A non-zero cring that has only the two trivial ideals is a field

lemma (in cring**)** trivialideals_fieldI:
  assumes carrnzero: "carrier R ≠ {**0**}"
    and haveideals: "{I. ideal I R} = {{**0**}, carrier R}"
  shows "field R"
⟨*proof*⟩

lemma (in field**)** all_ideals: "{I. ideal I R} = {{**0**}, carrier R}"
⟨*proof*⟩
lemma (in cring**)** trivialideals_eq_field:
  assumes carrnzero: "carrier R ≠ {**0**}"
  shows "({I. ideal I R} = {{**0**}, carrier R}) = field R"
  ⟨*proof*⟩

Like zeroprimeideal for domains

lemma (in field**)** zeromaximalideal: "maximalideal {**0**} R"
⟨*proof*⟩

lemma (in cring**)** zeromaximalideal_fieldI:
  assumes zeromax: "maximalideal {**0**} R"
  shows "field R"
⟨*proof*⟩

lemma (in cring**)** zeromaximalideal_eq_field: "maximalideal {**0**} R = field R"
  ⟨*proof*⟩

end

theory RingHom
imports Ideal
begin

14 Homomorphisms of Non-Commutative Rings

Lifting existing lemmas in a ring_hom_ring locale

locale ring_hom_ring = R?: ring R + S?: ring S
    for R **(structure) and** S **(structure)** +
  fixes h
  assumes homh: "h ∈ ring_hom R S"
  notes hom_mult [simp] = ring_hom_mult [OF homh]
    and hom_one [simp] = ring_hom_one [OF homh]

sublocale ring_hom_cring ⊆ ring: ring_hom_ring
  ⟨*proof*⟩

sublocale ring_hom_ring ⊆ abelian_group?: abelian_group_hom R S
⟨*proof*⟩

lemma (in ring_hom_ring**)** is_ring_hom_ring:
"ring_hom_ring R S h"
⟨*proof*⟩

lemma ring_hom_ringI:
fixes R **(structure) and** S **(structure)**
assumes "ring R" "ring S"
assumes hom_closed: "!!x. x ∈ carrier R ==> h x ∈ carrier S"
and compatible_mult: "⋀x y. [| x ∈ carrier R; y ∈ carrier R |]
==> h (x ⊗ y) = h x ⊗$_S$ h y"
and compatible_add: "⋀x y. [| x ∈ carrier R; y ∈ carrier R |]
==> h (x ⊕ y) = h x ⊕$_S$ h y"
and compatible_one: "h **1** = **1**$_S$"
shows "ring_hom_ring R S h"
⟨*proof*⟩

lemma ring_hom_ringI2:
assumes "ring R" "ring S"
assumes h: "h ∈ ring_hom R S"
shows "ring_hom_ring R S h"
⟨*proof*⟩

lemma ring_hom_ringI3:
fixes R **(structure) and** S **(structure)**
assumes "abelian_group_hom R S h" "ring R" "ring S"
assumes compatible_mult: "⋀x y. [| x ∈ carrier R; y ∈ carrier R |]
==> h (x ⊗ y) = h x ⊗$_S$ h y"
and compatible_one: "h **1** = **1**$_S$"
shows "ring_hom_ring R S h"
⟨*proof*⟩

lemma ring_hom_cringI:
assumes "ring_hom_ring R S h" "cring R" "cring S"
shows "ring_hom_cring R S h"
⟨*proof*⟩

14.1 The Kernel of a Ring Homomorphism

lemma (in ring_hom_ring**)** kernel_is_ideal: "ideal (a_kernel R S h) R"
⟨*proof*⟩

Elements of the kernel are mapped to zero

lemma (in abelian_group_hom**)** kernel_zero [simp]:
"i ∈ a_kernel R S h ⟹ h i = **0**$_S$"
⟨*proof*⟩

14.2 Cosets

Cosets of the kernel correspond to the elements of the image of the homomorphism

lemma (in ring_hom_ring**)** rcos_imp_homeq:
  assumes acarr: "a ∈ carrier R"
      and xrcos: "x ∈ a_kernel R S h +> a"
  shows "h x = h a"
⟨*proof*⟩

lemma (in ring_hom_ring**)** homeq_imp_rcos:
  assumes acarr: "a ∈ carrier R"
      and xcarr: "x ∈ carrier R"
      and hx: "h x = h a"
  shows "x ∈ a_kernel R S h +> a"
⟨*proof*⟩

corollary (in ring_hom_ring**)** rcos_eq_homeq:
  assumes acarr: "a ∈ carrier R"
  shows "(a_kernel R S h) +> a = {x ∈ carrier R. h x = h a}"
⟨*proof*⟩

lemma (in ring_hom_ring**)** hom_nat_pow:
  "x ∈ carrier R ⟹ h (x [^] (n :: nat)) = (h x) [^]$_S$ n"
  ⟨*proof*⟩

lemma (in ring_hom_ring**)** inj_on_domain:
  assumes "inj_on h (carrier R)"
  shows "domain S ⟹ domain R"
⟨*proof*⟩

end

theory UnivPoly
imports Module RingHom
begin

15 Univariate Polynomials

Polynomials are formalised as modules with additional operations for extracting coefficients from polynomials and for obtaining monomials from coefficients and exponents (record up_ring). The carrier set is a set of bounded functions from Nat to the coefficient domain. Bounded means that these functions return zero above a certain bound (the degree). There is a chapter on the formalisation of polynomials in the PhD thesis [1], which was

implemented with axiomatic type classes. This was later ported to Locales.

15.1 The Constructor for Univariate Polynomials

Functions with finite support.

```
locale bound =
  fixes z :: 'a
    and n :: nat
    and f :: "nat => 'a"
  assumes bound: "!!m. n < m ⟹ f m = z"

declare bound.intro [intro!]
  and bound.bound [dest]

lemma bound_below:
  assumes bound: "bound z m f" and nonzero: "f n ≠ z" shows "n ≤ m"
⟨proof⟩

record ('a, 'p) up_ring = "('a, 'p) module" +
  monom :: "['a, nat] => 'p"
  coeff :: "['p, nat] => 'a"

definition
  up :: "('a, 'm) ring_scheme => (nat => 'a) set"
  where "up R = {f. f ∈ UNIV → carrier R ∧ (∃n. bound 0_R n f)}"

definition UP :: "('a, 'm) ring_scheme => ('a, nat => 'a) up_ring"
  where "UP R = (|
   carrier = up R,
   mult = (λp∈up R. λq∈up R. λn. ⨁_R i ∈ {..n}. p i ⊗_R q (n-i)),
   one = (λi. if i=0 then 1_R else 0_R),
   zero = (λi. 0_R),
   add = (λp∈up R. λq∈up R. λi. p i ⊕_R q i),
   smult = (λa∈carrier R. λp∈up R. λi. a ⊗_R p i),
   monom = (λa∈carrier R. λn i. if i=n then a else 0_R),
   coeff = (λp∈up R. λn. p n)|)"
```

Properties of the set of polynomials up.

```
lemma mem_upI [intro]:
  "[| ⋀n. f n ∈ carrier R; ∃n. bound (zero R) n f |] ==> f ∈ up R"
  ⟨proof⟩

lemma mem_upD [dest]:
  "f ∈ up R ==> f n ∈ carrier R"
  ⟨proof⟩

context ring
begin
```

lemma bound_upD [dest]: "f ∈ up R ⟹ ∃n. bound **0** n f" ⟨*proof*⟩

lemma up_one_closed: "(λn. if n = 0 then **1** else **0**) ∈ up R" ⟨*proof*⟩

lemma up_smult_closed: "[| a ∈ carrier R; p ∈ up R |] ==> (λi. a ⊗ p i) ∈ up R" ⟨*proof*⟩

lemma up_add_closed:
"[| p ∈ up R; q ∈ up R |] ==> (λi. p i ⊕ q i) ∈ up R"
⟨*proof*⟩

lemma up_a_inv_closed:
"p ∈ up R ==> (λi. ⊖ (p i)) ∈ up R"
⟨*proof*⟩

lemma up_minus_closed:
"[| p ∈ up R; q ∈ up R |] ==> (λi. p i ⊖ q i) ∈ up R"
⟨*proof*⟩

lemma up_mult_closed:
"[| p ∈ up R; q ∈ up R |] ==>
(λn. ⨁i ∈ {..n}. p i ⊗ q (n-i)) ∈ up R"
⟨*proof*⟩

end

15.2 Effect of Operations on Coefficients

locale UP =
fixes R (**structure**) **and** P (**structure**)
defines P_def: "P == UP R"

locale UP_ring = UP + R?: ring R

locale UP_cring = UP + R?: cring R

sublocale UP_cring < UP_ring
⟨*proof*⟩

locale UP_domain = UP + R?: "domain" R

sublocale UP_domain < UP_cring
⟨*proof*⟩

context UP
begin

Temporarily declare P ≡ UP R as simp rule.

declare P_def [simp]

lemma up_eqI:
  assumes prem: "!!n. coeff P p n = coeff P q n" **and** R: "p ∈ carrier P" "q ∈ carrier P"
  shows "p = q"
⟨*proof*⟩

lemma coeff_closed [simp]:
  "p ∈ carrier P ==> coeff P p n ∈ carrier R" ⟨*proof*⟩

end

context UP_ring
begin

lemma coeff_monom [simp]:
  "a ∈ carrier R ==> coeff P (monom P a m) n = (if m=n then a else **0**)"
⟨*proof*⟩

lemma coeff_zero [simp]: "coeff P **0**$_P$ n = **0**" ⟨*proof*⟩

lemma coeff_one [simp]: "coeff P **1**$_P$ n = (if n=0 then **1** else **0**)"
  ⟨*proof*⟩

lemma coeff_smult [simp]:
  "[| a ∈ carrier R; p ∈ carrier P |] ==> coeff P (a ⊙$_P$ p) n = a ⊗ coeff P p n"
  ⟨*proof*⟩

lemma coeff_add [simp]:
  "[| p ∈ carrier P; q ∈ carrier P |] ==> coeff P (p ⊕$_P$ q) n = coeff P p n ⊕ coeff P q n"
  ⟨*proof*⟩

lemma coeff_mult [simp]:
  "[| p ∈ carrier P; q ∈ carrier P |] ==> coeff P (p ⊗$_P$ q) n = (⨁i ∈ {..n}. coeff P p i ⊗ coeff P q (n-i))"
  ⟨*proof*⟩

end

15.3 Polynomials Form a Ring.

context UP_ring
begin

Operations are closed over P.

lemma UP_mult_closed [simp]:
"[| p ∈ carrier P; q ∈ carrier P |] ==> p ⊗P q ∈ carrier P" ⟨*proof*⟩

lemma UP_one_closed [simp]:
"**1**P ∈ carrier P" ⟨*proof*⟩

lemma UP_zero_closed [intro, simp]:
"**0**P ∈ carrier P" ⟨*proof*⟩

lemma UP_a_closed [intro, simp]:
"[| p ∈ carrier P; q ∈ carrier P |] ==> p ⊕P q ∈ carrier P" ⟨*proof*⟩

lemma monom_closed [simp]:
"a ∈ carrier R ==> monom P a n ∈ carrier P" ⟨*proof*⟩

lemma UP_smult_closed [simp]:
"[| a ∈ carrier R; p ∈ carrier P |] ==> a ⊙P p ∈ carrier P" ⟨*proof*⟩

end

declare (in UP) P_def [simp del]

Algebraic ring properties

context UP_ring
begin

lemma UP_a_assoc:
assumes R: "p ∈ carrier P" "q ∈ carrier P" "r ∈ carrier P"
shows "(p ⊕P q) ⊕P r = p ⊕P (q ⊕P r)" ⟨*proof*⟩

lemma UP_l_zero [simp]:
assumes R: "p ∈ carrier P"
shows "**0**P ⊕P p = p" ⟨*proof*⟩

lemma UP_l_neg_ex:
assumes R: "p ∈ carrier P"
shows "∃q ∈ carrier P. q ⊕P p = **0**P"
⟨*proof*⟩

lemma UP_a_comm:
assumes R: "p ∈ carrier P" "q ∈ carrier P"
shows "p ⊕P q = q ⊕P p" ⟨*proof*⟩

lemma UP_m_assoc:
assumes R: "p ∈ carrier P" "q ∈ carrier P" "r ∈ carrier P"
shows "(p ⊗P q) ⊗P r = p ⊗P (q ⊗P r)"
⟨*proof*⟩

lemma UP_r_one [simp]:

assumes R: "p ∈ carrier P" **shows** "p ⊗P 1P = p"
⟨*proof*⟩

lemma UP_l_one [simp]:
assumes R: "p ∈ carrier P"
shows "1P ⊗P p = p"
⟨*proof*⟩

lemma UP_l_distr:
assumes R: "p ∈ carrier P" "q ∈ carrier P" "r ∈ carrier P"
shows "(p ⊕P q) ⊗P r = (p ⊗P r) ⊕P (q ⊗P r)"
⟨*proof*⟩

lemma UP_r_distr:
assumes R: "p ∈ carrier P" "q ∈ carrier P" "r ∈ carrier P"
shows "r ⊗P (p ⊕P q) = (r ⊗P p) ⊕P (r ⊗P q)"
⟨*proof*⟩

theorem UP_ring: "ring P"
⟨*proof*⟩

end

15.4 Polynomials Form a Commutative Ring.

context UP_cring
begin

lemma UP_m_comm:
assumes R1: "p ∈ carrier P" **and** R2: "q ∈ carrier P" **shows** "p ⊗P q = q ⊗P p"
⟨*proof*⟩

15.5 Polynomials over a commutative ring for a commutative ring

theorem UP_cring:
"cring P" ⟨*proof*⟩

end

context UP_ring
begin

lemma UP_a_inv_closed [intro, simp]:
"p ∈ carrier P ==> ⊖P p ∈ carrier P"
⟨*proof*⟩

lemma coeff_a_inv [simp]:

assumes R: "p ∈ carrier P"
shows "coeff P (⊖P p) n = ⊖ (coeff P p n)"
⟨*proof*⟩

end

sublocale UP_ring < P?: ring P ⟨*proof*⟩
sublocale UP_cring < P?: cring P ⟨*proof*⟩

15.6 Polynomials Form an Algebra

context UP_ring
begin

lemma UP_smult_l_distr:
"[| a ∈ carrier R; b ∈ carrier R; p ∈ carrier P |] ==>
(a ⊕ b) ⊙P p = a ⊙P p ⊕P b ⊙P p"
⟨*proof*⟩

lemma UP_smult_r_distr:
"[| a ∈ carrier R; p ∈ carrier P; q ∈ carrier P |] ==>
a ⊙P (p ⊕P q) = a ⊙P p ⊕P a ⊙P q"
⟨*proof*⟩

lemma UP_smult_assoc1:
"[| a ∈ carrier R; b ∈ carrier R; p ∈ carrier P |] ==>
(a ⊗ b) ⊙P p = a ⊙P (b ⊙P p)"
⟨*proof*⟩

lemma UP_smult_zero [simp]:
"p ∈ carrier P ==> **0** ⊙P p = **0**P"
⟨*proof*⟩

lemma UP_smult_one [simp]:
"p ∈ carrier P ==> **1** ⊙P p = p"
⟨*proof*⟩

lemma UP_smult_assoc2:
"[| a ∈ carrier R; p ∈ carrier P; q ∈ carrier P |] ==>
(a ⊙P p) ⊗P q = a ⊙P (p ⊗P q)"
⟨*proof*⟩

end

Interpretation of lemmas from algebra.

lemma (in UP_cring**)** UP_algebra:
"algebra R P" ⟨*proof*⟩

sublocale UP_cring < algebra R P ⟨*proof*⟩

15.7 Further Lemmas Involving Monomials

context UP_ring
begin

lemma monom_zero [simp]:
"monom P **0** n = **0**$_P$" ⟨*proof*⟩

lemma monom_mult_is_smult:
assumes R: "a ∈ carrier R" "p ∈ carrier P"
shows "monom P a 0 ⊗$_P$ p = a ⊙$_P$ p"
⟨*proof*⟩

lemma monom_one [simp]:
"monom P **1** 0 = **1**$_P$"
⟨*proof*⟩

lemma monom_add [simp]:
"[| a ∈ carrier R; b ∈ carrier R |] ==>
monom P (a ⊕ b) n = monom P a n ⊕$_P$ monom P b n"
⟨*proof*⟩

lemma monom_one_Suc:
"monom P **1** (Suc n) = monom P **1** n ⊗$_P$ monom P **1** 1"
⟨*proof*⟩

lemma monom_one_Suc2:
"monom P **1** (Suc n) = monom P **1** 1 ⊗$_P$ monom P **1** n"
⟨*proof*⟩

The following corollary follows from lemmas monom P **1** (Suc ?n) = monom P **1** ?n ⊗$_P$ monom P **1** 1 and monom P **1** (Suc ?n) = monom P **1** 1 ⊗$_P$ monom P **1** ?n, and is trivial in UP_cring

corollary monom_one_comm: **shows** "monom P **1** k ⊗$_P$ monom P **1** 1 = monom P **1** 1 ⊗$_P$ monom P **1** k"
⟨*proof*⟩

lemma monom_mult_smult:
"[| a ∈ carrier R; b ∈ carrier R |] ==> monom P (a ⊗ b) n = a ⊙$_P$ monom P b n"
⟨*proof*⟩

lemma monom_one_mult:
"monom P **1** (n + m) = monom P **1** n ⊗$_P$ monom P **1** m"
⟨*proof*⟩

lemma monom_one_mult_comm: "monom P **1** n ⊗$_P$ monom P **1** m = monom P **1** m ⊗$_P$ monom P **1** n"
⟨*proof*⟩

lemma monom_mult [simp]:
  assumes a_in_R: "a ∈ carrier R" **and** b_in_R: "b ∈ carrier R"
  shows "monom P (a ⊗ b) (n + m) = monom P a n ⊗$_P$ monom P b m"
⟨*proof*⟩

lemma monom_a_inv [simp]:
  "a ∈ carrier R ==> monom P (⊖ a) n = ⊖$_P$ monom P a n"
  ⟨*proof*⟩

lemma monom_inj:
  "inj_on (λa. monom P a n) (carrier R)"
⟨*proof*⟩

end

15.8 The Degree Function

definition
  deg :: "[('a, 'm) ring_scheme, nat => 'a] => nat"
  where "deg R p = (LEAST n. bound $\mathbf{0}_R$ n (coeff (UP R) p))"

context UP_ring
begin

lemma deg_aboveI:
  "[| (!!m. n < m ==> coeff P p m = **0**); p ∈ carrier P |] ==> deg R p <= n"
  ⟨*proof*⟩

lemma deg_aboveD:
  assumes "deg R p < m" **and** "p ∈ carrier P"
  shows "coeff P p m = **0**"
⟨*proof*⟩

lemma deg_belowI:
  assumes non_zero: "n ≠ 0 ⟹ coeff P p n ≠ **0**"
    and R: "p ∈ carrier P"
  shows "n ≤ deg R p"
— Logically, this is a slightly stronger version of deg_aboveD
⟨*proof*⟩

lemma lcoeff_nonzero_deg:
  assumes deg: "deg R p ≠ 0" **and** R: "p ∈ carrier P"
  shows "coeff P p (deg R p) ≠ **0**"
⟨*proof*⟩

lemma lcoeff_nonzero_nonzero:
assumes deg: "deg R p = 0" **and** nonzero: "p ≠ **0**$_P$" **and** R: "p ∈ carrier P"
shows "coeff P p 0 ≠ **0**"
⟨*proof*⟩

lemma lcoeff_nonzero:
assumes neq: "p ≠ **0**$_P$" **and** R: "p ∈ carrier P"
shows "coeff P p (deg R p) ≠ **0**"
⟨*proof*⟩

lemma deg_eqI:
"[| ⋀m. n < m ⟹ coeff P p m = **0**;
⋀n. n ≠ 0 ⟹ coeff P p n ≠ **0**; p ∈ carrier P |] ==> deg R p = n"
⟨*proof*⟩

Degree and polynomial operations

lemma deg_add [simp]:
"p ∈ carrier P ⟹ q ∈ carrier P ⟹
deg R (p ⊕$_P$ q) ≤ max (deg R p) (deg R q)"
⟨*proof*⟩

lemma deg_monom_le:
"a ∈ carrier R ⟹ deg R (monom P a n) ≤ n"
⟨*proof*⟩

lemma deg_monom [simp]:
"[| a ≠ **0**; a ∈ carrier R |] ==> deg R (monom P a n) = n"
⟨*proof*⟩

lemma deg_const [simp]:
assumes R: "a ∈ carrier R" **shows** "deg R (monom P a 0) = 0"
⟨*proof*⟩

lemma deg_zero [simp]:
"deg R **0**$_P$ = 0"
⟨*proof*⟩

lemma deg_one [simp]:
"deg R **1**$_P$ = 0"
⟨*proof*⟩

lemma deg_uminus [simp]:
assumes R: "p ∈ carrier P" **shows** "deg R (⊖$_P$ p) = deg R p"
⟨*proof*⟩

The following lemma is later *overwritten* by the most specific one for domains, deg_smult.

lemma deg_smult_ring [simp]:
  "[| a ∈ carrier R; p ∈ carrier P |] ==>
  deg R (a ⊙P p) ≤ (if a = **0** then 0 else deg R p)"
  ⟨*proof*⟩

end

context UP_domain
begin

lemma deg_smult [simp]:
  assumes R: "a ∈ carrier R" "p ∈ carrier P"
  shows "deg R (a ⊙P p) = (if a = **0** then 0 else deg R p)"
⟨*proof*⟩

end

context UP_ring
begin

lemma deg_mult_ring:
  assumes R: "p ∈ carrier P" "q ∈ carrier P"
  shows "deg R (p ⊗P q) ≤ deg R p + deg R q"
⟨*proof*⟩

end

context UP_domain
begin

lemma deg_mult [simp]:
  "[| p ≠ **0**P; q ≠ **0**P; p ∈ carrier P; q ∈ carrier P |] ==>
  deg R (p ⊗P q) = deg R p + deg R q"
⟨*proof*⟩

end

The following lemmas also can be lifted to UP_ring.

context UP_ring
begin

lemma coeff_finsum:
  assumes fin: "finite A"
  shows "p ∈ A → carrier P ==>
    coeff P (finsum P p A) k = (⨁i ∈ A. coeff P (p i) k)"
  ⟨*proof*⟩

lemma up_repr:
  assumes R: "p ∈ carrier P"

shows "(⨁P i ∈ {..deg R p}. monom P (coeff P p i) i) = p"
⟨*proof*⟩

lemma up_repr_le:
"[| deg R p <= n; p ∈ carrier P |] ==>
(⨁P i ∈ {..n}. monom P (coeff P p i) i) = p"
⟨*proof*⟩

end

15.9 Polynomials over Integral Domains

lemma domainI:
assumes cring: "cring R"
and one_not_zero: "one R ≠ zero R"
and integral: "⋀a b. [| mult R a b = zero R; a ∈ carrier R;
b ∈ carrier R |] ==> a = zero R ∨ b = zero R"
shows "domain R"
⟨*proof*⟩

context UP_domain
begin

lemma UP_one_not_zero:
"**1**P ≠ **0**P"
⟨*proof*⟩

lemma UP_integral:
"[| p ⊗P q = **0**P; p ∈ carrier P; q ∈ carrier P |] ==> p = **0**P ∨ q = **0**P"
⟨*proof*⟩

theorem UP_domain:
"domain P"
⟨*proof*⟩

end

Interpretation of theorems from domain.

sublocale UP_domain < "domain" P
⟨*proof*⟩

15.10 The Evaluation Homomorphism and Universal Property

lemma (**in** abelian_monoid) boundD_carrier:
"[| bound **0** n f; n < m |] ==> f m ∈ carrier G"
⟨*proof*⟩

context ring

begin

theorem diagonal_sum:
"[| f ∈ {..n + m::nat} → carrier R; g ∈ {..n + m} → carrier R |] ==>
(⨁k ∈ {..n + m}. ⨁i ∈ {..k}. f i ⊗ g (k - i)) =
(⨁k ∈ {..n + m}. ⨁i ∈ {..n + m - k}. f k ⊗ g i)"
⟨*proof*⟩

theorem cauchy_product:
assumes bf: "bound **0** n f" **and** bg: "bound **0** m g"
and Rf: "f ∈ {..n} → carrier R" **and** Rg: "g ∈ {..m} → carrier R"
shows "(⨁k ∈ {..n + m}. ⨁i ∈ {..k}. f i ⊗ g (k - i)) =
(⨁i ∈ {..n}. f i) ⊗ (⨁i ∈ {..m}. g i)"
⟨*proof*⟩

end

lemma (**in** UP_ring) const_ring_hom:
"(λa. monom P a 0) ∈ ring_hom R P"
⟨*proof*⟩

definition
eval :: "[('a, 'm) ring_scheme, ('b, 'n) ring_scheme,
'a => 'b, 'b, nat => 'a] => 'b"
where "eval R S phi s = (λp ∈ carrier (UP R).
⨁$_S$i ∈ {..deg R p}. phi (coeff (UP R) p i) ⊗$_S$ s [^]$_S$ i)"

context UP
begin

lemma eval_on_carrier:
fixes S (**structure**)
shows "p ∈ carrier P ==>
eval R S phi s p = (⨁$_S$ i ∈ {..deg R p}. phi (coeff P p i) ⊗$_S$ s [^]$_S$
i)"
⟨*proof*⟩

lemma eval_extensional:
"eval R S phi p ∈ extensional (carrier P)"
⟨*proof*⟩

end

The universal property of the polynomial ring

locale UP_pre_univ_prop = ring_hom_cring + UP_cring

locale UP_univ_prop = UP_pre_univ_prop +
fixes s **and** Eval
assumes indet_img_carrier [simp, intro]: "s ∈ carrier S"

defines Eval_def: "Eval == eval R S h s"

JE: I have moved the following lemma from Ring.thy and lifted then to the locale ring_hom_ring from ring_hom_cring.

JE: I was considering using it in eval_ring_hom, but that property does not hold for non commutative rings, so maybe it is not that necessary.

lemma (in ring_hom_ring**)** hom_finsum [simp]:
"f ∈ A → carrier R ⟹
h (finsum R f A) = finsum S (h ∘ f) A"
⟨*proof*⟩

context UP_pre_univ_prop
begin

theorem eval_ring_hom:
assumes S: "s ∈ carrier S"
shows "eval R S h s ∈ ring_hom P S"
⟨*proof*⟩

The following lemma could be proved in UP_cring with the additional assumption that h is closed.

lemma (in UP_pre_univ_prop**)** eval_const:
"[| s ∈ carrier S; r ∈ carrier R |] ==> eval R S h s (monom P r 0) = h r"
⟨*proof*⟩

Further properties of the evaluation homomorphism.

The following proof is complicated by the fact that in arbitrary rings one might have **1 = 0**.

lemma (in UP_pre_univ_prop**)** eval_monom1:
assumes S: "s ∈ carrier S"
shows "eval R S h s (monom P **1** 1) = s"
⟨*proof*⟩

end

Interpretation of ring homomorphism lemmas.

sublocale UP_univ_prop < ring_hom_cring P S Eval
⟨*proof*⟩

lemma (in UP_cring**)** monom_pow:
assumes R: "a ∈ carrier R"
shows "(monom P a n) [^]$_P$ m = monom P (a [^] m) (n * m)"
⟨*proof*⟩

lemma (in ring_hom_cring**)** hom_pow [simp]:

"x ∈ carrier R ==> h (x [^] n) = h x [^]$_S$ (n::nat)"
⟨*proof*⟩

lemma (in UP_univ_prop**)** Eval_monom:
"r ∈ carrier R ==> Eval (monom P r n) = h r ⊗$_S$ s [^]$_S$ n"
⟨*proof*⟩

lemma (in UP_pre_univ_prop**)** eval_monom:
assumes R: "r ∈ carrier R" **and** S: "s ∈ carrier S"
shows "eval R S h s (monom P r n) = h r ⊗$_S$ s [^]$_S$ n"
⟨*proof*⟩

lemma (in UP_univ_prop**)** Eval_smult:
"[| r ∈ carrier R; p ∈ carrier P |] ==> Eval (r ⊙$_P$ p) = h r ⊗$_S$ Eval p"
⟨*proof*⟩

lemma ring_hom_cringI:
assumes "cring R"
and "cring S"
and "h ∈ ring_hom R S"
shows "ring_hom_cring R S h"
⟨*proof*⟩

context UP_pre_univ_prop
begin

lemma UP_hom_unique:
assumes "ring_hom_cring P S Phi"
assumes Phi: "Phi (monom P **1** (Suc 0)) = s"
"!!r. r ∈ carrier R ==> Phi (monom P r 0) = h r"
assumes "ring_hom_cring P S Psi"
assumes Psi: "Psi (monom P **1** (Suc 0)) = s"
"!!r. r ∈ carrier R ==> Psi (monom P r 0) = h r"
and P: "p ∈ carrier P" **and** S: "s ∈ carrier S"
shows "Phi p = Psi p"
⟨*proof*⟩

lemma ring_homD:
assumes Phi: "Phi ∈ ring_hom P S"
shows "ring_hom_cring P S Phi"
⟨*proof*⟩

theorem UP_universal_property:
assumes S: "s ∈ carrier S"
shows "∃!Phi. Phi ∈ ring_hom P S ∩ extensional (carrier P) ∧
Phi (monom P **1** 1) = s ∧
(∀r ∈ carrier R. Phi (monom P r 0) = h r)"
⟨*proof*⟩

end

JE: The following lemma was added by me; it might be even lifted to a simpler locale

context monoid
begin

lemma nat_pow_eone[simp]: **assumes** x_in_G: "x ∈ carrier G" **shows** "x [^] (1::nat) = x"
⟨*proof*⟩

end

context UP_ring
begin

abbreviation lcoeff :: "(nat =>'a) => 'a" **where** "lcoeff p == coeff P p (deg R p)"

lemma lcoeff_nonzero2: **assumes** p_in_R: "p ∈ carrier P" **and** p_not_zero: "p ≠ 0$_P$" **shows** "lcoeff p ≠ **0**"
⟨*proof*⟩

15.11 The long division algorithm: some previous facts.

lemma coeff_minus [simp]:
assumes p: "p ∈ carrier P" **and** q: "q ∈ carrier P"
shows "coeff P (p ⊖$_P$ q) n = coeff P p n ⊖ coeff P q n"
⟨*proof*⟩

lemma lcoeff_closed [simp]: **assumes** p: "p ∈ carrier P" **shows** "lcoeff p ∈ carrier R"
⟨*proof*⟩

lemma deg_smult_decr: **assumes** a_in_R: "a ∈ carrier R" **and** f_in_P: "f ∈ carrier P" **shows** "deg R (a ⊙$_P$ f) ≤ deg R f"
⟨*proof*⟩

lemma coeff_monom_mult: **assumes** R: "c ∈ carrier R" **and** P: "p ∈ carrier P"
shows "coeff P (monom P c n ⊗$_P$ p) (m + n) = c ⊗ (coeff P p m)"
⟨*proof*⟩

lemma deg_lcoeff_cancel:
assumes p_in_P: "p ∈ carrier P" **and** q_in_P: "q ∈ carrier P" **and** r_in_P: "r ∈ carrier P"
and deg_r_nonzero: "deg R r ≠ 0"
and deg_R_p: "deg R p ≤ deg R r" **and** deg_R_q: "deg R q ≤ deg R r"

and coeff_R_p_eq_q: "coeff P p (deg R r) = $\ominus_R$ (coeff P q (deg R r))"
shows "deg R (p $\oplus_P$ q) < deg R r"
⟨*proof*⟩

lemma monom_deg_mult:
assumes f_in_P: "f ∈ carrier P" **and** g_in_P: "g ∈ carrier P" **and** deg_le: "deg R g ≤ deg R f"
and a_in_R: "a ∈ carrier R"
shows "deg R (g $\otimes_P$ monom P a (deg R f - deg R g)) ≤ deg R f"
⟨*proof*⟩

lemma deg_zero_impl_monom:
assumes f_in_P: "f ∈ carrier P" **and** deg_f: "deg R f = 0"
shows "f = monom P (coeff P f 0) 0"
⟨*proof*⟩

end

15.12 The long division proof for commutative rings

context UP_cring
begin

lemma exI3: **assumes** exist: "Pred x y z"
shows "∃ x y z. Pred x y z"
⟨*proof*⟩

Jacobson's Theorem 2.14

lemma long_div_theorem:
assumes g_in_P [simp]: "g ∈ carrier P" **and** f_in_P [simp]: "f ∈ carrier P"
and g_not_zero: "g ≠ $\mathbf{0}_P$"
shows "∃q r (k::nat). (q ∈ carrier P) ∧ (r ∈ carrier P) ∧ (lcoeff g)[^]$_R$k $\odot_P$ f = g $\otimes_P$ q $\oplus_P$ r ∧ (r = $\mathbf{0}_P$ ∨ deg R r < deg R g)"
⟨*proof*⟩

end

The remainder theorem as corollary of the long division theorem.

context UP_cring
begin

lemma deg_minus_monom:
assumes a: "a ∈ carrier R"
and R_not_trivial: "(carrier R ≠ {**0**})"
shows "deg R (monom P $\mathbf{1}_R$ 1 $\ominus_P$ monom P a 0) = 1"
(is "deg R ?g = 1")
⟨*proof*⟩

lemma lcoeff_monom:
assumes a: "a ∈ carrier R" **and** R_not_trivial: "(carrier R ≠ {**0**})"
shows "lcoeff (monom P $\mathbf{1}_R$ 1 ⊖$_P$ monom P a 0) = **1**"
⟨*proof*⟩

lemma deg_nzero_nzero:
assumes deg_p_nzero: "deg R p ≠ 0"
shows "p ≠ $\mathbf{0}_P$"
⟨*proof*⟩

lemma deg_monom_minus:
assumes a: "a ∈ carrier R"
and R_not_trivial: "carrier R ≠ {**0**}"
shows "deg R (monom P $\mathbf{1}_R$ 1 ⊖$_P$ monom P a 0) = 1"
(**is** "deg R ?g = 1")
⟨*proof*⟩

lemma eval_monom_expr:
assumes a: "a ∈ carrier R"
shows "eval R R id a (monom P $\mathbf{1}_R$ 1 ⊖$_P$ monom P a 0) = **0**"
(**is** "eval R R id a ?g = _")
⟨*proof*⟩

lemma remainder_theorem_exist:
assumes f: "f ∈ carrier P" **and** a: "a ∈ carrier R"
and R_not_trivial: "carrier R ≠ {**0**}"
shows "∃ q r. (q ∈ carrier P) ∧ (r ∈ carrier P) ∧ f = (monom P $\mathbf{1}_R$ 1 ⊖$_P$ monom P a 0) ⊗$_P$ q ⊕$_P$ r ∧ (deg R r = 0)"
(**is** "∃ q r. (q ∈ carrier P) ∧ (r ∈ carrier P) ∧ f = ?g ⊗$_P$ q ⊕$_P$ r ∧ (deg R r = 0)")
⟨*proof*⟩

lemma remainder_theorem_expression:
assumes f [simp]: "f ∈ carrier P" **and** a [simp]: "a ∈ carrier R"
and q [simp]: "q ∈ carrier P" **and** r [simp]: "r ∈ carrier P"
and R_not_trivial: "carrier R ≠ {**0**}"
and f_expr: "f = (monom P $\mathbf{1}_R$ 1 ⊖$_P$ monom P a 0) ⊗$_P$ q ⊕$_P$ r"
(**is** "f = ?g ⊗$_P$ q ⊕$_P$ r" **is** "f = ?gq ⊕$_P$ r")
and deg_r_0: "deg R r = 0"
shows "r = monom P (eval R R id a f) 0"
⟨*proof*⟩

corollary remainder_theorem:
assumes f [simp]: "f ∈ carrier P" **and** a [simp]: "a ∈ carrier R"
and R_not_trivial: "carrier R ≠ {**0**}"
shows "∃ q r. (q ∈ carrier P) ∧ (r ∈ carrier P) ∧
f = (monom P $\mathbf{1}_R$ 1 ⊖$_P$ monom P a 0) ⊗$_P$ q ⊕$_P$ monom P (eval R R id a f) 0"
(**is** "∃ q r. (q ∈ carrier P) ∧ (r ∈ carrier P) ∧ f = ?g ⊗$_P$ q ⊕$_P$ monom

```
P (eval R R id a f) 0")
```
⟨*proof*⟩

end

15.13 Sample Application of Evaluation Homomorphism

lemma UP_pre_univ_propI:
  assumes "cring R"
    and "cring S"
    and "h ∈ ring_hom R S"
  shows "UP_pre_univ_prop R S h"
  ⟨*proof*⟩

definition
  INTEG :: "int ring"
  where "INTEG = ⦇carrier = UNIV, mult = (*), one = 1, zero = 0, add = (+)⦈"

lemma INTEG_cring: "cring INTEG"
  ⟨*proof*⟩

lemma INTEG_id_eval:
  "UP_pre_univ_prop INTEG INTEG id"
  ⟨*proof*⟩

Interpretation now enables to import all theorems and lemmas valid in the context of homomorphisms between INTEG and UP INTEG globally.

interpretation INTEG: UP_pre_univ_prop INTEG INTEG id "UP INTEG"
  ⟨*proof*⟩

lemma INTEG_closed [intro, simp]:
  "z ∈ carrier INTEG"
  ⟨*proof*⟩

lemma INTEG_mult [simp]:
  "mult INTEG z w = z * w"
  ⟨*proof*⟩

lemma INTEG_pow [simp]:
  "pow INTEG z n = z ^ n"
  ⟨*proof*⟩

lemma "eval INTEG INTEG id 10 (monom (UP INTEG) 5 2) = 500"
  ⟨*proof*⟩

end

16 Generated Groups

theory Generated_Groups
imports Group Coset

begin

16.1 Generated Groups

inductive_set generate :: "('a, 'b) monoid_scheme ⇒ 'a set ⇒ 'a set"
for G **and** H **where**
one: "$\mathbf{1}_G$ ∈ generate G H"
| incl: "h ∈ H ⟹ h ∈ generate G H"
| inv: "h ∈ H ⟹ inv_G h ∈ generate G H"
| eng: "h1 ∈ generate G H ⟹ h2 ∈ generate G H ⟹ h1 $\otimes_G$ h2 ∈ generate G H"

16.1.1 Basic Properties

lemma (**in** group) generate_consistent:
assumes "K ⊆ H" "subgroup H G" **shows** "generate (G (| carrier := H |)) K = generate G K"
⟨*proof*⟩

lemma (**in** group) generate_in_carrier:
assumes "H ⊆ carrier G" **and** "h ∈ generate G H" **shows** "h ∈ carrier G"
⟨*proof*⟩

lemma (**in** group) generate_incl:
assumes "H ⊆ carrier G" **shows** "generate G H ⊆ carrier G"
⟨*proof*⟩

lemma (**in** group) generate_m_inv_closed:
assumes "H ⊆ carrier G" **and** "h ∈ generate G H" **shows** "(inv h) ∈ generate G H"
⟨*proof*⟩

lemma (**in** group) generate_is_subgroup:
assumes "H ⊆ carrier G" **shows** "subgroup (generate G H) G"
⟨*proof*⟩

lemma (**in** group) mono_generate:
assumes "K ⊆ H" **shows** "generate G K ⊆ generate G H"
⟨*proof*⟩

lemma (**in** group) generate_subgroup_incl:
assumes "K ⊆ H" "subgroup H G" **shows** "generate G K ⊆ H"
⟨*proof*⟩

lemma (in group) generate_minimal:
assumes "H ⊆ carrier G" **shows** "generate G H = ⋂ { H'. subgroup H' G ∧ H ⊆ H' }"
⟨*proof*⟩

lemma (in group) generateI:
assumes "subgroup E G" "H ⊆ E" **and** "⋀K. ⟦ subgroup K G; H ⊆ K ⟧ ⟹ E ⊆ K"
shows "E = generate G H"
⟨*proof*⟩

lemma (in group) normal_generateI:
assumes "H ⊆ carrier G" **and** "⋀h g. ⟦ h ∈ H; g ∈ carrier G ⟧ ⟹ g ⊗ h ⊗ (inv g) ∈ H"
shows "generate G H ◁ G"
⟨*proof*⟩

lemma (in group) subgroup_int_pow_closed:
assumes "subgroup H G" "h ∈ H" **shows** "h [^] (k :: int) ∈ H"
⟨*proof*⟩

lemma (in group) generate_pow:
assumes "a ∈ carrier G" **shows** "generate G { a } = { a [^] (k :: int) | k. k ∈ UNIV }"
⟨*proof*⟩

corollary (in group) generate_one: "generate G { **1** } = { **1** }"
⟨*proof*⟩

corollary (in group) generate_empty: "generate G {} = { **1** }"
⟨*proof*⟩

lemma (in group_hom)
"subgroup K G ⟹ subgroup (h ‘ K) H"
⟨*proof*⟩

lemma (in group_hom) generate_img:
assumes "K ⊆ carrier G" **shows** "generate H (h ‘ K) = h ‘ (generate G K)"
⟨*proof*⟩

16.2 Derived Subgroup

16.2.1 Definitions

abbreviation derived_set :: "('a, 'b) monoid_scheme ⇒ 'a set ⇒ 'a set"
where "derived_set G H ≡
⋃h1 ∈ H. (⋃h2 ∈ H. { h1 ⊗$_G$ h2 ⊗$_G$ (inv$_G$ h1) ⊗$_G$ (inv$_G$ h2) })"

definition derived :: "('a, 'b) monoid_scheme ⇒ 'a set ⇒ 'a set" **where**
"derived G H = generate G (derived_set G H)"

16.2.2 Basic Properties

lemma (**in** group) derived_set_incl:
assumes "K ⊆ H" "subgroup H G" **shows** "derived_set G K ⊆ H"
⟨*proof*⟩

lemma (**in** group) derived_incl:
assumes "K ⊆ H" "subgroup H G" **shows** "derived G K ⊆ H"
⟨*proof*⟩

lemma (**in** group) derived_set_in_carrier:
assumes "H ⊆ carrier G" **shows** "derived_set G H ⊆ carrier G"
⟨*proof*⟩

lemma (**in** group) derived_in_carrier:
assumes "H ⊆ carrier G" **shows** "derived G H ⊆ carrier G"
⟨*proof*⟩

lemma (**in** group) exp_of_derived_in_carrier:
assumes "H ⊆ carrier G" **shows** "(derived G ^^ n) H ⊆ carrier G"
⟨*proof*⟩

lemma (**in** group) derived_is_subgroup:
assumes "H ⊆ carrier G" **shows** "subgroup (derived G H) G"
⟨*proof*⟩

lemma (**in** group) exp_of_derived_is_subgroup:
assumes "subgroup H G" **shows** "subgroup ((derived G ^^ n) H) G"
⟨*proof*⟩

lemma (**in** group) exp_of_derived_is_subgroup':
assumes "H ⊆ carrier G" **shows** "subgroup ((derived G ^^ (Suc n)) H)
G"
⟨*proof*⟩

lemma (**in** group) mono_derived_set:
assumes "K ⊆ H" **shows** "derived_set G K ⊆ derived_set G H"
⟨*proof*⟩

lemma (**in** group) mono_derived:
assumes "K ⊆ H" **shows** "derived G K ⊆ derived G H"
⟨*proof*⟩

lemma (**in** group) mono_exp_of_derived:
assumes "K ⊆ H" **shows** "(derived G ^^ n) K ⊆ (derived G ^^ n) H"
⟨*proof*⟩

lemma (**in** group) derived_set_consistent:
assumes "K ⊆ H" "subgroup H G" **shows** "derived_set (G ⦇ carrier := H ⦈) K = derived_set G K"
⟨*proof*⟩

lemma (**in** group) derived_consistent:
assumes "K ⊆ H" "subgroup H G" **shows** "derived (G ⦇ carrier := H ⦈) K = derived G K"
⟨*proof*⟩

lemma (**in** comm_group) derived_eq_singleton:
assumes "H ⊆ carrier G" **shows** "derived G H = { 1 }"
⟨*proof*⟩

lemma (**in** group) derived_is_normal:
assumes "H ◁ G" **shows** "derived G H ◁ G"
⟨*proof*⟩

lemma (**in** group) normal_self: "carrier G ◁ G"
⟨*proof*⟩

corollary (**in** group) derived_self_is_normal: "derived G (carrier G) ◁ G"
⟨*proof*⟩

corollary (**in** group) derived_subgroup_is_normal:
assumes "subgroup H G" **shows** "derived G H ◁ G ⦇ carrier := H ⦈"
⟨*proof*⟩

corollary (**in** group) derived_quot_is_group: "group (G Mod (derived G (carrier G)))"
⟨*proof*⟩

lemma (**in** group) derived_quot_is_comm_group: "comm_group (G Mod (derived G (carrier G)))"
⟨*proof*⟩

corollary (**in** group) derived_quot_of_subgroup_is_comm_group:
assumes "subgroup H G" **shows** "comm_group ((G ⦇ carrier := H ⦈) Mod (derived G H))"
⟨*proof*⟩

proposition (**in** group) derived_minimal:
assumes "H ◁ G" **and** "comm_group (G Mod H)" **shows** "derived G (carrier G) ⊆ H"
⟨*proof*⟩

proposition (**in** group) derived_of_subgroup_minimal:

assumes "K ◁ G (| carrier := H |)" "subgroup H G" **and** "comm_group ((G (| carrier := H |)) Mod K)"
shows "derived G H ⊆ K"
⟨*proof*⟩

lemma (**in** group_hom) derived_img:
assumes "K ⊆ carrier G" **shows** "derived H (h ‘ K) = h ‘ (derived G K)"
⟨*proof*⟩

lemma (**in** group_hom) exp_of_derived_img:
assumes "K ⊆ carrier G" **shows** "(derived H ^^ n) (h ‘ K) = h ‘ ((derived G ^^ n) K)"
⟨*proof*⟩

16.2.3 Generated subgroup of a group

definition subgroup_generated :: "('a, 'b) monoid_scheme ⇒ 'a set ⇒ ('a, 'b) monoid_scheme"
where "subgroup_generated G S = G(|carrier := generate G (carrier G ∩ S)|)"

lemma carrier_subgroup_generated: "carrier (subgroup_generated G S) = generate G (carrier G ∩ S)"
⟨*proof*⟩

lemma (**in** group) subgroup_generated_subset_carrier_subset:
"S ⊆ carrier G ⟹ S ⊆ carrier(subgroup_generated G S)"
⟨*proof*⟩

lemma (**in** group) subgroup_generated_minimal:
"⟦subgroup H G; S ⊆ H⟧ ⟹ carrier(subgroup_generated G S) ⊆ H"
⟨*proof*⟩

lemma (**in** group) carrier_subgroup_generated_subset:
"carrier (subgroup_generated G A) ⊆ carrier G"
⟨*proof*⟩

lemma (**in** group) group_subgroup_generated [simp]: "group (subgroup_generated G S)"
⟨*proof*⟩

lemma (**in** group) abelian_subgroup_generated:
assumes "comm_group G"
shows "comm_group (subgroup_generated G S)" (**is** "comm_group ?GS")
⟨*proof*⟩

lemma (**in** group) subgroup_of_subgroup_generated:
assumes "H ⊆ B" "subgroup H G"

shows "subgroup H (subgroup_generated G B)"
⟨*proof*⟩

lemma carrier_subgroup_generated_alt:
assumes "Group.group G" "S ⊆ carrier G"
shows "carrier (subgroup_generated G S) = ⋂{H. subgroup H G ∧ carrier G ∩ S ⊆ H}"
⟨*proof*⟩

lemma one_subgroup_generated [simp]: "$\mathbf{1}_{\text{subgroup_generated G S}}$ = $\mathbf{1}_{\text{G}}$"
⟨*proof*⟩

lemma mult_subgroup_generated [simp]: "mult (subgroup_generated G S) = mult G"
⟨*proof*⟩

lemma (**in** group) inv_subgroup_generated [simp]:
assumes "f ∈ carrier (subgroup_generated G S)"
shows "$\text{inv}_{\text{subgroup_generated G S}}$ f = inv f"
⟨*proof*⟩

lemma subgroup_generated_restrict [simp]:
"subgroup_generated G (carrier G ∩ S) = subgroup_generated G S"
⟨*proof*⟩

lemma (**in** subgroup) carrier_subgroup_generated_subgroup [simp]:
"carrier (subgroup_generated G H) = H"
⟨*proof*⟩

lemma (**in** group) subgroup_subgroup_generated_iff:
"subgroup H (subgroup_generated G B) ⟷ subgroup H G ∧ H ⊆ carrier(subgroup_generated G B)"
(**is** "?lhs = ?rhs")
⟨*proof*⟩

lemma (**in** group) subgroup_subgroup_generated:
"subgroup (carrier(subgroup_generated G S)) G"
⟨*proof*⟩

lemma pow_subgroup_generated:
"pow (subgroup_generated G S) = (pow G :: 'a ⇒ nat ⇒ 'a)"
⟨*proof*⟩

lemma (**in** group) subgroup_generated2 [simp]: "subgroup_generated (subgroup_generated G S) S = subgroup_generated G S"
⟨*proof*⟩

lemma (**in** group) int_pow_subgroup_generated:
fixes n::int

assumes "x ∈ carrier (subgroup_generated G S)"
shows "x [^]$_{\text{subgroup_generated G S}}$ n = x [^]$_{\text{G}}$ n"
⟨*proof*⟩

lemma kernel_from_subgroup_generated [simp]:
"subgroup S G ⟹ kernel (subgroup_generated G S) H f = kernel G H f ∩ S"
⟨*proof*⟩

lemma kernel_to_subgroup_generated [simp]:
"kernel G (subgroup_generated H S) f = kernel G H f"
⟨*proof*⟩

16.3 And homomorphisms

lemma (**in** group) hom_from_subgroup_generated:
"h ∈ hom G H ⟹ h ∈ hom(subgroup_generated G A) H"
⟨*proof*⟩

lemma hom_into_subgroup:
"⟦h ∈ hom G G'; h ‘ (carrier G) ⊆ H⟧ ⟹ h ∈ hom G (subgroup_generated G' H)"
⟨*proof*⟩

lemma hom_into_subgroup_eq_gen:
"group G ⟹
h ∈ hom K (subgroup_generated G H)
⟷ h ∈ hom K G ∧ h ‘ (carrier K) ⊆ carrier(subgroup_generated G H)"
⟨*proof*⟩

lemma hom_into_subgroup_eq:
"⟦subgroup H G; group G⟧
⟹ (h ∈ hom K (subgroup_generated G H) ⟷ h ∈ hom K G ∧ h ‘ (carrier K) ⊆ H)"
⟨*proof*⟩

lemma (**in** group_hom) hom_between_subgroups:
assumes "h ‘ A ⊆ B"
shows "h ∈ hom (subgroup_generated G A) (subgroup_generated H B)"
⟨*proof*⟩

lemma (**in** group_hom) subgroup_generated_by_image:
assumes "S ⊆ carrier G"
shows "carrier (subgroup_generated H (h ‘ S)) = h ‘ (carrier(subgroup_generated G S))"
⟨*proof*⟩

lemma (**in** group_hom) iso_between_subgroups:
assumes "h ∈ iso G H" "S ⊆ carrier G" "h ‘ S = T"

shows "h ∈ iso (subgroup_generated G S) (subgroup_generated H T)"
⟨*proof*⟩

lemma (**in** group) subgroup_generated_group_carrier:
"subgroup_generated G (carrier G) = G"
⟨*proof*⟩

lemma iso_onto_image:
assumes "group G" "group H"
shows
"f ∈ iso G (subgroup_generated H (f ‘ carrier G)) ⟷ f ∈ hom G H
∧ inj_on f (carrier G)"
⟨*proof*⟩

lemma (**in** group) iso_onto_image:
"group H ⟹ f ∈ iso G (subgroup_generated H (f ‘ carrier G)) ⟷
f ∈ mon G H"
⟨*proof*⟩

end

17 Elementary Group Constructions

theory Elementary_Groups
imports Generated_Groups "HOL-Library.Infinite_Set"
begin

17.1 Direct sum/product lemmas

locale group_disjoint_sum = group G + AG: subgroup A G + BG: subgroup B
G **for** G (**structure**) **and** A B
begin

lemma subset_one: "A ∩ B ⊆ {**1**} ⟷ A ∩ B = {**1**}"
⟨*proof*⟩

lemma sub_id_iff: "A ∩ B ⊆ {**1**} ⟷ (∀x∈A. ∀y∈B. x ⊗ y = **1** ⟶ x =
1 ∧ y = **1**)"
(**is** "?lhs = ?rhs")
⟨*proof*⟩

lemma cancel: "A ∩ B ⊆ {**1**} ⟷ (∀x∈A. ∀y∈B. ∀x’∈A. ∀y’∈B. x ⊗ y
= x’ ⊗ y’ ⟶ x = x’ ∧ y = y’)"
(**is** "?lhs = ?rhs")
⟨*proof*⟩

lemma commuting_imp_normal1:
assumes sub: "carrier G ⊆ A <#> B"
and mult: "⋀x y. ⟦x ∈ A; y ∈ B⟧ ⟹ x ⊗ y = y ⊗ x"

shows "A ◁ G"
⟨*proof*⟩

lemma commuting_imp_normal2:
assumes"carrier G ⊆ A <#> B" "⋀x y. ⟦x ∈ A; y ∈ B⟧ ⟹ x ⊗ y = y ⊗ x"
shows "B ◁ G"
⟨*proof*⟩

lemma (**in** group) normal_imp_commuting:
assumes "A ◁ G" "B ◁ G" "A ∩ B ⊆ {1}" "x ∈ A" "y ∈ B"
shows "x ⊗ y = y ⊗ x"
⟨*proof*⟩

lemma normal_eq_commuting:
assumes "carrier G ⊆ A <#> B" "A ∩ B ⊆ {1}"
shows "A ◁ G ∧ B ◁ G ⟷ (∀x∈A. ∀y∈B. x ⊗ y = y ⊗ x)"
⟨*proof*⟩

lemma (**in** group) hom_group_mul_rev:
assumes "(λ(x,y). x ⊗ y) ∈ hom (subgroup_generated G A ×× subgroup_generated G B) G"
(**is** "?h ∈ hom ?P G")
and "x ∈ carrier G" "y ∈ carrier G" "x ∈ A" "y ∈ B"
shows "x ⊗ y = y ⊗ x"
⟨*proof*⟩

lemma hom_group_mul_eq:
"(λ(x,y). x ⊗ y) ∈ hom (subgroup_generated G A ×× subgroup_generated G B) G
⟷ (∀x∈A. ∀y∈B. x ⊗ y = y ⊗ x)"
(**is** "?lhs = ?rhs")
⟨*proof*⟩

lemma epi_group_mul_eq:
"(λ(x,y). x ⊗ y) ∈ epi (subgroup_generated G A ×× subgroup_generated G B) G
⟷ A <#> B = carrier G ∧ (∀x∈A. ∀y∈B. x ⊗ y = y ⊗ x)"
⟨*proof*⟩

lemma mon_group_mul_eq:
"(λ(x,y). x ⊗ y) ∈ mon (subgroup_generated G A ×× subgroup_generated G B) G
⟷ A ∩ B = {1} ∧ (∀x∈A. ∀y∈B. x ⊗ y = y ⊗ x)"
⟨*proof*⟩

lemma iso_group_mul_alt:

"(λ(x,y). x ⊗ y) ∈ iso (subgroup_generated G A ×× subgroup_generated G B) G
⟷ A ∩ B = {1} ∧ A <#> B = carrier G ∧ (∀x∈A. ∀y∈B. x ⊗ y = y ⊗ x)"
⟨*proof*⟩

lemma iso_group_mul_eq:
"(λ(x,y). x ⊗ y) ∈ iso (subgroup_generated G A ×× subgroup_generated G B) G
⟷ A ∩ B = {1} ∧ A <#> B = carrier G ∧ A ◁ G ∧ B ◁ G"
⟨*proof*⟩

lemma (**in** group) iso_group_mul_gen:
assumes "A ◁ G" "B ◁ G"
shows "(λ(x,y). x ⊗ y) ∈ iso (subgroup_generated G A ×× subgroup_generated G B) G
⟷ A ∩ B ⊆ {1} ∧ A <#> B = carrier G"
⟨*proof*⟩

lemma iso_group_mul:
assumes "comm_group G"
shows "((λ(x,y). x ⊗ y) ∈ iso (DirProd (subgroup_generated G A) (subgroup_generated G B)) G
⟷ A ∩ B ⊆ {1} ∧ A <#> B = carrier G)"
⟨*proof*⟩

end

17.2 The one-element group on a given object

definition singleton_group :: "'a ⇒ 'a monoid"
where "singleton_group a = (|carrier = {a}, monoid.mult = (λx y. a), one = a|)"

lemma singleton_group [simp]: "group (singleton_group a)"
⟨*proof*⟩

lemma singleton_abelian_group [simp]: "comm_group (singleton_group a)"
⟨*proof*⟩

lemma carrier_singleton_group [simp]: "carrier (singleton_group a) = {a}"
⟨*proof*⟩

lemma (**in** group) hom_into_singleton_iff [simp]:
"h ∈ hom G (singleton_group a) ⟷ h ∈ carrier G → {a}"
⟨*proof*⟩

declare group.hom_into_singleton_iff [simp]

lemma (**in** group) id_hom_singleton: "id ∈ hom (singleton_group 1) G"
⟨*proof*⟩

17.3 Similarly, trivial groups

definition trivial_group :: "('a, 'b) monoid_scheme ⇒ bool"
where "trivial_group G ≡ group G ∧ carrier G = {one G}"

lemma trivial_imp_finite_group:
"trivial_group G ⟹ finite(carrier G)"
⟨*proof*⟩

lemma trivial_singleton_group [simp]: "trivial_group(singleton_group a)"
⟨*proof*⟩

lemma (**in** group) trivial_group_subset:
"trivial_group G ⟷ carrier G ⊆ {one G}"
⟨*proof*⟩

lemma (**in** group) trivial_group: "trivial_group G ⟷ (∃a. carrier G = {a})"
⟨*proof*⟩

lemma (**in** group) trivial_group_alt:
"trivial_group G ⟷ (∃a. carrier G ⊆ {a})"
⟨*proof*⟩

lemma (**in** group) trivial_group_subgroup_generated:
assumes "S ⊆ {one G}"
shows "trivial_group(subgroup_generated G S)"
⟨*proof*⟩

lemma (**in** group) trivial_group_subgroup_generated_eq:
"trivial_group(subgroup_generated G s) ⟷ carrier G ∩ s ⊆ {one G}"
⟨*proof*⟩

lemma isomorphic_group_triviality1:
assumes "G ≅ H" "group H" "trivial_group G"
shows "trivial_group H"
⟨*proof*⟩

lemma isomorphic_group_triviality:
assumes "G ≅ H" "group G" "group H"
shows "trivial_group G ⟷ trivial_group H"
⟨*proof*⟩

lemma (in group_hom**)** kernel_from_trivial_group:
"trivial_group G ⟹ kernel G H h = carrier G"
⟨*proof*⟩

lemma (in group_hom**)** image_from_trivial_group:
"trivial_group G ⟹ h ‘ carrier G = {one H}"
⟨*proof*⟩

lemma (in group_hom**)** kernel_to_trivial_group:
"trivial_group H ⟹ kernel G H h = carrier G"
⟨*proof*⟩

17.4 The additive group of integers

definition integer_group
where "integer_group = ⦇carrier = UNIV, monoid.mult = (+), one = (0::int)⦈"

lemma group_integer_group [simp]: "group integer_group"
⟨*proof*⟩

lemma carrier_integer_group [simp]: "carrier integer_group = UNIV"
⟨*proof*⟩

lemma one_integer_group [simp]: "$\mathbf{1}_{\text{integer_group}}$ = 0"
⟨*proof*⟩

lemma mult_integer_group [simp]: "x $\otimes_{\text{integer_group}}$ y = x + y"
⟨*proof*⟩

lemma inv_integer_group [simp]: "$\text{inv}_{\text{integer_group}}$ x = -x"
⟨*proof*⟩

lemma abelian_integer_group: "comm_group integer_group"
⟨*proof*⟩

lemma group_nat_pow_integer_group [simp]:
fixes n::nat **and** x::int
shows "pow integer_group x n = int n * x"
⟨*proof*⟩

lemma group_int_pow_integer_group [simp]:
fixes n::int **and** x::int
shows "pow integer_group x n = n * x"
⟨*proof*⟩

lemma (in group**)** hom_integer_group_pow:
"x ∈ carrier G ⟹ pow G x ∈ hom integer_group G"
⟨*proof*⟩

17.5 Additive group of integers modulo n (n = 0 gives just the integers)

definition integer_mod_group :: "nat ⇒ int monoid"
where
"integer_mod_group n ≡
if n = 0 then integer_group
else (|carrier = {0..<int n}, monoid.mult = (λx y. (x+y) mod int n),
one = 0|)"

lemma carrier_integer_mod_group:
"carrier(integer_mod_group n) = (if n=0 then UNIV else {0..<int n})"
⟨*proof*⟩

lemma one_integer_mod_group[simp]: "one(integer_mod_group n) = 0"
⟨*proof*⟩

lemma mult_integer_mod_group[simp]: "monoid.mult(integer_mod_group n)
= (λx y. (x + y) mod int n)"
⟨*proof*⟩

lemma group_integer_mod_group [simp]: "group (integer_mod_group n)"
⟨*proof*⟩

lemma inv_integer_mod_group[simp]:
"x ∈ carrier (integer_mod_group n) ⟹ m_inv(integer_mod_group n) x
= (-x) mod int n"
⟨*proof*⟩

lemma pow_integer_mod_group [simp]:
fixes m::nat
shows "pow (integer_mod_group n) x m = (int m * x) mod int n"
⟨*proof*⟩

lemma int_pow_integer_mod_group:
"pow (integer_mod_group n) x m = (m * x) mod int n"
⟨*proof*⟩

lemma abelian_integer_mod_group [simp]: "comm_group(integer_mod_group
n)"
⟨*proof*⟩

lemma integer_mod_group_0 [simp]: "0 ∈ carrier(integer_mod_group n)"
⟨*proof*⟩

lemma integer_mod_group_1 [simp]: "1 ∈ carrier(integer_mod_group n)
⟷ (n ≠ 1)"
⟨*proof*⟩

lemma trivial_integer_mod_group: "trivial_group(integer_mod_group n) ⟷ n = 1"
  (**is** "?lhs = ?rhs")
⟨*proof*⟩

17.6 Cyclic groups

lemma (**in** group) subgroup_of_powers:
  "x ∈ carrier G ⟹ subgroup (range (λn::int. x [^] n)) G"
  ⟨*proof*⟩

lemma (**in** group) carrier_subgroup_generated_by_singleton:
  assumes "x ∈ carrier G"
  shows "carrier(subgroup_generated G {x}) = (range (λn::int. x [^] n))"
⟨*proof*⟩

definition cyclic_group
  where "cyclic_group G ≡ ∃x ∈ carrier G. subgroup_generated G {x} = G"

lemma (**in** group) cyclic_group:
  "cyclic_group G ⟷ (∃x ∈ carrier G. carrier G = range (λn::int. x [^] n))"
⟨*proof*⟩

lemma cyclic_integer_group [simp]: "cyclic_group integer_group"
⟨*proof*⟩

lemma nontrivial_integer_group [simp]: "¬ trivial_group integer_group"
  ⟨*proof*⟩

lemma (**in** group) cyclic_imp_abelian_group:
  "cyclic_group G ⟹ comm_group G"
  ⟨*proof*⟩

lemma trivial_imp_cyclic_group:
   "trivial_group G ⟹ cyclic_group G"
  ⟨*proof*⟩

lemma (**in** group) cyclic_group_alt:
   "cyclic_group G ⟷ (∃x. subgroup_generated G {x} = G)"
⟨*proof*⟩

lemma (**in** group) cyclic_group_generated:
  "cyclic_group (subgroup_generated G {x})"
  ⟨*proof*⟩

lemma (in group**)** cyclic_group_epimorphic_image:
assumes "h ∈ epi G H" "cyclic_group G" "group H"
shows "cyclic_group H"
⟨*proof*⟩

lemma isomorphic_group_cyclicity:
"⟦G ≅ H; group G; group H⟧ ⟹ cyclic_group G ⟷ cyclic_group H"
⟨*proof*⟩

end

theory Multiplicative_Group
imports
Complex_Main
Group
Coset
UnivPoly
Generated_Groups
Elementary_Groups
begin

18 Simplification Rules for Polynomials

lemma (in ring_hom_cring**)** hom_sub[simp]:
assumes "x ∈ carrier R" "y ∈ carrier R"
shows "h (x ⊖ y) = h x ⊖$_S$ h y"
⟨*proof*⟩

context UP_ring **begin**

lemma deg_nzero_nzero:
assumes deg_p_nzero: "deg R p ≠ 0"
shows "p ≠ **0**$_P$"
⟨*proof*⟩

lemma deg_add_eq:
assumes c: "p ∈ carrier P" "q ∈ carrier P"
assumes "deg R q ≠ deg R p"
shows "deg R (p ⊕$_P$ q) = max (deg R p) (deg R q)"
⟨*proof*⟩

lemma deg_minus_eq:
assumes "p ∈ carrier P" "q ∈ carrier P" "deg R q ≠ deg R p"
shows "deg R (p ⊖$_P$ q) = max (deg R p) (deg R q)"
⟨*proof*⟩

end

context UP_cring **begin**

lemma evalRR_add:
  assumes "p ∈ carrier P" "q ∈ carrier P"
  assumes x: "x ∈ carrier R"
  shows "eval R R id x (p ⊕$_P$ q) = eval R R id x p ⊕ eval R R id x q"
⟨*proof*⟩

lemma evalRR_sub:
  assumes "p ∈ carrier P" "q ∈ carrier P"
  assumes x: "x ∈ carrier R"
  shows "eval R R id x (p ⊖$_P$ q) = eval R R id x p ⊖ eval R R id x q"
⟨*proof*⟩

lemma evalRR_mult:
  assumes "p ∈ carrier P" "q ∈ carrier P"
  assumes x: "x ∈ carrier R"
  shows "eval R R id x (p ⊗$_P$ q) = eval R R id x p ⊗ eval R R id x q"
⟨*proof*⟩

lemma evalRR_monom:
  assumes a: "a ∈ carrier R" **and** x: "x ∈ carrier R"
  shows "eval R R id x (monom P a d) = a ⊗ x [^] d"
⟨*proof*⟩

lemma evalRR_one:
  assumes x: "x ∈ carrier R"
  shows "eval R R id x 1$_P$ = 1"
⟨*proof*⟩

lemma carrier_evalRR:
  assumes x: "x ∈ carrier R" **and** "p ∈ carrier P"
  shows "eval R R id x p ∈ carrier R"
⟨*proof*⟩

lemmas evalRR_simps = evalRR_add evalRR_sub evalRR_mult evalRR_monom evalRR_one carrier_evalRR

end

19 Properties of the Euler φ-function

In this section we prove that for every positive natural number the equation $\sum_{d|n}^{n} \varphi(d) = n$ holds.

lemma dvd_div_ge_1:
  fixes a b :: nat
  assumes "a ≥ 1" "b dvd a"

shows "a div b ≥ 1"
⟨*proof*⟩

lemma dvd_nat_bounds:
fixes n p :: nat
assumes "p > 0" "n dvd p"
shows "n > 0 ∧ n ≤ p"
⟨*proof*⟩

definition phi' :: "nat => nat"
where "phi' m = card {x. 1 ≤ x ∧ x ≤ m ∧ coprime x m}"

notation (latex **output**)
phi' (‹φ _›)

lemma phi'_nonzero:
assumes "m > 0"
shows "phi' m > 0"
⟨*proof*⟩

lemma dvd_div_eq_1:
fixes a b c :: nat
assumes "c dvd a" "c dvd b" "a div c = b div c"
shows "a = b" ⟨*proof*⟩

lemma dvd_div_eq_2:
fixes a b c :: nat
assumes "c>0" "a dvd c" "b dvd c" "c div a = c div b"
shows "a = b"
⟨*proof*⟩

lemma div_mult_mono:
fixes a b c :: nat
assumes "a > 0" "a≤d"
shows "a * b div d ≤ b"
⟨*proof*⟩

We arrive at the main result of this section: For every positive natural number the equation $\sum_{d|n} \varphi(d) = n$ holds.

The outline of the proof for this lemma is as follows: We count the n fractions $1/n$, ..., $(n-1)/n$, n/n. We analyze the reduced form $a/d = m/n$ for any of those fractions. We want to know how many fractions m/n have the reduced form denominator d. The condition $1 \leq m \leq n$ is equivalent to the condition $1 \leq a \leq d$. Therefore we want to know how many a with $1 \leq a \leq d$ exist, s.t. `gcd a d = 1`. This number is exactly φ `d`.

Finally, by counting the fractions m/n according to their reduced form denominator, we get:

(∑d | d dvd n. φ d) = n

. To formalize this proof in Isabelle, we analyze for an arbitrary divisor d of n

- the set of reduced form numerators {a. 1 ≤ a ∧ a ≤ d ∧ coprime a d}
- the set of numerators m, for which m/n has the reduced form denominator d, i.e. the set {m ∈ {1..n}. n div gcd m n = d}

We show that λa. a * n div d with the inverse λa. a div gcd a n is a bijection between theses sets, thus yielding the equality

φ d = card {m ∈ {1..n}. n div gcd m n = d}

This gives us

(∑d | d dvd n. φ d) = card (⋃d∈{d. d dvd n} {m ∈ {1..n}. n div gcd m n = d})

and by showing {1..n} ⊆ (⋃d∈{d. d dvd n} {m ∈ {1..n}. n div gcd m n = d}) (this is our counting argument) the thesis follows.

lemma sum_phi'_factors:
fixes n :: nat
assumes "n > 0"
shows "(∑d | d dvd n. phi' d) = n"
⟨*proof*⟩

20 Order of an Element of a Group

context group **begin**

definition (**in** group) ord :: "'a ⇒ nat" **where**
"ord x ≡ (@d. ∀n::nat. x [^] n = **1** ⟷ d dvd n)"

lemma (**in** group) pow_eq_id:
assumes "x ∈ carrier G"
shows "x [^] n = **1** ⟷ (ord x) dvd n"
⟨*proof*⟩

lemma (**in** group) pow_ord_eq_1 [simp]:
"x ∈ carrier G ⟹ x [^] ord x = **1**"
⟨*proof*⟩

lemma (**in** group) int_pow_eq_id:
assumes "x ∈ carrier G"
shows "(pow G x i = one G ⟷ int (ord x) dvd i)"

⟨*proof*⟩

lemma (**in** group) int_pow_eq:
"x ∈ carrier G ⟹ (x [^] m = x [^] n) ⟷ int (ord x) dvd (n - m)"
⟨*proof*⟩

lemma (**in** group) ord_eq_0:
"x ∈ carrier G ⟹ (ord x = 0 ⟷ (∀n::nat. n ≠ 0 ⟶ x [^] n ≠ 𝟭))"
⟨*proof*⟩

lemma (**in** group) ord_unique:
"x ∈ carrier G ⟹ ord x = d ⟷ (∀n. pow G x n = one G ⟷ d dvd
n)"
⟨*proof*⟩

lemma (**in** group) ord_eq_1:
"x ∈ carrier G ⟹ (ord x = 1 ⟷ x = 𝟭)"
⟨*proof*⟩

lemma (**in** group) ord_id [simp]:
"ord (one G) = 1"
⟨*proof*⟩

lemma (**in** group) ord_inv [simp]:
"x ∈ carrier G
⟹ ord (m_inv G x) = ord x"
⟨*proof*⟩

lemma (**in** group) ord_pow:
assumes "x ∈ carrier G" "k dvd ord x" "k ≠ 0"
shows "ord (pow G x k) = ord x div k"
⟨*proof*⟩

lemma (**in** group) ord_mul_divides:
assumes eq: "x ⊗ y = y ⊗ x" **and** xy: "x ∈ carrier G" "y ∈ carrier
G"
shows "ord (x ⊗ y) dvd (ord x * ord y)"
⟨*proof*⟩

lemma (**in** comm_group) abelian_ord_mul_divides:
"⟦x ∈ carrier G; y ∈ carrier G⟧
⟹ ord (x ⊗ y) dvd (ord x * ord y)"
⟨*proof*⟩

lemma ord_inj:
assumes a: "a ∈ carrier G"
shows "inj_on (λ x . a [^] x) {0 .. ord a - 1}"
⟨*proof*⟩

lemma ord_inj':
  assumes a: "a ∈ carrier G"
  shows "inj_on (λ x . a [^] x) {1 .. ord a}"
⟨*proof*⟩

lemma (**in** group) ord_ge_1:
  assumes finite: "finite (carrier G)" **and** a: "a ∈ carrier G"
  shows "ord a ≥ 1"
⟨*proof*⟩

lemma ord_elems:
  assumes "finite (carrier G)" "a ∈ carrier G"
  shows "{a[^]x | x. x ∈ (UNIV :: nat set)} = {a[^]x | x. x ∈ {0 .. ord a - 1}}" (**is** "?L = ?R")
⟨*proof*⟩

lemma (**in** group)
  assumes "x ∈ carrier G"
  shows finite_cyclic_subgroup:
        "finite(carrier(subgroup_generated G {x})) ⟷ (∃n::nat. n ≠ 0 ∧ x [^] n = 1)" (**is** "?fin ⟷ ?nat1")
    and infinite_cyclic_subgroup:
        "infinite(carrier(subgroup_generated G {x})) ⟷ (∀m n::nat. x [^] m = x [^] n ⟶ m = n)" (**is** "¬ ?fin ⟷ ?nateq")
    and finite_cyclic_subgroup_int:
        "finite(carrier(subgroup_generated G {x})) ⟷ (∃i::int. i ≠ 0 ∧ x [^] i = 1)" (**is** "?fin ⟷ ?int1")
    and infinite_cyclic_subgroup_int:
        "infinite(carrier(subgroup_generated G {x})) ⟷ (∀i j::int. x [^] i = x [^] j ⟶ i = j)" (**is** "¬ ?fin ⟷ ?inteq")
⟨*proof*⟩

lemma (**in** group) finite_cyclic_subgroup_order:
   "x ∈ carrier G ⟹ finite(carrier(subgroup_generated G {x})) ⟷ ord x ≠ 0"
  ⟨*proof*⟩

lemma (**in** group) infinite_cyclic_subgroup_order:
   "x ∈ carrier G ⟹ infinite (carrier(subgroup_generated G {x})) ⟷ ord x = 0"
  ⟨*proof*⟩

lemma generate_pow_on_finite_carrier:
  assumes "finite (carrier G)" **and** a: "a ∈ carrier G"
  shows "generate G { a } = { a [^] k | k. k ∈ (UNIV :: nat set) }"
⟨*proof*⟩

lemma ord_elems_inf_carrier:
 assumes "a ∈ carrier G" "ord a ≠ 0"

shows "{a[^]x | x. x ∈ (UNIV :: nat set)} = {a[^]x | x. x ∈ {0 .. ord a - 1}}" (**is** "?L = ?R")
⟨*proof*⟩

lemma generate_pow_nat:
assumes a: "a ∈ carrier G" **and** "ord a ≠ 0"
shows "generate G { a } = { a [^] k | k. k ∈ (UNIV :: nat set) }"
⟨*proof*⟩

lemma generate_pow_card:
assumes a: "a ∈ carrier G"
shows "ord a = card (generate G { a })"
⟨*proof*⟩

lemma (**in** group) cyclic_order_is_ord:
assumes "g ∈ carrier G"
shows "ord g = order (subgroup_generated G {g})"
⟨*proof*⟩

lemma ord_dvd_group_order:
assumes "a ∈ carrier G" **shows** "(ord a) dvd (order G)"
⟨*proof*⟩

lemma (**in** group) pow_order_eq_1:
assumes "a ∈ carrier G" **shows** "a [^] order G = 1"
⟨*proof*⟩

lemma dvd_gcd:
fixes a b :: nat
obtains q **where** "a * (b div gcd a b) = b*q"
⟨*proof*⟩

lemma (**in** group) ord_le_group_order:
assumes finite: "finite (carrier G)" **and** a: "a ∈ carrier G"
shows "ord a ≤ order G"
⟨*proof*⟩

lemma (**in** group) ord_pow_gen:
assumes "x ∈ carrier G"
shows "ord (pow G x k) = (if k = 0 then 1 else ord x div gcd (ord x) k)"
⟨*proof*⟩

lemma (**in** group)
assumes finite': "finite (carrier G)" "a ∈ carrier G"
shows pow_ord_eq_ord_iff: "group.ord G (a [^] k) = ord a ⟷ coprime k (ord a)" (**is** "?L ⟷ ?R")
⟨*proof*⟩

lemma element_generates_subgroup:
assumes finite[simp]: "finite (carrier G)"
assumes a[simp]: "a ∈ carrier G"
shows "subgroup {a [^] i | i. i ∈ {0 .. ord a - 1}} G"
⟨*proof*⟩

end

21 Number of Roots of a Polynomial

definition mult_of :: "('a, 'b) ring_scheme ⇒ 'a monoid" **where**
"mult_of R ≡ ⦇ carrier = carrier R - {$\mathbf{0}_R$}, mult = mult R, one = $\mathbf{1}_R$⦈"

lemma carrier_mult_of [simp]: "carrier (mult_of R) = carrier R - {$\mathbf{0}_R$}"
⟨*proof*⟩

lemma mult_mult_of [simp]: "mult (mult_of R) = mult R"
⟨*proof*⟩

lemma nat_pow_mult_of: "([^]$_{\text{mult_of R}}$) = (([^]$_R$) :: _ ⇒ nat ⇒ _)"
⟨*proof*⟩

lemma one_mult_of [simp]: "$\mathbf{1}_{\text{mult_of R}}$ = $\mathbf{1}_R$"
⟨*proof*⟩

lemmas mult_of_simps = carrier_mult_of mult_mult_of nat_pow_mult_of one_mult_of

context field
begin

lemma mult_of_is_Units: "mult_of R = units_of R"
⟨*proof*⟩

lemma m_inv_mult_of:
"⋀x. x ∈ carrier (mult_of R) ⟹ m_inv (mult_of R) x = m_inv R x"
⟨*proof*⟩

lemma (**in** field) field_mult_group: "group (mult_of R)"
⟨*proof*⟩

lemma finite_mult_of: "finite (carrier R) ⟹ finite (carrier (mult_of R))"
⟨*proof*⟩

lemma order_mult_of: "finite (carrier R) ⟹ order (mult_of R) = order R - 1"
⟨*proof*⟩

end

lemma (in monoid**)** Units_pow_closed :
  fixes d :: nat
  assumes "x ∈ Units G"
  shows "x [^] d ∈ Units G"
    ⟨*proof*⟩

lemma (in ring**)** r_right_minus_eq[simp]:
  assumes "a ∈ carrier R" "b ∈ carrier R"
  shows "a ⊖ b = **0** ⟷ a = b"
  ⟨*proof*⟩

context UP_cring **begin**

lemma is_UP_cring: "UP_cring R" ⟨*proof*⟩
lemma is_UP_ring:
  shows "UP_ring R" ⟨*proof*⟩

end

context UP_domain **begin**

lemma roots_bound:
  assumes f [simp]: "f ∈ carrier P"
  assumes f_not_zero: "f ≠ **0**$_P$"
  assumes finite: "finite (carrier R)"
  shows "finite {a ∈ carrier R . eval R R id a f = **0**} ∧
         card {a ∈ carrier R . eval R R id a f = **0**} ≤ deg R f" ⟨*proof*⟩

end

lemma (in domain**)** num_roots_le_deg :
  fixes p d :: nat
  assumes finite: "finite (carrier R)"
  assumes d_neq_zero: "d ≠ 0"
  shows "card {x ∈ carrier R. x [^] d = **1**} ≤ d"
⟨*proof*⟩

22 The Multiplicative Group of a Field

In this section we show that the multiplicative group of a finite field is generated by a single element, i.e. it is cyclic. The proof is inspired by the first proof given in the survey [2].

context field **begin**

lemma num_elems_of_ord_eq_phi':
assumes finite: "finite (carrier R)" **and** dvd: "d dvd order (mult_of R)"
and exists: "∃a∈carrier (mult_of R). group.ord (mult_of R) a = d"
shows "card {a ∈ carrier (mult_of R). group.ord (mult_of R) a = d} = phi' d"
⟨*proof*⟩

end

theorem (**in** field) finite_field_mult_group_has_gen :
assumes finite: "finite (carrier R)"
shows "∃ a ∈ carrier (mult_of R) . carrier (mult_of R) = {a[^]i | i::nat . i ∈ UNIV}"
⟨*proof*⟩

end

theory Group_Action
imports Bij Coset Congruence
begin

23 Group Actions

locale group_action =
fixes G (**structure**) **and** E **and** φ
assumes group_hom: "group_hom G (BijGroup E) φ"

definition
orbit :: "[_, 'a ⇒ 'b ⇒ 'b, 'b] ⇒ 'b set"
where "orbit G φ x = {(φ g) x | g. g ∈ carrier G}"

definition
orbits :: "[_, 'b set, 'a ⇒ 'b ⇒ 'b] ⇒ ('b set) set"
where "orbits G E φ = {orbit G φ x | x. x ∈ E}"

definition
stabilizer :: "[_, 'a ⇒ 'b ⇒ 'b, 'b] ⇒ 'a set"
where "stabilizer G φ x = {g ∈ carrier G. (φ g) x = x}"

definition
invariants :: "['b set, 'a ⇒ 'b ⇒ 'b, 'a] ⇒ 'b set"
where "invariants E φ g = {x ∈ E. (φ g) x = x}"

definition
normalizer :: "[_, 'a set] ⇒ 'a set"

where "normalizer G H =
stabilizer G (λg. λH ∈ {H. H ⊆ carrier G}. g <#$_G$ H #>$_G$ (inv$_G$ g)) H"

locale faithful_action = group_action +
assumes faithful: "inj_on φ (carrier G)"

locale transitive_action = group_action +
assumes unique_orbit: "⟦ x ∈ E; y ∈ E ⟧ ⟹ ∃g ∈ carrier G. (φ g) x = y"

23.1 Prelimineries

Some simple lemmas to make group action's properties more explicit

lemma (in group_action) id_eq_one: "(λx ∈ E. x) = φ **1**"
⟨*proof*⟩

lemma (in group_action) bij_prop0:
"⋀ g. g ∈ carrier G ⟹ (φ g) ∈ Bij E"
⟨*proof*⟩

lemma (in group_action) surj_prop:
"⋀ g. g ∈ carrier G ⟹ (φ g) ‘ E = E"
⟨*proof*⟩

lemma (in group_action) inj_prop:
"⋀ g. g ∈ carrier G ⟹ inj_on (φ g) E"
⟨*proof*⟩

lemma (in group_action) bij_prop1:
"⋀ g y. ⟦ g ∈ carrier G; y ∈ E ⟧ ⟹ ∃!x ∈ E. (φ g) x = y"
⟨*proof*⟩

lemma (in group_action) composition_rule:
assumes "x ∈ E" "g1 ∈ carrier G" "g2 ∈ carrier G"
shows "φ (g1 ⊗ g2) x = (φ g1) (φ g2 x)"
⟨*proof*⟩

lemma (in group_action) element_image:
assumes "g ∈ carrier G" **and** "x ∈ E" **and** "(φ g) x = y"
shows "y ∈ E"
⟨*proof*⟩

23.2 Orbits

We prove here that orbits form an equivalence relation

lemma (in group_action) orbit_sym_aux:
assumes "g ∈ carrier G"

and "x ∈ E"
and "(φ g) x = y"
shows "(φ (inv g)) y = x"
⟨*proof*⟩

lemma (**in** group_action) orbit_refl:
"x ∈ E ⟹ x ∈ orbit G φ x"
⟨*proof*⟩

lemma (**in** group_action) orbit_sym:
assumes "x ∈ E" **and** "y ∈ E" **and** "y ∈ orbit G φ x"
shows "x ∈ orbit G φ y"
⟨*proof*⟩

lemma (**in** group_action) orbit_trans:
assumes "x ∈ E" "y ∈ E" "z ∈ E"
and "y ∈ orbit G φ x" "z ∈ orbit G φ y"
shows "z ∈ orbit G φ x"
⟨*proof*⟩

lemma (**in** group_action) orbits_as_classes:
"classes⦇ carrier = E, eq = λx. λy. y ∈ orbit G φ x ⦈ = orbits G E φ"
⟨*proof*⟩

theorem (**in** group_action) orbit_partition:
"partition E (orbits G E φ)"
⟨*proof*⟩

corollary (**in** group_action) orbits_coverture:
"⋃ (orbits G E φ) = E"
⟨*proof*⟩

corollary (**in** group_action) disjoint_union:
assumes "orb1 ∈ (orbits G E φ)" "orb2 ∈ (orbits G E φ)"
shows "(orb1 = orb2) ∨ (orb1 ∩ orb2) = {}"
⟨*proof*⟩

corollary (**in** group_action) disjoint_sum:
assumes "finite E"
shows "(∑orb∈(orbits G E φ). ∑x∈orb. f x) = (∑x∈E. f x)"
⟨*proof*⟩

23.2.1 Transitive Actions

Transitive actions have only one orbit

lemma (**in** transitive_action) all_equivalent:
"⟦ x ∈ E; y ∈ E ⟧ ⟹ x .=⦇carrier = E, eq = λx y. y ∈ orbit G φ x⦈ y"
⟨*proof*⟩

proposition (**in** transitive_action) one_orbit:
  assumes "E ≠ {}"
  shows "card (orbits G E φ) = 1"
⟨*proof*⟩

23.3 Stabilizers

We show that stabilizers are subgroups from the acting group

lemma (**in** group_action) stabilizer_subset:
  "stabilizer G φ x ⊆ carrier G"
  ⟨*proof*⟩

lemma (**in** group_action) stabilizer_m_closed:
  assumes "x ∈ E" "g1 ∈ (stabilizer G φ x)" "g2 ∈ (stabilizer G φ x)"
  shows "(g1 ⊗ g2) ∈ (stabilizer G φ x)"
⟨*proof*⟩

lemma (**in** group_action) stabilizer_one_closed:
  assumes "x ∈ E"
  shows "1 ∈ (stabilizer G φ x)"
⟨*proof*⟩

lemma (**in** group_action) stabilizer_m_inv_closed:
  assumes "x ∈ E" "g ∈ (stabilizer G φ x)"
  shows "(inv g) ∈ (stabilizer G φ x)"
⟨*proof*⟩

theorem (**in** group_action) stabilizer_subgroup:
  assumes "x ∈ E"
  shows "subgroup (stabilizer G φ x) G"
  ⟨*proof*⟩

23.4 The Orbit-Stabilizer Theorem

In this subsection, we prove the Orbit-Stabilizer theorem. Our approach is to show the existence of a bijection between "rcosets (stabilizer G phi x)" and "orbit G phi x". Then we use Lagrange's theorem to find the cardinal of the first set.

23.4.1 Rcosets - Supporting Lemmas

corollary (**in** group_action) stab_rcosets_not_empty:
  assumes "x ∈ E" "R ∈ rcosets (stabilizer G φ x)"
  shows "R ≠ {}"
  ⟨*proof*⟩

corollary (**in** group_action) diff_stabilizes:

assumes "x ∈ E" "R ∈ rcosets (stabilizer G φ x)"
shows "⋀g1 g2. ⟦ g1 ∈ R; g2 ∈ R ⟧ ⟹ g1 ⊗ (inv g2) ∈ stabilizer G φ x"
⟨*proof*⟩

23.4.2 Bijection Between Rcosets and an Orbit - Definition and Supporting Lemmas

definition
orb_stab_fun :: "[_, ('a ⇒ 'b ⇒ 'b), 'a set, 'b] ⇒ 'b"
where "orb_stab_fun G φ R x = (φ (inv$_G$ (SOME h. h ∈ R))) x"

lemma (**in** group_action) orbit_stab_fun_is_well_defined0:
assumes "x ∈ E" "R ∈ rcosets (stabilizer G φ x)"
shows "⋀g1 g2. ⟦ g1 ∈ R; g2 ∈ R ⟧ ⟹ (φ (inv g1)) x = (φ (inv g2)) x"
⟨*proof*⟩

lemma (**in** group_action) orbit_stab_fun_is_well_defined1:
assumes "x ∈ E" "R ∈ rcosets (stabilizer G φ x)"
shows "⋀g. g ∈ R ⟹ (φ (inv (SOME h. h ∈ R))) x = (φ (inv g)) x"
⟨*proof*⟩

lemma (**in** group_action) orbit_stab_fun_is_inj:
assumes "x ∈ E"
and "R1 ∈ rcosets (stabilizer G φ x)"
and "R2 ∈ rcosets (stabilizer G φ x)"
and "φ (inv (SOME h. h ∈ R1)) x = φ (inv (SOME h. h ∈ R2)) x"
shows "R1 = R2"
⟨*proof*⟩

lemma (**in** group_action) orbit_stab_fun_is_surj:
assumes "x ∈ E" "y ∈ orbit G φ x"
shows "∃R ∈ rcosets (stabilizer G φ x). φ (inv (SOME h. h ∈ R)) x = y"
⟨*proof*⟩

proposition (**in** group_action) orbit_stab_fun_is_bij:
assumes "x ∈ E"
shows "bij_betw (λR. (φ (inv (SOME h. h ∈ R))) x) (rcosets (stabilizer G φ x)) (orbit G φ x)"
⟨*proof*⟩

23.4.3 The Theorem

theorem (**in** group_action) orbit_stabilizer_theorem:
assumes "x ∈ E"
shows "card (orbit G φ x) * card (stabilizer G φ x) = order G"
⟨*proof*⟩

23.5 The Burnside's Lemma

23.5.1 Sums and Cardinals

lemma card_as_sums:
  assumes "A = {x ∈ B. P x}" "finite B"
  shows "card A = (∑x∈B. (if P x then 1 else 0))"
⟨*proof*⟩

lemma sum_invertion:
  "⟦ finite A; finite B ⟧ ⟹ (∑x∈A. ∑y∈B. f x y) = (∑y∈B. ∑x∈A. f x y)"
⟨*proof*⟩

lemma (**in** group_action) card_stablizer_sum:
  assumes "finite (carrier G)" "orb ∈ (orbits G E φ)"
  shows "(∑x ∈ orb. card (stabilizer G φ x)) = order G"
⟨*proof*⟩

23.5.2 The Lemma

theorem (**in** group_action) burnside:
  assumes "finite (carrier G)" "finite E"
  shows "card (orbits G E φ) * order G = (∑g ∈ carrier G. card(invariants E φ g))"
⟨*proof*⟩

23.6 Action by Conjugation

23.6.1 Action Over Itself

A Group Acts by Conjugation Over Itself

lemma (**in** group) conjugation_is_inj:
  assumes "g ∈ carrier G" "h1 ∈ carrier G" "h2 ∈ carrier G"
    and "g ⊗ h1 ⊗ (inv g) = g ⊗ h2 ⊗ (inv g)"
    shows "h1 = h2"
  ⟨*proof*⟩

lemma (**in** group) conjugation_is_surj:
  assumes "g ∈ carrier G" "h ∈ carrier G"
  shows "g ⊗ ((inv g) ⊗ h ⊗ g) ⊗ (inv g) = h"
  ⟨*proof*⟩

lemma (**in** group) conjugation_is_bij:
  assumes "g ∈ carrier G"
  shows "bij_betw (λh ∈ carrier G. g ⊗ h ⊗ (inv g)) (carrier G) (carrier G)"
          (**is** "bij_betw ?φ (carrier G) (carrier G)")
  ⟨*proof*⟩

lemma (**in** group) conjugation_is_hom:
"(λg. λh ∈ carrier G. g ⊗ h ⊗ inv g) ∈ hom G (BijGroup (carrier G))"
⟨*proof*⟩

theorem (**in** group) action_by_conjugation:
"group_action G (carrier G) (λg. (λh ∈ carrier G. g ⊗ h ⊗ (inv g)))"
⟨*proof*⟩

23.6.2 Action Over The Set of Subgroups

A Group Acts by Conjugation Over The Set of Subgroups

lemma (**in** group) subgroup_conjugation_is_inj_aux:
assumes "g ∈ carrier G" "H1 ⊆ carrier G" "H2 ⊆ carrier G"
and "g <# H1 #> (inv g) = g <# H2 #> (inv g)"
shows "H1 ⊆ H2"
⟨*proof*⟩

lemma (**in** group) subgroup_conjugation_is_inj:
assumes "g ∈ carrier G" "H1 ⊆ carrier G" "H2 ⊆ carrier G"
and "g <# H1 #> (inv g) = g <# H2 #> (inv g)"
shows "H1 = H2"
⟨*proof*⟩

lemma (**in** group) subgroup_conjugation_is_surj0:
assumes "g ∈ carrier G" "H ⊆ carrier G"
shows "g <# ((inv g) <# H #> g) #> (inv g) = H"
⟨*proof*⟩

lemma (**in** group) subgroup_conjugation_is_surj1:
assumes "g ∈ carrier G" "subgroup H G"
shows "subgroup ((inv g) <# H #> g) G"
⟨*proof*⟩

lemma (**in** group) subgroup_conjugation_is_surj2:
assumes "g ∈ carrier G" "subgroup H G"
shows "subgroup (g <# H #> (inv g)) G"
⟨*proof*⟩

lemma (**in** group) subgroup_conjugation_is_bij:
assumes "g ∈ carrier G"
shows "bij_betw (λH ∈ {H. subgroup H G}. g <# H #> (inv g)) {H. subgroup H G} {H. subgroup H G}"
(**is** "bij_betw ?φ {H. subgroup H G} {H. subgroup H G}")
⟨*proof*⟩

lemma (**in** group) subgroup_conjugation_is_hom:
"(λg. λH ∈ {H. subgroup H G}. g <# H #> (inv g)) ∈ hom G (BijGroup {H. subgroup H G})"
⟨*proof*⟩

theorem (in group) action_by_conjugation_on_subgroups_set:
"group_action G {H. subgroup H G} (λg. λH ∈ {H. subgroup H G}. g <# H #> (inv g))"
⟨*proof*⟩

23.6.3 Action Over The Power Set

A Group Acts by Conjugation Over The Power Set

lemma (in group) subset_conjugation_is_bij:
assumes "g ∈ carrier G"
shows "bij_betw (λH ∈ {H. H ⊆ carrier G}. g <# H #> (inv g)) {H. H ⊆ carrier G} {H. H ⊆ carrier G}"
(is "bij_betw ?φ {H. H ⊆ carrier G} {H. H ⊆ carrier G}")
⟨*proof*⟩

lemma (in group) subset_conjugation_is_hom:
"(λg. λH ∈ {H. H ⊆ carrier G}. g <# H #> (inv g)) ∈ hom G (BijGroup {H. H ⊆ carrier G})"
⟨*proof*⟩

theorem (in group) action_by_conjugation_on_power_set:
"group_action G {H. H ⊆ carrier G} (λg. λH ∈ {H. H ⊆ carrier G}. g <# H #> (inv g))"
⟨*proof*⟩

corollary (in group) normalizer_imp_subgroup:
assumes "H ⊆ carrier G"
shows "subgroup (normalizer G H) G"
⟨*proof*⟩

23.7 Subgroup of an Acting Group

A Subgroup of an Acting Group Induces an Action

lemma (in group_action) induced_homomorphism:
assumes "subgroup H G"
shows "φ ∈ hom (G (|carrier := H|)) (BijGroup E)"
⟨*proof*⟩

theorem (in group_action) induced_action:
assumes "subgroup H G"
shows "group_action (G (|carrier := H|)) E φ"
⟨*proof*⟩

end

24 The Zassenhaus Lemma

theory Zassenhaus
  imports Coset Group_Action
begin

Proves the second isomorphism theorem and the Zassenhaus lemma.

24.1 Lemmas about normalizer

lemma (**in** group) subgroup_in_normalizer:
  assumes "subgroup H G"
  shows "normal H (G(|carrier:= (normalizer G H)|))"
⟨*proof*⟩

lemma (**in** group) normal_imp_subgroup_normalizer:
  assumes "subgroup H G"
    and "N ◁ (G(|carrier := H|))"
  shows "subgroup H (G(|carrier := normalizer G N|))"
⟨*proof*⟩

24.2 Second Isomorphism Theorem

lemma (**in** group) mult_norm_subgroup:
  assumes "normal N G"
    and "subgroup H G"
  shows "subgroup (N<#>H) G" ⟨*proof*⟩

lemma (**in** group) mult_norm_sub_in_sub:
  assumes "normal N (G(|carrier:=K|))"
  assumes "subgroup H (G(|carrier:=K|))"
  assumes "subgroup K G"
  shows "subgroup (N<#>H) (G(|carrier:=K|))"
⟨*proof*⟩

lemma (**in** group) subgroup_of_normal_set_mult:
  assumes "normal N G"
and "subgroup H G"
shows "subgroup H (G(|carrier := N <#> H|))"
⟨*proof*⟩

lemma (**in** group) normal_in_normal_set_mult:
  assumes "normal N G"
and "subgroup H G"
shows "normal N (G(|carrier := N <#> H|))"
⟨*proof*⟩

proposition (in group) weak_snd_iso_thme:
assumes "subgroup H G"
and "N◁G"
shows "(G(|carrier := N<#>H|) Mod N ≅ G(|carrier:=H|) Mod (N∩H))"
⟨*proof*⟩

theorem (in group) snd_iso_thme:
assumes "subgroup H G"
and "subgroup N G"
and "subgroup H (G(|carrier:= (normalizer G N)|))"
shows "(G(|carrier:= N<#>H|) Mod N) ≅ (G(|carrier:= H|) Mod (H∩N))"
⟨*proof*⟩

corollary (in group) snd_iso_thme_recip :
assumes "subgroup H G"
and "subgroup N G"
and "subgroup H (G(|carrier:= (normalizer G N)|))"
shows "(G(|carrier:= H<#>N|) Mod N) ≅ (G(|carrier:= H|) Mod (H∩N))"
⟨*proof*⟩

24.3 The Zassenhaus Lemma

lemma (in group) distinc:
assumes "subgroup H G"
and "H1◁G(|carrier := H|)"
and "subgroup K G"
and "K1◁G(|carrier:=K|)"
shows "subgroup (H∩K) (G(|carrier:=(normalizer G (H1<#>(H∩K1))) |))"
⟨*proof*⟩

lemma (in group) preliminary1:
assumes "subgroup H G"
and "H1◁G(|carrier := H|)"
and "subgroup K G"
and "K1◁G(|carrier:=K|)"
shows " (H∩K) ∩ (H1<#>(H∩K1)) = (H1∩K)<#>(H∩K1)"
⟨*proof*⟩

lemma (in group) preliminary2:
assumes "subgroup H G"
and "H1◁G(|carrier := H|)"
and "subgroup K G"
and "K1◁G(|carrier:=K|)"
shows "(H1<#>(H∩K1)) ◁ G(|carrier:=(H1<#>(H∩K))|)"
⟨*proof*⟩

proposition (**in** group) Zassenhaus_1:
  assumes "subgroup H G"
    and "H1⊲G⦇carrier := H⦈"
    and "subgroup K G"
    and "K1⊲G⦇carrier:=K⦈"
  shows "(G⦇carrier:= H1 <#> (H∩K)⦈ Mod (H1<#>H∩K1)) ≅ (G⦇carrier:= (H∩K)⦈ Mod ((H1∩K)<#>(H∩K1)))"
⟨*proof*⟩

theorem (**in** group) Zassenhaus:
  assumes "subgroup H G"
    and "H1⊲G⦇carrier := H⦈"
    and "subgroup K G"
    and "K1⊲G⦇carrier:=K⦈"
  shows "(G⦇carrier:= H1 <#> (H∩K)⦈ Mod (H1<#>(H∩K1))) ≅
        (G⦇carrier:= K1 <#> (H∩K)⦈ Mod (K1<#>(K∩H1)))"
⟨*proof*⟩

end

25 Divisibility in monoids and rings

theory Divisibility
  imports "HOL-Combinatorics.List_Permutation" Coset Group
begin

26 Factorial Monoids

26.1 Monoids with Cancellation Law

locale monoid_cancel = monoid +
  assumes l_cancel: "⟦c ⊗ a = c ⊗ b; a ∈ carrier G; b ∈ carrier G; c ∈ carrier G⟧ ⟹ a = b"
    and r_cancel: "⟦a ⊗ c = b ⊗ c; a ∈ carrier G; b ∈ carrier G; c ∈ carrier G⟧ ⟹ a = b"

lemma (**in** monoid) monoid_cancelI:
  assumes l_cancel: "⋀a b c. ⟦c ⊗ a = c ⊗ b; a ∈ carrier G; b ∈ carrier G; c ∈ carrier G⟧ ⟹ a = b"
    and r_cancel: "⋀a b c. ⟦a ⊗ c = b ⊗ c; a ∈ carrier G; b ∈ carrier G; c ∈ carrier G⟧ ⟹ a = b"
  shows "monoid_cancel G"
    ⟨*proof*⟩

lemma (**in** monoid_cancel) is_monoid_cancel: "monoid_cancel G" ⟨*proof*⟩

sublocale group ⊆ monoid_cancel
⟨*proof*⟩

locale comm_monoid_cancel = monoid_cancel + comm_monoid

lemma comm_monoid_cancelI:
fixes G (**structure**)
assumes "comm_monoid G"
assumes cancel: "⋀a b c. ⟦a ⊗ c = b ⊗ c; a ∈ carrier G; b ∈ carrier G; c ∈ carrier G⟧ ⟹ a = b"
shows "comm_monoid_cancel G"
⟨*proof*⟩

lemma (**in** comm_monoid_cancel) is_comm_monoid_cancel: "comm_monoid_cancel G"
⟨*proof*⟩

sublocale comm_group ⊆ comm_monoid_cancel ⟨*proof*⟩

26.2 Products of Units in Monoids

lemma (**in** monoid) prod_unit_l:
assumes abunit[simp]: "a ⊗ b ∈ Units G"
and aunit[simp]: "a ∈ Units G"
and carr[simp]: "a ∈ carrier G" "b ∈ carrier G"
shows "b ∈ Units G"
⟨*proof*⟩

lemma (**in** monoid) prod_unit_r:
assumes abunit[simp]: "a ⊗ b ∈ Units G"
and bunit[simp]: "b ∈ Units G"
and carr[simp]: "a ∈ carrier G" "b ∈ carrier G"
shows "a ∈ Units G"
⟨*proof*⟩

lemma (**in** comm_monoid) unit_factor:
assumes abunit: "a ⊗ b ∈ Units G"
and [simp]: "a ∈ carrier G" "b ∈ carrier G"
shows "a ∈ Units G"
⟨*proof*⟩

26.3 Divisibility and Association

26.3.1 Function definitions

definition factor :: "[_, 'a, 'a] ⇒ bool" (**infix** <dividesʔ> 65)
where "a dividesG b ⟷ (∃c∈carrier G. b = a ⊗G c)"

definition associated :: "[_, 'a, 'a] ⇒ bool" (**infix** ‹∼ı› 55)
  where "a ∼$_G$ b ⟷ a divides$_G$ b ∧ b divides$_G$ a"

abbreviation "division_rel G ≡ ⦇carrier = carrier G, eq = (∼$_G$), le = (divides$_G$)⦈"

definition properfactor :: "[_, 'a, 'a] ⇒ bool"
  where "properfactor G a b ⟷ a divides$_G$ b ∧ ¬(b divides$_G$ a)"

definition irreducible :: "[_, 'a] ⇒ bool"
  where "irreducible G a ⟷ a ∉ Units G ∧ (∀b∈carrier G. properfactor G b a ⟶ b ∈ Units G)"

definition prime :: "[_, 'a] ⇒ bool"
  where "prime G p ⟷
    p ∉ Units G ∧
    (∀a∈carrier G. ∀b∈carrier G. p divides$_G$ (a ⊗$_G$ b) ⟶ p divides$_G$ a ∨ p divides$_G$ b)"

26.3.2 Divisibility

lemma dividesI:
  fixes G (**structure**)
  assumes carr: "c ∈ carrier G"
    and p: "b = a ⊗ c"
  shows "a divides b"
  ⟨*proof*⟩

lemma dividesI' [intro]:
  fixes G (**structure**)
  assumes p: "b = a ⊗ c"
    and carr: "c ∈ carrier G"
  shows "a divides b"
  ⟨*proof*⟩

lemma dividesD:
  fixes G (**structure**)
  assumes "a divides b"
  shows "∃c∈carrier G. b = a ⊗ c"
  ⟨*proof*⟩

lemma dividesE [elim]:
  fixes G (**structure**)
  assumes d: "a divides b"
    and elim: "⋀c. ⟦b = a ⊗ c; c ∈ carrier G⟧ ⟹ P"
  shows "P"
⟨*proof*⟩

lemma (**in** monoid) divides_refl[simp, intro!]:

assumes carr: "a ∈ carrier G"
shows "a divides a"
⟨*proof*⟩

lemma (**in** monoid) divides_trans [trans]:
assumes dvds: "a divides b" "b divides c"
and acarr: "a ∈ carrier G"
shows "a divides c"
⟨*proof*⟩

lemma (**in** monoid) divides_mult_lI [intro]:
assumes "a divides b" "a ∈ carrier G" "c ∈ carrier G"
shows "(c ⊗ a) divides (c ⊗ b)"
⟨*proof*⟩

lemma (**in** monoid_cancel) divides_mult_l [simp]:
assumes carr: "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier G"
shows "(c ⊗ a) divides (c ⊗ b) = a divides b"
⟨*proof*⟩

lemma (**in** comm_monoid) divides_mult_rI [intro]:
assumes ab: "a divides b"
and carr: "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier G"
shows "(a ⊗ c) divides (b ⊗ c)"
⟨*proof*⟩

lemma (**in** comm_monoid_cancel) divides_mult_r [simp]:
assumes carr: "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier G"
shows "(a ⊗ c) divides (b ⊗ c) = a divides b"
⟨*proof*⟩

lemma (**in** monoid) divides_prod_r:
assumes ab: "a divides b"
and carr: "a ∈ carrier G" "c ∈ carrier G"
shows "a divides (b ⊗ c)"
⟨*proof*⟩

lemma (**in** comm_monoid) divides_prod_l:
assumes "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier G" "a divides b"
shows "a divides (c ⊗ b)"
⟨*proof*⟩

lemma (**in** monoid) unit_divides:
assumes uunit: "u ∈ Units G"
and acarr: "a ∈ carrier G"
shows "u divides a"
⟨*proof*⟩

lemma (in comm_monoid**)** divides_unit:
assumes udvd: "a divides u"
and carr: "a ∈ carrier G" "u ∈ Units G"
shows "a ∈ Units G"
⟨*proof*⟩

lemma (in comm_monoid**)** Unit_eq_dividesone:
assumes ucarr: "u ∈ carrier G"
shows "u ∈ Units G = u divides **1**"
⟨*proof*⟩

26.3.3 Association

lemma associatedI:
fixes G **(structure)**
assumes "a divides b" "b divides a"
shows "a ∼ b"
⟨*proof*⟩

lemma (in monoid**)** associatedI2:
assumes uunit[simp]: "u ∈ Units G"
and a: "a = b ⊗ u"
and bcarr: "b ∈ carrier G"
shows "a ∼ b"
⟨*proof*⟩

lemma (in monoid**)** associatedI2':
assumes "a = b ⊗ u"
and "u ∈ Units G"
and "b ∈ carrier G"
shows "a ∼ b"
⟨*proof*⟩

lemma associatedD:
fixes G **(structure)**
assumes "a ∼ b"
shows "a divides b"
⟨*proof*⟩

lemma (in monoid_cancel**)** associatedD2:
assumes assoc: "a ∼ b"
and carr: "a ∈ carrier G" "b ∈ carrier G"
shows "∃u∈Units G. a = b ⊗ u"
⟨*proof*⟩

lemma associatedE:
fixes G **(structure)**
assumes assoc: "a ∼ b"
and e: "⟦a divides b; b divides a⟧ ⟹ P"

shows "P"
⟨*proof*⟩

lemma (**in** monoid_cancel) associatedE2:
assumes assoc: "a ∼ b"
and e: "⋀u. ⟦a = b ⊗ u; u ∈ Units G⟧ ⟹ P"
and carr: "a ∈ carrier G" "b ∈ carrier G"
shows "P"
⟨*proof*⟩

lemma (**in** monoid) associated_refl [simp, intro!]:
assumes "a ∈ carrier G"
shows "a ∼ a"
⟨*proof*⟩

lemma (**in** monoid) associated_sym [sym]:
assumes "a ∼ b"
shows "b ∼ a"
⟨*proof*⟩

lemma (**in** monoid) associated_trans [trans]:
assumes "a ∼ b" "b ∼ c"
and "a ∈ carrier G" "c ∈ carrier G"
shows "a ∼ c"
⟨*proof*⟩

lemma (**in** monoid) division_equiv [intro, simp]: "equivalence (division_rel G)"
⟨*proof*⟩

26.3.4 Division and associativity

lemmas divides_antisym = associatedI

lemma (**in** monoid) divides_cong_l [trans]:
assumes "x ∼ x'" "x' divides y" "x ∈ carrier G"
shows "x divides y"
⟨*proof*⟩

lemma (**in** monoid) divides_cong_r [trans]:
assumes "x divides y" "y ∼ y'" "x ∈ carrier G"
shows "x divides y'"
⟨*proof*⟩

lemma (**in** monoid) division_weak_partial_order [simp, intro!]:
"weak_partial_order (division_rel G)"
⟨*proof*⟩

26.3.5 Multiplication and associativity

lemma (in monoid**)** mult_cong_r:
assumes "b ∼ b'" "a ∈ carrier G" "b ∈ carrier G" "b' ∈ carrier G"
shows "a ⊗ b ∼ a ⊗ b'"
⟨*proof*⟩

lemma (in comm_monoid**)** mult_cong_l:
assumes "a ∼ a'" "a ∈ carrier G" "a' ∈ carrier G" "b ∈ carrier G"
shows "a ⊗ b ∼ a' ⊗ b"
⟨*proof*⟩

lemma (in monoid_cancel**)** assoc_l_cancel:
assumes "a ∈ carrier G" "b ∈ carrier G" "b' ∈ carrier G" "a ⊗ b ∼ a ⊗ b'"
shows "b ∼ b'"
⟨*proof*⟩

lemma (in comm_monoid_cancel**)** assoc_r_cancel:
assumes "a ⊗ b ∼ a' ⊗ b" "a ∈ carrier G" "a' ∈ carrier G" "b ∈ carrier G"
shows "a ∼ a'"
⟨*proof*⟩

26.3.6 Units

lemma (in monoid_cancel**)** assoc_unit_l [trans]:
assumes "a ∼ b"
and "b ∈ Units G"
and "a ∈ carrier G"
shows "a ∈ Units G"
⟨*proof*⟩

lemma (in monoid_cancel**)** assoc_unit_r [trans]:
assumes aunit: "a ∈ Units G"
and asc: "a ∼ b"
and bcarr: "b ∈ carrier G"
shows "b ∈ Units G"
⟨*proof*⟩

lemma (in comm_monoid**)** Units_cong:
assumes aunit: "a ∈ Units G" **and** asc: "a ∼ b"
and bcarr: "b ∈ carrier G"
shows "b ∈ Units G"
⟨*proof*⟩

lemma (in monoid**)** Units_assoc:
assumes units: "a ∈ Units G" "b ∈ Units G"
shows "a ∼ b"
⟨*proof*⟩

lemma (**in** monoid) Units_are_ones: "Units G {.=}$_{\text{(division_rel G)}}$ {1}"
⟨*proof*⟩

lemma (**in** comm_monoid) Units_Lower: "Units G = Lower (division_rel G) (carrier G)"
⟨*proof*⟩

lemma (**in** monoid_cancel) associated_iff:
assumes "a ∈ carrier G" "b ∈ carrier G"
shows "a ∼ b ⟷ (∃c ∈ Units G. a = b ⊗ c)"
⟨*proof*⟩

26.3.7 Proper factors

lemma properfactorI:
fixes G (**structure**)
assumes "a divides b"
and "¬(b divides a)"
shows "properfactor G a b"
⟨*proof*⟩

lemma properfactorI2:
fixes G (**structure**)
assumes advdb: "a divides b"
and neq: "¬(a ∼ b)"
shows "properfactor G a b"
⟨*proof*⟩

lemma (**in** comm_monoid_cancel) properfactorI3:
assumes p: "p = a ⊗ b"
and nunit: "b ∉ Units G"
and carr: "a ∈ carrier G" "b ∈ carrier G"
shows "properfactor G a p"
⟨*proof*⟩

lemma properfactorE:
fixes G (**structure**)
assumes pf: "properfactor G a b"
and r: "⟦a divides b; ¬(b divides a)⟧ ⟹ P"
shows "P"
⟨*proof*⟩

lemma properfactorE2:
fixes G (**structure**)
assumes pf: "properfactor G a b"
and elim: "⟦a divides b; ¬(a ∼ b)⟧ ⟹ P"
shows "P"
⟨*proof*⟩

lemma (in monoid**)** properfactor_unitE:
assumes uunit: "u ∈ Units G"
and pf: "properfactor G a u"
and acarr: "a ∈ carrier G"
shows "P"
⟨*proof*⟩

lemma (in monoid**)** properfactor_divides:
assumes pf: "properfactor G a b"
shows "a divides b"
⟨*proof*⟩

lemma (in monoid**)** properfactor_trans1 [trans]:
assumes "a divides b" "properfactor G b c" "a ∈ carrier G" "c ∈ carrier G"
shows "properfactor G a c"
⟨*proof*⟩

lemma (in monoid**)** properfactor_trans2 [trans]:
assumes "properfactor G a b" "b divides c" "a ∈ carrier G" "b ∈ carrier G"
shows "properfactor G a c"
⟨*proof*⟩

lemma properfactor_lless:
fixes G **(structure)**
shows "properfactor G = lless (division_rel G)"
⟨*proof*⟩

lemma (in monoid**)** properfactor_cong_l [trans]:
assumes x'x: "x' ∼ x"
and pf: "properfactor G x y"
and carr: "x ∈ carrier G" "x' ∈ carrier G" "y ∈ carrier G"
shows "properfactor G x' y"
⟨*proof*⟩

lemma (in monoid**)** properfactor_cong_r [trans]:
assumes pf: "properfactor G x y"
and yy': "y ∼ y'"
and carr: "x ∈ carrier G" "y ∈ carrier G" "y' ∈ carrier G"
shows "properfactor G x y'"
⟨*proof*⟩

lemma (in monoid_cancel**)** properfactor_mult_lI [intro]:
assumes ab: "properfactor G a b"
and carr: "a ∈ carrier G" "c ∈ carrier G"
shows "properfactor G (c ⊗ a) (c ⊗ b)"
⟨*proof*⟩

lemma (in monoid_cancel**)** properfactor_mult_l [simp]:
assumes carr: "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier G"
shows "properfactor G (c ⊗ a) (c ⊗ b) = properfactor G a b"
⟨*proof*⟩

lemma (in comm_monoid_cancel**)** properfactor_mult_rI [intro]:
assumes ab: "properfactor G a b"
and carr: "a ∈ carrier G" "c ∈ carrier G"
shows "properfactor G (a ⊗ c) (b ⊗ c)"
⟨*proof*⟩

lemma (in comm_monoid_cancel**)** properfactor_mult_r [simp]:
assumes carr: "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier G"
shows "properfactor G (a ⊗ c) (b ⊗ c) = properfactor G a b"
⟨*proof*⟩

lemma (in monoid**)** properfactor_prod_r:
assumes ab: "properfactor G a b"
and carr[simp]: "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier G"
shows "properfactor G a (b ⊗ c)"
⟨*proof*⟩

lemma (in comm_monoid**)** properfactor_prod_l:
assumes ab: "properfactor G a b"
and carr[simp]: "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier G"
shows "properfactor G a (c ⊗ b)"
⟨*proof*⟩

26.4 Irreducible Elements and Primes

26.4.1 Irreducible elements

lemma irreducibleI:
fixes G **(structure)**
assumes "a ∉ Units G"
and "⋀b. ⟦b ∈ carrier G; properfactor G b a⟧ ⟹ b ∈ Units G"
shows "irreducible G a"
⟨*proof*⟩

lemma irreducibleE:
fixes G **(structure)**
assumes irr: "irreducible G a"
and elim: "⟦a ∉ Units G; ∀b. b ∈ carrier G ∧ properfactor G b a ⟶
b ∈ Units G⟧ ⟹ P"
shows "P"
⟨*proof*⟩

lemma irreducibleD:
fixes G **(structure)**

assumes irr: "irreducible G a"
and pf: "properfactor G b a"
and bcarr: "b ∈ carrier G"
shows "b ∈ Units G"
⟨*proof*⟩

lemma (**in** monoid_cancel) irreducible_cong [trans]:
assumes "irreducible G a" "a ∼ a'" "a ∈ carrier G" "a' ∈ carrier G"
shows "irreducible G a'"
⟨*proof*⟩

lemma (**in** monoid) irreducible_prod_rI:
assumes "irreducible G a" "b ∈ Units G" "a ∈ carrier G" "b ∈ carrier G"
shows "irreducible G (a ⊗ b)"
⟨*proof*⟩

lemma (**in** comm_monoid) irreducible_prod_lI:
assumes birr: "irreducible G b"
and aunit: "a ∈ Units G"
and carr [simp]: "a ∈ carrier G" "b ∈ carrier G"
shows "irreducible G (a ⊗ b)"
⟨*proof*⟩

lemma (**in** comm_monoid_cancel) irreducible_prodE [elim]:
assumes irr: "irreducible G (a ⊗ b)"
and carr[simp]: "a ∈ carrier G" "b ∈ carrier G"
and e1: "⟦irreducible G a; b ∈ Units G⟧ ⟹ P"
and e2: "⟦a ∈ Units G; irreducible G b⟧ ⟹ P"
shows P
⟨*proof*⟩

lemma divides_irreducible_condition:
assumes "irreducible G r" **and** "a ∈ carrier G"
shows "a divides$_G$ r ⟹ a ∈ Units G ∨ a ∼$_G$ r"
⟨*proof*⟩

26.4.2 Prime elements

lemma primeI:
fixes G (**structure**)
assumes "p ∉ Units G"
and "⋀a b. ⟦a ∈ carrier G; b ∈ carrier G; p divides (a ⊗ b)⟧ ⟹ p divides a ∨ p divides b"
shows "prime G p"
⟨*proof*⟩

lemma primeE:

fixes G **(structure)**
assumes pprime: "prime G p"
and e: "⟦p ∉ Units G; ∀a∈carrier G. ∀b∈carrier G.
p divides a ⊗ b ⟶ p divides a ∨ p divides b⟧ ⟹ P"
shows "P"
⟨*proof*⟩

lemma (in comm_monoid_cancel**)** prime_divides:
assumes carr: "a ∈ carrier G" "b ∈ carrier G"
and pprime: "prime G p"
and pdvd: "p divides a ⊗ b"
shows "p divides a ∨ p divides b"
⟨*proof*⟩

lemma (in monoid_cancel**)** prime_cong [trans]:
assumes "prime G p"
and pp': "p ∼ p'" "p ∈ carrier G" "p' ∈ carrier G"
shows "prime G p'"
⟨*proof*⟩

lemma (in comm_monoid_cancel**)** prime_irreducible:
assumes "prime G p"
shows "irreducible G p"
⟨*proof*⟩

lemma (in comm_monoid_cancel**)** prime_pow_divides_iff:
assumes "p ∈ carrier G" "a ∈ carrier G" "b ∈ carrier G" **and** "prime G p" **and** "¬ (p divides a)"
shows "(p [^] (n :: nat)) divides (a ⊗ b) ⟷ (p [^] n) divides b"
⟨*proof*⟩

26.5 Factorization and Factorial Monoids

26.5.1 Function definitions

definition factors :: "('a, _) monoid_scheme ⇒ 'a list ⇒ 'a ⇒ bool"
where "factors G fs a ⟷ (∀x ∈ (set fs). irreducible G x) ∧ foldr (⊗G) fs 1G = a"

definition wfactors :: "('a, _) monoid_scheme ⇒ 'a list ⇒ 'a ⇒ bool"
where "wfactors G fs a ⟷ (∀x ∈ (set fs). irreducible G x) ∧ foldr (⊗G) fs 1G ∼G a"

abbreviation list_assoc :: "('a, _) monoid_scheme ⇒ 'a list ⇒ 'a list ⇒ bool" (**infix** ‹[∼]› 44)
where "list_assoc G ≡ list_all2 (∼G)"

definition essentially_equal :: "('a, _) monoid_scheme ⇒ 'a list ⇒ 'a list ⇒ bool"
where "essentially_equal G fs1 fs2 ⟷ (∃fs1'. fs1 <~~> fs1' ∧ fs1'

```
[∼]G fs2)"


locale factorial_monoid = comm_monoid_cancel +
  assumes factors_exist: "⟦a ∈ carrier G; a ∉ Units G⟧ ⟹ ∃fs. set fs
⊆ carrier G ∧ factors G fs a"
    and factors_unique:
      "⟦factors G fs a; factors G fs' a; a ∈ carrier G; a ∉ Units G;
        set fs ⊆ carrier G; set fs' ⊆ carrier G⟧ ⟹ essentially_equal
G fs fs'"
```

26.5.2 Comparing lists of elements

Association on lists

```
lemma (in monoid) listassoc_refl [simp, intro]:
  assumes "set as ⊆ carrier G"
  shows "as [∼] as"
  ⟨proof⟩

lemma (in monoid) listassoc_sym [sym]:
  assumes "as [∼] bs"
    and "set as ⊆ carrier G"
    and "set bs ⊆ carrier G"
  shows "bs [∼] as"
  ⟨proof⟩

lemma (in monoid) listassoc_trans [trans]:
  assumes "as [∼] bs" and "bs [∼] cs"
    and "set as ⊆ carrier G" and "set bs ⊆ carrier G" and "set cs ⊆
carrier G"
  shows "as [∼] cs"
  ⟨proof⟩

lemma (in monoid_cancel) irrlist_listassoc_cong:
  assumes "∀a∈set as. irreducible G a"
    and "as [∼] bs"
    and "set as ⊆ carrier G" and "set bs ⊆ carrier G"
  shows "∀a∈set bs. irreducible G a"
  ⟨proof⟩
```

Permutations

```
lemma perm_map [intro]:
  assumes p: "a <~~> b"
  shows "map f a <~~> map f b"
  ⟨proof⟩

lemma perm_map_switch:
  assumes m: "map f a = map f b" and p: "b <~~> c"
  shows "∃d. a <~~> d ∧ map f d = map f c"
```

⟨*proof*⟩

lemma (in monoid**)** perm_assoc_switch:
  assumes a:"as [∼] bs" **and** p: "bs <~~> cs"
  shows "∃bs'. as <~~> bs' ∧ bs' [∼] cs"
⟨*proof*⟩

lemma (in monoid**)** perm_assoc_switch_r:
  assumes p: "as <~~> bs" **and** a:"bs [∼] cs"
  shows "∃bs'. as [∼] bs' ∧ bs' <~~> cs"
  ⟨*proof*⟩

declare perm_sym [sym]

lemma perm_setP:
  assumes perm: "as <~~> bs"
    and as: "P (set as)"
  shows "P (set bs)"
  ⟨*proof*⟩

lemmas (in monoid**)** perm_closed = perm_setP[of _ _ "λas. as ⊆ carrier G"]

lemmas (in monoid**)** irrlist_perm_cong = perm_setP[of _ _ "λas. ∀a∈as. irreducible G a"]

Essentially equal factorizations

lemma (in monoid**)** essentially_equalI:
  assumes ex: "fs1 <~~> fs1'" "fs1' [∼] fs2"
  shows "essentially_equal G fs1 fs2"
  ⟨*proof*⟩

lemma (in monoid**)** essentially_equalE:
  assumes ee: "essentially_equal G fs1 fs2"
    and e: "⋀fs1'. ⟦fs1 <~~> fs1'; fs1' [∼] fs2⟧ ⟹ P"
  shows "P"
  ⟨*proof*⟩

lemma (in monoid**)** ee_refl [simp,intro]:
  assumes carr: "set as ⊆ carrier G"
  shows "essentially_equal G as as"
  ⟨*proof*⟩

lemma (in monoid**)** ee_sym [sym]:
  assumes ee: "essentially_equal G as bs"
    and carr: "set as ⊆ carrier G" "set bs ⊆ carrier G"
  shows "essentially_equal G bs as"
  ⟨*proof*⟩

lemma (in monoid**)** ee_trans [trans]:
assumes ab: "essentially_equal G as bs" **and** bc: "essentially_equal G bs cs"
and ascarr: "set as ⊆ carrier G"
and bscarr: "set bs ⊆ carrier G"
and cscarr: "set cs ⊆ carrier G"
shows "essentially_equal G as cs"
⟨*proof*⟩

26.5.3 Properties of lists of elements

Multiplication of factors in a list

lemma (in monoid**)** multlist_closed [simp, intro]:
assumes ascarr: "set fs ⊆ carrier G"
shows "foldr (⊗) fs **1** ∈ carrier G"
⟨*proof*⟩

lemma (in comm_monoid**)** multlist_dividesI:
assumes "f ∈ set fs" **and** "set fs ⊆ carrier G"
shows "f divides (foldr (⊗) fs **1**)"
⟨*proof*⟩

lemma (in comm_monoid_cancel**)** multlist_listassoc_cong:
assumes "fs [∼] fs'"
and "set fs ⊆ carrier G" **and** "set fs' ⊆ carrier G"
shows "foldr (⊗) fs **1** ∼ foldr (⊗) fs' **1**"
⟨*proof*⟩

lemma (in comm_monoid**)** multlist_perm_cong:
assumes prm: "as <~~> bs"
and ascarr: "set as ⊆ carrier G"
shows "foldr (⊗) as **1** = foldr (⊗) bs **1**"
⟨*proof*⟩

lemma (in comm_monoid_cancel**)** multlist_ee_cong:
assumes "essentially_equal G fs fs'"
and "set fs ⊆ carrier G" **and** "set fs' ⊆ carrier G"
shows "foldr (⊗) fs **1** ∼ foldr (⊗) fs' **1**"
⟨*proof*⟩

26.5.4 Factorization in irreducible elements

lemma wfactorsI:
fixes G **(structure)**
assumes "∀f∈set fs. irreducible G f"
and "foldr (⊗) fs **1** ∼ a"
shows "wfactors G fs a"
⟨*proof*⟩

lemma wfactorsE:
  fixes G (**structure**)
  assumes wf: "wfactors G fs a"
    and e: "⟦∀f∈set fs. irreducible G f; foldr (⊗) fs **1** ∼ a⟧ ⟹ P"
  shows "P"
  ⟨*proof*⟩

lemma (**in** monoid) factorsI:
  assumes "∀f∈set fs. irreducible G f"
    and "foldr (⊗) fs **1** = a"
  shows "factors G fs a"
  ⟨*proof*⟩

lemma factorsE:
  fixes G (**structure**)
  assumes f: "factors G fs a"
    and e: "⟦∀f∈set fs. irreducible G f; foldr (⊗) fs **1** = a⟧ ⟹ P"
  shows "P"
  ⟨*proof*⟩

lemma (**in** monoid) factors_wfactors:
  assumes "factors G as a" **and** "set as ⊆ carrier G"
  shows "wfactors G as a"
  ⟨*proof*⟩

lemma (**in** monoid) wfactors_factors:
  assumes "wfactors G as a" **and** "set as ⊆ carrier G"
  shows "∃a'. factors G as a' ∧ a' ∼ a"
  ⟨*proof*⟩

lemma (**in** monoid) factors_closed [dest]:
  assumes "factors G fs a" **and** "set fs ⊆ carrier G"
  shows "a ∈ carrier G"
  ⟨*proof*⟩

lemma (**in** monoid) nunit_factors:
  assumes anunit: "a ∉ Units G"
    and fs: "factors G as a"
  shows "length as > 0"
⟨*proof*⟩

lemma (**in** monoid) unit_wfactors [simp]:
  assumes aunit: "a ∈ Units G"
  shows "wfactors G [] a"
  ⟨*proof*⟩

lemma (**in** comm_monoid_cancel) unit_wfactors_empty:
  assumes aunit: "a ∈ Units G"
    and wf: "wfactors G fs a"

```
    and carr[simp]: "set fs ⊆ carrier G"
  shows "fs = []"
```
⟨*proof*⟩

Comparing wfactors

```
lemma (in comm_monoid_cancel) wfactors_listassoc_cong_l:
  assumes fact: "wfactors G fs a"
    and asc: "fs [∼] fs'"
    and carr: "a ∈ carrier G"  "set fs ⊆ carrier G"  "set fs' ⊆ carrier
G"
  shows "wfactors G fs' a"
```
⟨*proof*⟩

```
lemma (in comm_monoid) wfactors_perm_cong_l:
  assumes "wfactors G fs a"
    and "fs <~~> fs'"
    and "set fs ⊆ carrier G"
  shows "wfactors G fs' a"
```
⟨*proof*⟩

```
lemma (in comm_monoid_cancel) wfactors_ee_cong_l [trans]:
  assumes ee: "essentially_equal G as bs"
    and bfs: "wfactors G bs b"
    and carr: "b ∈ carrier G"  "set as ⊆ carrier G"  "set bs ⊆ carrier
G"
  shows "wfactors G as b"
```
⟨*proof*⟩

```
lemma (in monoid) wfactors_cong_r [trans]:
  assumes fac: "wfactors G fs a" and aa': "a ∼ a'"
    and carr[simp]: "a ∈ carrier G"  "a' ∈ carrier G"  "set fs ⊆ carrier
G"
  shows "wfactors G fs a'"
```
⟨*proof*⟩

26.5.5 Essentially equal factorizations

```
lemma (in comm_monoid_cancel) unitfactor_ee:
  assumes uunit: "u ∈ Units G"
    and carr: "set as ⊆ carrier G"
  shows "essentially_equal G (as[0 := (as!0 ⊗ u)]) as"
    (is "essentially_equal G ?as' as")
```
⟨*proof*⟩

```
lemma (in comm_monoid_cancel) factors_cong_unit:
  assumes u: "u ∈ Units G"
    and a: "a ∉ Units G"
    and afs: "factors G as a"
    and ascarr: "set as ⊆ carrier G"
```

```
  shows "factors G (as[0 := (as!0 ⊗ u)]) (a ⊗ u)"
    (is "factors G ?as' ?a'")
⟨proof⟩

lemma (in comm_monoid) perm_wfactorsD:
  assumes prm: "as <~~> bs"
    and afs: "wfactors G as a"
    and bfs: "wfactors G bs b"
    and [simp]: "a ∈ carrier G"  "b ∈ carrier G"
    and ascarr [simp]: "set as ⊆ carrier G"
  shows "a ∼ b"
  ⟨proof⟩

lemma (in comm_monoid_cancel) listassoc_wfactorsD:
  assumes assoc: "as [∼] bs"
    and afs: "wfactors G as a"
    and bfs: "wfactors G bs b"
    and [simp]: "a ∈ carrier G"  "b ∈ carrier G"
    and [simp]: "set as ⊆ carrier G"  "set bs ⊆ carrier G"
  shows "a ∼ b"
  ⟨proof⟩

lemma (in comm_monoid_cancel) ee_wfactorsD:
  assumes ee: "essentially_equal G as bs"
    and afs: "wfactors G as a" and bfs: "wfactors G bs b"
    and [simp]: "a ∈ carrier G"  "b ∈ carrier G"
    and ascarr[simp]: "set as ⊆ carrier G" and bscarr[simp]: "set bs
⊆ carrier G"
  shows "a ∼ b"
  ⟨proof⟩

lemma (in comm_monoid_cancel) ee_factorsD:
  assumes ee: "essentially_equal G as bs"
    and afs: "factors G as a" and bfs:"factors G bs b"
    and "set as ⊆ carrier G"  "set bs ⊆ carrier G"
  shows "a ∼ b"
  ⟨proof⟩

lemma (in factorial_monoid) ee_factorsI:
  assumes ab: "a ∼ b"
    and afs: "factors G as a" and anunit: "a ∉ Units G"
    and bfs: "factors G bs b" and bnunit: "b ∉ Units G"
    and ascarr: "set as ⊆ carrier G" and bscarr: "set bs ⊆ carrier G"
  shows "essentially_equal G as bs"
⟨proof⟩

lemma (in factorial_monoid) ee_wfactorsI:
  assumes asc: "a ∼ b"
    and asf: "wfactors G as a" and bsf: "wfactors G bs b"
```

```
    and acarr[simp]: "a ∈ carrier G" and bcarr[simp]: "b ∈ carrier G"
    and ascarr[simp]: "set as ⊆ carrier G" and bscarr[simp]: "set bs
⊆ carrier G"
  shows "essentially_equal G as bs"
```
⟨*proof*⟩

lemma (in factorial_monoid**)** ee_wfactors:
```
  assumes asf: "wfactors G as a"
    and bsf: "wfactors G bs b"
    and acarr: "a ∈ carrier G" and bcarr: "b ∈ carrier G"
    and ascarr: "set as ⊆ carrier G" and bscarr: "set bs ⊆ carrier G"
  shows asc: "a ∼ b = essentially_equal G as bs"
```
⟨*proof*⟩

lemma (in factorial_monoid**)** wfactors_exist [intro, simp]:
```
  assumes acarr[simp]: "a ∈ carrier G"
  shows "∃fs. set fs ⊆ carrier G ∧ wfactors G fs a"
```
⟨*proof*⟩

lemma (in monoid**)** wfactors_prod_exists [intro, simp]:
```
  assumes "∀a ∈ set as. irreducible G a" and "set as ⊆ carrier G"
  shows "∃a. a ∈ carrier G ∧ wfactors G as a"
```
⟨*proof*⟩

lemma (in factorial_monoid**)** wfactors_unique:
```
  assumes "wfactors G fs a"
    and "wfactors G fs' a"
    and "a ∈ carrier G"
    and "set fs ⊆ carrier G"
    and "set fs' ⊆ carrier G"
  shows "essentially_equal G fs fs'"
```
⟨*proof*⟩

lemma (in monoid**)** factors_mult_single:
```
  assumes "irreducible G a" and "factors G fb b" and "a ∈ carrier G"
  shows "factors G (a # fb) (a ⊗ b)"
```
⟨*proof*⟩

lemma (in monoid_cancel**)** wfactors_mult_single:
```
  assumes f: "irreducible G a"  "wfactors G fb b"
    "a ∈ carrier G"  "b ∈ carrier G"  "set fb ⊆ carrier G"
  shows "wfactors G (a # fb) (a ⊗ b)"
```
⟨*proof*⟩

lemma (in monoid**)** factors_mult:
```
  assumes factors: "factors G fa a"  "factors G fb b"
    and ascarr: "set fa ⊆ carrier G"
    and bscarr: "set fb ⊆ carrier G"
  shows "factors G (fa @ fb) (a ⊗ b)"
```

⟨*proof*⟩

lemma (in comm_monoid_cancel**)** wfactors_mult [intro]:
  assumes asf: "wfactors G as a" **and** bsf:"wfactors G bs b"
    and acarr: "a ∈ carrier G" **and** bcarr: "b ∈ carrier G"
    and ascarr: "set as ⊆ carrier G" **and** bscarr:"set bs ⊆ carrier G"
  shows "wfactors G (as @ bs) (a ⊗ b)"
  ⟨*proof*⟩

lemma (in comm_monoid**)** factors_dividesI:
  assumes "factors G fs a"
    and "f ∈ set fs"
    and "set fs ⊆ carrier G"
  shows "f divides a"
  ⟨*proof*⟩

lemma (in comm_monoid**)** wfactors_dividesI:
  assumes p: "wfactors G fs a"
    and fscarr: "set fs ⊆ carrier G" **and** acarr: "a ∈ carrier G"
    and f: "f ∈ set fs"
  shows "f divides a"
  ⟨*proof*⟩

26.5.6 Factorial monoids and wfactors

lemma (in comm_monoid_cancel**)** factorial_monoidI:
  assumes wfactors_exists: "⋀a. ⟦ a ∈ carrier G; a ∉ Units G ⟧ ⟹ ∃fs. set fs ⊆ carrier G ∧ wfactors G fs a"
    and wfactors_unique:
      "⋀a fs fs'. ⟦a ∈ carrier G; set fs ⊆ carrier G; set fs' ⊆ carrier G;
        wfactors G fs a; wfactors G fs' a⟧ ⟹ essentially_equal G fs fs'"
  shows "factorial_monoid G"
⟨*proof*⟩

26.6 Factorizations as Multisets

Gives useful operations like intersection

abbreviation "assocs G x ≡ eq_closure_of (division_rel G) {x}"

definition "fmset G as = mset (map (assocs G) as)"

Helper lemmas

lemma (in monoid**)** assocs_repr_independence:
  assumes "y ∈ assocs G x" "x ∈ carrier G"
  shows "assocs G x = assocs G y"
  ⟨*proof*⟩

lemma (in monoid**)** assocs_self:
assumes "x ∈ carrier G"
shows "x ∈ assocs G x"
⟨*proof*⟩

lemma (in monoid**)** assocs_repr_independenceD:
assumes repr: "assocs G x = assocs G y" **and** ycarr: "y ∈ carrier G"
shows "y ∈ assocs G x"
⟨*proof*⟩

lemma (in comm_monoid**)** assocs_assoc:
assumes "a ∈ assocs G b" "b ∈ carrier G"
shows "a ∼ b"
⟨*proof*⟩

lemmas (in comm_monoid**)** assocs_eqD = assocs_repr_independenceD[THEN assocs_assoc]

26.6.1 Comparing multisets

lemma (in monoid**)** fmset_perm_cong:
assumes prm: "as <~~> bs"
shows "fmset G as = fmset G bs"
⟨*proof*⟩

lemma (in comm_monoid_cancel**)** eqc_listassoc_cong:
assumes "as [∼] bs" **and** "set as ⊆ carrier G" **and** "set bs ⊆ carrier G"
shows "map (assocs G) as = map (assocs G) bs"
⟨*proof*⟩

lemma (in comm_monoid_cancel**)** fmset_listassoc_cong:
assumes "as [∼] bs"
and "set as ⊆ carrier G" **and** "set bs ⊆ carrier G"
shows "fmset G as = fmset G bs"
⟨*proof*⟩

lemma (in comm_monoid_cancel**)** ee_fmset:
assumes ee: "essentially_equal G as bs"
and ascarr: "set as ⊆ carrier G" **and** bscarr: "set bs ⊆ carrier G"
shows "fmset G as = fmset G bs"
⟨*proof*⟩

lemma (in comm_monoid_cancel**)** fmset_ee:
assumes mset: "fmset G as = fmset G bs"
and ascarr: "set as ⊆ carrier G" **and** bscarr: "set bs ⊆ carrier G"
shows "essentially_equal G as bs"
⟨*proof*⟩

lemma (in comm_monoid_cancel**)** ee_is_fmset:

assumes "set as ⊆ carrier G" **and** "set bs ⊆ carrier G"
shows "essentially_equal G as bs = (fmset G as = fmset G bs)"
⟨*proof*⟩

26.6.2 Interpreting multisets as factorizations

lemma (in monoid**)** mset_fmsetEx:
assumes elems: "⋀X. X ∈ set_mset Cs ⟹ ∃x. P x ∧ X = assocs G x"
shows "∃cs. (∀c ∈ set cs. P c) ∧ fmset G cs = Cs"
⟨*proof*⟩

lemma (in monoid**)** mset_wfactorsEx:
assumes elems: "⋀X. X ∈ set_mset Cs ⟹ ∃x. (x ∈ carrier G ∧ irreducible G x) ∧ X = assocs G x"
shows "∃c cs. c ∈ carrier G ∧ set cs ⊆ carrier G ∧ wfactors G cs c ∧ fmset G cs = Cs"
⟨*proof*⟩

26.6.3 Multiplication on multisets

lemma (in factorial_monoid**)** mult_wfactors_fmset:
assumes afs: "wfactors G as a"
and bfs: "wfactors G bs b"
and cfs: "wfactors G cs (a ⊗ b)"
and carr: "a ∈ carrier G" "b ∈ carrier G"
"set as ⊆ carrier G" "set bs ⊆ carrier G" "set cs ⊆ carrier G"
shows "fmset G cs = fmset G as + fmset G bs"
⟨*proof*⟩

lemma (in factorial_monoid**)** mult_factors_fmset:
assumes afs: "factors G as a"
and bfs: "factors G bs b"
and cfs: "factors G cs (a ⊗ b)"
and "set as ⊆ carrier G" "set bs ⊆ carrier G" "set cs ⊆ carrier G"
shows "fmset G cs = fmset G as + fmset G bs"
⟨*proof*⟩

lemma (in comm_monoid_cancel**)** fmset_wfactors_mult:
assumes mset: "fmset G cs = fmset G as + fmset G bs"
and carr: "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier G"
"set as ⊆ carrier G" "set bs ⊆ carrier G" "set cs ⊆ carrier G"
and fs: "wfactors G as a" "wfactors G bs b" "wfactors G cs c"
shows "c ∼ a ⊗ b"
⟨*proof*⟩

26.6.4 Divisibility on multisets

lemma (in factorial_monoid**)** divides_fmsubset:

```
  assumes ab: "a divides b"
    and afs: "wfactors G as a"
    and bfs: "wfactors G bs b"
    and carr: "a ∈ carrier G"  "b ∈ carrier G"  "set as ⊆ carrier G"
"set bs ⊆ carrier G"
  shows "fmset G as ⊆# fmset G bs"
  ⟨proof⟩

lemma (in comm_monoid_cancel) fmsubset_divides:
  assumes msubset: "fmset G as ⊆# fmset G bs"
    and afs: "wfactors G as a"
    and bfs: "wfactors G bs b"
    and acarr: "a ∈ carrier G"
    and bcarr: "b ∈ carrier G"
    and ascarr: "set as ⊆ carrier G"
    and bscarr: "set bs ⊆ carrier G"
  shows "a divides b"
⟨proof⟩

lemma (in factorial_monoid) divides_as_fmsubset:
  assumes "wfactors G as a"
    and "wfactors G bs b"
    and "a ∈ carrier G"
    and "b ∈ carrier G"
    and "set as ⊆ carrier G"
    and "set bs ⊆ carrier G"
  shows "a divides b = (fmset G as ⊆# fmset G bs)"
  ⟨proof⟩
```

Proper factors on multisets

```
lemma (in factorial_monoid) fmset_properfactor:
  assumes asubb: "fmset G as ⊆# fmset G bs"
    and anb: "fmset G as ≠ fmset G bs"
    and "wfactors G as a"
    and "wfactors G bs b"
    and "a ∈ carrier G"
    and "b ∈ carrier G"
    and "set as ⊆ carrier G"
    and "set bs ⊆ carrier G"
  shows "properfactor G a b"
⟨proof⟩

lemma (in factorial_monoid) properfactor_fmset:
  assumes "properfactor G a b"
    and "wfactors G as a"
    and "wfactors G bs b"
    and "a ∈ carrier G"
    and "b ∈ carrier G"
    and "set as ⊆ carrier G"
```

and "set bs ⊆ carrier G"
shows "fmset G as ⊆# fmset G bs"
⟨*proof*⟩

lemma (**in** factorial_monoid) properfactor_fmset_ne:
assumes pf: "properfactor G a b"
and "wfactors G as a"
and "wfactors G bs b"
and "a ∈ carrier G"
and "b ∈ carrier G"
and "set as ⊆ carrier G"
and "set bs ⊆ carrier G"
shows "fmset G as ≠ fmset G bs"
⟨*proof*⟩

26.7 Irreducible Elements are Prime

lemma (**in** factorial_monoid) irreducible_prime:
assumes pirr: "irreducible G p" **and** pcarr: "p ∈ carrier G"
shows "prime G p"
⟨*proof*⟩
lemma (**in** factorial_monoid) factors_irreducible_prime:
assumes pirr: "irreducible G p" **and** pcarr: "p ∈ carrier G"
shows "prime G p"
⟨*proof*⟩

26.8 Greatest Common Divisors and Lowest Common Multiples

26.8.1 Definitions

definition isgcd :: "[('a,_) monoid_scheme, 'a, 'a, 'a] ⇒ bool"
(<(<notation=<mixfix gcdof>>_ gcdof↕ _ _)> [81,81,81] 80)
where "x gcdof$_G$ a b ⟷ x divides$_G$ a ∧ x divides$_G$ b ∧
(∀y∈carrier G. (y divides$_G$ a ∧ y divides$_G$ b ⟶ y divides$_G$ x))"

definition islcm :: "[_, 'a, 'a, 'a] ⇒ bool"
(<(<notation=<mixfix lcmof>>_ lcmof↕ _ _)> [81,81,81] 80)
where "x lcmof$_G$ a b ⟷ a divides$_G$ x ∧ b divides$_G$ x ∧
(∀y∈carrier G. (a divides$_G$ y ∧ b divides$_G$ y ⟶ x divides$_G$ y))"

definition somegcd :: "('a,_) monoid_scheme ⇒ 'a ⇒ 'a ⇒ 'a"
where "somegcd G a b = (SOME x. x ∈ carrier G ∧ x gcdof$_G$ a b)"

definition somelcm :: "('a,_) monoid_scheme ⇒ 'a ⇒ 'a ⇒ 'a"
where "somelcm G a b = (SOME x. x ∈ carrier G ∧ x lcmof$_G$ a b)"

definition "SomeGcd G A = Lattice.inf (division_rel G) A"

locale gcd_condition_monoid = comm_monoid_cancel +
  assumes gcdof_exists: "⟦a ∈ carrier G; b ∈ carrier G⟧ ⟹ ∃c. c ∈ carrier G ∧ c gcdof a b"

locale primeness_condition_monoid = comm_monoid_cancel +
  assumes irreducible_prime: "⟦a ∈ carrier G; irreducible G a⟧ ⟹ prime G a"

locale divisor_chain_condition_monoid = comm_monoid_cancel +
  assumes division_wellfounded: "wf {(x, y). x ∈ carrier G ∧ y ∈ carrier G ∧ properfactor G x y}"

26.8.2 Connections to Lattice.thy

lemma gcdof_greatestLower:
  fixes G (**structure**)
  assumes carr[simp]: "a ∈ carrier G" "b ∈ carrier G"
  shows "(x ∈ carrier G ∧ x gcdof a b) = greatest (division_rel G) x (Lower (division_rel G) {a, b})"
  ⟨*proof*⟩

lemma lcmof_leastUpper:
  fixes G (**structure**)
  assumes carr[simp]: "a ∈ carrier G" "b ∈ carrier G"
  shows "(x ∈ carrier G ∧ x lcmof a b) = least (division_rel G) x (Upper (division_rel G) {a, b})"
  ⟨*proof*⟩

lemma somegcd_meet:
  fixes G (**structure**)
  assumes carr: "a ∈ carrier G" "b ∈ carrier G"
  shows "somegcd G a b = meet (division_rel G) a b"
  ⟨*proof*⟩

lemma (**in** monoid) isgcd_divides_l:
  assumes "a divides b"
    and "a ∈ carrier G" "b ∈ carrier G"
  shows "a gcdof a b"
  ⟨*proof*⟩

lemma (**in** monoid) isgcd_divides_r:
  assumes "b divides a"
    and "a ∈ carrier G" "b ∈ carrier G"
  shows "b gcdof a b"
  ⟨*proof*⟩

26.8.3 Existence of gcd and lcm

lemma (**in** factorial_monoid) gcdof_exists:
  assumes acarr: "a ∈ carrier G"

and bcarr: "b ∈ carrier G"
shows "∃c. c ∈ carrier G ∧ c gcdof a b"
⟨*proof*⟩

lemma (**in** factorial_monoid) lcmof_exists:
assumes acarr: "a ∈ carrier G"
and bcarr: "b ∈ carrier G"
shows "∃c. c ∈ carrier G ∧ c lcmof a b"
⟨*proof*⟩

26.9 Conditions for Factoriality

26.9.1 Gcd condition

lemma (**in** gcd_condition_monoid) division_weak_lower_semilattice [simp]:
"weak_lower_semilattice (division_rel G)"
⟨*proof*⟩

lemma (**in** gcd_condition_monoid) gcdof_cong_l:
assumes "a' ∼ a" "a gcdof b c" "a' ∈ carrier G" **and** carr': "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier G"
shows "a' gcdof b c"
⟨*proof*⟩

lemma (**in** gcd_condition_monoid) gcd_closed [simp]:
assumes "a ∈ carrier G" "b ∈ carrier G"
shows "somegcd G a b ∈ carrier G"
⟨*proof*⟩

lemma (**in** gcd_condition_monoid) gcd_isgcd:
assumes "a ∈ carrier G" "b ∈ carrier G"
shows "(somegcd G a b) gcdof a b"
⟨*proof*⟩

lemma (**in** gcd_condition_monoid) gcd_exists:
assumes "a ∈ carrier G" "b ∈ carrier G"
shows "∃x∈carrier G. x = somegcd G a b"
⟨*proof*⟩

lemma (**in** gcd_condition_monoid) gcd_divides_l:
assumes "a ∈ carrier G" "b ∈ carrier G"
shows "(somegcd G a b) divides a"
⟨*proof*⟩

lemma (**in** gcd_condition_monoid) gcd_divides_r:
assumes "a ∈ carrier G" "b ∈ carrier G"
shows "(somegcd G a b) divides b"
⟨*proof*⟩

lemma (**in** gcd_condition_monoid) gcd_divides:

assumes "z divides x" "z divides y"
and L: "x ∈ carrier G" "y ∈ carrier G" "z ∈ carrier G"
shows "z divides (somegcd G x y)"
⟨*proof*⟩

lemma (**in** gcd_condition_monoid) gcd_cong_l:
assumes "x ∼ x'" "x ∈ carrier G" "x' ∈ carrier G" "y ∈ carrier G"
shows "somegcd G x y ∼ somegcd G x' y"
⟨*proof*⟩

lemma (**in** gcd_condition_monoid) gcd_cong_r:
assumes "y ∼ y'" "x ∈ carrier G" "y ∈ carrier G" "y' ∈ carrier G"
shows "somegcd G x y ∼ somegcd G x y'"
⟨*proof*⟩

lemma (**in** gcd_condition_monoid) gcdI:
assumes dvd: "a divides b" "a divides c"
and others: "⋀y. ⟦y∈carrier G; y divides b; y divides c⟧ ⟹ y divides a"
and acarr: "a ∈ carrier G" **and** bcarr: "b ∈ carrier G" **and** ccarr: "c ∈ carrier G"
shows "a ∼ somegcd G b c"
⟨*proof*⟩

lemma (**in** gcd_condition_monoid) gcdI2:
assumes "a gcdof b c" **and** "a ∈ carrier G" **and** "b ∈ carrier G" **and** "c ∈ carrier G"
shows "a ∼ somegcd G b c"
⟨*proof*⟩

lemma (**in** gcd_condition_monoid) SomeGcd_ex:
assumes "finite A" "A ⊆ carrier G" "A ≠ {}"
shows "∃x ∈ carrier G. x = SomeGcd G A"
⟨*proof*⟩

lemma (**in** gcd_condition_monoid) gcd_assoc:
assumes "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier G"
shows "somegcd G (somegcd G a b) c ∼ somegcd G a (somegcd G b c)"
⟨*proof*⟩

lemma (**in** gcd_condition_monoid) gcd_mult:
assumes acarr: "a ∈ carrier G" **and** bcarr: "b ∈ carrier G" **and** ccarr: "c ∈ carrier G"
shows "c ⊗ somegcd G a b ∼ somegcd G (c ⊗ a) (c ⊗ b)"
⟨*proof*⟩

lemma (**in** monoid) assoc_subst:
assumes ab: "a ∼ b"
and cP: "∀a b. a ∈ carrier G ∧ b ∈ carrier G ∧ a ∼ b

```
      ⟶ f a ∈ carrier G ∧ f b ∈ carrier G ∧ f a ∼ f b"
    and carr: "a ∈ carrier G"  "b ∈ carrier G"
  shows "f a ∼ f b"
  ⟨proof⟩

lemma (in gcd_condition_monoid) relprime_mult:
  assumes abrelprime: "somegcd G a b ∼ 1"
    and acrelprime: "somegcd G a c ∼ 1"
    and carr[simp]: "a ∈ carrier G"  "b ∈ carrier G"  "c ∈ carrier G"
  shows "somegcd G a (b ⊗ c) ∼ 1"
⟨proof⟩

lemma (in gcd_condition_monoid) primeness_condition: "primeness_condition_monoid
G"
⟨proof⟩


sublocale gcd_condition_monoid ⊆ primeness_condition_monoid
  ⟨proof⟩
```

26.9.2 Divisor chain condition

```
lemma (in divisor_chain_condition_monoid) wfactors_exist:
  assumes acarr: "a ∈ carrier G"
  shows "∃as. set as ⊆ carrier G ∧ wfactors G as a"
⟨proof⟩
```

26.9.3 Primeness condition

```
lemma (in comm_monoid_cancel) multlist_prime_pos:
  assumes aprime: "prime G a" and carr: "a ∈ carrier G"
     and as: "set as ⊆ carrier G" "a divides (foldr (⊗) as 1)"
   shows "∃i<length as. a divides (as!i)"
  ⟨proof⟩

proposition (in primeness_condition_monoid) wfactors_unique:
  assumes "wfactors G as a"  "wfactors G as' a"
    and "a ∈ carrier G"  "set as ⊆ carrier G"  "set as' ⊆ carrier G"
  shows "essentially_equal G as as'"
  ⟨proof⟩
```

26.9.4 Application to factorial monoids

Number of factors for wellfoundedness

```
definition factorcount :: "_ ⇒ 'a ⇒ nat"
  where "factorcount G a =
    (THE c. ∀as. set as ⊆ carrier G ∧ wfactors G as a ⟶ c = length
as)"
```

lemma (in monoid**)** ee_length:
  assumes ee: "essentially_equal G as bs"
  shows "length as = length bs"
  ⟨*proof*⟩

lemma (in factorial_monoid**)** factorcount_exists:
  assumes carr[simp]: "a ∈ carrier G"
  shows "∃c. ∀as. set as ⊆ carrier G ∧ wfactors G as a ⟶ c = length as"
⟨*proof*⟩

lemma (in factorial_monoid**)** factorcount_unique:
  assumes afs: "wfactors G as a"
    and acarr[simp]: "a ∈ carrier G" **and** ascarr: "set as ⊆ carrier G"
  shows "factorcount G a = length as"
⟨*proof*⟩

lemma (in factorial_monoid**)** divides_fcount:
  assumes dvd: "a divides b"
    and acarr: "a ∈ carrier G"
    and bcarr:"b ∈ carrier G"
  shows "factorcount G a ≤ factorcount G b"
⟨*proof*⟩

lemma (in factorial_monoid**)** associated_fcount:
  assumes acarr: "a ∈ carrier G"
    and bcarr: "b ∈ carrier G"
    and asc: "a ∼ b"
  shows "factorcount G a = factorcount G b"
  ⟨*proof*⟩

lemma (in factorial_monoid**)** properfactor_fcount:
  assumes acarr: "a ∈ carrier G" **and** bcarr:"b ∈ carrier G"
    and pf: "properfactor G a b"
  shows "factorcount G a < factorcount G b"
⟨*proof*⟩

sublocale factorial_monoid ⊆ divisor_chain_condition_monoid
  ⟨*proof*⟩

sublocale factorial_monoid ⊆ primeness_condition_monoid
  ⟨*proof*⟩

lemma (in factorial_monoid**)** primeness_condition: "primeness_condition_monoid G" ⟨*proof*⟩

lemma (in factorial_monoid**)** gcd_condition [simp]: "gcd_condition_monoid G"

⟨*proof*⟩

sublocale factorial_monoid ⊆ gcd_condition_monoid
⟨*proof*⟩

lemma (in factorial_monoid**)** division_weak_lattice [simp]: "weak_lattice (division_rel G)"
⟨*proof*⟩

26.10 Factoriality Theorems

theorem factorial_condition_one:
"divisor_chain_condition_monoid G ∧ primeness_condition_monoid G ⟷ factorial_monoid G"
⟨*proof*⟩

theorem factorial_condition_two:
"divisor_chain_condition_monoid G ∧ gcd_condition_monoid G ⟷ factorial_monoid G"
⟨*proof*⟩

end

theory QuotRing
imports RingHom
begin

27 Quotient Rings

27.1 Multiplication on Cosets

definition rcoset_mult :: "[('a, _) ring_scheme, 'a set, 'a set, 'a set] ⇒ 'a set"
(‹(‹open_block notation=‹mixfix rcoset_mult››[mod _:] _ ⨂ı _)› [81,81,81] 80)
where "rcoset_mult R I A B = (⋃a∈A. ⋃b∈B. I +>$_R$ (a ⊗$_R$ b))"

rcoset_mult fulfils the properties required by congruences

lemma (in ideal**)** rcoset_mult_add:
assumes "x ∈ carrier R" "y ∈ carrier R"
shows "[mod I:] (I +> x) ⨂ (I +> y) = I +> (x ⊗ y)"
⟨*proof*⟩

27.2 Quotient Ring Definition

definition FactRing :: "[('a,'b) ring_scheme, 'a set] ⇒ ('a set) ring"
(**infixl** ‹Quot› 65)
where "FactRing R I =

```
    (|carrier = a_rcosets_R I, mult = rcoset_mult R I,
      one = (I +>_R 1_R), zero = I, add = set_add R|)"

lemmas FactRing_simps = FactRing_def A_RCOSETS_defs a_r_coset_def[symmetric]
```

27.3 Factorization over General Ideals

The quotient is a ring

```
lemma (in ideal) quotient_is_ring: "ring (R Quot I)"
```
⟨*proof*⟩

This is a ring homomorphism

```
lemma (in ideal) rcos_ring_hom: "((+>) I) ∈ ring_hom R (R Quot I)"
```
⟨*proof*⟩

```
lemma (in ideal) rcos_ring_hom_ring: "ring_hom_ring R (R Quot I) ((+>)
I)"
```
⟨*proof*⟩

The quotient of a cring is also commutative

```
lemma (in ideal) quotient_is_cring:
  assumes "cring R"
  shows "cring (R Quot I)"
```
⟨*proof*⟩

Cosets as a ring homomorphism on crings

```
lemma (in ideal) rcos_ring_hom_cring:
  assumes "cring R"
  shows "ring_hom_cring R (R Quot I) ((+>) I)"
```
⟨*proof*⟩

27.4 Factorization over Prime Ideals

The quotient ring generated by a prime ideal is a domain

```
lemma (in primeideal) quotient_is_domain: "domain (R Quot I)"
```
⟨*proof*⟩

Generating right cosets of a prime ideal is a homomorphism on commutative rings

```
lemma (in primeideal) rcos_ring_hom_cring: "ring_hom_cring R (R Quot
I) ((+>) I)"
```
⟨*proof*⟩

27.5 Factorization over Maximal Ideals

In a commutative ring, the quotient ring over a maximal ideal is a field. The proof follows "W. Adkins, S. Weintraub: Algebra – An Approach via Module Theory"

proposition (**in** maximalideal) quotient_is_field:
assumes "cring R"
shows "field (R Quot I)"
⟨*proof*⟩

lemma (**in** ring_hom_ring) trivial_hom_iff:
"(h ‘ (carrier R) = { $\mathbf{0}_S$ }) = (a_kernel R S h = carrier R)"
⟨*proof*⟩

lemma (**in** ring_hom_ring) trivial_ker_imp_inj:
assumes "a_kernel R S h = { **0** }"
shows "inj_on h (carrier R)"
⟨*proof*⟩

lemma (**in** ring_hom_ring) inj_iff_trivial_ker:
shows "inj_on h (carrier R) ⟷ a_kernel R S h = { **0** }"
⟨*proof*⟩

corollary ring_hom_in_hom:
assumes "h ∈ ring_hom R S" **shows** "h ∈ hom R S" **and** "h ∈ hom (add_monoid R) (add_monoid S)"
⟨*proof*⟩

corollary set_add_hom:
assumes "h ∈ ring_hom R S" "I ⊆ carrier R" **and** "J ⊆ carrier R"
shows "h ‘ (I <+>$_R$ J) = h ‘ I <+>$_S$ h ‘ J"
⟨*proof*⟩

corollary a_coset_hom:
assumes "h ∈ ring_hom R S" "I ⊆ carrier R" "a ∈ carrier R"
shows "h ‘ (a <+$_R$ I) = h a <+$_S$ (h ‘ I)" **and** "h ‘ (I +>$_R$ a) = (h ‘ I) +>$_S$ h a"
⟨*proof*⟩

corollary (**in** ring_hom_ring) set_add_ker_hom:
assumes "I ⊆ carrier R"
shows "h ‘ (I <+> (a_kernel R S h)) = h ‘ I" **and** "h ‘ ((a_kernel R S h) <+> I) = h ‘ I"
⟨*proof*⟩

lemma (**in** ring_hom_ring) non_trivial_field_hom_imp_inj:
assumes R: "field R"
and h: "h ‘ (carrier R) ≠ { $\mathbf{0}_S$ }"

shows "inj_on h (carrier R)"
⟨*proof*⟩

lemma "field R ⟹ cring R"
⟨*proof*⟩

lemma non_trivial_field_hom_is_inj:
assumes "h ∈ ring_hom R S" **and** "field R" **and** "field S"
shows "inj_on h (carrier R)"
⟨*proof*⟩

lemma (**in** ring_hom_ring) img_is_add_subgroup:
assumes "subgroup H (add_monoid R)"
shows "subgroup (h ‘ H) (add_monoid S)"
⟨*proof*⟩

lemma (**in** ring) ring_ideal_imp_quot_ideal:
assumes "ideal I R"
and A: "ideal J R"
shows "ideal ((+>) I ‘ J) (R Quot I)"
⟨*proof*⟩

lemma (**in** ring_hom_ring) ideal_vimage:
assumes "ideal I S"
shows "ideal { r ∈ carrier R. h r ∈ I } R"
⟨*proof*⟩

lemma (**in** ring) canonical_proj_vimage_in_carrier:
assumes "ideal I R"
and A: "J ⊆ carrier (R Quot I)"
shows "⋃ J ⊆ carrier R"
⟨*proof*⟩

lemma (**in** ring) canonical_proj_vimage_mem_iff:
assumes "ideal I R" "J ⊆ carrier (R Quot I)"
and a: "a ∈ carrier R"
shows "(a ∈ ⋃ J) = (I +> a ∈ J)"
⟨*proof*⟩

corollary (**in** ring) quot_ideal_imp_ring_ideal:
assumes "ideal I R"
shows "ideal J (R Quot I) ⟹ ideal (⋃ J) R"
⟨*proof*⟩

lemma (**in** ring) ideal_incl_iff:
assumes "ideal I R" "ideal J R"
shows "(I ⊆ J) = (J = (⋃ j ∈ J. I +> j))"
⟨*proof*⟩

theorem (in ring**)** quot_ideal_correspondence:
assumes "ideal I R"
shows "bij_betw (λJ. (+>) I ‘ J) { J. ideal J R ∧ I ⊆ J } { J . ideal J (R Quot I) }"
⟨*proof*⟩

lemma (in cring**)** quot_domain_imp_primeideal:
assumes "ideal P R"
and A: "domain (R Quot P)"
shows "primeideal P R"
⟨*proof*⟩

lemma (in cring**)** quot_domain_iff_primeideal:
assumes "ideal P R"
shows "domain (R Quot P) = primeideal P R"
⟨*proof*⟩

27.6 Isomorphism

definition
ring_iso :: "_ ⇒ _ ⇒ ('a ⇒ 'b) set"
where "ring_iso R S = { h. h ∈ ring_hom R S ∧ bij_betw h (carrier R) (carrier S) }"

definition
is_ring_iso :: "_ ⇒ _ ⇒ bool" (**infixr** ‹≃› 60)
where "R ≃ S = (ring_iso R S ≠ {})"

definition
morphic_prop :: "_ ⇒ ('a ⇒ bool) ⇒ bool"
where "morphic_prop R P =
((P $\mathbf{1}_R$) ∧
(∀r ∈ carrier R. P r) ∧
(∀r1 ∈ carrier R. ∀r2 ∈ carrier R. P (r1 $\otimes_R$ r2)) ∧
(∀r1 ∈ carrier R. ∀r2 ∈ carrier R. P (r1 $\oplus_R$ r2)))"

lemma ring_iso_memI:
fixes R **(structure) and** S **(structure)**
assumes "⋀x. x ∈ carrier R ⟹ h x ∈ carrier S"
and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ h (x ⊗ y) = h x $\otimes_S$ h y"
and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ h (x ⊕ y) = h x $\oplus_S$ h y"
and "h $\mathbf{1}$ = $\mathbf{1}_S$"
and "bij_betw h (carrier R) (carrier S)"
shows "h ∈ ring_iso R S"
⟨*proof*⟩

lemma ring_iso_memE:

fixes R **(structure) and** S **(structure)**
assumes "h ∈ ring_iso R S"
shows "⋀x. x ∈ carrier R ⟹ h x ∈ carrier S"
and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ h (x ⊗ y) = h x ⊗$_S$ h y"
and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ h (x ⊕ y) = h x ⊕$_S$ h y"
and "h 1 = 1$_S$"
and "bij_betw h (carrier R) (carrier S)"
⟨*proof*⟩

lemma morphic_propI:
fixes R **(structure)**
assumes "P 1"
and "⋀r. r ∈ carrier R ⟹ P r"
and "⋀r1 r2. ⟦ r1 ∈ carrier R; r2 ∈ carrier R ⟧ ⟹ P (r1 ⊗ r2)"
and "⋀r1 r2. ⟦ r1 ∈ carrier R; r2 ∈ carrier R ⟧ ⟹ P (r1 ⊕ r2)"
shows "morphic_prop R P"
⟨*proof*⟩

lemma morphic_propE:
fixes R **(structure)**
assumes "morphic_prop R P"
shows "P 1"
and "⋀r. r ∈ carrier R ⟹ P r"
and "⋀r1 r2. ⟦ r1 ∈ carrier R; r2 ∈ carrier R ⟧ ⟹ P (r1 ⊗ r2)"
and "⋀r1 r2. ⟦ r1 ∈ carrier R; r2 ∈ carrier R ⟧ ⟹ P (r1 ⊕ r2)"
⟨*proof*⟩

lemma (in ring**)** ring_hom_restrict:
assumes "f ∈ ring_hom R S" **and** "⋀r. r ∈ carrier R ⟹ f r = g r" **shows** "g ∈ ring_hom R S"
⟨*proof*⟩

lemma (in ring**)** ring_iso_restrict:
assumes "f ∈ ring_iso R S" **and** "⋀r. r ∈ carrier R ⟹ f r = g r" **shows** "g ∈ ring_iso R S"
⟨*proof*⟩

lemma ring_iso_morphic_prop:
assumes "f ∈ ring_iso R S"
and "morphic_prop R P"
and "⋀r. P r ⟹ f r = g r"
shows "g ∈ ring_iso R S"
⟨*proof*⟩

lemma (in ring**)** ring_hom_imp_img_ring:

assumes "h ∈ ring_hom R S"
shows "ring (S (| carrier := h ‘ (carrier R), zero := h **0** |))" (**is** "ring ?h_img")
⟨*proof*⟩

lemma (**in** ring) ring_iso_imp_img_ring:
assumes "h ∈ ring_iso R S"
shows "ring (S (| zero := h **0** |))"
⟨*proof*⟩

lemma (**in** cring) ring_iso_imp_img_cring:
assumes "h ∈ ring_iso R S"
shows "cring (S (| zero := h **0** |))" (**is** "cring ?h_img")
⟨*proof*⟩

lemma (**in** domain) ring_iso_imp_img_domain:
assumes "h ∈ ring_iso R S"
shows "domain (S (| zero := h **0** |))" (**is** "domain ?h_img")
⟨*proof*⟩

lemma (**in** field) ring_iso_imp_img_field:
assumes "h ∈ ring_iso R S"
shows "field (S (| zero := h **0** |))" (**is** "field ?h_img")
⟨*proof*⟩

lemma ring_iso_same_card: "R ≃ S ⟹ card (carrier R) = card (carrier S)"
⟨*proof*⟩

lemma ring_iso_set_refl: "id ∈ ring_iso R R"
⟨*proof*⟩

corollary ring_iso_refl: "R ≃ R"
⟨*proof*⟩

lemma ring_iso_set_trans:
"⟦ f ∈ ring_iso R S; g ∈ ring_iso S Q ⟧ ⟹ (g ∘ f) ∈ ring_iso R Q"
⟨*proof*⟩

corollary ring_iso_trans: "⟦ R ≃ S; S ≃ Q ⟧ ⟹ R ≃ Q"
⟨*proof*⟩

lemma ring_iso_set_sym:
assumes "ring R" **and** h: "h ∈ ring_iso R S"
shows "(inv_into (carrier R) h) ∈ ring_iso S R"
⟨*proof*⟩

corollary ring_iso_sym:

assumes "ring R"
shows "R ≃ S ⟹ S ≃ R"
⟨*proof*⟩

lemma (**in** ring_hom_ring) the_elem_simp [simp]:
assumes x: "x ∈ carrier R"
shows "the_elem (h ‘ ((a_kernel R S h) +> x)) = h x"
⟨*proof*⟩

lemma (**in** ring_hom_ring) the_elem_inj:
assumes "X ∈ carrier (R Quot (a_kernel R S h))"
and "Y ∈ carrier (R Quot (a_kernel R S h))"
and Eq: "the_elem (h ‘ X) = the_elem (h ‘ Y)"
shows "X = Y"
⟨*proof*⟩

lemma (**in** ring_hom_ring) quot_mem:
"X ∈ carrier (R Quot (a_kernel R S h)) ⟹ ∃x ∈ carrier R. X = (a_kernel R S h) +> x"
⟨*proof*⟩

lemma (**in** ring_hom_ring) the_elem_wf:
assumes "X ∈ carrier (R Quot (a_kernel R S h))"
shows "∃y ∈ carrier S. (h ‘ X) = { y }"
⟨*proof*⟩

corollary (**in** ring_hom_ring) the_elem_wf’:
"X ∈ carrier (R Quot (a_kernel R S h)) ⟹ ∃r ∈ carrier R. (h ‘ X) = { h r }"
⟨*proof*⟩

lemma (**in** ring_hom_ring) the_elem_hom:
"(λX. the_elem (h ‘ X)) ∈ ring_hom (R Quot (a_kernel R S h)) S"
⟨*proof*⟩

lemma (**in** ring_hom_ring) the_elem_surj:
"(λX. (the_elem (h ‘ X))) ‘ carrier (R Quot (a_kernel R S h)) = (h ‘ (carrier R))"
⟨*proof*⟩

proposition (**in** ring_hom_ring) FactRing_iso_set_aux:
"(λX. the_elem (h ‘ X)) ∈ ring_iso (R Quot (a_kernel R S h)) (S (| carrier := h ‘ (carrier R) |))"
⟨*proof*⟩

theorem (**in** ring_hom_ring) FactRing_iso_set:
assumes "h ‘ carrier R = carrier S"
shows "(λX. the_elem (h ‘ X)) ∈ ring_iso (R Quot (a_kernel R S h)) S"

⟨*proof*⟩

corollary (in ring_hom_ring**)** FactRing_iso:
assumes "h ‘ carrier R = carrier S"
shows "R Quot (a_kernel R S h) ≃ S"
⟨*proof*⟩

corollary (in ring**)** FactRing_zeroideal:
shows "R Quot { **0** } ≃ R" **and** "R ≃ R Quot { **0** }"
⟨*proof*⟩

lemma (in ring_hom_ring**)** img_is_ring: "ring (S (| carrier := h ‘ (carrier R) |))"
⟨*proof*⟩

lemma (in ring_hom_ring**)** img_is_cring:
assumes "cring S"
shows "cring (S (| carrier := h ‘ (carrier R) |))"
⟨*proof*⟩

lemma (in ring_hom_ring**)** img_is_domain:
assumes "domain S"
shows "domain (S (| carrier := h ‘ (carrier R) |))"
⟨*proof*⟩

proposition (in ring_hom_ring**)** primeideal_vimage:
assumes R: "cring R"
and A: "primeideal P S"
shows "primeideal { r ∈ carrier R. h r ∈ P } R"
⟨*proof*⟩

end

theory IntRing
imports "HOL-Computational_Algebra.Primes" QuotRing Lattice
begin

28 The Ring of Integers

28.1 Some properties of int

lemma dvds_eq_abseq:
fixes k :: int
shows "l dvd k ∧ k dvd l ⟷ |l| = |k|"
⟨*proof*⟩

28.2 $\mathcal{Z}$: The Set of Integers as Algebraic Structure

abbreviation int_ring :: "int ring" (‹$\mathcal{Z}$›)
  where "int_ring ≡ ⦇carrier = UNIV, mult = (*), one = 1, zero = 0, add = (+)⦈"

lemma int_Zcarr [intro!, simp]: "k ∈ carrier $\mathcal{Z}$"
  ⟨*proof*⟩

lemma int_is_cring: "cring $\mathcal{Z}$"
⟨*proof*⟩

28.3 Interpretations

Since definitions of derived operations are global, their interpretation needs to be done as early as possible — that is, with as few assumptions as possible.

interpretation int: monoid $\mathcal{Z}$
  rewrites "carrier $\mathcal{Z}$ = UNIV"
    and "mult $\mathcal{Z}$ x y = x * y"
    and "one $\mathcal{Z}$ = 1"
    and "pow $\mathcal{Z}$ x n = x^n"
⟨*proof*⟩

interpretation int: comm_monoid $\mathcal{Z}$
  rewrites "finprod $\mathcal{Z}$ f A = prod f A"
⟨*proof*⟩

interpretation int: abelian_monoid $\mathcal{Z}$
  rewrites int_carrier_eq: "carrier $\mathcal{Z}$ = UNIV"
    and int_zero_eq: "zero $\mathcal{Z}$ = 0"
    and int_add_eq: "add $\mathcal{Z}$ x y = x + y"
    and int_finsum_eq: "finsum $\mathcal{Z}$ f A = sum f A"
⟨*proof*⟩

interpretation int: abelian_group $\mathcal{Z}$

  rewrites "carrier $\mathcal{Z}$ = UNIV"
    and "zero $\mathcal{Z}$ = 0"
    and "add $\mathcal{Z}$ x y = x + y"
    and "finsum $\mathcal{Z}$ f A = sum f A"
    and int_a_inv_eq: "a_inv $\mathcal{Z}$ x = - x"
    and int_a_minus_eq: "a_minus $\mathcal{Z}$ x y = x - y"
⟨*proof*⟩

interpretation int: "domain" $\mathcal{Z}$
  rewrites "carrier $\mathcal{Z}$ = UNIV"
    and "zero $\mathcal{Z}$ = 0"
    and "add $\mathcal{Z}$ x y = x + y"

and "finsum $\mathcal{Z}$ f A = sum f A"
and "a_inv $\mathcal{Z}$ x = - x"
and "a_minus $\mathcal{Z}$ x y = x - y"
⟨*proof*⟩

Removal of occurrences of UNIV in interpretation result — experimental.

lemma UNIV:
"x ∈ UNIV ⟷ True"
"A ⊆ UNIV ⟷ True"
"(∀x ∈ UNIV. P x) ⟷ (∀x. P x)"
"(∃x ∈ UNIV. P x) ⟷ (∃x. P x)"
"(True ⟶ Q) ⟷ Q"
"(True ⟹ PROP R) ≡ PROP R"
⟨*proof*⟩

interpretation int :
partial_order "(|carrier = UNIV::int set, eq = (=), le = (≤)|)"
rewrites "carrier (|carrier = UNIV::int set, eq = (=), le = (≤)|) = UNIV"
and "le (|carrier = UNIV::int set, eq = (=), le = (≤)|) x y = (x ≤ y)"
and "lless (|carrier = UNIV::int set, eq = (=), le = (≤)|) x y = (x < y)"
⟨*proof*⟩

interpretation int :
lattice "(|carrier = UNIV::int set, eq = (=), le = (≤)|)"
rewrites "join (|carrier = UNIV::int set, eq = (=), le = (≤)|) x y = max x y"
and "meet (|carrier = UNIV::int set, eq = (=), le = (≤)|) x y = min x y"
⟨*proof*⟩

interpretation int :
total_order "(|carrier = UNIV::int set, eq = (=), le = (≤)|)"
⟨*proof*⟩

28.4 Generated Ideals of $\mathcal{Z}$

lemma int_Idl: "Idl$_{\mathcal{Z}}$ {a} = {x * a | x. True}"
⟨*proof*⟩

lemma multiples_principalideal: "principalideal {x * a | x. True } $\mathcal{Z}$"
⟨*proof*⟩

lemma prime_primeideal:
assumes prime: "Factorial_Ring.prime p"
shows "primeideal (Idl$_{\mathcal{Z}}$ {p}) $\mathcal{Z}$"
⟨*proof*⟩

28.5 Ideals and Divisibility

lemma int_Idl_subset_ideal: "Idl$_{\mathcal{Z}}$ {k} ⊆ Idl$_{\mathcal{Z}}$ {l} = (k ∈ Idl$_{\mathcal{Z}}$ {l})"
⟨*proof*⟩

lemma Idl_subset_eq_dvd: "Idl$_{\mathcal{Z}}$ {k} ⊆ Idl$_{\mathcal{Z}}$ {l} ⟷ l dvd k"
⟨*proof*⟩

lemma dvds_eq_Idl: "l dvd k ∧ k dvd l ⟷ Idl$_{\mathcal{Z}}$ {k} = Idl$_{\mathcal{Z}}$ {l}"
⟨*proof*⟩

lemma Idl_eq_abs: "Idl$_{\mathcal{Z}}$ {k} = Idl$_{\mathcal{Z}}$ {l} ⟷ |l| = |k|"
⟨*proof*⟩

28.6 Ideals and the Modulus

definition ZMod :: "int ⇒ int ⇒ int set"
where "ZMod k r = (Idl$_{\mathcal{Z}}$ {k}) +>$_{\mathcal{Z}}$ r"

lemmas ZMod_defs =
ZMod_def genideal_def

lemma rcos_zfact:
assumes kIl: "k ∈ ZMod l r"
shows "∃x. k = x * l + r"
⟨*proof*⟩

lemma ZMod_imp_zmod:
assumes zmods: "ZMod m a = ZMod m b"
shows "a mod m = b mod m"
⟨*proof*⟩

lemma ZMod_mod: "ZMod m a = ZMod m (a mod m)"
⟨*proof*⟩

lemma zmod_imp_ZMod:
assumes modeq: "a mod m = b mod m"
shows "ZMod m a = ZMod m b"
⟨*proof*⟩

corollary ZMod_eq_mod: "ZMod m a = ZMod m b ⟷ a mod m = b mod m"
⟨*proof*⟩

28.7 Factorization

definition ZFact :: "int ⇒ int set ring"
where "ZFact k = $\mathcal{Z}$ Quot (Idl$_{\mathcal{Z}}$ {k})"

lemmas ZFact_defs = ZFact_def FactRing_def

lemma ZFact_is_cring: "cring (ZFact k)"
⟨*proof*⟩

lemma ZFact_zero: "carrier (ZFact 0) = (⋃a. {{a}})"
⟨*proof*⟩

lemma ZFact_one: "carrier (ZFact 1) = {UNIV}"
⟨*proof*⟩

lemma ZFact_prime_is_domain:
assumes pprime: "Factorial_Ring.prime p"
shows "domain (ZFact p)"
⟨*proof*⟩

end

theory Weak_Morphisms
imports QuotRing

begin

29 Weak Morphisms

The definition of ring isomorphism, as well as the definition of group isomorphism, doesn't enforce any algebraic constraint to the structure of the schemes involved. This seems unnatural, but it turns out to be very useful: in order to prove that a scheme B satisfy certain algebraic constraints, it's sufficient to prove those for a scheme A and show the existence of an isomorphism between A and B. In this section, we explore this idea in a different way: given a scheme A and a function f, we build a scheme B with an algebraic structure of same strength as A where f is an homomorphism from A to B.

29.1 Definitions

locale weak_group_morphism = normal H G **for** f **and** H **and** G **(structure)**
+
assumes inj_mod_subgroup: "⟦ a ∈ carrier G; b ∈ carrier G ⟧ ⟹ f a
= f b ⟷ a ⊗ (inv b) ∈ H"

locale weak_ring_morphism = ideal I R **for** f **and** I **and** R **(structure)** +
assumes inj_mod_ideal: "⟦ a ∈ carrier R; b ∈ carrier R ⟧ ⟹ f a =
f b ⟷ a ⊖ b ∈ I"

definition image_group :: "('a ⇒ 'b) ⇒ ('a, 'c) monoid_scheme ⇒ 'b monoid"

where "image_group f G ≡
(| carrier = f ‘ (carrier G),
mult = (λa b. f ((inv_into (carrier G) f a) ⊗G (inv_into
(carrier G) f b))),
one = f **1**G |)"

definition image_ring :: "('a ⇒ 'b) ⇒ ('a, 'c) ring_scheme ⇒ 'b ring"
where "image_ring f R ≡ monoid.extend (image_group f R)
(| zero = f **0**R,
add = (λa b. f ((inv_into (carrier R) f a) ⊕R (inv_into
(carrier R) f b))) |)"

29.2 Weak Group Morphisms

lemma image_group_carrier: "carrier (image_group f G) = f ‘ (carrier
G)"
⟨*proof*⟩

lemma image_group_one: "one (image_group f G) = f **1**G"
⟨*proof*⟩

lemma weak_group_morphismsI:
assumes "H ◁ G" **and** "⋀a b. ⟦ a ∈ carrier G; b ∈ carrier G ⟧ ⟹ f
a = f b ⟷ a ⊗G (invG b) ∈ H"
shows "weak_group_morphism f H G"
⟨*proof*⟩

lemma image_group_truncate:
fixes R :: "('a, 'b) monoid_scheme"
shows "monoid.truncate (image_group f R) = image_group f (monoid.truncate
R)"
⟨*proof*⟩

lemma image_ring_truncate: "monoid.truncate (image_ring f R) = image_group
f R"
⟨*proof*⟩

lemma (**in** ring) weak_add_group_morphism:
assumes "weak_ring_morphism f I R" **shows** "weak_group_morphism f I (add_monoid
R)"
⟨*proof*⟩

lemma (**in** group) weak_group_morphism_range:
assumes "weak_group_morphism f H G" **and** "a ∈ carrier G" **shows** "f ‘
(H #> a) = { f a }"
⟨*proof*⟩

lemma (**in** group) vimage_eq_rcoset:
assumes "weak_group_morphism f H G" **and** "a ∈ carrier G"

shows "{ b ∈ carrier G. f b = f a } = H #> a" **and** "{ b ∈ carrier G. f b = f a } = a <# H"
⟨*proof*⟩

lemma (**in** group) weak_group_morphism_ker:
assumes "weak_group_morphism f H G" **shows** "kernel G (image_group f G) f = H"
⟨*proof*⟩

lemma (**in** group) weak_group_morphism_inv_into:
assumes "weak_group_morphism f H G" **and** "a ∈ carrier G"
obtains h h' **where** "h ∈ H" "inv_into (carrier G) f (f a) = h ⊗ a"
and "h' ∈ H" "inv_into (carrier G) f (f a) = a ⊗ h'"
⟨*proof*⟩

proposition (**in** group) weak_group_morphism_is_iso:
assumes "weak_group_morphism f H G" **shows** "(λx. the_elem (f ‘ x)) ∈ iso (G Mod H) (image_group f G)"
⟨*proof*⟩

corollary (**in** group) image_group_is_group:
assumes "weak_group_morphism f H G" **shows** "group (image_group f G)"
⟨*proof*⟩

corollary (**in** group) weak_group_morphism_is_hom:
assumes "weak_group_morphism f H G" **shows** "f ∈ hom G (image_group f G)"
⟨*proof*⟩

corollary (**in** group) weak_group_morphism_group_hom:
assumes "weak_group_morphism f H G" **shows** "group_hom G (image_group f G) f"
⟨*proof*⟩

29.3 Weak Ring Morphisms

lemma image_ring_carrier: "carrier (image_ring f R) = f ‘ (carrier R)"
⟨*proof*⟩

lemma image_ring_one: "one (image_ring f R) = f $\mathbf{1}_R$"
⟨*proof*⟩

lemma image_ring_zero: "zero (image_ring f R) = f $\mathbf{0}_R$"
⟨*proof*⟩

lemma weak_ring_morphismI:
assumes "ideal I R" **and** "⋀a b. ⟦ a ∈ carrier R; b ∈ carrier R ⟧ ⟹ f a = f b ⟷ a $\ominus_R$ b ∈ I"
shows "weak_ring_morphism f I R"

⟨*proof*⟩

lemma (**in** ring) weak_ring_morphism_range:
assumes "weak_ring_morphism f I R" **and** "a ∈ carrier R" **shows** "f ‘ (I +> a) = { f a }"
⟨*proof*⟩

lemma (**in** ring) vimage_eq_a_rcoset:
assumes "weak_ring_morphism f I R" **and** "a ∈ carrier R" **shows** "{ b ∈ carrier R. f b = f a } = I +> a"
⟨*proof*⟩

lemma (**in** ring) weak_ring_morphism_ker:
assumes "weak_ring_morphism f I R" **shows** "a_kernel R (image_ring f R) f = I"
⟨*proof*⟩

lemma (**in** ring) weak_ring_morphism_inv_into:
assumes "weak_ring_morphism f I R" **and** "a ∈ carrier R"
obtains i **where** "i ∈ I" "inv_into (carrier R) f (f a) = i ⊕ a"
⟨*proof*⟩

proposition (**in** ring) weak_ring_morphism_is_iso:
assumes "weak_ring_morphism f I R" **shows** "(λx. the_elem (f ‘ x)) ∈ ring_iso (R Quot I) (image_ring f R)"
⟨*proof*⟩

corollary (**in** ring) image_ring_zero’:
assumes "weak_ring_morphism f I R" **shows** "the_elem (f ‘ $\mathbf{0}_{\text{R Quot I}}$) = $\mathbf{0}_{\text{image_ring f R}}$"
⟨*proof*⟩

corollary (**in** ring) image_ring_is_ring:
assumes "weak_ring_morphism f I R" **shows** "ring (image_ring f R)"
⟨*proof*⟩

corollary (**in** ring) image_ring_is_field:
assumes "weak_ring_morphism f I R" **and** "field (R Quot I)" **shows** "field (image_ring f R)"
⟨*proof*⟩

corollary (**in** ring) weak_ring_morphism_is_hom:
assumes "weak_ring_morphism f I R" **shows** "f ∈ ring_hom R (image_ring f R)"
⟨*proof*⟩

corollary (**in** ring) weak_ring_morphism_ring_hom:
assumes "weak_ring_morphism f I R" **shows** "ring_hom_ring R (image_ring f R) f"

⟨*proof*⟩

29.4 Injective Functions

If the fuction is injective, we don't need to impose any algebraic restriction to the input scheme in order to state an isomorphism.

lemma inj_imp_image_group_iso:
assumes "inj_on f (carrier G)" **shows** "f ∈ iso G (image_group f G)"
⟨*proof*⟩

lemma inj_imp_image_group_inv_iso:
assumes "inj f" **shows** "Hilbert_Choice.inv f ∈ iso (image_group f G) G"
⟨*proof*⟩

lemma inj_imp_image_ring_iso:
assumes "inj_on f (carrier R)" **shows** "f ∈ ring_iso R (image_ring f R)"
⟨*proof*⟩

lemma inj_imp_image_ring_inv_iso:
assumes "inj f" **shows** "Hilbert_Choice.inv f ∈ ring_iso (image_ring f R) R"
⟨*proof*⟩

lemma (**in** group) inj_imp_image_group_is_group:
assumes "inj_on f (carrier G)" **shows** "group (image_group f G)"
⟨*proof*⟩

lemma (**in** ring) inj_imp_image_ring_is_ring:
assumes "inj_on f (carrier R)" **shows** "ring (image_ring f R)"
⟨*proof*⟩

lemma (**in** domain) inj_imp_image_ring_is_domain:
assumes "inj_on f (carrier R)" **shows** "domain (image_ring f R)"
⟨*proof*⟩

lemma (**in** field) inj_imp_image_ring_is_field:
assumes "inj_on f (carrier R)" **shows** "field (image_ring f R)"
⟨*proof*⟩

30 Examples

In a lot of different contexts, the lack of dependent types make some definitions quite complicated. The tools developed in this theory give us a way to change the type of a scheme and preserve all of its algebraic properties. We show, in this section, how to make use of this feature in order to solve

the problem mentioned above.

30.1 Direct Product

abbreviation nil_monoid :: "('a list) monoid"
  where "nil_monoid ≡ ⦇ carrier = { [] }, mult = (λa b. []), one = [] ⦈"

definition DirProd_list :: "(('a, 'b) monoid_scheme) list ⇒ ('a list) monoid"
  where "DirProd_list Gs = foldr (λG H. image_group (λ(x, xs). x # xs) (G ×× H)) Gs nil_monoid"

30.1.1 Basic Properties

lemma DirProd_list_carrier:
  shows "carrier (DirProd_list (G # Gs)) = (λ(x, xs). x # xs) ‘ (carrier G × carrier (DirProd_list Gs))"
  ⟨*proof*⟩

lemma DirProd_list_one:
  shows "one (DirProd_list Gs) = foldr (λG tl. (one G) # tl) Gs []"
  ⟨*proof*⟩

lemma DirProd_list_carrier_mem:
  assumes "gs ∈ carrier (DirProd_list Gs)"
  shows "length gs = length Gs" **and** "⋀i. i < length Gs ⟹ (gs ! i) ∈ carrier (Gs ! i)"
⟨*proof*⟩

lemma DirProd_list_carrier_memI:
  assumes "length gs = length Gs" **and** "⋀i. i < length Gs ⟹ (gs ! i) ∈ carrier (Gs ! i)"
  shows "gs ∈ carrier (DirProd_list Gs)"
  ⟨*proof*⟩

lemma inj_on_DirProd_carrier:
  shows "inj_on (λ(g, gs). g # gs) (carrier (G ×× (DirProd_list Gs)))"
  ⟨*proof*⟩

lemma DirProd_list_is_group:
  assumes "⋀i. i < length Gs ⟹ group (Gs ! i)" **shows** "group (DirProd_list Gs)"
  ⟨*proof*⟩

lemma DirProd_list_iso:
  "(λ(g, gs). g # gs) ∈ iso (G ×× (DirProd_list Gs)) (DirProd_list (G # Gs))"
  ⟨*proof*⟩

end

theory Ring_Divisibility
imports Ideal Divisibility QuotRing Multiplicative_Group

begin

definition mult_of :: "('a, 'b) ring_scheme ⇒ 'a monoid" **where**
"mult_of R ≡ ⦇ carrier = carrier R - {$\mathbf{0}_R$}, mult = mult R, one = $\mathbf{1}_R$⦈"

lemma carrier_mult_of [simp]: "carrier (mult_of R) = carrier R - {$\mathbf{0}_R$}"
⟨*proof*⟩

lemma mult_mult_of [simp]: "mult (mult_of R) = mult R"
⟨*proof*⟩

lemma nat_pow_mult_of: "([^]$_{\text{mult_of R}}$) = (([^]$_R$) :: _ ⇒ nat ⇒ _)"
⟨*proof*⟩

lemma one_mult_of [simp]: "$\mathbf{1}_{\text{mult_of R}}$ = $\mathbf{1}_R$"
⟨*proof*⟩

31 The Arithmetic of Rings

In this section we study the links between the divisibility theory and that of rings

31.1 Definitions

locale factorial_domain = domain + factorial_monoid "mult_of R"

locale noetherian_ring = ring +
assumes finetely_gen: "ideal I R ⟹ ∃A ⊆ carrier R. finite A ∧ I = Idl A"

locale noetherian_domain = noetherian_ring + domain

locale principal_domain = domain +
assumes exists_gen: "ideal I R ⟹ ∃a ∈ carrier R. I = PIdl a"

locale euclidean_domain =
domain R **for** R (**structure**) + **fixes** φ :: "'a ⇒ nat"
assumes euclidean_function:
" ⟦ a ∈ carrier R - { **0** }; b ∈ carrier R - { **0** } ⟧ ⟹

```
   ∃q r. q ∈ carrier R ∧ r ∈ carrier R ∧ a = (b ⊗ q) ⊕ r ∧ ((r = 0)
∨ (φ r < φ b))"
```

definition ring_irreducible :: "('a, 'b) ring_scheme ⇒ 'a ⇒ bool" (‹ring'_irreducible› ›)
  where "ring_irreducible$_R$ a ⟷ (a ≠ **0**$_R$) ∧ (irreducible R a)"

definition ring_prime :: "('a, 'b) ring_scheme ⇒ 'a ⇒ bool" (‹ring'_prime› ›)
  where "ring_prime$_R$ a ⟷ (a ≠ **0**$_R$) ∧ (prime R a)"

31.2 The cancellative monoid of a domain.

sublocale domain < mult_of: comm_monoid_cancel "mult_of R"
  rewrites "mult (mult_of R) = mult R"
      and "one (mult_of R) = one R"
  ⟨*proof*⟩

sublocale factorial_domain < mult_of: factorial_monoid "mult_of R"
  rewrites "mult (mult_of R) = mult R"
      and "one (mult_of R) = one R"
  ⟨*proof*⟩

lemma (**in** ring) noetherian_ringI:
  assumes "⋀I. ideal I R ⟹ ∃A ⊆ carrier R. finite A ∧ I = Idl A"
  shows "noetherian_ring R"
  ⟨*proof*⟩

lemma (**in** domain) euclidean_domainI:
  assumes "⋀a b. ⟦ a ∈ carrier R - { **0** }; b ∈ carrier R - { **0** } ⟧ ⟹
          ∃q r. q ∈ carrier R ∧ r ∈ carrier R ∧ a = (b ⊗ q) ⊕ r ∧
((r = **0**) ∨ (φ r < φ b))"
  shows "euclidean_domain R φ"
  ⟨*proof*⟩

31.3 Passing from R to Ring_Divisibility.mult_of R and vice-versa.

lemma divides_mult_imp_divides [simp]: "a divides$_{(\text{mult_of R})}$ b ⟹ a divides$_R$
b"
  ⟨*proof*⟩

lemma (**in** domain) divides_imp_divides_mult [simp]:
  "⟦ a ∈ carrier R; b ∈ carrier R - { **0** } ⟧ ⟹ a divides$_R$ b ⟹ a divides$_{(\text{mult_of R})}$
b"
  ⟨*proof*⟩

lemma (**in** cring) divides_one:
  assumes "a ∈ carrier R"
  shows "a divides **1** ⟷ a ∈ Units R"
  ⟨*proof*⟩

lemma (in ring**)** one_divides:
assumes "a ∈ carrier R" **shows** "**1** divides a"
⟨*proof*⟩

lemma (in ring**)** divides_zero:
assumes "a ∈ carrier R" **shows** "a divides **0**"
⟨*proof*⟩

lemma (in ring**)** zero_divides:
shows "**0** divides a ⟷ a = **0**"
⟨*proof*⟩

lemma (in domain**)** divides_mult_zero:
assumes "a ∈ carrier R" **shows** "a divides$_{\text{(mult_of R)}}$ **0** ⟹ a = **0**"
⟨*proof*⟩

lemma (in ring**)** divides_mult:
assumes "a ∈ carrier R" "c ∈ carrier R"
shows "a divides b ⟹ (c ⊗ a) divides (c ⊗ b)"
⟨*proof*⟩

lemma (in domain**)** mult_divides:
assumes "a ∈ carrier R" "b ∈ carrier R" "c ∈ carrier R - { **0** }"
shows "(c ⊗ a) divides (c ⊗ b) ⟹ a divides b"
⟨*proof*⟩

lemma (in domain**)** assoc_iff_assoc_mult:
assumes "a ∈ carrier R" **and** "b ∈ carrier R"
shows "a ∼ b ⟷ a ∼$_{\text{(mult_of R)}}$ b"
⟨*proof*⟩

lemma (in domain**)** Units_mult_eq_Units [simp]: "Units (mult_of R) = Units R"
⟨*proof*⟩

lemma (in domain**)** ring_associated_iff:
assumes "a ∈ carrier R" "b ∈ carrier R"
shows "a ∼ b ⟷ (∃u ∈ Units R. a = u ⊗ b)"
⟨*proof*⟩

lemma (in domain**)** properfactor_mult_imp_properfactor:
"⟦ a ∈ carrier R; b ∈ carrier R ⟧ ⟹ properfactor (mult_of R) b a ⟹ properfactor R b a"
⟨*proof*⟩

lemma (in domain**)** properfactor_imp_properfactor_mult:
"⟦ a ∈ carrier R - { **0** }; b ∈ carrier R ⟧ ⟹ properfactor R b a ⟹ properfactor (mult_of R) b a"

⟨*proof*⟩

lemma (in domain) properfactor_of_zero:
assumes "b ∈ carrier R"
shows "¬ properfactor (mult_of R) b **0**" **and** "properfactor R b **0** ⟷ b ≠ **0**"
⟨*proof*⟩

31.4 Irreducible

The following lemmas justify the need for a definition of irreducible specific to rings: for `irreducible R`, we need to suppose we are not in a field (which is plausible, but ¬ `field R` is an assumption we want to avoid; for `irreducible (Ring_Divisibility.mult_of R)`, zero is allowed.

lemma (in domain) zero_is_irreducible_mult:
shows "irreducible (mult_of R) **0**"
⟨*proof*⟩

lemma (in domain) zero_is_irreducible_iff_field:
shows "irreducible R **0** ⟷ field R"
⟨*proof*⟩

lemma (in domain) irreducible_imp_irreducible_mult:
"⟦ a ∈ carrier R; irreducible R a ⟧ ⟹ irreducible (mult_of R) a"
⟨*proof*⟩

lemma (in domain) irreducible_mult_imp_irreducible:
"⟦ a ∈ carrier R - { **0** }; irreducible (mult_of R) a ⟧ ⟹ irreducible R a"
⟨*proof*⟩

lemma (in domain) ring_irreducibleE:
assumes "r ∈ carrier R" **and** "ring_irreducible r"
shows "r ≠ **0**" "irreducible R r" "irreducible (mult_of R) r" "r ∉ Units R"
and "⋀a b. ⟦ a ∈ carrier R; b ∈ carrier R ⟧ ⟹ r = a ⊗ b ⟹ a ∈ Units R ∨ b ∈ Units R"
⟨*proof*⟩

lemma (in domain) ring_irreducibleI:
assumes "r ∈ carrier R - { **0** }" "r ∉ Units R"
and "⋀a b. ⟦ a ∈ carrier R; b ∈ carrier R ⟧ ⟹ r = a ⊗ b ⟹ a ∈ Units R ∨ b ∈ Units R"
shows "ring_irreducible r"
⟨*proof*⟩

lemma (in domain) ring_irreducibleI':
assumes "r ∈ carrier R - { **0** }" "irreducible (mult_of R) r"

shows "ring_irreducible r"
⟨*proof*⟩

31.5 Primes

lemma (**in** domain) zero_is_prime: "prime R **0**" "prime (mult_of R) **0**"
⟨*proof*⟩

lemma (**in** domain) prime_eq_prime_mult:
assumes "p ∈ carrier R"
shows "prime R p ⟷ prime (mult_of R) p"
⟨*proof*⟩

lemma (**in** domain) ring_primeE:
assumes "p ∈ carrier R" "ring_prime p"
shows "p ≠ **0**" "prime (mult_of R) p" "prime R p"
⟨*proof*⟩

lemma (**in** ring) ring_primeI:
assumes "p ≠ **0**" "prime R p" **shows** "ring_prime p"
⟨*proof*⟩

lemma (**in** domain) ring_primeI':
assumes "p ∈ carrier R - { **0** }" "prime (mult_of R) p"
shows "ring_prime p"
⟨*proof*⟩

31.6 Basic Properties

lemma (**in** cring) to_contain_is_to_divide:
assumes "a ∈ carrier R" "b ∈ carrier R"
shows "PIdl b ⊆ PIdl a ⟷ a divides b"
⟨*proof*⟩

lemma (**in** cring) associated_iff_same_ideal:
assumes "a ∈ carrier R" "b ∈ carrier R"
shows "a ∼ b ⟷ PIdl a = PIdl b"
⟨*proof*⟩

lemma (**in** cring) ideal_eq_carrier_iff:
assumes "a ∈ carrier R"
shows "carrier R = PIdl a ⟷ a ∈ Units R"
⟨*proof*⟩

lemma (**in** domain) primeideal_iff_prime:
assumes "p ∈ carrier R - { **0** }"
shows "primeideal (PIdl p) R ⟷ ring_prime p"
⟨*proof*⟩

31.7 Noetherian Rings

lemma (in ring**)** chain_Union_is_ideal:
assumes "subset.chain { I. ideal I R } C"
shows "ideal (if C = {} then { **0** } else (⋃C)) R"
⟨*proof*⟩

lemma (in noetherian_ring**)** ideal_chain_is_trivial:
assumes "C ≠ {}" "subset.chain { I. ideal I R } C"
shows "⋃C ∈ C"
⟨*proof*⟩

lemma (in ring**)** trivial_ideal_chain_imp_noetherian:
assumes "⋀C. ⟦ C ≠ {}; subset.chain { I. ideal I R } C ⟧ ⟹ ⋃C ∈ C"
shows "noetherian_ring R"
⟨*proof*⟩

lemma (in noetherian_domain**)** factorization_property:
assumes "a ∈ carrier R - { **0** }" "a ∉ Units R"
shows "∃fs. set fs ⊆ carrier (mult_of R) ∧ wfactors (mult_of R) fs a" **(is** "?factorizable a"**)**
⟨*proof*⟩

lemma (in noetherian_domain**)** exists_irreducible_divisor:
assumes "a ∈ carrier R - { **0** }" **and** "a ∉ Units R"
obtains b **where** "b ∈ carrier R" **and** "ring_irreducible b" **and** "b divides a"
⟨*proof*⟩

31.8 Principal Domains

sublocale principal_domain ⊆ noetherian_domain
⟨*proof*⟩

lemma (in principal_domain**)** irreducible_imp_maximalideal:
assumes "p ∈ carrier R"
and "ring_irreducible p"
shows "maximalideal (PIdl p) R"
⟨*proof*⟩

corollary (in principal_domain**)** primeness_condition:
assumes "p ∈ carrier R"
shows "ring_irreducible p ⟷ ring_prime p"
⟨*proof*⟩

lemma (in principal_domain**)** domain_iff_prime:
assumes "a ∈ carrier R - { **0** }"
shows "domain (R Quot (PIdl a)) ⟷ ring_prime a"
⟨*proof*⟩

lemma (in principal_domain**)** field_iff_prime:
assumes "a ∈ carrier R - { **0** }"
shows "field (R Quot (PIdl a)) ⟷ ring_prime a"
⟨*proof*⟩

sublocale principal_domain < mult_of: primeness_condition_monoid "mult_of R"
rewrites "mult (mult_of R) = mult R"
and "one (mult_of R) = one R"
⟨*proof*⟩

sublocale principal_domain < mult_of: factorial_monoid "mult_of R"
rewrites "mult (mult_of R) = mult R"
and "one (mult_of R) = one R"
⟨*proof*⟩

sublocale principal_domain ⊆ factorial_domain
⟨*proof*⟩

lemma (in principal_domain**)** ideal_sum_iff_gcd:
assumes "a ∈ carrier R" "b ∈ carrier R" "d ∈ carrier R"
shows "PIdl d = PIdl a <+>$_R$ PIdl b ⟷ d gcdof a b"
⟨*proof*⟩

lemma (in principal_domain**)** bezout_identity:
assumes "a ∈ carrier R" "b ∈ carrier R"
shows "PIdl a <+>$_R$ PIdl b = PIdl (somegcd R a b)"
⟨*proof*⟩

31.9 Euclidean Domains

sublocale euclidean_domain ⊆ principal_domain
⟨*proof*⟩

sublocale field ⊆ euclidean_domain R "λ_. 0"
⟨*proof*⟩

end

theory Subrings
imports Ring RingHom QuotRing Multiplicative_Group
begin

32 Subrings

32.1 Definitions

locale subring =
  subgroup H "add_monoid R" + submonoid H R **for** H **and** R (**structure**)

locale subcring = subring +
  assumes sub_m_comm: "⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗ h2 = h2 ⊗ h1"

locale subdomain = subcring +
  assumes sub_one_not_zero [simp]: "**1** ≠ **0**"
  assumes subintegral: "⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗ h2 = **0** ⟹ h1 = **0**
∨ h2 = **0**"

locale subfield = subdomain K R **for** K **and** R (**structure**) +
  assumes subfield_Units: "Units (R (| carrier := K |)) = K - { **0** }"

32.2 Basic Properties

32.2.1 Subrings

lemma (**in** ring) subringI:
  assumes "H ⊆ carrier R"
    and "**1** ∈ H"
    and "⋀h. h ∈ H ⟹ ⊖ h ∈ H"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗ h2 ∈ H"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊕ h2 ∈ H"
  shows "subring H R"
  ⟨*proof*⟩

lemma subringE:
  assumes "subring H R"
  shows "H ⊆ carrier R"
    and "$\mathbf{0}_R$ ∈ H"
    and "$\mathbf{1}_R$ ∈ H"
    and "H ≠ {}"
    and "⋀h. h ∈ H ⟹ $\ominus_R$ h ∈ H"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 $\otimes_R$ h2 ∈ H"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 $\oplus_R$ h2 ∈ H"
  ⟨*proof*⟩

lemma (**in** ring) carrier_is_subring: "subring (carrier R) R"
  ⟨*proof*⟩

lemma (**in** ring) subring_inter:
  assumes "subring I R" **and** "subring J R"
  shows "subring (I ∩ J) R"
  ⟨*proof*⟩

lemma (in ring**)** subring_Inter:
assumes "⋀I. I ∈ S ⟹ subring I R" **and** "S ≠ {}"
shows "subring (⋂S) R"
⟨*proof*⟩

lemma (in ring**)** subring_is_ring:
assumes "subring H R" **shows** "ring (R ⦇ carrier := H ⦈)"
⟨*proof*⟩

lemma (in ring**)** ring_incl_imp_subring:
assumes "H ⊆ carrier R"
and "ring (R ⦇ carrier := H ⦈)"
shows "subring H R"
⟨*proof*⟩

lemma (in ring**)** subring_iff:
assumes "H ⊆ carrier R"
shows "subring H R ⟷ ring (R ⦇ carrier := H ⦈)"
⟨*proof*⟩

32.2.2 Subcrings

lemma (in ring**)** subcringI:
assumes "subring H R"
and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗ h2 = h2 ⊗ h1"
shows "subcring H R"
⟨*proof*⟩

lemma (in cring**)** subcringI':
assumes "subring H R"
shows "subcring H R"
⟨*proof*⟩

lemma subcringE:
assumes "subcring H R"
shows "H ⊆ carrier R"
and "$\mathbf{0}_R$ ∈ H"
and "$\mathbf{1}_R$ ∈ H"
and "H ≠ {}"
and "⋀h. h ∈ H ⟹ ⊖$_R$ h ∈ H"
and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗$_R$ h2 ∈ H"
and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊕$_R$ h2 ∈ H"
and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗$_R$ h2 = h2 ⊗$_R$ h1"
⟨*proof*⟩

lemma (in cring**)** carrier_is_subcring: "subcring (carrier R) R"
⟨*proof*⟩

lemma (in ring**)** subcring_inter:

assumes "subcring I R" **and** "subcring J R"
shows "subcring (I ∩ J) R"
⟨*proof*⟩

lemma (in ring**)** subcring_Inter:
assumes "⋀I. I ∈ S ⟹ subcring I R" **and** "S ≠ {}"
shows "subcring (⋂S) R"
⟨*proof*⟩

lemma (in ring**)** subcring_iff:
assumes "H ⊆ carrier R"
shows "subcring H R ⟷ cring (R (| carrier := H |))"
⟨*proof*⟩

32.2.3 Subdomains

lemma (in ring**)** subdomainI:
assumes "subcring H R"
and "1 ≠ 0"
and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗ h2 = 0 ⟹ h1 = 0 ∨ h2 = 0"
shows "subdomain H R"
⟨*proof*⟩

lemma (in domain**)** subdomainI':
assumes "subring H R"
shows "subdomain H R"
⟨*proof*⟩

lemma subdomainE:
assumes "subdomain H R"
shows "H ⊆ carrier R"
and "$\mathbf{0}_R$ ∈ H"
and "$\mathbf{1}_R$ ∈ H"
and "H ≠ {}"
and "⋀h. h ∈ H ⟹ ⊖$_R$ h ∈ H"
and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗$_R$ h2 ∈ H"
and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊕$_R$ h2 ∈ H"
and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗$_R$ h2 = h2 ⊗$_R$ h1"
and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗$_R$ h2 = $\mathbf{0}_R$ ⟹ h1 = $\mathbf{0}_R$ ∨ h2 = $\mathbf{0}_R$"
and "$\mathbf{1}_R$ ≠ $\mathbf{0}_R$"
⟨*proof*⟩

lemma (in ring**)** subdomain_iff:
assumes "H ⊆ carrier R"
shows "subdomain H R ⟷ domain (R (| carrier := H |))"
⟨*proof*⟩

lemma (in domain**)** subring_is_domain:
assumes "subring H R" **shows** "domain (R ⦇ carrier := H ⦈)"
⟨*proof*⟩

lemma (in ring**)** subdomain_is_domain:
assumes "subdomain H R" **shows** "domain (R ⦇ carrier := H ⦈)"
⟨*proof*⟩

32.2.4 Subfields

lemma (in ring**)** subfieldI:
assumes "subcring K R" **and** "Units (R ⦇ carrier := K ⦈) = K - { **0** }"
shows "subfield K R"
⟨*proof*⟩

lemma (in field**)** subfieldI':
assumes "subring K R" **and** "⋀k. k ∈ K - { **0** } ⟹ inv k ∈ K"
shows "subfield K R"
⟨*proof*⟩

lemma (in field**)** carrier_is_subfield: "subfield (carrier R) R"
⟨*proof*⟩

lemma subfieldE:
assumes "subfield K R"
shows "subring K R" **and** "subcring K R"
and "K ⊆ carrier R"
and "⋀k1 k2. ⟦ k1 ∈ K; k2 ∈ K ⟧ ⟹ k1 ⊗$_R$ k2 = k2 ⊗$_R$ k1"
and "⋀k1 k2. ⟦ k1 ∈ K; k2 ∈ K ⟧ ⟹ k1 ⊗$_R$ k2 = **0**$_R$ ⟹ k1 = **0**$_R$ ∨ k2 = **0**$_R$"
and "**1**$_R$ ≠ **0**$_R$"
⟨*proof*⟩

lemma (in ring**)** subfield_m_inv:
assumes "subfield K R" **and** "k ∈ K - { **0** }"
shows "inv k ∈ K - { **0** }" **and** "k ⊗ inv k = **1**" **and** "inv k ⊗ k = **1**"
⟨*proof*⟩

lemma (in ring**)** subfield_m_inv_simprule:
assumes "subfield K R"
shows "⟦ k ∈ K - { **0** }; a ∈ carrier R ⟧ ⟹ k ⊗ a ∈ K ⟹ a ∈ K"
⟨*proof*⟩

lemma (in ring**)** subfield_iff:
shows "⟦ field (R ⦇ carrier := K ⦈); K ⊆ carrier R ⟧ ⟹ subfield K R"
and "subfield K R ⟹ field (R ⦇ carrier := K ⦈)"
⟨*proof*⟩

lemma (in field**)** subgroup_mult_of :
assumes "subfield K R"
shows "subgroup (K - {**0**}) (mult_of R)"
⟨*proof*⟩

32.3 Subring Homomorphisms

lemma (in ring**)** hom_imp_img_subring:
assumes "h ∈ ring_hom R S" **and** "subring K R"
shows "ring (S (| carrier := h ‘ K, one := h **1**, zero := h **0** |))"
⟨*proof*⟩

lemma (in ring_hom_ring**)** img_is_subring:
assumes "subring K R" **shows** "subring (h ‘ K) S"
⟨*proof*⟩

lemma (in ring_hom_ring**)** img_is_subfield:
assumes "subfield K R" **and** "$\mathbf{1}_S \neq \mathbf{0}_S$"
shows "inj_on h K" **and** "subfield (h ‘ K) S"
⟨*proof*⟩

lemma (in ring_hom_ring**)** induced_ring_hom:
assumes "subring K R" **shows** "ring_hom_ring (R (| carrier := K |)) S h"
⟨*proof*⟩

lemma (in ring_hom_ring**)** inj_on_subgroup_iff_trivial_ker:
assumes "subring K R"
shows "inj_on h K ⟷ a_kernel (R (| carrier := K |)) S h = { **0** }"
⟨*proof*⟩

lemma (in ring_hom_ring**)** inv_ring_hom:
assumes "inj_on h K" **and** "subring K R"
shows "ring_hom_ring (S (| carrier := h ‘ K |)) R (inv_into K h)"
⟨*proof*⟩

end

theory Polynomials
imports Ring Ring_Divisibility Subrings

begin

33 Polynomials

33.1 Definitions

abbreviation lead_coeff :: "'a list ⇒ 'a"
  where "lead_coeff ≡ hd"

abbreviation degree :: "'a list ⇒ nat"
  where "degree p ≡ length p - 1"

definition polynomial :: "_ ⇒ 'a set ⇒ 'a list ⇒ bool" (<polynomial⇘ ⇙>)
  where "polynomial$_R$ K p ⟷ p = [] ∨ (set p ⊆ K ∧ lead_coeff p ≠ $\mathbf{0}_R$)"

definition (**in** ring) monom :: "'a ⇒ nat ⇒ 'a list"
  where "monom a n = a # (replicate n $\mathbf{0}_R$)"

fun (**in** ring) eval :: "'a list ⇒ 'a ⇒ 'a"
  where
    "eval [] = (λ_. **0**)"
  | "eval p = (λx. ((lead_coeff p) ⊗ (x [^] (degree p))) ⊕ (eval (tl p) x))"

fun (**in** ring) coeff :: "'a list ⇒ nat ⇒ 'a"
  where
    "coeff [] = (λ_. **0**)"
  | "coeff p = (λi. if i = degree p then lead_coeff p else (coeff (tl p)) i)"

fun (**in** ring) normalize :: "'a list ⇒ 'a list"
  where
    "normalize [] = []"
  | "normalize p = (if lead_coeff p ≠ **0** then p else normalize (tl p))"

fun (**in** ring) poly_add :: "'a list ⇒ 'a list ⇒ 'a list"
  where "poly_add p1 p2 =
           (if length p1 ≥ length p2
            then normalize (map2 (⊕) p1 ((replicate (length p1 - length p2) **0**) @ p2))
            else poly_add p2 p1)"

fun (**in** ring) poly_mult :: "'a list ⇒ 'a list ⇒ 'a list"
  where
    "poly_mult [] p2 = []"
  | "poly_mult p1 p2 =
       poly_add ((map (λa. lead_coeff p1 ⊗ a) p2) @ (replicate (degree p1) **0**)) (poly_mult (tl p1) p2)"

fun (**in** ring) dense_repr :: "'a list ⇒ ('a × nat) list"
  where
    "dense_repr [] = []"

```
  | "dense_repr p = (if lead_coeff p ≠ 0
                     then (lead_coeff p, degree p) # (dense_repr (tl p))
                     else (dense_repr (tl p)))"

fun (in ring) poly_of_dense :: "('a × nat) list ⇒ 'a list"
  where "poly_of_dense dl = foldr (λ(a, n) l. poly_add (monom a n) l)
dl []"

definition (in ring) poly_of_const :: "'a ⇒ 'a list"
  where "poly_of_const = (λk. normalize [ k ])"
```

33.2 Basic Properties

```
context ring
begin

lemma polynomialI [intro]: "⟦ set p ⊆ K; lead_coeff p ≠ 0 ⟧ ⟹ polynomial
K p"
  ⟨proof⟩

lemma polynomial_incl: "polynomial K p ⟹ set p ⊆ K"
  ⟨proof⟩

lemma monom_in_carrier [intro]: "a ∈ carrier R ⟹ set (monom a n) ⊆
carrier R"
  ⟨proof⟩

lemma lead_coeff_not_zero: "polynomial K (a # p) ⟹ a ∈ K - { 0 }"
  ⟨proof⟩

lemma zero_is_polynomial [intro]: "polynomial K []"
  ⟨proof⟩

lemma const_is_polynomial [intro]: "a ∈ K - { 0 } ⟹ polynomial K [
a ]"
  ⟨proof⟩

lemma normalize_gives_polynomial: "set p ⊆ K ⟹ polynomial K (normalize
p)"
  ⟨proof⟩

lemma normalize_in_carrier: "set p ⊆ carrier R ⟹ set (normalize p)
⊆ carrier R"
  ⟨proof⟩

lemma normalize_polynomial: "polynomial K p ⟹ normalize p = p"
  ⟨proof⟩

lemma normalize_idem: "normalize ((normalize p) @ q) = normalize (p @
```

q)"
⟨*proof*⟩

lemma normalize_length_le: "length (normalize p) ≤ length p"
⟨*proof*⟩

lemma eval_in_carrier: "⟦ set p ⊆ carrier R; x ∈ carrier R ⟧ ⟹ (eval p) x ∈ carrier R"
⟨*proof*⟩

lemma coeff_in_carrier [simp]: "set p ⊆ carrier R ⟹ (coeff p) i ∈ carrier R"
⟨*proof*⟩

lemma lead_coeff_simp [simp]: "p ≠ [] ⟹ (coeff p) (degree p) = lead_coeff p"
⟨*proof*⟩

lemma coeff_list: "map (coeff p) (rev [0..< length p]) = p"
⟨*proof*⟩

lemma coeff_nth: "i < length p ⟹ (coeff p) i = p ! (length p - 1 - i)"
⟨*proof*⟩

lemma coeff_iff_length_cond:
assumes "length p1 = length p2"
shows "p1 = p2 ⟷ coeff p1 = coeff p2"
⟨*proof*⟩

lemma coeff_img_restrict: "(coeff p) ‘ {..< length p} = set p"
⟨*proof*⟩

lemma coeff_length: "⋀i. i ≥ length p ⟹ (coeff p) i = **0**"
⟨*proof*⟩

lemma coeff_degree: "⋀i. i > degree p ⟹ (coeff p) i = **0**"
⟨*proof*⟩

lemma replicate_zero_coeff [simp]: "coeff (replicate n **0**) = (λ_. **0**)"
⟨*proof*⟩

lemma scalar_coeff: "a ∈ carrier R ⟹ coeff (map (λb. a ⊗ b) p) = (λi. a ⊗ (coeff p) i)"
⟨*proof*⟩

lemma monom_coeff: "coeff (monom a n) = (λi. if i = n then a else **0**)"
⟨*proof*⟩

lemma coeff_img:
"(coeff p) ‘ {..< length p} = set p"
"(coeff p) ‘ { length p ..} = { **0** }"
"(coeff p) ‘ UNIV = (set p) ∪ { **0** }"
⟨*proof*⟩

lemma degree_def':
assumes "polynomial K p"
shows "degree p = (LEAST n. ∀i. i > n ⟶ (coeff p) i = **0**)"
⟨*proof*⟩

lemma coeff_iff_polynomial_cond:
assumes "polynomial K p1" **and** "polynomial K p2"
shows "p1 = p2 ⟷ coeff p1 = coeff p2"
⟨*proof*⟩

lemma normalize_lead_coeff:
assumes "length (normalize p) < length p"
shows "lead_coeff p = **0**"
⟨*proof*⟩

lemma normalize_length_lt:
assumes "lead_coeff p = **0**" **and** "length p > 0"
shows "length (normalize p) < length p"
⟨*proof*⟩

lemma normalize_length_eq:
assumes "lead_coeff p ≠ **0**"
shows "length (normalize p) = length p"
⟨*proof*⟩

lemma normalize_replicate_zero: "normalize ((replicate n **0**) @ p) = normalize p"
⟨*proof*⟩

lemma normalize_def':
shows "p = (replicate (length p - length (normalize p)) **0**) @
(drop (length p - length (normalize p)) p)" (**is** ?statement1)
and "normalize p = drop (length p - length (normalize p)) p" (**is** ?statement2)
⟨*proof*⟩

corollary normalize_trick:
shows "p = (replicate (length p - length (normalize p)) **0**) @ (normalize p)"
⟨*proof*⟩

lemma normalize_coeff: "coeff p = coeff (normalize p)"
⟨*proof*⟩

lemma append_coeff:
"coeff (p @ q) = (λi. if i < length q then (coeff q) i else (coeff p) (i - length q))"
⟨*proof*⟩

lemma prefix_replicate_zero_coeff: "coeff p = coeff ((replicate n **0**) @ p)"
⟨*proof*⟩

context
fixes K :: "'a set" **assumes** K: "subring K R"
begin

lemma polynomial_in_carrier [intro]: "polynomial K p ⟹ set p ⊆ carrier R"
⟨*proof*⟩

lemma carrier_polynomial [intro]: "polynomial K p ⟹ polynomial (carrier R) p"
⟨*proof*⟩

lemma append_is_polynomial: "⟦ polynomial K p; p ≠ [] ⟧ ⟹ polynomial K (p @ (replicate n **0**))"
⟨*proof*⟩

lemma lead_coeff_in_carrier: "polynomial K (a # p) ⟹ a ∈ carrier R - { **0** }"
⟨*proof*⟩

lemma monom_is_polynomial [intro]: "a ∈ K - { **0** } ⟹ polynomial K (monom a n)"
⟨*proof*⟩

lemma eval_poly_in_carrier: "⟦ polynomial K p; x ∈ carrier R ⟧ ⟹ (eval p) x ∈ carrier R"
⟨*proof*⟩

lemma poly_coeff_in_carrier [simp]: "polynomial K p ⟹ coeff p i ∈ carrier R"
⟨*proof*⟩

end

33.3 Polynomial Addition

context
fixes K :: "'a set" **assumes** K: "subring K R"
begin

lemma poly_add_is_polynomial:
  assumes "set p1 ⊆ K" **and** "set p2 ⊆ K"
  shows "polynomial K (poly_add p1 p2)"
⟨*proof*⟩

lemma poly_add_closed: "⟦ polynomial K p1; polynomial K p2 ⟧ ⟹ polynomial K (poly_add p1 p2)"
  ⟨*proof*⟩

lemma poly_add_length_eq:
  assumes "polynomial K p1" "polynomial K p2" **and** "length p1 ≠ length p2"
  shows "length (poly_add p1 p2) = max (length p1) (length p2)"
⟨*proof*⟩

lemma poly_add_degree_eq:
  assumes "polynomial K p1" "polynomial K p2" **and** "degree p1 ≠ degree p2"
  shows "degree (poly_add p1 p2) = max (degree p1) (degree p2)"
  ⟨*proof*⟩

end

lemma poly_add_in_carrier:
  "⟦ set p1 ⊆ carrier R; set p2 ⊆ carrier R ⟧ ⟹ set (poly_add p1 p2) ⊆ carrier R"
  ⟨*proof*⟩

lemma poly_add_length_le: "length (poly_add p1 p2) ≤ max (length p1) (length p2)"
⟨*proof*⟩

lemma poly_add_degree: "degree (poly_add p1 p2) ≤ max (degree p1) (degree p2)"
  ⟨*proof*⟩

lemma poly_add_coeff_aux:
  assumes "length p1 ≥ length p2"
  shows "coeff (poly_add p1 p2) = (λi. ((coeff p1) i) ⊕ ((coeff p2) i))"
⟨*proof*⟩

lemma poly_add_coeff:
  assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
  shows "coeff (poly_add p1 p2) = (λi. ((coeff p1) i) ⊕ ((coeff p2) i))"
⟨*proof*⟩

lemma poly_add_comm:

assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
shows "poly_add p1 p2 = poly_add p2 p1"
⟨*proof*⟩

lemma poly_add_monom:
assumes "set p ⊆ carrier R" **and** "a ∈ carrier R - { **0** }"
shows "poly_add (monom a (length p)) p = a # p"
⟨*proof*⟩

lemma poly_add_append_replicate:
assumes "set p ⊆ carrier R" "set q ⊆ carrier R"
shows "poly_add (p @ (replicate (length q) **0**)) q = normalize (p @ q)"
⟨*proof*⟩

lemma poly_add_append_zero:
assumes "set p ⊆ carrier R" "set q ⊆ carrier R"
shows "poly_add (p @ [**0**]) (q @ [**0**]) = normalize ((poly_add p q)
@ [**0**])"
⟨*proof*⟩

lemma poly_add_normalize_aux:
assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
shows "poly_add p1 p2 = poly_add (normalize p1) p2"
⟨*proof*⟩

lemma poly_add_normalize:
assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
shows "poly_add p1 p2 = poly_add (normalize p1) p2"
and "poly_add p1 p2 = poly_add p1 (normalize p2)"
and "poly_add p1 p2 = poly_add (normalize p1) (normalize p2)"
⟨*proof*⟩

lemma poly_add_zero':
assumes "set p ⊆ carrier R"
shows "poly_add p [] = normalize p" **and** "poly_add [] p = normalize
p"
⟨*proof*⟩

lemma poly_add_zero:
assumes "subring K R" "polynomial K p"
shows "poly_add p [] = p" **and** "poly_add [] p = p"
⟨*proof*⟩

lemma poly_add_replicate_zero':
assumes "set p ⊆ carrier R"
shows "poly_add p (replicate n **0**) = normalize p" **and** "poly_add (replicate
n **0**) p = normalize p"
⟨*proof*⟩

lemma poly_add_replicate_zero:
assumes "subring K R" "polynomial K p"
shows "poly_add p (replicate n **0**) = p" **and** "poly_add (replicate n **0**) p = p"
⟨*proof*⟩

33.4 Dense Representation

lemma dense_repr_replicate_zero: "dense_repr ((replicate n **0**) @ p) = dense_repr p"
⟨*proof*⟩

lemma dense_repr_normalize: "dense_repr (normalize p) = dense_repr p"
⟨*proof*⟩

lemma polynomial_dense_repr:
assumes "polynomial K p" **and** "p ≠ []"
shows "dense_repr p = (lead_coeff p, degree p) # dense_repr (normalize (tl p))"
⟨*proof*⟩

lemma monom_decomp:
assumes "subring K R" "polynomial K p"
shows "p = poly_of_dense (dense_repr p)"
⟨*proof*⟩

33.5 Polynomial Multiplication

lemma poly_mult_is_polynomial:
assumes "subring K R" "set p1 ⊆ K" **and** "set p2 ⊆ K"
shows "polynomial K (poly_mult p1 p2)"
⟨*proof*⟩

lemma poly_mult_closed:
assumes "subring K R"
shows "⟦ polynomial K p1; polynomial K p2 ⟧ ⟹ polynomial K (poly_mult p1 p2)"
⟨*proof*⟩

lemma poly_mult_in_carrier:
"⟦ set p1 ⊆ carrier R; set p2 ⊆ carrier R ⟧ ⟹ set (poly_mult p1 p2) ⊆ carrier R"
⟨*proof*⟩

lemma poly_mult_coeff:
assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
shows "coeff (poly_mult p1 p2) = (λi. ⨁ k ∈ {..i}. (coeff p1) k ⊗ (coeff p2) (i - k))"
⟨*proof*⟩

lemma poly_mult_zero:
assumes "set p ⊆ carrier R"
shows "poly_mult [] p = []" **and** "poly_mult p [] = []"
⟨*proof*⟩

lemma poly_mult_l_distr':
assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R" "set p3 ⊆ carrier R"
shows "poly_mult (poly_add p1 p2) p3 = poly_add (poly_mult p1 p3) (poly_mult p2 p3)"
⟨*proof*⟩

lemma poly_mult_l_distr:
assumes "subring K R" "polynomial K p1" "polynomial K p2" "polynomial K p3"
shows "poly_mult (poly_add p1 p2) p3 = poly_add (poly_mult p1 p3) (poly_mult p2 p3)"
⟨*proof*⟩

lemma poly_mult_prepend_replicate_zero:
assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
shows "poly_mult p1 p2 = poly_mult ((replicate n **0**) @ p1) p2"
⟨*proof*⟩

lemma poly_mult_normalize:
assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
shows "poly_mult p1 p2 = poly_mult (normalize p1) p2"
⟨*proof*⟩

lemma poly_mult_append_zero:
assumes "set p ⊆ carrier R" "set q ⊆ carrier R"
shows "poly_mult (p @ [**0**]) q = normalize ((poly_mult p q) @ [**0**])"
⟨*proof*⟩

end

33.6 Properties Within a Domain

context domain
begin

lemma one_is_polynomial [intro]: "subring K R ⟹ polynomial K [**1**]"
⟨*proof*⟩

lemma poly_mult_comm:
assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
shows "poly_mult p1 p2 = poly_mult p2 p1"
⟨*proof*⟩

lemma poly_mult_r_distr':
  assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R" "set p3 ⊆ carrier R"
  shows "poly_mult p1 (poly_add p2 p3) = poly_add (poly_mult p1 p2) (poly_mult p1 p3)"
  ⟨*proof*⟩

lemma poly_mult_r_distr:
  assumes "subring K R" "polynomial K p1" "polynomial K p2" "polynomial K p3"
  shows "poly_mult p1 (poly_add p2 p3) = poly_add (poly_mult p1 p2) (poly_mult p1 p3)"
  ⟨*proof*⟩

lemma poly_mult_replicate_zero:
  assumes "set p ⊆ carrier R"
  shows "poly_mult (replicate n **0**) p = []"
    and "poly_mult p (replicate n **0**) = []"
⟨*proof*⟩

lemma poly_mult_const':
  assumes "set p ⊆ carrier R" "a ∈ carrier R"
  shows "poly_mult [a] p = normalize (map (λb. a ⊗ b) p)"
    and "poly_mult p [a] = normalize (map (λb. a ⊗ b) p)"
⟨*proof*⟩

lemma poly_mult_const:
  assumes "subring K R" "polynomial K p" "a ∈ K - { **0** }"
  shows "poly_mult [a] p = map (λb. a ⊗ b) p"
    and "poly_mult p [a] = map (λb. a ⊗ b) p"
⟨*proof*⟩

lemma poly_mult_semiassoc:
  assumes "set p ⊆ carrier R" "set q ⊆ carrier R" **and** "a ∈ carrier R"
  shows "poly_mult (poly_mult [a] p) q = poly_mult [a] (poly_mult p q)"
⟨*proof*⟩

Note that "polynomial (carrier R) p" and "subring K p; polynomial K p" are "equivalent" assumptions for any lemma in ring which the result doesn't depend on K, because carrier is a subring and a polynomial for a subset of the carrier is a carrier polynomial. The decision between one of them should be based on how the lemma is going to be used and proved. These are some tips: (a) Lemmas about the algebraic structure of polynomials should use the latter option. (b) Also, if the lemma deals with lots of polynomials, then the latter option is preferred. (c) If the proof is going to be much easier with the first option, do not hesitate.

lemma poly_mult_monom':

assumes "set p ⊆ carrier R" "a ∈ carrier R"
shows "poly_mult (monom a n) p = normalize ((map ((⊗) a) p) @ (replicate n **0**))"
⟨*proof*⟩

lemma poly_mult_monom:
assumes "polynomial (carrier R) p" "a ∈ carrier R - { **0** }"
shows "poly_mult (monom a n) p =
(if p = [] then [] else (poly_mult [a] p) @ (replicate n **0**))"
⟨*proof*⟩

lemma poly_mult_one':
assumes "set p ⊆ carrier R"
shows "poly_mult [**1**] p = normalize p" **and** "poly_mult p [**1**] = normalize p"
⟨*proof*⟩

lemma poly_mult_one:
assumes "subring K R" "polynomial K p"
shows "poly_mult [**1**] p = p" **and** "poly_mult p [**1**] = p"
⟨*proof*⟩

lemma poly_mult_lead_coeff_aux:
assumes "subring K R" "polynomial K p1" "polynomial K p2" **and** "p1 ≠ []" **and** "p2 ≠ []"
shows "(coeff (poly_mult p1 p2)) (degree p1 + degree p2) = (lead_coeff p1) ⊗ (lead_coeff p2)"
⟨*proof*⟩

lemma poly_mult_degree_eq:
assumes "subring K R" "polynomial K p1" "polynomial K p2"
shows "degree (poly_mult p1 p2) = (if p1 = [] ∨ p2 = [] then 0 else (degree p1) + (degree p2))"
⟨*proof*⟩

lemma poly_mult_integral:
assumes "subring K R" "polynomial K p1" "polynomial K p2"
shows "poly_mult p1 p2 = [] ⟹ p1 = [] ∨ p2 = []"
⟨*proof*⟩

lemma poly_mult_lead_coeff:
assumes "subring K R" "polynomial K p1" "polynomial K p2" **and** "p1 ≠ []" **and** "p2 ≠ []"
shows "lead_coeff (poly_mult p1 p2) = (lead_coeff p1) ⊗ (lead_coeff p2)"
⟨*proof*⟩

lemma poly_mult_append_zero_lcancel:

assumes "subring K R" **and** "polynomial K p" "polynomial K q"
shows "poly_mult (p @ [**0**]) q = r @ [**0**] ⟹ poly_mult p q = r"
⟨*proof*⟩

lemma poly_mult_append_zero_rcancel:
assumes "subring K R" **and** "polynomial K p" "polynomial K q"
shows "poly_mult p (q @ [**0**]) = r @ [**0**] ⟹ poly_mult p q = r"
⟨*proof*⟩

end

33.7 Algebraic Structure of Polynomials

definition univ_poly :: "('a, 'b) ring_scheme ⇒'a set ⇒ ('a list) ring"
(<(<open_block notation=<postfix X>>_ [X]ₓ)> 80)
where "univ_poly R K =
(| carrier = { p. polynomial$_R$ K p },
mult = ring.poly_mult R,
one = [**1**$_R$],
zero = [],
add = ring.poly_add R |)"

These lemmas allow you to unfold one field of the record at a time.

lemma univ_poly_carrier: "polynomial$_R$ K p ⟷ p ∈ carrier (K[X]$_R$)"
⟨*proof*⟩

lemma univ_poly_mult: "mult (K[X]$_R$) = ring.poly_mult R"
⟨*proof*⟩

lemma univ_poly_one: "one (K[X]$_R$) = [**1**$_R$]"
⟨*proof*⟩

lemma univ_poly_zero: "zero (K[X]$_R$) = []"
⟨*proof*⟩

lemma univ_poly_add: "add (K[X]$_R$) = ring.poly_add R"
⟨*proof*⟩

lemma univ_poly_zero_closed [intro]: "[] ∈ carrier (K[X]$_R$)"
⟨*proof*⟩

context domain
begin

lemma poly_mult_monom_assoc:
assumes "set p ⊆ carrier R" "set q ⊆ carrier R" **and** "a ∈ carrier R"

```
    shows "poly_mult (poly_mult (monom a n) p) q =
           poly_mult (monom a n) (poly_mult p q)"
```
⟨*proof*⟩

context
  fixes K :: "'a set" **assumes** K: "subring K R"
begin

lemma univ_poly_is_monoid: "monoid (K[X])"
  ⟨*proof*⟩

declare poly_add.simps[simp del]

lemma univ_poly_is_abelian_monoid: "abelian_monoid (K[X])"
  ⟨*proof*⟩

lemma univ_poly_is_abelian_group: "abelian_group (K[X])"
⟨*proof*⟩

lemma univ_poly_is_ring: "ring (K[X])"
⟨*proof*⟩

lemma univ_poly_is_cring: "cring (K[X])"
⟨*proof*⟩

lemma univ_poly_is_domain: "domain (K[X])"
⟨*proof*⟩

declare poly_add.simps[simp]

lemma univ_poly_a_inv_def':
  assumes "p ∈ carrier (K[X])" **shows** "$\ominus_{K[X]}$ p = map (λa. ⊖ a) p"
⟨*proof*⟩

corollary univ_poly_a_inv_length:
  assumes "p ∈ carrier (K[X])" **shows** "length ($\ominus_{K[X]}$ p) = length p"
  ⟨*proof*⟩

corollary univ_poly_a_inv_degree:
  assumes "p ∈ carrier (K[X])" **shows** "degree ($\ominus_{K[X]}$ p) = degree p"
  ⟨*proof*⟩

33.8 Long Division Theorem

lemma long_division_theorem:
  assumes "polynomial K p" **and** "polynomial K b" "b ≠ []"

```
    and "lead_coeff b ∈ Units (R ⦇ carrier := K ⦈)"
  shows "∃q r. polynomial K q ∧ polynomial K r ∧
              p = (b ⊗_K[X] q) ⊕_K[X] r ∧ (r = [] ∨ degree r < degree
b)"
    (is "∃q r. ?long_division p q r")
```
⟨*proof*⟩

end

end

```
lemma (in domain) field_long_division_theorem:
  assumes "subfield K R" "polynomial K p" and "polynomial K b" "b ≠
[]"
  shows "∃q r. polynomial K q ∧ polynomial K r ∧
              p = (b ⊗_K[X] q) ⊕_K[X] r ∧ (r = [] ∨ degree r < degree
b)"
```
⟨*proof*⟩

The same theorem as above, but now, everything is in a shell.

```
lemma (in domain) field_long_division_theorem_shell:
  assumes "subfield K R" "p ∈ carrier (K[X])" and "b ∈ carrier (K[X])"
"b ≠ 0_K[X]"
  shows "∃q r. q ∈ carrier (K[X]) ∧ r ∈ carrier (K[X]) ∧
              p = (b ⊗_K[X] q) ⊕_K[X] r ∧ (r = 0_K[X] ∨ degree r < degree
b)"
```
⟨*proof*⟩

33.9 Consistency Rules

```
lemma polynomial_consistent [simp]:
  shows "polynomial_(R ⦇ carrier := K ⦈) K p ⟹ polynomial_R K p"
```
⟨*proof*⟩

```
lemma (in ring) eval_consistent [simp]:
  assumes "subring K R" shows "ring.eval (R ⦇ carrier := K ⦈) = eval"
```
⟨*proof*⟩

```
lemma (in ring) coeff_consistent [simp]:
  assumes "subring K R" shows "ring.coeff (R ⦇ carrier := K ⦈) = coeff"
```
⟨*proof*⟩

```
lemma (in ring) normalize_consistent [simp]:
  assumes "subring K R" shows "ring.normalize (R ⦇ carrier := K ⦈) =
normalize"
```
⟨*proof*⟩

```
lemma (in ring) poly_add_consistent [simp]:
```

assumes "subring K R" **shows** "ring.poly_add (R ⦇ carrier := K ⦈) = poly_add"

⟨*proof*⟩

lemma (in ring**)** poly_mult_consistent [simp]:
assumes "subring K R" **shows** "ring.poly_mult (R ⦇ carrier := K ⦈) = poly_mult"
⟨*proof*⟩

lemma (in domain**)** univ_poly_a_inv_consistent:
assumes "subring K R" "p ∈ carrier (K[X])"
shows "$\ominus_{K[X]}$ p = $\ominus_{(carrier\ R)[X]}$ p"
⟨*proof*⟩

lemma (in domain**)** univ_poly_a_minus_consistent:
assumes "subring K R" "q ∈ carrier (K[X])"
shows "p $\ominus_{K[X]}$ q = p $\ominus_{(carrier\ R)[X]}$ q"
⟨*proof*⟩

lemma (in ring**)** univ_poly_consistent:
assumes "subring K R"
shows "univ_poly (R ⦇ carrier := K ⦈) = univ_poly R"
⟨*proof*⟩

33.9.1 Corollaries

corollary (in ring**)** subfield_long_division_theorem_shell:
assumes "subfield K R" "p ∈ carrier (K[X])" **and** "b ∈ carrier (K[X])" "b ≠ $\mathbf{0}_{K[X]}$"
shows "∃q r. q ∈ carrier (K[X]) ∧ r ∈ carrier (K[X]) ∧
p = (b $\otimes_{K[X]}$ q) $\oplus_{K[X]}$ r ∧ (r = $\mathbf{0}_{K[X]}$ ∨ degree r < degree b)"
⟨*proof*⟩

corollary (in domain**)** univ_poly_is_euclidean:
assumes "subfield K R" **shows** "euclidean_domain (K[X]) degree"
⟨*proof*⟩

corollary (in domain**)** univ_poly_is_principal:
assumes "subfield K R" **shows** "principal_domain (K[X])"
⟨*proof*⟩

33.10 The Evaluation Homomorphism

lemma (in ring**)** eval_replicate:
assumes "set p ⊆ carrier R" "a ∈ carrier R"
shows "eval ((replicate n **0**) @ p) a = eval p a"
⟨*proof*⟩

lemma (in ring**)** eval_normalize:

assumes "set p ⊆ carrier R" "a ∈ carrier R"
shows "eval (normalize p) a = eval p a"
⟨*proof*⟩

lemma (**in** ring) eval_poly_add_aux:
assumes "set p ⊆ carrier R" "set q ⊆ carrier R" **and** "length p = length q" **and** "a ∈ carrier R"
shows "eval (poly_add p q) a = (eval p a) ⊕ (eval q a)"
⟨*proof*⟩

lemma (**in** ring) eval_poly_add:
assumes "set p ⊆ carrier R" "set q ⊆ carrier R" **and** "a ∈ carrier R"
shows "eval (poly_add p q) a = (eval p a) ⊕ (eval q a)"
⟨*proof*⟩

lemma (**in** ring) eval_append_aux:
assumes "set p ⊆ carrier R" **and** "b ∈ carrier R" **and** "a ∈ carrier R"
shows "eval (p @ [b]) a = ((eval p a) ⊗ a) ⊕ b"
⟨*proof*⟩

lemma (**in** ring) eval_append:
assumes "set p ⊆ carrier R" "set q ⊆ carrier R" **and** "a ∈ carrier R"
shows "eval (p @ q) a = ((eval p a) ⊗ (a [^] (length q))) ⊕ (eval q a)"
⟨*proof*⟩

lemma (**in** ring) eval_monom:
assumes "b ∈ carrier R" **and** "a ∈ carrier R"
shows "eval (monom b n) a = b ⊗ (a [^] n)"
⟨*proof*⟩

lemma (**in** cring) eval_poly_mult:
assumes "set p ⊆ carrier R" "set q ⊆ carrier R" **and** "a ∈ carrier R"
shows "eval (poly_mult p q) a = (eval p a) ⊗ (eval q a)"
⟨*proof*⟩

proposition (**in** cring) eval_is_hom:
assumes "subring K R" **and** "a ∈ carrier R"
shows "(λp. (eval p) a) ∈ ring_hom (K[X]) R"
⟨*proof*⟩

theorem (**in** domain) eval_cring_hom:
assumes "subring K R" **and** "a ∈ carrier R"
shows "ring_hom_cring (K[X]) R (λp. (eval p) a)"
⟨*proof*⟩

corollary (**in** domain) eval_ring_hom:
assumes "subring K R" **and** "a ∈ carrier R"

shows "ring_hom_ring (K[X]) R (λp. (eval p) a)"
⟨*proof*⟩

33.11 Homomorphisms

lemma (**in** ring_hom_ring) eval_hom':
assumes "a ∈ carrier R" **and** "set p ⊆ carrier R"
shows "h (R.eval p a) = eval (map h p) (h a)"
⟨*proof*⟩

lemma (**in** ring_hom_ring) eval_hom:
assumes "subring K R" **and** "a ∈ carrier R" **and** "p ∈ carrier (K[X])"
shows "h (R.eval p a) = eval (map h p) (h a)"
⟨*proof*⟩

lemma (**in** ring_hom_ring) coeff_hom':
assumes "set p ⊆ carrier R" **shows** "h (R.coeff p i) = coeff (map h p) i"
⟨*proof*⟩

lemma (**in** ring_hom_ring) poly_add_hom':
assumes "set p ⊆ carrier R" **and** "set q ⊆ carrier R"
shows "normalize (map h (R.poly_add p q)) = poly_add (map h p) (map h q)"
⟨*proof*⟩

lemma (**in** ring_hom_ring) poly_mult_hom':
assumes "set p ⊆ carrier R" **and** "set q ⊆ carrier R"
shows "normalize (map h (R.poly_mult p q)) = poly_mult (map h p) (map h q)"
⟨*proof*⟩

33.12 The X Variable

definition var :: "_ ⇒ 'a list" (‹X⇘_⇙›)
where "X_R = [$\mathbf{1}_R$, $\mathbf{0}_R$]"

lemma (**in** ring) eval_var:
assumes "x ∈ carrier R" **shows** "eval X x = x"
⟨*proof*⟩

lemma (**in** domain) var_closed:
assumes "subring K R" **shows** "X ∈ carrier (K[X])" **and** "polynomial K X"
⟨*proof*⟩

lemma (**in** domain) poly_mult_var':
assumes "set p ⊆ carrier R"
shows "poly_mult X p = normalize (p @ [**0**])"
and "poly_mult p X = normalize (p @ [**0**])"

⟨*proof*⟩

lemma (in domain**)** poly_mult_var:
assumes "subring K R" "p ∈ carrier (K[X])"
shows "p $\otimes_{K[X]}$ X = (if p = [] then [] else p @ [**0**])"
⟨*proof*⟩

lemma (in domain**)** var_pow_closed:
assumes "subring K R" **shows** "X [^]$_{K[X]}$ (n :: nat) ∈ carrier (K[X])"
⟨*proof*⟩

lemma (in domain**)** unitary_monom_eq_var_pow:
assumes "subring K R" **shows** "monom **1** n = X [^]$_{K[X]}$ n"
⟨*proof*⟩

lemma (in domain**)** monom_eq_var_pow:
assumes "subring K R" "a ∈ carrier R - { **0** }"
shows "monom a n = [a] $\otimes_{K[X]}$ (X [^]$_{K[X]}$ n)"
⟨*proof*⟩

lemma (in domain**)** eval_rewrite:
assumes "subring K R" **and** "p ∈ carrier (K[X])"
shows "p = (ring.eval (K[X])) (map poly_of_const p) X"
⟨*proof*⟩

lemma (in ring**)** dense_repr_set_fst:
assumes "set p ⊆ K" **shows** "fst ‘ (set (dense_repr p)) ⊆ K - { **0** }"
⟨*proof*⟩

lemma (in ring**)** dense_repr_set_snd:
shows "snd ‘ (set (dense_repr p)) ⊆ {..< length p}"
⟨*proof*⟩

lemma (in domain**)** dense_repr_monom_closed:
assumes "subring K R" "set p ⊆ K"
shows "t ∈ set (dense_repr p) ⟹ monom (fst t) (snd t) ∈ carrier (K[X])"
⟨*proof*⟩

lemma (in domain**)** monom_finsum_decomp:
assumes "subring K R" "p ∈ carrier (K[X])"
shows "p = ($\bigoplus_{K[X]}$ t ∈ set (dense_repr p). monom (fst t) (snd t))"
⟨*proof*⟩

lemma (in domain**)** var_pow_finsum_decomp:
assumes "subring K R" "p ∈ carrier (K[X])"
shows "p = ($\bigoplus_{K[X]}$ t ∈ set (dense_repr p). [fst t] $\otimes_{K[X]}$ (X [^]$_{K[X]}$ (snd t)))"
⟨*proof*⟩

corollary (in domain**)** hom_var_pow_finsum:
assumes "subring K R" **and** "p ∈ carrier (K[X])" "ring_hom_ring (K[X]) A h"
shows "h p = (⨁$_A$ t ∈ set (dense_repr p). h [fst t] ⊗$_A$ (h X [^]$_A$ (snd t)))"
⟨*proof*⟩

corollary (in domain**)** determination_of_hom:
assumes "subring K R"
and "ring_hom_ring (K[X]) A h" "ring_hom_ring (K[X]) A g"
and "⋀k. k ∈ K ⟹ h [k] = g [k]" **and** "h X = g X"
shows "⋀p. p ∈ carrier (K[X]) ⟹ h p = g p"
⟨*proof*⟩

corollary (in domain**)** eval_as_unique_hom:
assumes "subring K R" "x ∈ carrier R"
and "ring_hom_ring (K[X]) R h"
and "⋀k. k ∈ K ⟹ h [k] = k" **and** "h X = x"
shows "⋀p. p ∈ carrier (K[X]) ⟹ h p = eval p x"
⟨*proof*⟩

33.13 The Constant Term

definition (in ring**)** const_term :: "'a list ⇒ 'a"
where "const_term p = eval p **0**"

lemma (in ring**)** const_term_eq_last:
assumes "set p ⊆ carrier R" **and** "a ∈ carrier R"
shows "const_term (p @ [a]) = a"
⟨*proof*⟩

lemma (in ring**)** const_term_not_zero:
assumes "const_term p ≠ **0**" **shows** "p ≠ []"
⟨*proof*⟩

lemma (in ring**)** const_term_explicit:
assumes "set p ⊆ carrier R" "p ≠ []" **and** "const_term p = a"
obtains p' **where** "set p' ⊆ carrier R" **and** "p = p' @ [a]"
⟨*proof*⟩

lemma (in ring**)** const_term_zero:
assumes "subring K R" "polynomial K p" "p ≠ []" **and** "const_term p = **0**"
obtains p' **where** "polynomial K p'" "p' ≠ []" **and** "p = p' @ [**0**]"
⟨*proof*⟩

lemma (in cring**)** const_term_simprules:
shows "⋀p. set p ⊆ carrier R ⟹ const_term p ∈ carrier R"
and "⋀p q. ⟦ set p ⊆ carrier R; set q ⊆ carrier R ⟧ ⟹

```
                const_term (poly_mult p q) = const_term p ⊗ const_term
q"
    and "⋀p q. ⟦ set p ⊆ carrier R; set q ⊆ carrier R ⟧ ⟹
                const_term (poly_add  p q) = const_term p ⊕ const_term
q"
```
⟨*proof*⟩

```
lemma (in domain) const_term_simprules_shell:
  assumes "subring K R"
  shows "⋀p. p ∈ carrier (K[X]) ⟹ const_term p ∈ K"
    and "⋀p q. ⟦ p ∈ carrier (K[X]); q ∈ carrier (K[X]) ⟧ ⟹
                const_term (p ⊗_K[X] q) = const_term p ⊗ const_term q"
    and "⋀p q. ⟦ p ∈ carrier (K[X]); q ∈ carrier (K[X]) ⟧ ⟹
                const_term (p ⊕_K[X] q) = const_term p ⊕ const_term q"
    and "⋀p. p ∈ carrier (K[X]) ⟹ const_term (⊖_K[X] p) = ⊖ (const_term
p)"
```
⟨*proof*⟩

33.14 The Canonical Embedding of K in K[X]

```
lemma (in ring) poly_of_const_consistent:
  assumes "subring K R" shows "ring.poly_of_const (R (| carrier := K |))
= poly_of_const"
```
⟨*proof*⟩

```
lemma (in domain) canonical_embedding_is_hom:
  assumes "subring K R" shows "poly_of_const ∈ ring_hom (R (| carrier
:= K |)) (K[X])"
```
⟨*proof*⟩

```
lemma (in domain) canonical_embedding_ring_hom:
  assumes "subring K R" shows "ring_hom_ring (R (| carrier := K |)) (K[X])
poly_of_const"
```
⟨*proof*⟩

```
lemma (in field) poly_of_const_over_carrier:
  shows "poly_of_const ‘ (carrier R) = { p ∈ carrier ((carrier R)[X]).
degree p = 0 }"
```
⟨*proof*⟩

```
lemma (in ring) poly_of_const_over_subfield:
  assumes "subfield K R" shows "poly_of_const ‘ K = { p ∈ carrier (K[X]).
degree p = 0 }"
```
⟨*proof*⟩

```
lemma (in field) univ_poly_carrier_subfield_of_consts:
  "subfield (poly_of_const ‘ (carrier R)) ((carrier R)[X])"
```
⟨*proof*⟩

proposition (**in** ring) univ_poly_subfield_of_consts:
assumes "subfield K R" **shows** "subfield (poly_of_const ‘ K) (K[X])"
⟨*proof*⟩

end

theory Embedded_Algebras
imports Subrings Generated_Groups
begin

34 Definitions

locale embedded_algebra =
K?: subfield K R + R?: ring R **for** K :: "'a set" **and** R :: "('a, 'b) ring_scheme"
(**structure**)

definition (**in** ring) line_extension :: "'a set ⇒ 'a ⇒ 'a set ⇒ 'a set"
where "line_extension K a E = (K #> a) <+>$_R$ E"

fun (**in** ring) Span :: "'a set ⇒ 'a list ⇒ 'a set"
where "Span K Us = foldr (line_extension K) Us { **0** }"

fun (**in** ring) combine :: "'a list ⇒ 'a list ⇒ 'a"
where
"combine (k # Ks) (u # Us) = (k ⊗ u) ⊕ (combine Ks Us)"
| "combine Ks Us = **0**"

inductive (**in** ring) independent :: "'a set ⇒ 'a list ⇒ bool"
where
li_Nil [simp, intro]: "independent K []"
| li_Cons: "⟦ u ∈ carrier R; u ∉ Span K Us; independent K Us ⟧ ⟹ independent K (u # Us)"

inductive (**in** ring) dimension :: "nat ⇒ 'a set ⇒ 'a set ⇒ bool"
where
zero_dim [simp, intro]: "dimension 0 K { **0** }"
| Suc_dim: "⟦ v ∈ carrier R; v ∉ E; dimension n K E ⟧ ⟹ dimension (Suc n) K (line_extension K v E)"

34.0.1 Syntactic Definitions

abbreviation (**in** ring) dependent :: "'a set ⇒ 'a list ⇒ bool"
where "dependent K U ≡ ¬ independent K U"

definition over :: "('a ⇒ 'b) ⇒ 'a ⇒ 'b" (**infixl** <over> 65)
where "f over a = f a"

context ring
begin

34.1 Basic Properties - First Part

lemma line_extension_consistent:
assumes "subring K R" **shows** "ring.line_extension (R ⦇ carrier := K ⦈) = line_extension"
⟨*proof*⟩

lemma Span_consistent:
assumes "subring K R" **shows** "ring.Span (R ⦇ carrier := K ⦈) = Span"
⟨*proof*⟩

lemma combine_in_carrier [simp, intro]:
"⟦ set Ks ⊆ carrier R; set Us ⊆ carrier R ⟧ ⟹ combine Ks Us ∈ carrier R"
⟨*proof*⟩

lemma combine_r_distr:
"⟦ set Ks ⊆ carrier R; set Us ⊆ carrier R ⟧ ⟹
k ∈ carrier R ⟹ k ⊗ (combine Ks Us) = combine (map ((⊗) k) Ks) Us"
⟨*proof*⟩

lemma combine_l_distr:
"⟦ set Ks ⊆ carrier R; set Us ⊆ carrier R ⟧ ⟹
u ∈ carrier R ⟹ (combine Ks Us) ⊗ u = combine Ks (map (λu'. u' ⊗ u) Us)"
⟨*proof*⟩

lemma combine_eq_foldr:
"combine Ks Us = foldr (λ(k, u). λl. (k ⊗ u) ⊕ l) (zip Ks Us) **0**"
⟨*proof*⟩

lemma combine_replicate:
"set Us ⊆ carrier R ⟹ combine (replicate (length Us) **0**) Us = **0**"
⟨*proof*⟩

lemma combine_take:
"combine (take (length Us) Ks) Us = combine Ks Us"
⟨*proof*⟩

lemma combine_append_zero:
"set Us ⊆ carrier R ⟹ combine (Ks @ [**0**]) Us = combine Ks Us"
⟨*proof*⟩

lemma combine_prepend_replicate:

"⟦ set Ks ⊆ carrier R; set Us ⊆ carrier R ⟧ ⟹
combine ((replicate n **0**) @ Ks) Us = combine Ks (drop n Us)"
⟨*proof*⟩

lemma combine_append_replicate:
"set Us ⊆ carrier R ⟹ combine (Ks @ (replicate n **0**)) Us = combine Ks Us"
⟨*proof*⟩

lemma combine_append:
assumes "length Ks = length Us"
and "set Ks ⊆ carrier R" "set Us ⊆ carrier R"
and "set Ks' ⊆ carrier R" "set Vs ⊆ carrier R"
shows "(combine Ks Us) ⊕ (combine Ks' Vs) = combine (Ks @ Ks') (Us @ Vs)"
⟨*proof*⟩

lemma combine_add:
assumes "length Ks = length Us" **and** "length Ks' = length Us"
and "set Ks ⊆ carrier R" "set Ks' ⊆ carrier R" "set Us ⊆ carrier R"
shows "(combine Ks Us) ⊕ (combine Ks' Us) = combine (map2 (⊕) Ks Ks') Us"
⟨*proof*⟩

lemma combine_normalize:
assumes "set Ks ⊆ carrier R" "set Us ⊆ carrier R" "combine Ks Us = a"
obtains Ks'
where "set (take (length Us) Ks) ⊆ set Ks'" "set Ks' ⊆ set (take (length Us) Ks) ∪ { **0** }"
and "length Ks' = length Us" "combine Ks' Us = a"
⟨*proof*⟩

lemma line_extension_mem_iff: "u ∈ line_extension K a E ⟷ (∃k ∈ K. ∃v ∈ E. u = k ⊗ a ⊕ v)"
⟨*proof*⟩

lemma line_extension_in_carrier:
assumes "K ⊆ carrier R" "a ∈ carrier R" "E ⊆ carrier R"
shows "line_extension K a E ⊆ carrier R"
⟨*proof*⟩

lemma Span_in_carrier:
assumes "K ⊆ carrier R" "set Us ⊆ carrier R"
shows "Span K Us ⊆ carrier R"
⟨*proof*⟩

34.2 Some Basic Properties of Linear Independence

lemma independent_in_carrier: "independent K Us ⟹ set Us ⊆ carrier R"
⟨*proof*⟩

lemma independent_backwards:
"independent K (u # Us) ⟹ u ∉ Span K Us"
"independent K (u # Us) ⟹ independent K Us"
"independent K (u # Us) ⟹ u ∈ carrier R"
⟨*proof*⟩

lemma dimension_independent [intro]: "independent K Us ⟹ dimension (length Us) K (Span K Us)"
⟨*proof*⟩

Now, we fix K, a subfield of the ring. Many lemmas would also be true for weaker structures, but our interest is to work with subfields, so generalization could be the subject of a future work.

context
fixes K :: "'a set" **assumes** K: "subfield K R"
begin

34.3 Basic Properties - Second Part

lemmas subring_props [simp] =
subringE[OF subfieldE(1)[OF K]]

lemma line_extension_is_subgroup:
assumes "subgroup E (add_monoid R)" "a ∈ carrier R"
shows "subgroup (line_extension K a E) (add_monoid R)"
⟨*proof*⟩

corollary Span_is_add_subgroup:
"set Us ⊆ carrier R ⟹ subgroup (Span K Us) (add_monoid R)"
⟨*proof*⟩

lemma line_extension_smult_closed:
assumes "⋀k v. ⟦ k ∈ K; v ∈ E ⟧ ⟹ k ⊗ v ∈ E" **and** "E ⊆ carrier R" "a ∈ carrier R"
shows "⋀k u. ⟦ k ∈ K; u ∈ line_extension K a E ⟧ ⟹ k ⊗ u ∈ line_extension K a E"
⟨*proof*⟩

lemma Span_subgroup_props [simp]:
assumes "set Us ⊆ carrier R"
shows "Span K Us ⊆ carrier R"
and "**0** ∈ Span K Us"
and "⋀v1 v2. ⟦ v1 ∈ Span K Us; v2 ∈ Span K Us ⟧ ⟹ (v1 ⊕ v2) ∈ Span K Us"

and "⋀v. v ∈ Span K Us ⟹ (⊖ v) ∈ Span K Us"
⟨*proof*⟩

lemma Span_smult_closed [simp]:
assumes "set Us ⊆ carrier R"
shows "⋀k v. ⟦ k ∈ K; v ∈ Span K Us ⟧ ⟹ k ⊗ v ∈ Span K Us"
⟨*proof*⟩

lemma Span_m_inv_simprule [simp]:
assumes "set Us ⊆ carrier R"
shows "⟦ k ∈ K - { **0** }; a ∈ carrier R ⟧ ⟹ k ⊗ a ∈ Span K Us ⟹ a ∈ Span K Us"
⟨*proof*⟩

34.4 Span as Linear Combinations

We show that Span is the set of linear combinations

lemma line_extension_of_combine_set:
assumes "u ∈ carrier R"
shows "line_extension K u { combine Ks Us | Ks. set Ks ⊆ K } =
{ combine Ks (u # Us) | Ks. set Ks ⊆ K }"
(is "?line_extension = ?combinations")
⟨*proof*⟩

lemma Span_eq_combine_set:
assumes "set Us ⊆ carrier R" **shows** "Span K Us = { combine Ks Us | Ks. set Ks ⊆ K }"
⟨*proof*⟩

lemma line_extension_of_combine_set_length_version:
assumes "u ∈ carrier R"
shows "line_extension K u { combine Ks Us | Ks. length Ks = length Us ∧ set Ks ⊆ K } =
{ combine Ks (u # Us) | Ks. length Ks = length (u # Us) ∧ set Ks ⊆ K }"
(is "?line_extension = ?combinations")
⟨*proof*⟩

lemma Span_eq_combine_set_length_version:
assumes "set Us ⊆ carrier R"
shows "Span K Us = { combine Ks Us | Ks. length Ks = length Us ∧ set Ks ⊆ K }"
⟨*proof*⟩

34.4.1 Corollaries

corollary Span_mem_iff_length_version:
assumes "set Us ⊆ carrier R"

shows "a ∈ Span K Us ⟷ (∃Ks. set Ks ⊆ K ∧ length Ks = length Us ∧ a = combine Ks Us)"
⟨*proof*⟩

corollary Span_mem_imp_non_trivial_combine:
assumes "set Us ⊆ carrier R" **and** "a ∈ Span K Us"
obtains k Ks
where "k ∈ K - { **0** }" "set Ks ⊆ K" "length Ks = length Us" "combine (k # Ks) (a # Us) = **0**"
⟨*proof*⟩

corollary Span_mem_iff:
assumes "set Us ⊆ carrier R" **and** "a ∈ carrier R"
shows "a ∈ Span K Us ⟷ (∃k ∈ K - { **0** }. ∃Ks. set Ks ⊆ K ∧ combine (k # Ks) (a # Us) = **0**)"
(**is** "?in_Span ⟷ ?exists_combine")
⟨*proof*⟩

34.5 Span as the minimal subgroup that contains K <#> set Us

Now we show the link between Span and Group.generate

lemma mono_Span:
assumes "set Us ⊆ carrier R" **and** "u ∈ carrier R"
shows "Span K Us ⊆ Span K (u # Us)"
⟨*proof*⟩

lemma Span_min:
assumes "set Us ⊆ carrier R" **and** "subgroup E (add_monoid R)"
shows "K <#> (set Us) ⊆ E ⟹ Span K Us ⊆ E"
⟨*proof*⟩

lemma Span_eq_generate:
assumes "set Us ⊆ carrier R" **shows** "Span K Us = generate (add_monoid R) (K <#> (set Us))"
⟨*proof*⟩

34.5.1 Corollaries

corollary Span_same_set:
assumes "set Us ⊆ carrier R"
shows "set Us = set Vs ⟹ Span K Us = Span K Vs"
⟨*proof*⟩

corollary Span_incl: "set Us ⊆ carrier R ⟹ K <#> (set Us) ⊆ Span K Us"
⟨*proof*⟩

corollary Span_base_incl: "set Us ⊆ carrier R ⟹ set Us ⊆ Span K Us"
⟨*proof*⟩

corollary mono_Span_sublist:
  assumes "set Us ⊆ set Vs" "set Vs ⊆ carrier R"
  shows "Span K Us ⊆ Span K Vs"
  ⟨*proof*⟩

corollary mono_Span_append:
  assumes "set Us ⊆ carrier R" "set Vs ⊆ carrier R"
  shows "Span K Us ⊆ Span K (Us @ Vs)"
    and "Span K Us ⊆ Span K (Vs @ Us)"
  ⟨*proof*⟩

corollary mono_Span_subset:
  assumes "set Us ⊆ Span K Vs" "set Vs ⊆ carrier R"
  shows "Span K Us ⊆ Span K Vs"
⟨*proof*⟩

lemma Span_strict_incl:
  assumes "set Us ⊆ carrier R" "set Vs ⊆ carrier R"
  shows "Span K Us ⊂ Span K Vs ⟹ (∃v ∈ set Vs. v ∉ Span K Us)"
⟨*proof*⟩

lemma Span_append_eq_set_add:
  assumes "set Us ⊆ carrier R" **and** "set Vs ⊆ carrier R"
  shows "Span K (Us @ Vs) = (Span K Us <+>$_R$ Span K Vs)"
  ⟨*proof*⟩

34.6 Characterisation of Linearly Independent "Sets"

declare independent_backwards [intro]
declare independent_in_carrier [intro]

lemma independent_distinct: "independent K Us ⟹ distinct Us"
⟨*proof*⟩

lemma independent_strict_incl:
  assumes "independent K (u # Us)" **shows** "Span K Us ⊂ Span K (u # Us)"
⟨*proof*⟩

corollary independent_replacement:
  assumes "independent K (u # Us)" **and** "independent K Vs"
  shows "Span K (u # Us) ⊆ Span K Vs ⟹ (∃v ∈ set Vs. independent K
(v # Us))"
⟨*proof*⟩

lemma independent_split:
  assumes "independent K (Us @ Vs)"
  shows "independent K Vs"
    and "independent K Us"

and "Span K Us ∩ Span K Vs = { **0** }"
⟨*proof*⟩

lemma independent_append:
assumes "independent K Us" **and** "independent K Vs" **and** "Span K Us ∩ Span K Vs = { **0** }"
shows "independent K (Us @ Vs)"
⟨*proof*⟩

lemma independent_imp_trivial_combine:
assumes "independent K Us"
shows "⋀Ks. ⟦ set Ks ⊆ K; combine Ks Us = **0** ⟧ ⟹ set (take (length Us) Ks) ⊆ { **0** }"
⟨*proof*⟩

lemma non_trivial_combine_imp_dependent:
assumes "set Ks ⊆ K" **and** "combine Ks Us = **0**" **and** "¬ set (take (length Us) Ks) ⊆ { **0** }"
shows "dependent K Us"
⟨*proof*⟩

lemma trivial_combine_imp_independent:
assumes "set Us ⊆ carrier R"
and "⋀Ks. ⟦ set Ks ⊆ K; combine Ks Us = **0** ⟧ ⟹ set (take (length Us) Ks) ⊆ { **0** }"
shows "independent K Us"
⟨*proof*⟩

corollary dependent_imp_non_trivial_combine:
assumes "set Us ⊆ carrier R" **and** "dependent K Us"
obtains Ks **where** "length Ks = length Us" "combine Ks Us = **0**" "set Ks ⊆ K" "set Ks ≠ { **0** }"
⟨*proof*⟩

corollary unique_decomposition:
assumes "independent K Us"
shows "a ∈ Span K Us ⟹ ∃!Ks. set Ks ⊆ K ∧ length Ks = length Us ∧ a = combine Ks Us"
⟨*proof*⟩

34.7 Replacement Theorem

lemma independent_rotate1_aux:
"independent K (u # Us @ Vs) ⟹ independent K ((Us @ [u]) @ Vs)"
⟨*proof*⟩

corollary independent_rotate1:
"independent K (Us @ Vs) ⟹ independent K ((rotate1 Us) @ Vs)"
⟨*proof*⟩

corollary independent_same_set:
assumes "set Us = set Vs" **and** "length Us = length Vs"
shows "independent K Us ⟹ independent K Vs"
⟨*proof*⟩

lemma replacement_theorem:
assumes "independent K (Us' @ Us)" **and** "independent K Vs"
and "Span K (Us' @ Us) ⊆ Span K Vs"
shows "∃Vs'. set Vs' ⊆ set Vs ∧ length Vs' = length Us' ∧ independent K (Vs' @ Us)"
⟨*proof*⟩

corollary independent_length_le:
assumes "independent K Us" **and** "independent K Vs"
shows "set Us ⊆ Span K Vs ⟹ length Us ≤ length Vs"
⟨*proof*⟩

34.8 Dimension

lemma exists_base:
assumes "dimension n K E"
shows "∃Vs. set Vs ⊆ carrier R ∧ independent K Vs ∧ length Vs = n ∧ Span K Vs = E"
(is "∃Vs. ?base K Vs E n")
⟨*proof*⟩

lemma dimension_zero: "dimension 0 K E ⟹ E = { **0** }"
⟨*proof*⟩

lemma dimension_one [iff]: "dimension 1 K K"
⟨*proof*⟩

lemma dimensionI:
assumes "independent K Us" "Span K Us = E"
shows "dimension (length Us) K E"
⟨*proof*⟩

lemma space_subgroup_props:
assumes "dimension n K E"
shows "E ⊆ carrier R"
and "**0** ∈ E"
and "⋀v1 v2. ⟦ v1 ∈ E; v2 ∈ E ⟧ ⟹ (v1 ⊕ v2) ∈ E"
and "⋀v. v ∈ E ⟹ (⊖ v) ∈ E"
and "⋀k v. ⟦ k ∈ K; v ∈ E ⟧ ⟹ k ⊗ v ∈ E"
and "⟦ k ∈ K - { **0** }; a ∈ carrier R ⟧ ⟹ k ⊗ a ∈ E ⟹ a ∈ E"
⟨*proof*⟩

lemma independent_length_le_dimension:
assumes "dimension n K E" **and** "independent K Us" "set Us ⊆ E"
shows "length Us ≤ n"
⟨*proof*⟩

lemma dimension_is_inj:
assumes "dimension n K E" **and** "dimension m K E"
shows "n = m"
⟨*proof*⟩

corollary independent_length_eq_dimension:
assumes "dimension n K E" **and** "independent K Us" "set Us ⊆ E"
shows "length Us = n ⟷ Span K Us = E"
⟨*proof*⟩

lemma complete_base:
assumes "dimension n K E" **and** "independent K Us" "set Us ⊆ E"
shows "∃Vs. length (Vs @ Us) = n ∧ independent K (Vs @ Us) ∧ Span K (Vs @ Us) = E"
⟨*proof*⟩

lemma filter_base:
assumes "set Us ⊆ carrier R"
obtains Vs **where** "set Vs ⊆ carrier R" **and** "independent K Vs" **and** "Span K Vs = Span K Us"
⟨*proof*⟩

lemma dimension_backwards:
"dimension (Suc n) K E ⟹ ∃v ∈ carrier R. ∃E'. dimension n K E' ∧ v ∉ E' ∧ E = line_extension K v E'"
⟨*proof*⟩

lemma dimension_direct_sum_space:
assumes "dimension n K E" **and** "dimension m K F" **and** "E ∩ F = { **0** }"
shows "dimension (n + m) K (E <+>$_R$ F)"
⟨*proof*⟩

lemma dimension_sum_space:
assumes "dimension n K E" **and** "dimension m K F" **and** "dimension k K (E ∩ F)"
shows "dimension (n + m - k) K (E <+>$_R$ F)"
⟨*proof*⟩

end

end

lemma (in ring**)** telescopic_base_aux:
assumes "subfield K R" "subfield F R"
and "dimension n K F" **and** "dimension 1 F E"
shows "dimension n K E"
⟨*proof*⟩

lemma (in ring**)** telescopic_base:
assumes "subfield K R" "subfield F R"
and "dimension n K F" **and** "dimension m F E"
shows "dimension (n * m) K E"
⟨*proof*⟩

context ring_hom_ring
begin

lemma combine_hom:
"⟦ set Ks ⊆ carrier R; set Us ⊆ carrier R ⟧ ⟹ combine (map h Ks) (map h Us) = h (R.combine Ks Us)"
⟨*proof*⟩

lemma line_extension_hom:
assumes "K ⊆ carrier R" "a ∈ carrier R" "E ⊆ carrier R"
shows "line_extension (h ‘ K) (h a) (h ‘ E) = h ‘ R.line_extension K a E"
⟨*proof*⟩

lemma Span_hom:
assumes "K ⊆ carrier R" "set Us ⊆ carrier R"
shows "Span (h ‘ K) (map h Us) = h ‘ R.Span K Us"
⟨*proof*⟩

lemma inj_on_subgroup_iff_trivial_ker:
assumes "subgroup H (add_monoid R)"
shows "inj_on h H ⟷ a_kernel (R (| carrier := H |)) S h = { **0** }"
⟨*proof*⟩

corollary inj_on_Span_iff_trivial_ker:
assumes "subfield K R" "set Us ⊆ carrier R"
shows "inj_on h (R.Span K Us) ⟷ a_kernel (R (| carrier := R.Span K Us |)) S h = { **0** }"
⟨*proof*⟩

context
fixes K :: "'a set" **assumes** K: "subfield K R" **and** one_zero: "$\mathbf{1}_S \neq \mathbf{0}_S$"
begin

lemma inj_hom_preserves_independent:

assumes "inj_on h (R.Span K Us)"
and "R.independent K Us" **shows** "independent (h ‘ K) (map h Us)"
⟨*proof*⟩

corollary inj_hom_dimension:
assumes "inj_on h E"
and "R.dimension n K E" **shows** "dimension n (h ‘ K) (h ‘ E)"
⟨*proof*⟩

corollary rank_nullity_theorem:
assumes "R.dimension n K E" **and** "R.dimension m K (a_kernel (R ⦇ carrier := E ⦈) S h)"
shows "dimension (n - m) (h ‘ K) (h ‘ E)"
⟨*proof*⟩

end

end

lemma (**in** ring_hom_ring)
assumes "subfield K R" **and** "set Us ⊆ carrier R" **and** "$\mathbf{1}_S \neq \mathbf{0}_S$"
and "independent (h ‘ K) (map h Us)" **shows** "R.independent K Us"
⟨*proof*⟩

34.9 Finite Dimension

definition (**in** ring) finite_dimension :: "’a set ⇒ ’a set ⇒ bool"
where "finite_dimension K E ⟷ (∃n. dimension n K E)"

abbreviation (**in** ring) infinite_dimension :: "’a set ⇒ ’a set ⇒ bool"
where "infinite_dimension K E ≡ ¬ finite_dimension K E"

definition (**in** ring) dim :: "’a set ⇒ ’a set ⇒ nat"
where "dim K E = (THE n. dimension n K E)"

locale subalgebra = subgroup V "add_monoid R" **for** K **and** V **and** R (**structure**)
+
assumes smult_closed: "⟦ k ∈ K; v ∈ V ⟧ ⟹ k ⊗ v ∈ V"

34.9.1 Basic Properties

lemma (**in** ring) unique_dimension:
assumes "subfield K R" **and** "finite_dimension K E" **shows** "∃!n. dimension n K E"
⟨*proof*⟩

lemma (**in** ring) finite_dimensionI:
assumes "dimension n K E" **shows** "finite_dimension K E"
⟨*proof*⟩

lemma (in ring**)** finite_dimensionE:
assumes "subfield K R" **and** "finite_dimension K E" **shows** "dimension ((dim over K) E) K E"
⟨*proof*⟩

lemma (in ring**)** dimI:
assumes "subfield K R" **and** "dimension n K E" **shows** "(dim over K) E = n"
⟨*proof*⟩

lemma (in ring**)** finite_dimensionE' [elim]:
assumes "finite_dimension K E" **and** "⋀n. dimension n K E ⟹ P" **shows** P
⟨*proof*⟩

lemma (in ring**)** Span_finite_dimension:
assumes "subfield K R" **and** "set Us ⊆ carrier R"
shows "finite_dimension K (Span K Us)"
⟨*proof*⟩

lemma (in ring**)** carrier_is_subalgebra:
assumes "K ⊆ carrier R" **shows** "subalgebra K (carrier R) R"
⟨*proof*⟩

lemma (in ring**)** subalgebra_in_carrier:
assumes "subalgebra K V R" **shows** "V ⊆ carrier R"
⟨*proof*⟩

lemma (in ring**)** subalgebra_inter:
assumes "subalgebra K V R" **and** "subalgebra K V' R" **shows** "subalgebra K (V ∩ V') R"
⟨*proof*⟩

lemma (in ring_hom_ring**)** img_is_subalgebra:
assumes "K ⊆ carrier R" **and** "subalgebra K V R" **shows** "subalgebra (h ‘ K) (h ‘ V) S"
⟨*proof*⟩

lemma (in ring**)** ideal_is_subalgebra:
assumes "K ⊆ carrier R" "ideal I R" **shows** "subalgebra K I R"
⟨*proof*⟩

lemma (in ring**)** Span_is_subalgebra:
assumes "subfield K R" "set Us ⊆ carrier R" **shows** "subalgebra K (Span K Us) R"
⟨*proof*⟩

lemma (in ring**)** finite_dimension_imp_subalgebra:
assumes "subfield K R" "finite_dimension K E" **shows** "subalgebra K E

R"
⟨*proof*⟩

lemma (in ring**)** subalgebra_Span_incl:
assumes "subfield K R" **and** "subalgebra K V R" "set Us ⊆ V" **shows** "Span K Us ⊆ V"
⟨*proof*⟩

lemma (in ring**)** Span_subalgebra_minimal:
assumes "subfield K R" "set Us ⊆ carrier R"
shows "Span K Us = ⋂ { V. subalgebra K V R ∧ set Us ⊆ V }"
⟨*proof*⟩

lemma (in ring**)** Span_subalgebraI:
assumes "subfield K R"
and "subalgebra K E R" "set Us ⊆ E"
and "⋀V. ⟦ subalgebra K V R; set Us ⊆ V ⟧ ⟹ E ⊆ V"
shows "E = Span K Us"
⟨*proof*⟩

lemma (in ring**)** subalbegra_incl_imp_finite_dimension:
assumes "subfield K R" **and** "finite_dimension K E"
and "subalgebra K V R" "V ⊆ E" **shows** "finite_dimension K V"
⟨*proof*⟩

lemma (in ring_hom_ring**)** infinite_dimension_hom:
assumes "subfield K R" **and** "$1_S \neq 0_S$" **and** "inj_on h E" **and** "subalgebra K E R"
shows "R.infinite_dimension K E ⟹ infinite_dimension (h ‘ K) (h ‘ E)"
⟨*proof*⟩

34.9.2 Reformulation of some lemmas in this new language.

lemma (in ring**)** sum_space_dim:
assumes "subfield K R" "finite_dimension K E" "finite_dimension K F"
shows "finite_dimension K (E <+>$_R$ F)"
and "((dim over K) (E <+>$_R$ F)) = ((dim over K) E) + ((dim over K) F) - ((dim over K) (E ∩ F))"
⟨*proof*⟩

lemma (in ring**)** telescopic_base_dim:
assumes "subfield K R" "subfield F R" **and** "finite_dimension K F" **and** "finite_dimension F E"
shows "finite_dimension K E" **and** "(dim over K) E = ((dim over K) F) * ((dim over F) E)"
⟨*proof*⟩

end

theory Polynomial_Divisibility
imports Polynomials Embedded_Algebras "HOL-Library.Multiset"

begin

35 Divisibility of Polynomials

35.1 Definitions

abbreviation poly_ring :: "_ ⇒ ('a list) ring"
where "poly_ring R ≡ univ_poly R (carrier R)"

abbreviation pirreducible :: "_ ⇒ 'a set ⇒ 'a list ⇒ bool" (‹pirreducible‹ ›)
where "pirreducible$_R$ K p ≡ ring_irreducible$_{(univ_poly\ R\ K)}$ p"

abbreviation pprime :: "_ ⇒ 'a set ⇒ 'a list ⇒ bool" (‹pprime‹ ›)
where "pprime$_R$ K p ≡ ring_prime$_{(univ_poly\ R\ K)}$ p"

definition pdivides :: "_ ⇒ 'a list ⇒ 'a list ⇒ bool" (**infix** ‹pdivides‹ ›
65)
where "p pdivides$_R$ q = p divides$_{(univ_poly\ R\ (carrier\ R))}$ q"

definition rupture :: "_ ⇒ 'a set ⇒ 'a list ⇒ (('a list) set) ring"
(‹Rupt‹ ›)
where "Rupt$_R$ K p = (K[X]$_R$) Quot (PIdl$_{K[X]_R}$ p)"

abbreviation (**in** ring) rupture_surj :: "'a set ⇒ 'a list ⇒ 'a list ⇒
('a list) set"
where "rupture_surj K p ≡ (λq. (PIdl$_{K[X]}$ p) +>$_{K[X]}$ q)"

35.2 Basic Properties

lemma (**in** ring) carrier_polynomial_shell [intro]:
assumes "subring K R" **and** "p ∈ carrier (K[X])" **shows** "p ∈ carrier
(poly_ring R)"
⟨*proof*⟩

lemma (**in** domain) pdivides_zero:
assumes "subring K R" **and** "p ∈ carrier (K[X])" **shows** "p pdivides []"
⟨*proof*⟩

lemma (**in** domain) zero_pdivides_zero: "[] pdivides []"
⟨*proof*⟩

lemma (**in** domain) zero_pdivides:
shows "[] pdivides p ⟷ p = []"
⟨*proof*⟩

lemma (in domain**)** pprime_iff_pirreducible:
assumes "subfield K R" **and** "p ∈ carrier (K[X])"
shows "pprime K p ⟷ pirreducible K p"
⟨*proof*⟩

lemma (in domain**)** pirreducibleE:
assumes "subring K R" "p ∈ carrier (K[X])" "pirreducible K p"
shows "p ≠ []" "p ∉ Units (K[X])"
and "⋀q r. ⟦ q ∈ carrier (K[X]); r ∈ carrier (K[X])⟧ ⟹
p = q ⊗$_{K[X]}$ r ⟹ q ∈ Units (K[X]) ∨ r ∈ Units (K[X])"
⟨*proof*⟩

lemma (in domain**)** pirreducibleI:
assumes "subring K R" "p ∈ carrier (K[X])" "p ≠ []" "p ∉ Units (K[X])"
and "⋀q r. ⟦ q ∈ carrier (K[X]); r ∈ carrier (K[X])⟧ ⟹
p = q ⊗$_{K[X]}$ r ⟹ q ∈ Units (K[X]) ∨ r ∈ Units (K[X])"
shows "pirreducible K p"
⟨*proof*⟩

lemma (in domain**)** univ_poly_carrier_units_incl:
shows "Units ((carrier R) [X]) ⊆ { [k] | k. k ∈ carrier R - { **0** }
}"
⟨*proof*⟩

lemma (in field**)** univ_poly_carrier_units:
"Units ((carrier R) [X]) = { [k] | k. k ∈ carrier R - { **0** } }"
⟨*proof*⟩

lemma (in domain**)** univ_poly_units_incl:
assumes "subring K R" **shows** "Units (K[X]) ⊆ { [k] | k. k ∈ K - {
0 } }"
⟨*proof*⟩

lemma (in ring**)** univ_poly_units:
assumes "subfield K R" **shows** "Units (K[X]) = { [k] | k. k ∈ K - {
0 } }"
⟨*proof*⟩

lemma (in domain**)** univ_poly_units':
assumes "subfield K R" **shows** "p ∈ Units (K[X]) ⟷ p ∈ carrier (K[X])
∧ p ≠ [] ∧ degree p = 0"
⟨*proof*⟩

corollary (in domain**)** rupture_one_not_zero:
assumes "subfield K R" **and** "p ∈ carrier (K[X])" **and** "degree p > 0"
shows "**1**$_{Rupt\ K\ p}$ ≠ **0**$_{Rupt\ K\ p}$"
⟨*proof*⟩

corollary (**in** ring) pirreducible_degree:
  assumes "subfield K R" "p ∈ carrier (K[X])" "pirreducible K p"
  shows "degree p ≥ 1"
⟨*proof*⟩

corollary (**in** domain) univ_poly_not_field:
  assumes "subring K R" **shows** "¬ field (K[X])"
⟨*proof*⟩

lemma (**in** domain) rupture_is_field_iff_pirreducible:
  assumes "subfield K R" **and** "p ∈ carrier (K[X])"
  shows "field (Rupt K p) ⟷ pirreducible K p"
⟨*proof*⟩

lemma (**in** domain) rupture_surj_hom:
  assumes "subring K R" **and** "p ∈ carrier (K[X])"
  shows "(rupture_surj K p) ∈ ring_hom (K[X]) (Rupt K p)"
    and "ring_hom_ring (K[X]) (Rupt K p) (rupture_surj K p)"
⟨*proof*⟩

corollary (**in** domain) rupture_surj_norm_is_hom:
  assumes "subring K R" **and** "p ∈ carrier (K[X])"
  shows "((rupture_surj K p) ∘ poly_of_const) ∈ ring_hom (R ⦇ carrier := K ⦈) (Rupt K p)"
  ⟨*proof*⟩

lemma (**in** domain) norm_map_in_poly_ring_carrier:
  assumes "p ∈ carrier (poly_ring R)" **and** "⋀a. a ∈ carrier R ⟹ f a ∈ carrier (poly_ring R)"
  shows "ring.normalize (poly_ring R) (map f p) ∈ carrier (poly_ring (poly_ring R))"
⟨*proof*⟩

lemma (**in** domain) map_in_poly_ring_carrier:
  assumes "p ∈ carrier (poly_ring R)" **and** "⋀a. a ∈ carrier R ⟹ f a ∈ carrier (poly_ring R)"
    and "⋀a. a ≠ **0** ⟹ f a ≠ []"
  shows "map f p ∈ carrier (poly_ring (poly_ring R))"
⟨*proof*⟩

lemma (**in** domain) map_norm_in_poly_ring_carrier:
  assumes "subring K R" **and** "p ∈ carrier (K[X])"
  shows "map poly_of_const p ∈ carrier (poly_ring (K[X]))"
  ⟨*proof*⟩

lemma (**in** domain) polynomial_rupture:
  assumes "subring K R" **and** "p ∈ carrier (K[X])"
  shows "(ring.eval (Rupt K p)) (map ((rupture_surj K p) ∘ poly_of_const) p) (rupture_surj K p X) = $\mathbf{0}_{\text{Rupt K p}}$"

⟨*proof*⟩

35.3 Division

definition (in ring**)** long_divides :: "'a list ⇒ 'a list ⇒ ('a list × 'a list) ⇒ bool"
  where "long_divides p q t ⟷
          — i (t ∈ carrier (poly_ring R) × carrier (poly_ring R))
∧
          — ii (p = (q ⊗$_{\text{poly_ring R}}$ (fst t)) ⊕$_{\text{poly_ring R}}$ (snd t)) ∧
          — iii (snd t = [] ∨ degree (snd t) < degree q)"

definition (in ring**)** long_division :: "'a list ⇒ 'a list ⇒ ('a list × 'a list)"
  where "long_division p q = (THE t. long_divides p q t)"

definition (in ring**)** pdiv :: "'a list ⇒ 'a list ⇒ 'a list" (**infixl** <pdiv> 65)
  where "p pdiv q = (if q = [] then [] else fst (long_division p q))"

definition (in ring**)** pmod :: "'a list ⇒ 'a list ⇒ 'a list" (**infixl** <pmod> 65)
  where "p pmod q = (if q = [] then p else snd (long_division p q))"

lemma (in ring**)** long_dividesI:
  assumes "b ∈ carrier (poly_ring R)" **and** "r ∈ carrier (poly_ring R)"
      and "p = (q ⊗$_{\text{poly_ring R}}$ b) ⊕$_{\text{poly_ring R}}$ r" **and** "r = [] ∨ degree r < degree q"
    shows "long_divides p q (b, r)"
  ⟨*proof*⟩

lemma (in domain**)** exists_long_division:
  assumes "subfield K R" **and** "p ∈ carrier (K[X])" **and** "q ∈ carrier (K[X])" "q ≠ []"
  obtains b r **where** "b ∈ carrier (K[X])" **and** "r ∈ carrier (K[X])" **and** "long_divides p q (b, r)"
  ⟨*proof*⟩

lemma (in domain**)** exists_unique_long_division:
  assumes "subfield K R" **and** "p ∈ carrier (K[X])" **and** "q ∈ carrier (K[X])" "q ≠ []"
  shows "∃!t. long_divides p q t"
⟨*proof*⟩

lemma (in domain**)** long_divisionE:
  assumes "subfield K R" **and** "p ∈ carrier (K[X])" **and** "q ∈ carrier (K[X])" "q ≠ []"
  shows "long_divides p q (p pdiv q, p pmod q)"

⟨*proof*⟩

lemma (in domain**)** long_divisionI:
assumes "subfield K R" **and** "p ∈ carrier (K[X])" **and** "q ∈ carrier (K[X])" "q ≠ []"
shows "long_divides p q (b, r) ⟹ (b, r) = (p pdiv q, p pmod q)"
⟨*proof*⟩

lemma (in domain**)** long_division_closed:
assumes "subfield K R" **and** "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
shows "p pdiv q ∈ carrier (K[X])" **and** "p pmod q ∈ carrier (K[X])"
⟨*proof*⟩

lemma (in domain**)** pdiv_pmod:
assumes "subfield K R" **and** "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
shows "p = (q ⊗$_{K[X]}$ (p pdiv q)) ⊕$_{K[X]}$ (p pmod q)"
⟨*proof*⟩

lemma (in domain**)** pmod_degree:
assumes "subfield K R" **and** "p ∈ carrier (K[X])" **and** "q ∈ carrier (K[X])" "q ≠ []"
shows "p pmod q = [] ∨ degree (p pmod q) < degree q"
⟨*proof*⟩

lemma (in domain**)** pmod_const:
assumes "subfield K R" **and** "p ∈ carrier (K[X])" "q ∈ carrier (K[X])" **and** "degree q > degree p"
shows "p pdiv q = []" **and** "p pmod q = p"
⟨*proof*⟩

lemma (in domain**)** long_division_zero:
assumes "subfield K R" **and** "q ∈ carrier (K[X])" **shows** "[] pdiv q = []" **and** "[] pmod q = []"
⟨*proof*⟩

lemma (in domain**)** long_division_a_inv:
assumes "subfield K R" **and** "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
shows "((⊖$_{K[X]}$ p) pdiv q) = ⊖$_{K[X]}$ (p pdiv q)" (**is** "?pdiv")
and "((⊖$_{K[X]}$ p) pmod q) = ⊖$_{K[X]}$ (p pmod q)" (**is** "?pmod")
⟨*proof*⟩

lemma (in domain**)** long_division_add:
assumes "subfield K R" **and** "a ∈ carrier (K[X])" "b ∈ carrier (K[X])" "q ∈ carrier (K[X])"
shows "(a ⊕$_{K[X]}$ b) pdiv q = (a pdiv q) ⊕$_{K[X]}$ (b pdiv q)" (**is** "?pdiv")
and "(a ⊕$_{K[X]}$ b) pmod q = (a pmod q) ⊕$_{K[X]}$ (b pmod q)" (**is** "?pmod")
⟨*proof*⟩

lemma (in domain**)** long_division_add_iff:

assumes "subfield K R"
and "a ∈ carrier (K[X])" "b ∈ carrier (K[X])" "c ∈ carrier (K[X])"
"q ∈ carrier (K[X])"
shows "a pmod q = b pmod q ⟷ (a $\oplus_{K[X]}$ c) pmod q = (b $\oplus_{K[X]}$ c) pmod q"
⟨*proof*⟩

lemma (**in** domain) pdivides_iff:
assumes "subfield K R" **and** "polynomial K p" "polynomial K q"
shows "p pdivides q ⟷ p $\text{divides}_{K[X]}$ q"
⟨*proof*⟩

lemma (**in** domain) pdivides_iff_shell:
assumes "subfield K R" **and** "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
shows "p pdivides q ⟷ p $\text{divides}_{K[X]}$ q"
⟨*proof*⟩

lemma (**in** domain) pmod_zero_iff_pdivides:
assumes "subfield K R" **and** "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
shows "p pmod q = [] ⟷ q pdivides p"
⟨*proof*⟩

lemma (**in** domain) same_pmod_iff_pdivides:
assumes "subfield K R" **and** "a ∈ carrier (K[X])" "b ∈ carrier (K[X])"
"q ∈ carrier (K[X])"
shows "a pmod q = b pmod q ⟷ q pdivides (a $\ominus_{K[X]}$ b)"
⟨*proof*⟩

lemma (**in** domain) pdivides_imp_degree_le:
assumes "subring K R" **and** "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
"q ≠ []"
shows "p pdivides q ⟹ degree p ≤ degree q"
⟨*proof*⟩

lemma (**in** domain) pprimeE:
assumes "subfield K R" "p ∈ carrier (K[X])" "pprime K p"
shows "p ≠ []" "p ∉ Units (K[X])"
and "⋀q r. ⟦ q ∈ carrier (K[X]); r ∈ carrier (K[X])⟧ ⟹
p pdivides (q $\otimes_{K[X]}$ r) ⟹ p pdivides q ∨ p pdivides r"
⟨*proof*⟩

lemma (**in** domain) pprimeI:
assumes "subfield K R" "p ∈ carrier (K[X])" "p ≠ []" "p ∉ Units (K[X])"
and "⋀q r. ⟦ q ∈ carrier (K[X]); r ∈ carrier (K[X])⟧ ⟹
p pdivides (q $\otimes_{K[X]}$ r) ⟹ p pdivides q ∨ p pdivides r"
shows "pprime K p"
⟨*proof*⟩

lemma (in domain**)** associated_polynomials_iff:
assumes "subfield K R" **and** "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
shows "p $\sim_{K[X]}$ q $\longleftrightarrow$ (∃k ∈ K - { **0** }. p = [k] $\otimes_{K[X]}$ q)"
⟨*proof*⟩

corollary (in domain**)** associated_polynomials_imp_same_length:
assumes "subring K R" **and** "p ∈ carrier (K[X])" **and** "q ∈ carrier (K[X])"
shows "p $\sim_{K[X]}$ q $\Longrightarrow$ length p = length q"
⟨*proof*⟩

lemma (in ring**)** divides_pirreducible_condition:
assumes "pirreducible K q" **and** "p ∈ carrier (K[X])"
shows "p $\text{divides}_{K[X]}$ q $\Longrightarrow$ p ∈ Units (K[X]) ∨ p $\sim_{K[X]}$ q"
⟨*proof*⟩

35.4 Polynomial Power

lemma (in domain**)** polynomial_pow_not_zero:
assumes "p ∈ carrier (poly_ring R)" **and** "p ≠ []"
shows "p $[\hat{}]_{\text{poly_ring R}}$ (n::nat) ≠ []"
⟨*proof*⟩

lemma (in domain**)** subring_polynomial_pow_not_zero:
assumes "subring K R" **and** "p ∈ carrier (K[X])" **and** "p ≠ []"
shows "p $[\hat{}]_{K[X]}$ (n::nat) ≠ []"
⟨*proof*⟩

lemma (in domain**)** polynomial_pow_degree:
assumes "p ∈ carrier (poly_ring R)"
shows "degree (p $[\hat{}]_{\text{poly_ring R}}$ n) = n * degree p"
⟨*proof*⟩

lemma (in domain**)** subring_polynomial_pow_degree:
assumes "subring K R" **and** "p ∈ carrier (K[X])"
shows "degree (p $[\hat{}]_{K[X]}$ n) = n * degree p"
⟨*proof*⟩

lemma (in domain**)** polynomial_pow_division:
assumes "p ∈ carrier (poly_ring R)" **and** "(n::nat) ≤ m"
shows "(p $[\hat{}]_{\text{poly_ring R}}$ n) pdivides (p $[\hat{}]_{\text{poly_ring R}}$ m)"
⟨*proof*⟩

lemma (in domain**)** subring_polynomial_pow_division:
assumes "subring K R" **and** "p ∈ carrier (K[X])" **and** "(n::nat) ≤ m"
shows "(p $[\hat{}]_{K[X]}$ n) $\text{divides}_{K[X]}$ (p $[\hat{}]_{K[X]}$ m)"
⟨*proof*⟩

lemma (in domain**)** pirreducible_pow_pdivides_iff:

assumes "subfield K R" "p ∈ carrier (K[X])" "q ∈ carrier (K[X])" "r ∈ carrier (K[X])"
and "pirreducible K p" **and** "¬ (p pdivides q)"
shows "(p [^]$_{K[X]}$ (n :: nat)) pdivides (q ⊗$_{K[X]}$ r) ⟷ (p [^]$_{K[X]}$ n) pdivides r"
⟨*proof*⟩

lemma (**in** domain) subring_degree_one_imp_pirreducible:
assumes "subring K R" **and** "a ∈ Units (R ⦇ carrier := K ⦈)" **and** "b ∈ K"
shows "pirreducible K [a, b]"
⟨*proof*⟩

lemma (**in** domain) degree_one_imp_pirreducible:
assumes "subfield K R" **and** "p ∈ carrier (K[X])" **and** "degree p = 1"
shows "pirreducible K p"
⟨*proof*⟩

lemma (**in** ring) degree_oneE[elim]:
assumes "p ∈ carrier (K[X])" **and** "degree p = 1"
and "⋀a b. ⟦ a ∈ K; a ≠ **0**; b ∈ K; p = [a, b] ⟧ ⟹ P"
shows P
⟨*proof*⟩

lemma (**in** domain) subring_degree_one_associatedI:
assumes "subring K R" **and** "a ∈ K" "a' ∈ K" **and** "b ∈ K" **and** "a ⊗ a' = **1**"
shows "[a , b] ∼$_{K[X]}$ [**1**, a' ⊗ b]"
⟨*proof*⟩

lemma (**in** domain) degree_one_associatedI:
assumes "subfield K R" **and** "p ∈ carrier (K[X])" **and** "degree p = 1"
shows "p ∼$_{K[X]}$ [**1**, inv (lead_coeff p) ⊗ (const_term p)]"
⟨*proof*⟩

35.5 Ideals

lemma (**in** domain) exists_unique_gen:
assumes "subfield K R" "ideal I (K[X])" "I ≠ { [] }"
shows "∃!p ∈ carrier (K[X]). lead_coeff p = **1** ∧ I = PIdl$_{K[X]}$ p"
(**is** "∃!p. ?generator p")
⟨*proof*⟩

proposition (**in** domain) exists_unique_pirreducible_gen:
assumes "subfield K R" "ring_hom_ring (K[X]) R h"
and "a_kernel (K[X]) R h ≠ { [] }" "a_kernel (K[X]) R h ≠ carrier (K[X])"
shows "∃!p ∈ carrier (K[X]). pirreducible K p ∧ lead_coeff p = **1** ∧ a_kernel (K[X]) R h = PIdl$_{K[X]}$ p"

```
    (is "∃!p. ?generator p")
```
⟨*proof*⟩

```
lemma (in domain) cgenideal_pirreducible:
  assumes "subfield K R" and "p ∈ carrier (K[X])" "pirreducible K p"

  shows "⟦ pirreducible K q; q ∈ PIdl_K[X] p ⟧ ⟹ p ~_K[X] q"
```
⟨*proof*⟩

35.6 Roots and Multiplicity

```
definition (in ring) is_root :: "'a list ⇒ 'a ⇒ bool"
  where "is_root p x ⟷ (x ∈ carrier R ∧ eval p x = 𝟬 ∧ p ≠ [])"

definition (in ring) alg_mult :: "'a list ⇒ 'a ⇒ nat"
  where "alg_mult p x =
           (if p = [] then 0 else
             (if x ∈ carrier R then Greatest (λ n. ([ 𝟭, ⊖ x ] [^]_poly_ring R
n) pdivides p) else 0))"

definition (in ring) roots :: "'a list ⇒ 'a multiset"
  where "roots p = Abs_multiset (alg_mult p)"

definition (in ring) roots_on :: "'a set ⇒ 'a list ⇒ 'a multiset"
  where "roots_on K p = roots p ∩# mset_set K"

definition (in ring) splitted :: "'a list ⇒ bool"
  where "splitted p ⟷ size (roots p) = degree p"

definition (in ring) splitted_on :: "'a set ⇒ 'a list ⇒ bool"
  where "splitted_on K p ⟷ size (roots_on K p) = degree p"

lemma (in domain) pdivides_imp_root_sharing:
  assumes "p ∈ carrier (poly_ring R)" "p pdivides q" and "a ∈ carrier
R"
  shows "eval p a = 𝟬 ⟹ eval q a = 𝟬"
```
⟨*proof*⟩

```
lemma (in domain) degree_one_root:
  assumes "subfield K R" and "p ∈ carrier (K[X])" and "degree p = 1"
  shows "eval p (⊖ (inv (lead_coeff p) ⊗ (const_term p))) = 𝟬"
    and "inv (lead_coeff p) ⊗ (const_term p) ∈ K"
```
⟨*proof*⟩
```
lemma (in domain) is_root_imp_pdivides:
  assumes "p ∈ carrier (poly_ring R)"
  shows "is_root p x ⟹ [ 𝟭, ⊖ x ] pdivides p"
```
⟨*proof*⟩

```
lemma (in domain) pdivides_imp_is_root:
```

assumes "p ≠ []" **and** "x ∈ carrier R"
shows "[**1**, ⊖ x] pdivides p ⟹ is_root p x"
⟨*proof*⟩

lemma (**in** domain) associated_polynomials_imp_same_is_root:
assumes "p ∈ carrier (poly_ring R)" **and** "q ∈ carrier (poly_ring R)" **and** "p ∼poly_ring R q"
shows "is_root p x ⟷ is_root q x"
⟨*proof*⟩

lemma (**in** ring) monic_degree_one_root_condition:
assumes "a ∈ carrier R" **shows** "is_root [**1**, ⊖ a] b ⟷ a = b"
⟨*proof*⟩

lemma (**in** field) degree_one_root_condition:
assumes "p ∈ carrier (poly_ring R)" **and** "degree p = 1"
shows "is_root p x ⟷ x = ⊖ (inv (lead_coeff p) ⊗ (const_term p))"
⟨*proof*⟩

lemma (**in** domain) is_root_poly_mult_imp_is_root:
assumes "p ∈ carrier (poly_ring R)" **and** "q ∈ carrier (poly_ring R)"
shows "is_root (p ⊗poly_ring R q) x ⟹ (is_root p x) ∨ (is_root q x)"
⟨*proof*⟩

lemma (**in** domain) degree_zero_imp_not_is_root:
assumes "p ∈ carrier (poly_ring R)" **and** "degree p = 0" **shows** "¬ is_root p x"
⟨*proof*⟩

lemma (**in** domain) finite_number_of_roots:
assumes "p ∈ carrier (poly_ring R)" **shows** "finite { x. is_root p x }"
⟨*proof*⟩

lemma (**in** domain) alg_multE:
assumes "x ∈ carrier R" **and** "p ∈ carrier (poly_ring R)" **and** "p ≠ []"
shows "([**1**, ⊖ x] [^]poly_ring R (alg_mult p x)) pdivides p"
and "⋀n. ([**1**, ⊖ x] [^]poly_ring R n) pdivides p ⟹ n ≤ alg_mult p x"
⟨*proof*⟩

lemma (**in** domain) le_alg_mult_imp_pdivides:
assumes "x ∈ carrier R" **and** "p ∈ carrier (poly_ring R)"
shows "n ≤ alg_mult p x ⟹ ([**1**, ⊖ x] [^]poly_ring R n) pdivides p"
⟨*proof*⟩

lemma (**in** domain) alg_mult_gt_zero_iff_is_root:

assumes "p ∈ carrier (poly_ring R)" **shows** "alg_mult p x > 0 ⟷ is_root p x"
⟨*proof*⟩

lemma (**in** domain) alg_mult_eq_count_roots:
assumes "p ∈ carrier (poly_ring R)" **shows** "alg_mult p = count (roots p)"
⟨*proof*⟩

lemma (**in** domain) roots_mem_iff_is_root:
assumes "p ∈ carrier (poly_ring R)" **shows** "x ∈# roots p ⟷ is_root p x"
⟨*proof*⟩

lemma (**in** domain) degree_zero_imp_empty_roots:
assumes "p ∈ carrier (poly_ring R)" **and** "degree p = 0" **shows** "roots p = {#}"
⟨*proof*⟩

lemma (**in** domain) degree_zero_imp_splitted:
assumes "p ∈ carrier (poly_ring R)" **and** "degree p = 0" **shows** "splitted p"
⟨*proof*⟩

lemma (**in** domain) roots_inclI':
assumes "p ∈ carrier (poly_ring R)" **and** "⋀a. ⟦ a ∈ carrier R; p ≠ [] ⟧ ⟹ alg_mult p a ≤ count m a"
shows "roots p ⊆# m"
⟨*proof*⟩

lemma (**in** domain) roots_inclI:
assumes "p ∈ carrier (poly_ring R)" **and** "q ∈ carrier (poly_ring R)" "q ≠ []"
and "⋀a. ⟦ a ∈ carrier R; p ≠ [] ⟧ ⟹ ([1, ⊖ a] [^]$_{\text{poly_ring R}}$ (alg_mult p a)) pdivides q"
shows "roots p ⊆# roots q"
⟨*proof*⟩

lemma (**in** domain) pdivides_imp_roots_incl:
assumes "p ∈ carrier (poly_ring R)" **and** "q ∈ carrier (poly_ring R)" "q ≠ []"
shows "p pdivides q ⟹ roots p ⊆# roots q"
⟨*proof*⟩

lemma (**in** domain) associated_polynomials_imp_same_roots:
assumes "p ∈ carrier (poly_ring R)" **and** "q ∈ carrier (poly_ring R)" **and** "p ∼$_{\text{poly_ring R}}$ q"
shows "roots p = roots q"
⟨*proof*⟩

lemma (in domain**)** monic_degree_one_roots:
assumes "a ∈ carrier R" **shows** "roots [1 , ⊖ a] = {# a #}"
⟨*proof*⟩

lemma (in domain**)** degree_one_roots:
assumes "a ∈ carrier R" "a' ∈ carrier R" **and** "b ∈ carrier R" **and** "a ⊗ a' = 1"
shows "roots [a , b] = {# ⊖ (a' ⊗ b) #}"
⟨*proof*⟩

lemma (in field**)** degree_one_imp_singleton_roots:
assumes "p ∈ carrier (poly_ring R)" **and** "degree p = 1"
shows "roots p = {# ⊖ (inv (lead_coeff p) ⊗ (const_term p)) #}"
⟨*proof*⟩

lemma (in field**)** degree_one_imp_splitted:
assumes "p ∈ carrier (poly_ring R)" **and** "degree p = 1" **shows** "splitted p"
⟨*proof*⟩

lemma (in field**)** no_roots_imp_same_roots:
assumes "p ∈ carrier (poly_ring R)" "p ≠ []" **and** "q ∈ carrier (poly_ring R)"
shows "roots p = {#} ⟹ roots (p ⊗$_{\text{poly_ring R}}$ q) = roots q"
⟨*proof*⟩

lemma (in field**)** poly_mult_degree_one_monic_imp_same_roots:
assumes "a ∈ carrier R" **and** "p ∈ carrier (poly_ring R)" "p ≠ []"
shows "roots ([1, ⊖ a] ⊗$_{\text{poly_ring R}}$ p) = add_mset a (roots p)"
⟨*proof*⟩

lemma (in domain**)** not_empty_rootsE[elim]:
assumes "p ∈ carrier (poly_ring R)" **and** "roots p ≠ {#}"
and "⋀a. ⟦ a ∈ carrier R; a ∈# roots p;
[1, ⊖ a] ∈ carrier (poly_ring R); [1, ⊖ a] pdivides p ⟧ ⟹ P"
shows P
⟨*proof*⟩

lemma (in field**)** associated_polynomials_imp_same_roots:
assumes "p ∈ carrier (poly_ring R)" "p ≠ []" **and** "q ∈ carrier (poly_ring R)" "q ≠ []"
shows "roots (p ⊗$_{\text{poly_ring R}}$ q) = roots p + roots q"
⟨*proof*⟩

lemma (in field**)** size_roots_le_degree:
assumes "p ∈ carrier (poly_ring R)" **shows** "size (roots p) ≤ degree p"

⟨*proof*⟩

lemma (in domain**)** pirreducible_roots:
assumes "p ∈ carrier (poly_ring R)" **and** "pirreducible (carrier R) p"
and "degree p ≠ 1"
shows "roots p = {#}"
⟨*proof*⟩

lemma (in field**)** pirreducible_imp_not_splitted:
assumes "p ∈ carrier (poly_ring R)" **and** "pirreducible (carrier R) p"
and "degree p ≠ 1"
shows "¬ splitted p"
⟨*proof*⟩

lemma (in field**)**
assumes "p ∈ carrier (poly_ring R)" **and** "q ∈ carrier (poly_ring R)"
and "pirreducible (carrier R) p" **and** "degree p ≠ 1"
shows "roots (p ⊗$_{\text{poly_ring R}}$ q) = roots q"
⟨*proof*⟩

lemma (in field**)** trivial_factors_imp_splitted:
assumes "p ∈ carrier (poly_ring R)"
and "⋀q. ⟦ q ∈ carrier (poly_ring R); pirreducible (carrier R) q;
q pdivides p ⟧ ⟹ degree q ≤ 1"
shows "splitted p"
⟨*proof*⟩

lemma (in field**)** pdivides_imp_splitted:
assumes "p ∈ carrier (poly_ring R)" **and** "q ∈ carrier (poly_ring R)"
"q ≠ []" **and** "splitted q"
shows "p pdivides q ⟹ splitted p"
⟨*proof*⟩

lemma (in field**)** splitted_imp_trivial_factors:
assumes "p ∈ carrier (poly_ring R)" "p ≠ []" **and** "splitted p"
shows "⋀q. ⟦ q ∈ carrier (poly_ring R); pirreducible (carrier R) q;
q pdivides p ⟧ ⟹ degree q = 1"
⟨*proof*⟩

35.7 Link between pmod and rupture_surj

lemma (in domain**)** rupture_surj_composed_with_pmod:
assumes "subfield K R" **and** "p ∈ carrier (K[X])" **and** "q ∈ carrier (K[X])"
shows "rupture_surj K p q = rupture_surj K p (q pmod p)"
⟨*proof*⟩

corollary (in domain**)** rupture_carrier_as_pmod_image:
assumes "subfield K R" **and** "p ∈ carrier (K[X])"
shows "(rupture_surj K p) ` ((λq. q pmod p) ` (carrier (K[X]))) = carrier

```
(Rupt K p)"
    (is "?lhs = ?rhs")
```
⟨*proof*⟩

lemma (**in** domain) rupture_surj_inj_on:
  assumes "subfield K R" **and** "p ∈ carrier (K[X])"
  shows "inj_on (rupture_surj K p) ((λq. q pmod p) ‘ (carrier (K[X])))"
⟨*proof*⟩

35.8 Dimension

definition (**in** ring) exp_base :: "'a ⇒ nat ⇒ 'a list"
  where "exp_base x n = map (λi. x [^] i) (rev [0..< n])"

lemma (**in** ring) exp_base_closed:
  assumes "x ∈ carrier R" **shows** "set (exp_base x n) ⊆ carrier R"
  ⟨*proof*⟩

lemma (**in** ring) exp_base_append:
  shows "exp_base x (n + m) = (map (λi. x [^] i) (rev [n..< n + m]))
@ exp_base x n"
  ⟨*proof*⟩

lemma (**in** ring) drop_exp_base:
  shows "drop n (exp_base x m) = exp_base x (m - n)"
⟨*proof*⟩

lemma (**in** ring) combine_eq_eval:
  shows "combine Ks (exp_base x (length Ks)) = eval Ks x"
  ⟨*proof*⟩

lemma (**in** domain) pmod_image_characterization:
  assumes "subfield K R" **and** "p ∈ carrier (K[X])" **and** "p ≠ []"
  shows "(λq. q pmod p) ‘ carrier (K[X]) = { q ∈ carrier (K[X]). length
q ≤ degree p }"
⟨*proof*⟩

lemma (**in** domain) Span_var_pow_base:
  assumes "subfield K R"
  shows "ring.Span (K[X]) (poly_of_const ‘ K) (ring.exp_base (K[X]) X
n) =
         { q ∈ carrier (K[X]). length q ≤ n }" (**is** "?lhs = ?rhs")
⟨*proof*⟩

lemma (**in** domain) var_pow_base_independent:
  assumes "subfield K R"
  shows "ring.independent (K[X]) (poly_of_const ‘ K) (ring.exp_base (K[X])
X n)"
⟨*proof*⟩

lemma (in domain**)** bounded_degree_dimension:
assumes "subfield K R"
shows "ring.dimension (K[X]) n (poly_of_const ‘ K) { q ∈ carrier (K[X]). length q ≤ n }"
⟨*proof*⟩

corollary (in domain**)** univ_poly_infinite_dimension:
assumes "subfield K R" **shows** "ring.infinite_dimension (K[X]) (poly_of_const ‘ K) (carrier (K[X]))"
⟨*proof*⟩

corollary (in domain**)** rupture_dimension:
assumes "subfield K R" **and** "p ∈ carrier (K[X])" **and** "degree p > 0"
shows "ring.dimension (Rupt K p) (degree p) ((rupture_surj K p) ‘ poly_of_const ‘ K) (carrier (Rupt K p))"
⟨*proof*⟩

end

theory Indexed_Polynomials
imports Weak_Morphisms "HOL-Library.Multiset" Polynomial_Divisibility

begin

36 Indexed Polynomials

In this theory, we build a basic framework to the study of polynomials on letters indexed by a set. The main interest is to then apply these concepts to the construction of the algebraic closure of a field.

36.1 Definitions

We formalize indexed monomials as multisets with its support a subset of the index set. On top of those, we build indexed polynomials which are simply functions mapping a monomial to its coefficient.

definition (in ring**)** indexed_const :: "'a ⇒ ('c multiset ⇒ 'a)"
where "indexed_const k = (λm. if m = {#} then k else **0**)"

definition (in ring**)** indexed_pmult :: "('c multiset ⇒ 'a) ⇒ 'c ⇒ ('c multiset ⇒ 'a)" **(infixl** ‹⨂› 65**)**
where "indexed_pmult P i = (λm. if i ∈# m then P (m - {# i #}) else **0**)"

definition (in ring**)** indexed_padd :: "_ ⇒ _ ⇒ ('c multiset ⇒ 'a)" **(infixl** ‹⨁› 65**)**

where "indexed_padd P Q = (λm. (P m) ⊕ (Q m))"

definition (**in** ring) indexed_var :: "'c ⇒ ('c multiset ⇒ 'a)" (‹$\mathcal{X}$_›)
where "indexed_var i = (indexed_const **1**) ⨂ i"

definition (**in** ring) index_free :: "('c multiset ⇒ 'a) ⇒ 'c ⇒ bool"
where "index_free P i ⟷ (∀m. i ∈# m ⟶ P m = **0**)"

definition (**in** ring) carrier_coeff :: "('c multiset ⇒ 'a) ⇒ bool"
where "carrier_coeff P ⟷ (∀m. P m ∈ carrier R)"

inductive_set (**in** ring) indexed_pset :: "'c set ⇒ 'a set ⇒ ('c multiset
⇒ 'a) set"
(‹(‹open_block notation=‹postfix $\mathcal{X}$››_ [$\mathcal{X}$_])› 80)
for I **and** K **where**
indexed_const: "k ∈ K ⟹ indexed_const k ∈ (K[$\mathcal{X}_I$])"
| indexed_padd: "⟦ P ∈ (K[$\mathcal{X}_I$]); Q ∈ (K[$\mathcal{X}_I$]) ⟧ ⟹ P ⨁ Q ∈ (K[$\mathcal{X}_I$])"
| indexed_pmult: "⟦ P ∈ (K[$\mathcal{X}_I$]); i ∈ I ⟧ ⟹ P ⨂ i ∈ (K[$\mathcal{X}_I$])"

fun (**in** ring) indexed_eval_aux :: "('c multiset ⇒ 'a) list ⇒ 'c ⇒ ('c
multiset ⇒ 'a)"
where "indexed_eval_aux Ps i = foldr (λP Q. (Q ⨂ i) ⨁ P) Ps (indexed_const
0)"

fun (**in** ring) indexed_eval :: "('c multiset ⇒ 'a) list ⇒ 'c ⇒ ('c multiset
⇒ 'a)"
where "indexed_eval Ps i = indexed_eval_aux (rev Ps) i"

36.2 Basic Properties

lemma (**in** ring) carrier_coeffE:
assumes "carrier_coeff P" **shows** "P m ∈ carrier R"
⟨*proof*⟩

lemma (**in** ring) indexed_zero_def: "indexed_const **0** = (λ_. **0**)"
⟨*proof*⟩

lemma (**in** ring) indexed_const_index_free: "index_free (indexed_const
k) i"
⟨*proof*⟩

lemma (**in** domain) indexed_var_not_index_free: "¬ index_free $\mathcal{X}_i$ i"
⟨*proof*⟩

lemma (**in** ring) indexed_pmult_zero [simp]:
shows "indexed_pmult (indexed_const **0**) i = indexed_const **0**"
⟨*proof*⟩

lemma (**in** ring) indexed_padd_zero:

assumes "carrier_coeff P" **shows** "P $\bigoplus$ (indexed_const **0**) = P" **and** "(indexed_const **0**) $\bigoplus$ P = P"
⟨*proof*⟩

lemma (**in** ring) indexed_padd_const:
shows "(indexed_const k1) $\bigoplus$ (indexed_const k2) = indexed_const (k1 $\oplus$ k2)"
⟨*proof*⟩

lemma (**in** ring) indexed_const_in_carrier:
assumes "K $\subseteq$ carrier R" **and** "k $\in$ K" **shows** "$\bigwedge$m. (indexed_const k) m $\in$ carrier R"
⟨*proof*⟩

lemma (**in** ring) indexed_padd_in_carrier:
assumes "carrier_coeff P" **and** "carrier_coeff Q" **shows** "carrier_coeff (indexed_padd P Q)"
⟨*proof*⟩

lemma (**in** ring) indexed_pmult_in_carrier:
assumes "carrier_coeff P" **shows** "carrier_coeff (P $\bigotimes$ i)"
⟨*proof*⟩

lemma (**in** ring) indexed_eval_aux_in_carrier:
assumes "list_all carrier_coeff Ps" **shows** "carrier_coeff (indexed_eval_aux Ps i)"
⟨*proof*⟩

lemma (**in** ring) indexed_eval_in_carrier:
assumes "list_all carrier_coeff Ps" **shows** "carrier_coeff (indexed_eval Ps i)"
⟨*proof*⟩

lemma (**in** ring) indexed_pset_in_carrier:
assumes "K $\subseteq$ carrier R" **and** "P $\in$ (K[$\mathcal{X}_{\text{I}}$])" **shows** "carrier_coeff P"
⟨*proof*⟩

36.3 Indexed Eval

lemma (**in** ring) exists_indexed_eval_aux_monomial:
assumes "carrier_coeff P" **and** "list_all carrier_coeff Qs"
and "count n i = k" **and** "P n $\neq$ **0**" **and** "list_all (λQ. index_free Q i) Qs"
obtains m **where** "count m i = length Qs + k" **and** "(indexed_eval_aux (Qs @ [P]) i) m $\neq$ **0**"
⟨*proof*⟩

lemma (**in** ring) indexed_eval_aux_monomial_degree_le:
assumes "list_all carrier_coeff Ps" **and** "list_all (λP. index_free P

```
i) Ps"
    and "(indexed_eval_aux Ps i) m ≠ 0" shows "count m i ≤ length Ps
- 1"
  ⟨proof⟩

lemma (in ring) indexed_eval_aux_is_inj:
  assumes "list_all carrier_coeff Ps" and "list_all (λP. index_free P
i) Ps"
      and "list_all carrier_coeff Qs" and "list_all (λQ. index_free Q
i) Qs"
    and "indexed_eval_aux Ps i = indexed_eval_aux Qs i" and "length Ps
= length Qs"
  shows "Ps = Qs"
  ⟨proof⟩

lemma (in ring) indexed_eval_aux_is_inj':
  assumes "list_all carrier_coeff Ps" and "list_all (λP. index_free P
i) Ps"
      and "list_all carrier_coeff Qs" and "list_all (λQ. index_free Q
i) Qs"
      and "carrier_coeff P" and "index_free P i" "P ≠ indexed_const 0"
      and "carrier_coeff Q" and "index_free Q i" "Q ≠ indexed_const 0"
    and "indexed_eval_aux (Ps @ [ P ]) i = indexed_eval_aux (Qs @ [ Q
]) i"
  shows "Ps = Qs" and "P = Q"
⟨proof⟩

lemma (in ring) exists_indexed_eval_monomial:
  assumes "carrier_coeff P" and "list_all carrier_coeff Qs"
    and "P n ≠ 0" and "list_all (λQ. index_free Q i) Qs"
  obtains m where "count m i = length Qs + (count n i)" and "(indexed_eval
(P # Qs) i) m ≠ 0"
  ⟨proof⟩

corollary (in ring) exists_indexed_eval_monomial':
  assumes "carrier_coeff P" and "list_all carrier_coeff Qs"
    and "P ≠ indexed_const 0" and "list_all (λQ. index_free Q i) Qs"
  obtains m where "count m i ≥ length Qs" and "(indexed_eval (P # Qs)
i) m ≠ 0"
⟨proof⟩

lemma (in ring) indexed_eval_monomial_degree_le:
  assumes "list_all carrier_coeff Ps" and "list_all (λP. index_free P
i) Ps"
    and "(indexed_eval Ps i) m ≠ 0" shows "count m i ≤ length Ps - 1"
  ⟨proof⟩

lemma (in ring) indexed_eval_is_inj:
  assumes "list_all carrier_coeff Ps" and "list_all (λP. index_free P
```

```
i) Ps"
      and "list_all carrier_coeff Qs" and "list_all (λQ. index_free Q
i) Qs"
      and "carrier_coeff P" and "index_free P i" "P ≠ indexed_const 0"
      and "carrier_coeff Q" and "index_free Q i" "Q ≠ indexed_const 0"
    and "indexed_eval (P # Ps) i = indexed_eval (Q # Qs) i"
  shows "Ps = Qs" and "P = Q"
⟨proof⟩

lemma (in ring) indexed_eval_inj_on_carrier:
  assumes "⋀P. P ∈ carrier L ⟹ carrier_coeff P" and "⋀P. P ∈ carrier
L ⟹ index_free P i" and "0_L = indexed_const 0"
  shows "inj_on (λPs. indexed_eval Ps i) (carrier (poly_ring L))"
⟨proof⟩
```

36.4 Link with Weak Morphisms

We study some elements of the contradiction needed in the algebraic closure existence proof.

```
context ring
begin

lemma (in ring) indexed_padd_index_free:
  assumes "index_free P i" and "index_free Q i" shows "index_free (P
⊕ Q) i"
  ⟨proof⟩

lemma (in ring) indexed_pmult_index_free:
  assumes "index_free P j" and "i ≠ j" shows "index_free (P ⊗ i) j"
  ⟨proof⟩

lemma (in ring) indexed_eval_index_free:
  assumes "list_all (λP. index_free P j) Ps" and "i ≠ j" shows "index_free
(indexed_eval Ps i) j"
⟨proof⟩

context
  fixes L :: "(('c multiset) ⇒ 'a) ring" and i :: 'c
  assumes hyps:
    — i   "field L"
    — ii  "⋀P. P ∈ carrier L ⟹ carrier_coeff P"
    — iii "⋀P. P ∈ carrier L ⟹ index_free P i"
    — iv  "0_L = indexed_const 0"
begin

interpretation L: field L
  ⟨proof⟩

interpretation UP: principal_domain "poly_ring L"
```

⟨*proof*⟩

abbreviation eval_pmod
where "eval_pmod q ≡ (λp. indexed_eval (L.pmod p q) i)"

abbreviation image_poly
where "image_poly q ≡ image_ring (eval_pmod q) (poly_ring L)"

lemma indexed_eval_is_weak_ring_morphism:
assumes "q ∈ carrier (poly_ring L)" **shows** "weak_ring_morphism (eval_pmod q) (PIdl$_{\text{poly_ring L q}}$) (poly_ring L)"
⟨*proof*⟩

lemma eval_norm_eq_id:
assumes "q ∈ carrier (poly_ring L)" **and** "degree q > 0" **and** "a ∈ carrier L"
shows "((eval_pmod q) ∘ (ring.poly_of_const L)) a = a"
⟨*proof*⟩

lemma image_poly_iso_incl:
assumes "q ∈ carrier (poly_ring L)" **and** "degree q > 0" **shows** "id ∈ ring_hom L (image_poly q)"
⟨*proof*⟩

lemma image_poly_is_field:
assumes "q ∈ carrier (poly_ring L)" **and** "pirreducible$_{\text{L}}$ (carrier L) q" **shows** "field (image_poly q)"
⟨*proof*⟩

lemma image_poly_index_free:
assumes "q ∈ carrier (poly_ring L)" **and** "P ∈ carrier (image_poly q)" **and** "¬ index_free P j" "i ≠ j"
obtains Q **where** "Q ∈ carrier L" **and** "¬ index_free Q j"
⟨*proof*⟩

lemma eval_pmod_var:
assumes "indexed_const ∈ ring_hom R L" **and** "q ∈ carrier (poly_ring L)" **and** "degree q > 1"
shows "(eval_pmod q) X$_{\text{L}}$ = $\mathcal{X}_{\text{i}}$" **and** "$\mathcal{X}_{\text{i}}$ ∈ carrier (image_poly q)"
⟨*proof*⟩

lemma image_poly_eval_indexed_var:
assumes "indexed_const ∈ ring_hom R L"
and "q ∈ carrier (poly_ring L)" **and** "degree q > 1" **and** "pirreducible$_{\text{L}}$ (carrier L) q"
shows "(ring.eval (image_poly q)) q $\mathcal{X}_{\text{i}}$ = $\mathbf{0}_{\text{image_poly q}}$"
⟨*proof*⟩

end

end

end

theory Finite_Extensions
imports Embedded_Algebras Polynomials Polynomial_Divisibility

begin

37 Finite Extensions

37.1 Definitions

definition (**in** ring) transcendental :: "'a set ⇒ 'a ⇒ bool"
where "transcendental K x ⟷ inj_on (λp. eval p x) (carrier (K[X]))"

abbreviation (**in** ring) algebraic :: "'a set ⇒ 'a ⇒ bool"
where "algebraic K x ≡ ¬ transcendental K x"

definition (**in** ring) Irr :: "'a set ⇒ 'a ⇒ 'a list"
where "Irr K x = (THE p. p ∈ carrier (K[X]) ∧ pirreducible K p ∧ eval p x = **0** ∧ lead_coeff p = **1**)"

inductive_set (**in** ring) simple_extension :: "'a set ⇒ 'a ⇒ 'a set"
for K **and** x **where**
zero [simp, intro]: "**0** ∈ simple_extension K x" |
lin: "⟦ k1 ∈ simple_extension K x; k2 ∈ K ⟧ ⟹ (k1 ⊗ x) ⊕ k2 ∈ simple_extension K x"

fun (**in** ring) finite_extension :: "'a set ⇒ 'a list ⇒ 'a set"
where "finite_extension K xs = foldr (λx K'. simple_extension K' x) xs K"

37.2 Basic Properties

lemma (**in** ring) transcendental_consistent:
assumes "subring K R" **shows** "transcendental = ring.transcendental (R ⦇ carrier := K ⦈)"
⟨*proof*⟩

lemma (**in** ring) algebraic_consistent:
assumes "subring K R" **shows** "algebraic = ring.algebraic (R ⦇ carrier := K ⦈)"
⟨*proof*⟩

lemma (in ring**)** eval_transcendental:
assumes "(transcendental over K) x" "p ∈ carrier (K[X])" "eval p x = 0" **shows** "p = []"
⟨*proof*⟩

lemma (in ring**)** transcendental_imp_trivial_ker:
shows "(transcendental over K) x ⟹ a_kernel (K[X]) R (λp. eval p x) = { [] }"
⟨*proof*⟩

lemma (in ring**)** non_trivial_ker_imp_algebraic:
shows "a_kernel (K[X]) R (λp. eval p x) ≠ { [] } ⟹ (algebraic over K) x"
⟨*proof*⟩

lemma (in domain**)** trivial_ker_imp_transcendental:
assumes "subring K R" **and** "x ∈ carrier R"
shows "a_kernel (K[X]) R (λp. eval p x) = { [] } ⟹ (transcendental over K) x"
⟨*proof*⟩

lemma (in domain**)** algebraic_imp_non_trivial_ker:
assumes "subring K R" **and** "x ∈ carrier R"
shows "(algebraic over K) x ⟹ a_kernel (K[X]) R (λp. eval p x) ≠ { [] }"
⟨*proof*⟩

lemma (in domain**)** algebraicE:
assumes "subring K R" **and** "x ∈ carrier R" "(algebraic over K) x"
obtains p **where** "p ∈ carrier (K[X])" "p ≠ []" "eval p x = 0"
⟨*proof*⟩

lemma (in ring**)** algebraicI:
assumes "p ∈ carrier (K[X])" "p ≠ []" **and** "eval p x = 0" **shows** "(algebraic over K) x"
⟨*proof*⟩

lemma (in ring**)** transcendental_mono:
assumes "K ⊆ K'" "(transcendental over K') x" **shows** "(transcendental over K) x"
⟨*proof*⟩

corollary (in ring**)** algebraic_mono:
assumes "K ⊆ K'" "(algebraic over K) x" **shows** "(algebraic over K') x"
⟨*proof*⟩

lemma (in domain**)** zero_is_algebraic:
assumes "subring K R" **shows** "(algebraic over K) 0"

⟨*proof*⟩

lemma (in domain**)** algebraic_self:
assumes "subring K R" **and** "k ∈ K" **shows** "(algebraic over K) k"
⟨*proof*⟩

lemma (in domain**)** ker_diff_carrier:
assumes "subring K R"
shows "a_kernel (K[X]) R (λp. eval p x) ≠ carrier (K[X])"
⟨*proof*⟩

37.3 Minimal Polynomial

lemma (in domain**)** minimal_polynomial_is_unique:
assumes "subfield K R" **and** "x ∈ carrier R" "(algebraic over K) x"
shows "∃!p ∈ carrier (K[X]). pirreducible K p ∧ eval p x = **0** ∧ lead_coeff p = **1**"
(is "∃!p. ?minimal_poly p")
⟨*proof*⟩

lemma (in domain**)** IrrE:
assumes "subfield K R" **and** "x ∈ carrier R" "(algebraic over K) x"
shows "Irr K x ∈ carrier (K[X])" **and** "pirreducible K (Irr K x)"
and "lead_coeff (Irr K x) = **1**" **and** "eval (Irr K x) x = **0**"
⟨*proof*⟩

lemma (in domain**)** Irr_generates_ker:
assumes "subfield K R" **and** "x ∈ carrier R" "(algebraic over K) x"
shows "a_kernel (K[X]) R (λp. eval p x) = $\text{PIdl}_{\text{K[X]}}$ (Irr K x)"
⟨*proof*⟩

lemma (in domain**)** Irr_minimal:
assumes "subfield K R" **and** "x ∈ carrier R" "(algebraic over K) x"
and "p ∈ carrier (K[X])" "eval p x = **0**" **shows** "(Irr K x) pdivides p"
⟨*proof*⟩

lemma (in domain**)** rupture_of_Irr:
assumes "subfield K R" **and** "x ∈ carrier R" "(algebraic over K) x" **shows** "field (Rupt K (Irr K x))"
⟨*proof*⟩

37.4 Simple Extensions

lemma (in ring**)** simple_extension_consistent:
assumes "subring K R" **shows** "ring.simple_extension (R (| carrier := K |)) = simple_extension"
⟨*proof*⟩

lemma (in ring**)** mono_simple_extension:

assumes "K ⊆ K'" **shows** "simple_extension K x ⊆ simple_extension K' x"
⟨*proof*⟩

lemma (in ring**)** simple_extension_incl:
assumes "K ⊆ carrier R" **and** "x ∈ carrier R" **shows** "K ⊆ simple_extension K x"
⟨*proof*⟩

lemma (in ring**)** simple_extension_mem:
assumes "subring K R" **and** "x ∈ carrier R" **shows** "x ∈ simple_extension K x"
⟨*proof*⟩

lemma (in ring**)** simple_extension_carrier:
assumes "x ∈ carrier R" **shows** "simple_extension (carrier R) x = carrier R"
⟨*proof*⟩

lemma (in ring**)** simple_extension_in_carrier:
assumes "K ⊆ carrier R" **and** "x ∈ carrier R" **shows** "simple_extension K x ⊆ carrier R"
⟨*proof*⟩

lemma (in ring**)** simple_extension_subring_incl:
assumes "subring K' R" **and** "K ⊆ K'" "x ∈ K'" **shows** "simple_extension K x ⊆ K'"
⟨*proof*⟩

lemma (in ring**)** simple_extension_as_eval_img:
assumes "K ⊆ carrier R" "x ∈ carrier R"
shows "simple_extension K x = (λp. eval p x) ‘ carrier (K[X])"
⟨*proof*⟩

corollary (in domain**)** simple_extension_is_subring:
assumes "subring K R" "x ∈ carrier R" **shows** "subring (simple_extension K x) R"
⟨*proof*⟩

corollary (in domain**)** simple_extension_minimal:
assumes "subring K R" "x ∈ carrier R"
shows "simple_extension K x = ⋂ { K'. subring K' R ∧ K ⊆ K' ∧ x ∈ K' }"
⟨*proof*⟩

corollary (in domain**)** simple_extension_isomorphism:
assumes "subring K R" "x ∈ carrier R"
shows "(K[X]) Quot (a_kernel (K[X]) R (λp. eval p x)) ≃ R ⦇ carrier := simple_extension K x ⦈"

⟨*proof*⟩

corollary (**in** domain) simple_extension_of_algebraic:
assumes "subfield K R" **and** "x ∈ carrier R" "(algebraic over K) x"
shows "Rupt K (Irr K x) ≃ R ⦇ carrier := simple_extension K x ⦈"
⟨*proof*⟩

corollary (**in** domain) simple_extension_of_transcendental:
assumes "subring K R" **and** "x ∈ carrier R" "(transcendental over K) x"
shows "K[X] ≃ R ⦇ carrier := simple_extension K x ⦈"
⟨*proof*⟩

proposition (**in** domain) simple_extension_subfield_imp_algebraic:
assumes "subring K R" "x ∈ carrier R"
shows "subfield (simple_extension K x) R ⟹ (algebraic over K) x"
⟨*proof*⟩

proposition (**in** domain) simple_extension_is_subfield:
assumes "subfield K R" "x ∈ carrier R"
shows "subfield (simple_extension K x) R ⟷ (algebraic over K) x"
⟨*proof*⟩

37.5 Link between dimension of K-algebras and algebraic extensions

lemma (**in** domain) exp_base_independent:
assumes "subfield K R" "x ∈ carrier R" "(algebraic over K) x"
shows "independent K (exp_base x (degree (Irr K x)))"
⟨*proof*⟩

lemma (**in** ring) Span_eq_eval_img:
assumes "subfield K R" "x ∈ carrier R"
shows "Span K (exp_base x n) = (λp. eval p x) ‘ { p ∈ carrier (K[X]). length p ≤ n }"
(**is** "?Span = ?eval_img")
⟨*proof*⟩

lemma (**in** domain) Span_exp_base:
assumes "subfield K R" "x ∈ carrier R" "(algebraic over K) x"
shows "Span K (exp_base x (degree (Irr K x))) = simple_extension K x"
⟨*proof*⟩

corollary (**in** domain) dimension_simple_extension:
assumes "subfield K R" "x ∈ carrier R" "(algebraic over K) x"
shows "dimension (degree (Irr K x)) K (simple_extension K x)"
⟨*proof*⟩

lemma (**in** ring) finite_dimension_imp_algebraic:

assumes "subfield K R" "subring F R" **and** "finite_dimension K F"
shows "x ∈ F ⟹ (algebraic over K) x"
⟨*proof*⟩

corollary (**in** domain) simple_extension_dim:
assumes "subfield K R" "x ∈ carrier R" "(algebraic over K) x"
shows "(dim over K) (simple_extension K x) = degree (Irr K x)"
⟨*proof*⟩

corollary (**in** domain) finite_dimension_simple_extension:
assumes "subfield K R" "x ∈ carrier R"
shows "finite_dimension K (simple_extension K x) ⟷ (algebraic over K) x"
⟨*proof*⟩

37.6 Finite Extensions

lemma (**in** ring) finite_extension_consistent:
assumes "subring K R" **shows** "ring.finite_extension (R ⦇ carrier := K ⦈) = finite_extension"
⟨*proof*⟩

lemma (**in** ring) mono_finite_extension:
assumes "K ⊆ K'" **shows** "finite_extension K xs ⊆ finite_extension K' xs"
⟨*proof*⟩

lemma (**in** ring) finite_extension_carrier:
assumes "set xs ⊆ carrier R" **shows** "finite_extension (carrier R) xs = carrier R"
⟨*proof*⟩

lemma (**in** ring) finite_extension_in_carrier:
assumes "K ⊆ carrier R" **and** "set xs ⊆ carrier R" **shows** "finite_extension K xs ⊆ carrier R"
⟨*proof*⟩

lemma (**in** ring) finite_extension_subring_incl:
assumes "subring K' R" **and** "K ⊆ K'" "set xs ⊆ K'" **shows** "finite_extension K xs ⊆ K'"
⟨*proof*⟩

lemma (**in** ring) finite_extension_incl_aux:
assumes "K ⊆ carrier R" **and** "x ∈ carrier R" "set xs ⊆ carrier R"
shows "finite_extension K xs ⊆ finite_extension K (x # xs)"
⟨*proof*⟩

lemma (**in** ring) finite_extension_incl:
assumes "K ⊆ carrier R" **and** "set xs ⊆ carrier R" **shows** "K ⊆ finite_extension

K xs"
⟨*proof*⟩

lemma (**in** ring) finite_extension_as_eval_img:
assumes "K ⊆ carrier R" **and** "x ∈ carrier R" "set xs ⊆ carrier R"
shows "finite_extension K (x # xs) = (λp. eval p x) ‘ carrier ((finite_extension K xs) [X])"
⟨*proof*⟩

lemma (**in** domain) finite_extension_is_subring:
assumes "subring K R" "set xs ⊆ carrier R" **shows** "subring (finite_extension K xs) R"
⟨*proof*⟩

corollary (**in** domain) finite_extension_mem:
assumes subring: "subring K R"
shows "set xs ⊆ carrier R ⟹ set xs ⊆ finite_extension K xs"
⟨*proof*⟩

corollary (**in** domain) finite_extension_minimal:
assumes "subring K R" "set xs ⊆ carrier R"
shows "finite_extension K xs = ⋂ { K’. subring K’ R ∧ K ⊆ K’ ∧ set xs ⊆ K’ }"
⟨*proof*⟩

corollary (**in** domain) finite_extension_same_set:
assumes "subring K R" "set xs ⊆ carrier R" "set xs = set ys"
shows "finite_extension K xs = finite_extension K ys"
⟨*proof*⟩

The reciprocal is also true, but it is more subtle.

proposition (**in** domain) finite_extension_is_subfield:
assumes "subfield K R" "set xs ⊆ carrier R"
shows "(⋀x. x ∈ set xs ⟹ (algebraic over K) x) ⟹ subfield (finite_extension K xs) R"
⟨*proof*⟩

proposition (**in** domain) finite_extension_finite_dimension:
assumes "subfield K R" "set xs ⊆ carrier R"
shows "(⋀x. x ∈ set xs ⟹ (algebraic over K) x) ⟹ finite_dimension K (finite_extension K xs)"
and "finite_dimension K (finite_extension K xs) ⟹ (⋀x. x ∈ set xs ⟹ (algebraic over K) x)"
⟨*proof*⟩

corollary (**in** domain) finite_extesion_mem_imp_algebraic:
assumes "subfield K R" "set xs ⊆ carrier R" **and** "⋀x. x ∈ set xs ⟹ (algebraic over K) x"
shows "y ∈ finite_extension K xs ⟹ (algebraic over K) y"

⟨*proof*⟩

corollary (in domain**)** simple_extesion_mem_imp_algebraic:
assumes "subfield K R" "x ∈ carrier R" "(algebraic over K) x"
shows "y ∈ simple_extension K x ⟹ (algebraic over K) y"
⟨*proof*⟩

37.7 Arithmetic of algebraic numbers

We show that the set of algebraic numbers of a field over a subfield K is a subfield itself.

lemma (in field**)** subfield_of_algebraics:
assumes "subfield K R" **shows** "subfield { x ∈ carrier R. (algebraic over K) x } R"
⟨*proof*⟩

end

theory Algebraic_Closure
imports Indexed_Polynomials Polynomial_Divisibility Finite_Extensions

begin

38 Algebraic Closure

38.1 Definitions

inductive iso_incl :: "'a ring ⇒ 'a ring ⇒ bool" **(infixl** ‹≲› 65**) for** A B
where iso_inclI [intro]: "id ∈ ring_hom A B ⟹ iso_incl A B"

definition law_restrict :: "('a, 'b) ring_scheme ⇒ 'a ring"
where "law_restrict R ≡ (ring.truncate R)
⦇ mult := (λa ∈ carrier R. λb ∈ carrier R. a ⊗$_R$ b),
add := (λa ∈ carrier R. λb ∈ carrier R. a ⊕$_R$ b) ⦈"

definition (in ring**)** σ :: "'a list ⇒ ((('a list × nat) multiset) ⇒ 'a) list"
where "σ P = map indexed_const P"

definition (in ring**)** extensions :: "((('a list × nat) multiset) ⇒ 'a) ring set"
where "extensions ≡ { L — such that.
— i (field L) ∧
— ii (indexed_const ∈ ring_hom R L) ∧
— iii (∀𝒫 ∈ carrier L. carrier_coeff 𝒫) ∧
— iv (∀𝒫 ∈ carrier L. ∀P ∈ carrier (poly_ring R). ∀i.

¬ index_free $\mathcal{P}$ (P, i) ⟶
$\mathcal{X}_{(P,\ i)}$ ∈ carrier L ∧ (ring.eval L) (σ P) $\mathcal{X}_{(P,\ i)}$
= $\mathbf{0}_L$) }"

abbreviation (**in** ring) restrict_extensions :: "(('a list × nat) multiset)
⇒ 'a) ring set" (‹$\mathcal{S}$›)
where "$\mathcal{S}$ ≡ law_restrict ‘ extensions"

38.2 Basic Properties

lemma law_restrict_carrier: "carrier (law_restrict R) = carrier R"
⟨*proof*⟩

lemma law_restrict_one: "one (law_restrict R) = one R"
⟨*proof*⟩

lemma law_restrict_zero: "zero (law_restrict R) = zero R"
⟨*proof*⟩

lemma law_restrict_mult: "monoid.mult (law_restrict R) = (λa ∈ carrier
R. λb ∈ carrier R. a ⊗$_R$ b)"
⟨*proof*⟩

lemma law_restrict_add: "add (law_restrict R) = (λa ∈ carrier R. λb
∈ carrier R. a ⊕$_R$ b)"
⟨*proof*⟩

lemma (**in** ring) law_restrict_is_ring: "ring (law_restrict R)"
⟨*proof*⟩

lemma (**in** field) law_restrict_is_field: "field (law_restrict R)"
⟨*proof*⟩

lemma law_restrict_iso_imp_eq:
assumes "id ∈ ring_iso (law_restrict A) (law_restrict B)" **and** "ring
A" **and** "ring B"
shows "law_restrict A = law_restrict B"
⟨*proof*⟩

lemma law_restrict_hom: "h ∈ ring_hom A B ⟷ h ∈ ring_hom (law_restrict
A) (law_restrict B)"
⟨*proof*⟩

lemma iso_incl_hom: "A ≲ B ⟷ (law_restrict A) ≲ (law_restrict B)"
⟨*proof*⟩

38.3 Partial Order

lemma iso_incl_backwards:
assumes "A ≲ B" **shows** "id ∈ ring_hom A B"

⟨*proof*⟩

lemma iso_incl_antisym_aux:
assumes "A $\lesssim$ B" **and** "B $\lesssim$ A" **shows** "id ∈ ring_iso A B"
⟨*proof*⟩

lemma iso_incl_refl: "A $\lesssim$ A"
⟨*proof*⟩

lemma iso_incl_trans:
assumes "A $\lesssim$ B" **and** "B $\lesssim$ C" **shows** "A $\lesssim$ C"
⟨*proof*⟩

lemma (**in** ring) iso_incl_antisym:
assumes "A ∈ $\mathcal{S}$" "B ∈ $\mathcal{S}$" **and** "A $\lesssim$ B" "B $\lesssim$ A" **shows** "A = B"
⟨*proof*⟩

lemma (**in** ring) iso_incl_partial_order: "partial_order_on $\mathcal{S}$ (relation_of ($\lesssim$) $\mathcal{S}$)"
⟨*proof*⟩

lemma iso_inclE:
assumes "ring A" **and** "ring B" **and** "A $\lesssim$ B" **shows** "ring_hom_ring A B id"
⟨*proof*⟩

lemma iso_incl_imp_same_eval:
assumes "ring A" **and** "ring B" **and** "A $\lesssim$ B" **and** "a ∈ carrier A" **and** "set p ⊆ carrier A"
shows "(ring.eval A) p a = (ring.eval B) p a"
⟨*proof*⟩

38.4 Extensions Non Empty

lemma (**in** ring) indexed_const_is_inj: "inj indexed_const"
⟨*proof*⟩

lemma (**in** ring) indexed_const_inj_on: "inj_on indexed_const (carrier R)"
⟨*proof*⟩

lemma (**in** field) extensions_non_empty: "$\mathcal{S}$ ≠ {}"
⟨*proof*⟩

38.5 Chains

definition union_ring :: "(('a, 'c) ring_scheme) set ⇒ 'a ring"
where "union_ring C =
(| carrier = ($\bigcup$(carrier ‘ C)),

```
        monoid.mult = (λa b. (monoid.mult (SOME R. R ∈ C ∧ a ∈ carrier
R ∧ b ∈ carrier R) a b)),
                one = one (SOME R. R ∈ C),
               zero = zero (SOME R. R ∈ C),
                add = (λa b. (add (SOME R. R ∈ C ∧ a ∈ carrier R ∧ b
∈ carrier R) a b)) ⦇"
```

lemma union_ring_carrier: "carrier (union_ring C) = (⋃(carrier ‘ C))"
⟨*proof*⟩

context
fixes C :: "'a ring set"
assumes field_chain: "⋀R. R ∈ C ⟹ field R" **and** chain: "⋀R S. ⟦ R ∈ C; S ∈ C ⟧ ⟹ R ≲ S ∨ S ≲ R"
begin

lemma ring_chain: "R ∈ C ⟹ ring R"
⟨*proof*⟩

lemma same_one_same_zero:
assumes "R ∈ C" **shows** "$1_{\text{union_ring C}}$ = 1_{R}" **and** "$0_{\text{union_ring C}}$ = 0_{R}"
⟨*proof*⟩

lemma same_laws:
assumes "R ∈ C" **and** "a ∈ carrier R" **and** "b ∈ carrier R"
shows "a $\otimes_{\text{union_ring C}}$ b = a $\otimes_{R}$ b" **and** "a $\oplus_{\text{union_ring C}}$ b = a $\oplus_{R}$ b"
⟨*proof*⟩

lemma exists_superset_carrier:
assumes "finite S" **and** "S ≠ {}" **and** "S ⊆ carrier (union_ring C)"
shows "∃R ∈ C. S ⊆ carrier R"
⟨*proof*⟩

lemma union_ring_is_monoid:
assumes "C ≠ {}" **shows** "comm_monoid (union_ring C)"
⟨*proof*⟩

lemma union_ring_is_abelian_group:
assumes "C ≠ {}" **shows** "cring (union_ring C)"
⟨*proof*⟩

lemma union_ring_is_field :
assumes "C ≠ {}" **shows** "field (union_ring C)"
⟨*proof*⟩

lemma union_ring_is_upper_bound:
assumes "R ∈ C" **shows** "R ≲ union_ring C"
⟨*proof*⟩

end

38.6 Zorn

lemma (in ring**)** exists_core_chain:
assumes "C ∈ Chains (relation_of (≲) 𝒮)" **obtains** C' **where** "C' ⊆ extensions" **and** "C = law_restrict ‘ C'"
⟨*proof*⟩

lemma (in ring**)** core_chain_is_chain:
assumes "law_restrict ‘ C ∈ Chains (relation_of (≲) 𝒮)" **shows** "⋀R S. ⟦ R ∈ C; S ∈ C ⟧ ⟹ R ≲ S ∨ S ≲ R"
⟨*proof*⟩

lemma (in field**)** exists_maximal_extension:
shows "∃M ∈ 𝒮. ∀L ∈ 𝒮. M ≲ L ⟶ L = M"
⟨*proof*⟩

38.7 Existence of roots

lemma polynomial_hom:
assumes "h ∈ ring_hom R S" **and** "field R" **and** "field S"
shows "p ∈ carrier (poly_ring R) ⟹ (map h p) ∈ carrier (poly_ring S)"
⟨*proof*⟩

lemma (in ring_hom_ring**)** subfield_polynomial_hom:
assumes "subfield K R" **and** "$\mathbf{1}_S \neq \mathbf{0}_S$"
shows "p ∈ carrier ($K[X]_R$) ⟹ (map h p) ∈ carrier ($(h ‘ K)[X]_S$)"
⟨*proof*⟩

lemma (in field**)** exists_root:
assumes "M ∈ extensions" **and** "⋀L. ⟦ L ∈ extensions; M ≲ L ⟧ ⟹ law_restrict L = law_restrict M"
and "P ∈ carrier (poly_ring R)"
shows "(ring.splitted M) (σ P)"
⟨*proof*⟩

lemma (in field**)** exists_extension_with_roots:
shows "∃L ∈ extensions. ∀P ∈ carrier (poly_ring R). (ring.splitted L) (σ P)"
⟨*proof*⟩

38.8 Existence of Algebraic Closure

locale algebraic_closure = field L + subfield K L **for** L **(structure) and**
K +

```
  assumes algebraic_extension: "x ∈ carrier L ⟹ (algebraic over K)
x"
    and roots_over_subfield: "P ∈ carrier (K[X]) ⟹ splitted P"

locale algebraically_closed = field L for L (structure) +
  assumes roots_over_carrier: "P ∈ carrier (poly_ring L) ⟹ splitted
P"

definition (in field) alg_closure :: "(('a list × nat) multiset ⇒ 'a)
ring"
  where "alg_closure = (SOME L — such that.
          — i  algebraic_closure L (indexed_const ‘ (carrier R)) ∧
          — ii indexed_const ∈ ring_hom R L)"

lemma algebraic_hom:
  assumes "h ∈ ring_hom R S" and "field R" and "field S" and "subfield
K R" and "x ∈ carrier R"
  shows "((ring.algebraic R) over K) x ⟹ ((ring.algebraic S) over (h
‘ K)) (h x)"
⟨proof⟩

lemma (in field) exists_closure:
  obtains L :: "((('a list × nat) multiset) ⇒ 'a) ring"
  where "algebraic_closure L (indexed_const ‘ (carrier R))" and "indexed_const
∈ ring_hom R L"
⟨proof⟩

lemma (in field) alg_closureE:
  shows "algebraic_closure alg_closure (indexed_const ‘ (carrier R))"
    and "indexed_const ∈ ring_hom R alg_closure"
  ⟨proof⟩

lemma (in field) algebraically_closedI':
  assumes "⋀p. ⟦ p ∈ carrier (poly_ring R); degree p > 1 ⟧ ⟹ splitted
p"
  shows "algebraically_closed R"
⟨proof⟩

lemma (in field) algebraically_closedI:
  assumes "⋀p. ⟦ p ∈ carrier (poly_ring R); degree p > 1 ⟧ ⟹ ∃x ∈
carrier R. eval p x = 0"
  shows "algebraically_closed R"
⟨proof⟩

sublocale algebraic_closure ⊆ algebraically_closed
⟨proof⟩

end
```

```
theory Algebraic_Closure_Type
imports
  Algebraic_Closure
  "HOL-Computational_Algebra.Computational_Algebra"
  "HOL-Computational_Algebra.Field_as_Ring"
begin


definition (in ring_1) ring_of_type_algebra :: "'a ring"
  where "ring_of_type_algebra = ⦇
    carrier = UNIV, monoid.mult = (λx y. x * y),
    one = 1,
    ring.zero = 0,
    add = (λ x y. x + y) ⦈"

lemma (in comm_ring_1) ring_from_type_algebra [intro]:
  "ring (ring_of_type_algebra :: 'a ring)"
⟨proof⟩

lemma (in comm_ring_1) cring_from_type_algebra [intro]:
  "cring (ring_of_type_algebra :: 'a ring)"
⟨proof⟩

lemma (in Fields.field) field_from_type_algebra [intro]:
  "field (ring_of_type_algebra :: 'a ring)"
⟨proof⟩
```

38.9 Definition

```
typedef (overloaded) 'a :: field alg_closure =
  "carrier (field.alg_closure (ring_of_type_algebra :: 'a :: field ring))"
⟨proof⟩

setup_lifting type_definition_alg_closure

instantiation alg_closure :: (field) field
begin

context
  fixes L K
  defines "K ≡ (ring_of_type_algebra :: 'a :: field ring)"
  defines "L ≡ field.alg_closure K"
begin

interpretation K: field K
  ⟨proof⟩

interpretation algebraic_closure L "range K.indexed_const"
```

⟨*proof*⟩

lift_definition zero_alg_closure :: "'a alg_closure" **is** "ring.zero L"
⟨*proof*⟩

lift_definition one_alg_closure :: "'a alg_closure" **is** "monoid.one L"
⟨*proof*⟩

lift_definition plus_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"
is "ring.add L"
⟨*proof*⟩

lift_definition minus_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"
is "a_minus L"
⟨*proof*⟩

lift_definition times_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"
is "monoid.mult L"
⟨*proof*⟩

lift_definition uminus_alg_closure :: "'a alg_closure ⇒ 'a alg_closure"
is "a_inv L"
⟨*proof*⟩

lift_definition inverse_alg_closure :: "'a alg_closure ⇒ 'a alg_closure"
is "λx. if x = ring.zero L then ring.zero L else m_inv L x"
⟨*proof*⟩

lift_definition divide_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"
is "λx y. if y = ring.zero L then ring.zero L else monoid.mult L x (m_inv L y)"
⟨*proof*⟩

end

instance ⟨*proof*⟩

end

38.10 The algebraic closure is algebraically closed

instance alg_closure :: (field) alg_closed_field
⟨*proof*⟩

38.11 Converting between the base field and the closure

context
  fixes L K
  defines "K ≡ (ring_of_type_algebra :: 'a :: field ring)"
  defines "L ≡ field.alg_closure K"
begin

interpretation K: field K
  ⟨*proof*⟩

interpretation algebraic_closure L "range K.indexed_const"
⟨*proof*⟩

lemma alg_closure_hom: "K.indexed_const ∈ Ring.ring_hom K L"
  ⟨*proof*⟩

lift_definition to_ac :: "'a :: field ⇒ 'a alg_closure"
  is "ring.indexed_const K"
  by (fold K_def, fold L_def) (use mem_carrier **in** blast)

lemma to_ac_0 [simp]: "to_ac (0 :: 'a) = 0"
⟨*proof*⟩

lemma to_ac_1 [simp]: "to_ac (1 :: 'a) = 1"
⟨*proof*⟩

lemma to_ac_add [simp]: "to_ac (x + y :: 'a) = to_ac x + to_ac y"
⟨*proof*⟩

lemma to_ac_minus [simp]: "to_ac (-x :: 'a) = -to_ac x"
  ⟨*proof*⟩

lemma to_ac_diff [simp]: "to_ac (x - y :: 'a) = to_ac x - to_ac y"
  ⟨*proof*⟩

lemma to_ac_mult [simp]: "to_ac (x * y :: 'a) = to_ac x * to_ac y"
⟨*proof*⟩

lemma to_ac_inverse [simp]: "to_ac (inverse x :: 'a) = inverse (to_ac
x)"
  ⟨*proof*⟩

lemma to_ac_divide [simp]: "to_ac (x / y :: 'a) = to_ac x / to_ac y"
  ⟨*proof*⟩

lemma to_ac_power [simp]: "to_ac (x ^ n) = to_ac x ^ n"
  ⟨*proof*⟩

lemma to_ac_of_nat [simp]: "to_ac (of_nat n) = of_nat n"

⟨*proof*⟩

lemma to_ac_of_int [simp]: "to_ac (of_int n) = of_int n"
⟨*proof*⟩

lemma to_ac_numeral [simp]: "to_ac (numeral n) = numeral n"
⟨*proof*⟩

lemma to_ac_sum: "to_ac (∑x∈A. f x) = (∑x∈A. to_ac (f x))"
⟨*proof*⟩

lemma to_ac_prod: "to_ac (∏x∈A. f x) = (∏x∈A. to_ac (f x))"
⟨*proof*⟩

lemma to_ac_sum_list: "to_ac (sum_list xs) = (∑x←xs. to_ac x)"
⟨*proof*⟩

lemma to_ac_prod_list: "to_ac (prod_list xs) = (∏x←xs. to_ac x)"
⟨*proof*⟩

lemma to_ac_sum_mset: "to_ac (sum_mset xs) = (∑x∈#xs. to_ac x)"
⟨*proof*⟩

lemma to_ac_prod_mset: "to_ac (prod_mset xs) = (∏x∈#xs. to_ac x)"
⟨*proof*⟩

end

lemma (**in** ring) indexed_const_eq_iff [simp]:
"indexed_const x = (indexed_const y :: 'c multiset ⇒ 'a) ⟷ x = y"
⟨*proof*⟩

lemma inj_to_ac: "inj to_ac"
⟨*proof*⟩

lemma to_ac_eq_iff [simp]: "to_ac x = to_ac y ⟷ x = y"
⟨*proof*⟩

lemma to_ac_eq_0_iff [simp]: "to_ac x = 0 ⟷ x = 0"
and to_ac_eq_0_iff' [simp]: "0 = to_ac x ⟷ x = 0"
and to_ac_eq_1_iff [simp]: "to_ac x = 1 ⟷ x = 1"
and to_ac_eq_1_iff' [simp]: "1 = to_ac x ⟷ x = 1"
⟨*proof*⟩

definition of_ac :: "'a :: field alg_closure ⇒ 'a" **where**
"of_ac x = (if x ∈ range to_ac then inv_into UNIV to_ac x else 0)"

lemma of_ac_eqI: "to_ac x = y ⟹ of_ac y = x"

⟨*proof*⟩

lemma of_ac_0 [simp]: "of_ac 0 = 0"
  and of_ac_1 [simp]: "of_ac 1 = 1"
  ⟨*proof*⟩

lemma of_ac_to_ac [simp]: "of_ac (to_ac x) = x"
  ⟨*proof*⟩

lemma to_ac_of_ac: "x ∈ range to_ac ⟹ to_ac (of_ac x) = x"
  ⟨*proof*⟩

lemma CHAR_alg_closure [simp]:
  "CHAR('a :: field alg_closure) = CHAR('a)"
⟨*proof*⟩

instance alg_closure :: (field_char_0) field_char_0
⟨*proof*⟩

bundle alg_closure_syntax
begin
notation to_ac (‹(‹open_block notation=‹postfix ↑››_↑)› [1000] 999)
notation of_ac (‹(‹open_block notation=‹postfix ↓››_↓)› [1000] 999)
end

bundle alg_closure_syntax'
begin
notation (**output**) to_ac (‹_›)
notation (**output**) of_ac (‹_›)
end

38.12 The algebraic closure is an algebraic extension

The algebraic closure is an algebraic extension, i.e. every element in it is a root of some non-zero polynomial in the base field.

theorem alg_closure_algebraic:
  fixes x :: "'a :: field alg_closure"
  obtains p :: "'a poly" **where** "p ≠ 0" "poly (map_poly to_ac p) x = 0"
⟨*proof*⟩

instantiation alg_closure :: (field)
  "{unique_euclidean_ring, normalization_euclidean_semiring, normalization_semidom_multipli
begin

definition [simp]: "normalize_alg_closure = (normalize_field :: 'a alg_closure

```
⇒ _)"
definition [simp]: "unit_factor_alg_closure = (unit_factor_field :: 'a
alg_closure ⇒ _)"
definition [simp]: "modulo_alg_closure = (mod_field :: 'a alg_closure ⇒
_)"
definition [simp]: "euclidean_size_alg_closure = (euclidean_size_field
:: 'a alg_closure ⇒ _)"
definition [simp]: "division_segment (x :: 'a alg_closure) = 1"

instance
  ⟨proof⟩

end

instantiation alg_closure :: (field) euclidean_ring_gcd
begin

definition gcd_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"
where
  "gcd_alg_closure = Euclidean_Algorithm.gcd"
definition lcm_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"
where
  "lcm_alg_closure = Euclidean_Algorithm.lcm"
definition Gcd_alg_closure :: "'a alg_closure set ⇒ 'a alg_closure" where
 "Gcd_alg_closure = Euclidean_Algorithm.Gcd"
definition Lcm_alg_closure :: "'a alg_closure set ⇒ 'a alg_closure" where
 "Lcm_alg_closure = Euclidean_Algorithm.Lcm"

instance ⟨proof⟩

end

instance alg_closure :: (field) semiring_gcd_mult_normalize
  ⟨proof⟩

end
```

39 The Residue Ring as a HOL-Algebra Structure

```
theory Residues_Algebra
  imports "HOL-Number_Theory.Cong" "HOL-Number_Theory.Totient"
    "HOL-Algebra.Multiplicative_Group"
begin
```

The `residue_ring` construction and the `residues` and `residues_prime` locales, split out of `Residues` so that the residue-number-theory proper no longer depends on HOL-Algebra. Nothing here is new: it is the record-based development as it stood, kept because a substantial body of work outside this theory — `Padic_Ints`, `Padic_Field`, `Crypto_Standards`, `Schoenhage_Strassen`,

`Elliptic_Curves_Group_Law` and others — reasons inside these locales and about `carrier R`, and so cannot be served by a carrier-set replacement.

The theorems of elementary residue number theory — Euler, Fermat, Wilson, the existence of a primitive root, the bound on the number of roots — are proved natively in `Residues` and are not repeated here. What remains below are the locale-internal forms, which those developments extend.

The statement below is about a type-class ring, but its proof goes through the record-based ring, so it belongs with the HOL-Algebra material rather than with `Residues`. `Perfect_Fields` uses it.

lemma (in `ring_1`**)** `CHAR_dvd_CARD: "CHAR('a) dvd card (UNIV :: 'a set)"`
⟨*proof*⟩

39.1 A locale for residue rings

```
definition residue_ring :: "int ⇒ int ring"
  where
    "residue_ring m =
      (|carrier = {0..m - 1},
       monoid.mult = λx y. (x * y) mod m,
       one = 1,
       zero = 0,
       add = λx y. (x + y) mod m|)"

locale residues =
  fixes m :: int and R (structure)
  assumes m_gt_one: "m > 1"
  defines R_m_def: "R ≡ residue_ring m"
begin

lemma abelian_group: "abelian_group R"
⟨proof⟩

lemma comm_monoid: "comm_monoid R"
⟨proof⟩

interpretation comm_monoid R
  ⟨proof⟩

lemma cring: "cring R"
⟨proof⟩

end

sublocale residues < cring
  ⟨proof⟩
```

context residues
begin

These lemmas translate back and forth between internal and external concepts.

lemma res_carrier_eq: "carrier R = {0..m - 1}"
⟨*proof*⟩

lemma res_add_eq: "x ⊕ y = (x + y) mod m"
⟨*proof*⟩

lemma res_mult_eq: "x ⊗ y = (x * y) mod m"
⟨*proof*⟩

lemma res_zero_eq: "**0** = 0"
⟨*proof*⟩

lemma res_one_eq: "**1** = 1"
⟨*proof*⟩

lemma res_units_eq: "Units R = {x. 0 < x ∧ x < m ∧ coprime x m}" (**is** "_ = ?rhs")
⟨*proof*⟩

lemma res_neg_eq: "⊖ x = (- x) mod m"
⟨*proof*⟩

lemma finite [iff]: "finite (carrier R)"
⟨*proof*⟩

lemma finite_Units [iff]: "finite (Units R)"
⟨*proof*⟩

The function a ↦ a mod m maps the integers to the residue classes. The following lemmas show that this mapping respects addition and multiplication on the integers.

lemma mod_in_carrier [iff]: "a mod m ∈ carrier R"
⟨*proof*⟩

lemma add_cong: "(x mod m) ⊕ (y mod m) = (x + y) mod m"
⟨*proof*⟩

lemma mult_cong: "(x mod m) ⊗ (y mod m) = (x * y) mod m"
⟨*proof*⟩

lemma zero_cong: "**0** = 0"
⟨*proof*⟩

lemma one_cong: "**1** = 1 mod m"

⟨*proof*⟩

lemma pow_cong: "(x mod m) [^] n = x^n mod m"
⟨*proof*⟩

lemma neg_cong: "⊖ (x mod m) = (- x) mod m"
⟨*proof*⟩

lemma (**in** residues) prod_cong: "finite A ⟹ (⨂i∈A. (f i) mod m) = (∏i∈A. f i) mod m"
⟨*proof*⟩

lemma (**in** residues) sum_cong: "finite A ⟹ (⨁i∈A. (f i) mod m) = (∑i∈A. f i) mod m"
⟨*proof*⟩

lemma mod_in_res_units [simp]:
assumes "1 < m" **and** "coprime a m"
shows "a mod m ∈ Units R"
⟨*proof*⟩

lemma res_eq_to_cong: "(a mod m) = (b mod m) ⟷ [a = b] (mod m)"
⟨*proof*⟩

Simplifying with these will translate a ring equation in R to a congruence.

lemmas res_to_cong_simps =
add_cong mult_cong pow_cong one_cong
prod_cong sum_cong neg_cong res_eq_to_cong

Other useful facts about the residue ring.

lemma one_eq_neg_one: "**1** = ⊖ **1** ⟹ m = 2"
⟨*proof*⟩

end

39.2 Prime residues

locale residues_prime =
fixes p :: nat **and** R (**structure**)
assumes p_prime [intro]: "prime p"
defines "R ≡ residue_ring (int p)"

sublocale residues_prime < residues p
⟨*proof*⟩

context residues_prime
begin

lemma p_coprime_left:

```
  "coprime p a ⟷ ¬ p dvd a"
```
⟨*proof*⟩

lemma p_coprime_right:
```
  "coprime a p  ⟷ ¬ p dvd a"
```
⟨*proof*⟩

lemma p_coprime_left_int:
```
  "coprime (int p) a ⟷ ¬ int p dvd a"
```
⟨*proof*⟩

lemma p_coprime_right_int:
```
  "coprime a (int p) ⟷ ¬ int p dvd a"
```
⟨*proof*⟩

lemma is_field: "field R"
⟨*proof*⟩

lemma res_prime_units_eq: "Units R = {1..p - 1}"
⟨*proof*⟩

end

sublocale residues_prime < field
⟨*proof*⟩

39.3 Euler's theorem

lemma (**in** residues) totatives_eq:
```
  "totatives (nat m) = nat ‘ Units R"
```
⟨*proof*⟩

lemma (**in** residues) totient_eq:
```
  "totient (nat m) = card (Units R)"
```
⟨*proof*⟩

lemma (**in** residues_prime) prime_totient_eq: "totient p = p - 1"
⟨*proof*⟩

lemma (**in** residues) euler_theorem:
assumes "coprime a m"
shows "[a ^ totient (nat m) = 1] (mod m)"
⟨*proof*⟩

39.4 Wilson's theorem

lemma (**in** field) inv_pair_lemma: "x ∈ Units R ⟹ y ∈ Units R ⟹
```
    {x, inv x} ≠ {y, inv y} ⟹ {x, inv x} ∩ {y, inv y} = {}"
```
⟨*proof*⟩

lemma (in residues_prime**)** wilson_theorem1:
  assumes a: "p > 2"
  shows "[fact (p - 1) = (-1::int)] (mod p)"
⟨*proof*⟩

lemma mod_nat_int_pow_eq:
  fixes n :: nat **and** p a :: int
  shows "a ≥ 0 ⟹ p ≥ 0 ⟹ (nat a ^ n) mod (nat p) = nat ((a ^ n) mod p)"
  ⟨*proof*⟩

end

theory Ideal_Product
  imports Ideal
begin

40 Product of Ideals

In this section, we study the structure of the set of ideals of a given ring.

inductive_set
  ideal_prod :: "[('a, 'b) ring_scheme, 'a set, 'a set] ⇒ 'a set" (**infixl** ‹·ı› 80)
  for R **and** I **and** J **where**
    prod: "⟦ i ∈ I; j ∈ J ⟧ ⟹ i ⊗$_R$ j ∈ ideal_prod R I J"
  | sum: "⟦ s1 ∈ ideal_prod R I J; s2 ∈ ideal_prod R I J ⟧ ⟹ s1 ⊕$_R$ s2 ∈ ideal_prod R I J"

definition ideals_set :: "('a, 'b) ring_scheme ⇒ ('a set) ring"
  where "ideals_set R = (| carrier = { I. ideal I R },
                              mult = ideal_prod R,
                               one = carrier R,
                              zero = { $\mathbf{0}_R$ },
                               add = set_add R |)"

40.1 Basic Properties

lemma (in ring**)** ideal_prod_in_carrier:
  assumes "ideal I R" "ideal J R"
  shows "I · J ⊆ carrier R"
⟨*proof*⟩

lemma (in ring**)** ideal_prod_inter:
  assumes "ideal I R" "ideal J R"
  shows "I · J ⊆ I ∩ J"
⟨*proof*⟩

lemma (in ring**)** ideal_prod_is_ideal:
  assumes "ideal I R" "ideal J R"
  shows "ideal (I · J) R"
⟨*proof*⟩

lemma (in ring**)** ideal_prod_eq_genideal:
  assumes "ideal I R" "ideal J R"
  shows "I · J = Idl (I <#> J)"
⟨*proof*⟩

lemma (in ring**)** ideal_prod_simp:
  assumes "ideal I R" "ideal J R"
  shows "I = I <+> (I · J)"
⟨*proof*⟩

lemma (in ring**)** ideal_prod_one:
  assumes "ideal I R"
  shows "I · (carrier R) = I"
⟨*proof*⟩

lemma (in ring**)** ideal_prod_zero:
  assumes "ideal I R"
  shows "I · { **0** } = { **0** }"
⟨*proof*⟩

lemma (in ring**)** ideal_prod_assoc:
  assumes "ideal I R" "ideal J R" "ideal K R"
  shows "(I · J) · K = I · (J · K)"
⟨*proof*⟩

lemma (in ring**)** ideal_prod_r_distr:
  assumes "ideal I R" "ideal J R" "ideal K R"
  shows "I · (J <+> K) = (I · J) <+> (I · K)"
⟨*proof*⟩

lemma (in cring**)** ideal_prod_commute:
  assumes "ideal I R" "ideal J R"
  shows "I · J = J · I"
⟨*proof*⟩

The following result would also be true for locale ring

lemma (in cring**)** ideal_prod_distr:
  assumes "ideal I R" "ideal J R" "ideal K R"
  shows "I · (J <+> K) = (I · J) <+> (I · K)"
    and "(J <+> K) · I = (J · I) <+> (K · I)"
  ⟨*proof*⟩

lemma (in cring**)** ideal_prod_eq_inter:
  assumes "ideal I R" "ideal J R"
    and "I <+> J = carrier R"
  shows "I · J = I ∩ J"
⟨*proof*⟩

40.2 Structure of the Set of Ideals

We focus on commutative rings for convenience.

lemma (in cring**)** ideals_set_is_semiring: "semiring (ideals_set R)"
⟨*proof*⟩

lemma (in cring**)** ideals_set_is_comm_monoid: "comm_monoid (ideals_set R)"
⟨*proof*⟩

lemma (in cring**)** ideal_prod_eq_Inter_aux:
  assumes "I: {..(Suc n)} → { J. ideal J R }"
    and "⋀i j. ⟦ i ≤ Suc n; j ≤ Suc n ⟧ ⟹
                i ≠ j ⟹ (I i) <+> (I j) = carrier R"
  shows "(⨂(ideals_set R) k ∈ {..n}. I k) <+> (I (Suc n)) = carrier R"
⟨*proof*⟩

theorem (in cring**)** ideal_prod_eq_Inter:
  assumes "I: {..n :: nat} → { J. ideal J R }"
    and "⋀i j. ⟦ i ∈ {..n}; j ∈ {..n} ⟧ ⟹ i ≠ j ⟹ (I i) <+> (I j) = carrier R"
  shows "(⨂(ideals_set R) k ∈ {..n}. I k) = (⋂ k ∈ {..n}. I k)" ⟨*proof*⟩

corollary (in cring**)** inter_plus_ideal_eq_carrier:
  assumes "⋀i. i ≤ Suc n ⟹ ideal (I i) R"
      and "⋀i j. ⟦ i ≤ Suc n; j ≤ Suc n; i ≠ j ⟧ ⟹ I i <+> I j = carrier R"
  shows "(⋂ i ≤ n. I i) <+> (I (Suc n)) = carrier R"
  ⟨*proof*⟩

corollary (in cring**)** inter_plus_ideal_eq_carrier_arbitrary:
  assumes "⋀i. i ≤ Suc n ⟹ ideal (I i) R"
      and "⋀i j. ⟦ i ≤ Suc n; j ≤ Suc n; i ≠ j ⟧ ⟹ I i <+> I j = carrier R"
      and "j ≤ Suc n"
  shows "(⋂ i ∈ ({..(Suc n)} - { j }). I i) <+> (I j) = carrier R"
⟨*proof*⟩

40.3 Another Characterization of Prime Ideals

With product of ideals being defined, we can give another definition of a prime ideal

lemma (in ring**)** primeideal_divides_ideal_prod:
  assumes "primeideal P R" "ideal I R" "ideal J R"
    and "I · J ⊆ P"
  shows "I ⊆ P ∨ J ⊆ P"
⟨*proof*⟩

lemma (in cring**)** divides_ideal_prod_imp_primeideal:
  assumes "ideal P R"
    and "P ≠ carrier R"
    and "⋀I J. ⟦ ideal I R; ideal J R; I · J ⊆ P ⟧ ⟹ I ⊆ P ∨ J ⊆ P"
  shows "primeideal P R"
⟨*proof*⟩

end

theory Chinese_Remainder
  imports Weak_Morphisms Ideal_Product

begin

41 Direct Product of Rings

41.1 Definitions

definition RDirProd :: "('a, 'n) ring_scheme ⇒ ('b, 'm) ring_scheme ⇒
('a × 'b) ring"
  where "RDirProd R S = monoid.extend (R ×× S)
          ⦇ zero = one ((add_monoid R) ×× (add_monoid S)),
            add = mult ((add_monoid R) ×× (add_monoid S)) ⦈ "

abbreviation nil_ring :: "('a list) ring"
  where "nil_ring ≡ monoid.extend nil_monoid ⦇ zero = [], add = (λa b.
[]) ⦈"

definition RDirProd_list :: "(('a, 'n) ring_scheme) list ⇒ ('a list) ring"
  where "RDirProd_list Rs = foldr (λR S. image_ring (λ(a, as). a # as)
(RDirProd R S)) Rs nil_ring"

41.2 Basic Properties

lemma RDirProd_carrier: "carrier (RDirProd R S) = carrier R × carrier
S"
  ⟨*proof*⟩

lemma RDirProd_add_monoid [simp]: "add_monoid (RDirProd R S) = (add_monoid
R) ×× (add_monoid S)"
  ⟨*proof*⟩

lemma RDirProd_ring:
assumes "ring R" **and** "ring S" **shows** "ring (RDirProd R S)"
⟨*proof*⟩

lemma RDirProd_iso1:
"(λ(x, y). (y, x)) ∈ ring_iso (RDirProd R S) (RDirProd S R)"
⟨*proof*⟩

lemma RDirProd_iso2:
"(λ(x, (y, z)). ((x, y), z)) ∈ ring_iso (RDirProd R (RDirProd S T))
(RDirProd (RDirProd R S) T)"
⟨*proof*⟩

lemma RDirProd_iso3:
"(λ((x, y), z). (x, (y, z))) ∈ ring_iso (RDirProd (RDirProd R S) T)
(RDirProd R (RDirProd S T))"
⟨*proof*⟩

lemma RDirProd_iso4:
assumes "f ∈ ring_iso R S" **shows** "(λ(r, t). (f r, t)) ∈ ring_iso (RDirProd
R T) (RDirProd S T)"
⟨*proof*⟩

lemma RDirProd_iso5:
assumes "f ∈ ring_iso S T" **shows** "(λ(r, s). (r, f s)) ∈ ring_iso (RDirProd
R S) (RDirProd R T)"
⟨*proof*⟩

lemma RDirProd_iso6:
assumes "f ∈ ring_iso R R'" **and** "g ∈ ring_iso S S'"
shows "(λ(r, s). (f r, g s)) ∈ ring_iso (RDirProd R S) (RDirProd R'
S')"
⟨*proof*⟩

lemma RDirProd_iso7:
shows "(λa. (a, [])) ∈ ring_iso R (RDirProd R nil_ring)"
⟨*proof*⟩

lemma RDirProd_hom1:
shows "(λa. (a, a)) ∈ ring_hom R (RDirProd R R)"
⟨*proof*⟩

lemma RDirProd_hom2:
assumes "f ∈ ring_hom S T"
shows "(λ(x, y). (x, f y)) ∈ ring_hom (RDirProd R S) (RDirProd R T)"
and "(λ(x, y). (f x, y)) ∈ ring_hom (RDirProd S R) (RDirProd T R)"
⟨*proof*⟩

lemma RDirProd_hom3:

assumes "f ∈ ring_hom R R'" **and** "g ∈ ring_hom S S'"
shows "(λ(r, s). (f r, g s)) ∈ ring_hom (RDirProd R S) (RDirProd R' S')"
⟨*proof*⟩

41.3 Direct Product of a List of Rings

lemma RDirProd_list_nil [simp]: "RDirProd_list [] = nil_ring"
⟨*proof*⟩

lemma nil_ring_simprules [simp]:
"carrier nil_ring = { [] }" **and** "one nil_ring = []" **and** "zero nil_ring = []"
⟨*proof*⟩

lemma RDirProd_list_truncate:
shows "monoid.truncate (RDirProd_list Rs) = DirProd_list Rs"
⟨*proof*⟩

lemma RDirProd_list_carrier_def':
shows "carrier (RDirProd_list Rs) = carrier (DirProd_list Rs)"
⟨*proof*⟩

lemma RDirProd_list_carrier:
shows "carrier (RDirProd_list (G # Gs)) = (λ(x, xs). x # xs) ` (carrier G × carrier (RDirProd_list Gs))"
⟨*proof*⟩

lemma RDirProd_list_one:
shows "one (RDirProd_list Rs) = foldr (λR tl. (one R) # tl) Rs []"
⟨*proof*⟩

lemma RDirProd_list_zero:
shows "zero (RDirProd_list Rs) = foldr (λR tl. (zero R) # tl) Rs []"
⟨*proof*⟩

lemma RDirProd_list_zero':
shows "zero (RDirProd_list (R # Rs)) = (zero R) # (zero (RDirProd_list Rs))"
⟨*proof*⟩

lemma RDirProd_list_carrier_mem:
assumes "as ∈ carrier (RDirProd_list Rs)"
shows "length as = length Rs" **and** "⋀i. i < length Rs ⟹ (as ! i) ∈ carrier (Rs ! i)"
⟨*proof*⟩

lemma RDirProd_list_carrier_memI:
assumes "length as = length Rs" **and** "⋀i. i < length Rs ⟹ (as ! i)

∈ carrier (Rs ! i)"
shows "as ∈ carrier (RDirProd_list Rs)"
⟨*proof*⟩

lemma inj_on_RDirProd_carrier:
shows "inj_on (λ(a, as). a # as) (carrier (RDirProd R (RDirProd_list Rs)))"
⟨*proof*⟩

lemma RDirProd_list_is_ring:
assumes "⋀i. i < length Rs ⟹ ring (Rs ! i)" **shows** "ring (RDirProd_list Rs)"
⟨*proof*⟩

lemma RDirProd_list_iso1:
"(λ(a, as). a # as) ∈ ring_iso (RDirProd R (RDirProd_list Rs)) (RDirProd_list (R # Rs))"
⟨*proof*⟩

lemma RDirProd_list_iso2:
"Hilbert_Choice.inv (λ(a, as). a # as) ∈ ring_iso (RDirProd_list (R # Rs)) (RDirProd R (RDirProd_list Rs))"
⟨*proof*⟩

lemma RDirProd_list_iso3:
"(λa. [a]) ∈ ring_iso R (RDirProd_list [R])"
⟨*proof*⟩

lemma RDirProd_list_hom1:
"(λ(a, as). a # as) ∈ ring_hom (RDirProd R (RDirProd_list Rs)) (RDirProd_list (R # Rs))"
⟨*proof*⟩

lemma RDirProd_list_hom2:
assumes "f ∈ ring_hom R S" **shows** "(λa. [f a]) ∈ ring_hom R (RDirProd_list [S])"
⟨*proof*⟩

42 Chinese Remainder Theorem

42.1 Definitions

abbreviation (**in** ring) canonical_proj :: "'a set ⇒ 'a set ⇒ 'a ⇒ 'a set × 'a set"
where "canonical_proj I J ≡ (λa. (I +> a, J +> a))"

definition (**in** ring) canonical_proj_ext :: "(nat ⇒ 'a set) ⇒ nat ⇒ 'a ⇒ ('a set) list"
where "canonical_proj_ext I n = (λa. map (λi. (I i) +> a) [0..< Suc

```
n])"
```

42.2 Chinese Remainder Simple

lemma (in ring**)** canonical_proj_is_surj:
assumes "ideal I R" "ideal J R" **and** "I <+> J = carrier R"
shows "(canonical_proj I J) ‘ carrier R = carrier (RDirProd (R Quot I) (R Quot J))"
⟨*proof*⟩

lemma (in ring**)** canonical_proj_ker:
assumes "ideal I R" **and** "ideal J R"
shows "a_kernel R (RDirProd (R Quot I) (R Quot J)) (canonical_proj I J) = I ∩ J"
⟨*proof*⟩

lemma (in ring**)** canonical_proj_is_hom:
assumes "ideal I R" **and** "ideal J R"
shows "(canonical_proj I J) ∈ ring_hom R (RDirProd (R Quot I) (R Quot J))"
⟨*proof*⟩

lemma (in ring**)** canonical_proj_ring_hom:
assumes "ideal I R" **and** "ideal J R"
shows "ring_hom_ring R (RDirProd (R Quot I) (R Quot J)) (canonical_proj I J)"
⟨*proof*⟩

theorem (in ring**)** chinese_remainder_simple:
assumes "ideal I R" "ideal J R" **and** "I <+> J = carrier R"
shows "R Quot (I ∩ J) ≃ RDirProd (R Quot I) (R Quot J)"
⟨*proof*⟩

42.3 Chinese Remainder Generalized

lemma (in ring**)** canonical_proj_ext_zero [simp]: "(canonical_proj_ext I 0) = (λa. [(I 0) +> a])"
⟨*proof*⟩

lemma (in ring**)** canonical_proj_ext_tl:
"(λa. canonical_proj_ext I (Suc n) a) = (λa. ((I 0) +> a) # (canonical_proj_ext (λi. I (Suc i)) n a))"
⟨*proof*⟩

lemma (in ring**)** canonical_proj_ext_is_hom:
assumes "⋀i. i ≤ n ⟹ ideal (I i) R"
shows "(canonical_proj_ext I n) ∈ ring_hom R (RDirProd_list (map (λi. R Quot (I i)) [0..< Suc n]))"
⟨*proof*⟩

lemma (in ring**)** RDirProd_Quot_list_is_ring:
assumes "⋀i. i ≤ n ⟹ ideal (I i) R" **shows** "ring (RDirProd_list (map (λi. R Quot (I i)) [0..< Suc n]))"
⟨*proof*⟩

lemma (in ring**)** canonical_proj_ext_ring_hom:
assumes "⋀i. i ≤ n ⟹ ideal (I i) R"
shows "ring_hom_ring R (RDirProd_list (map (λi. R Quot (I i)) [0..< Suc n])) (canonical_proj_ext I n)"
⟨*proof*⟩

lemma (in ring**)** canonical_proj_ext_ker:
assumes "⋀i. i ≤ (n :: nat) ⟹ ideal (I i) R"
shows "a_kernel R (RDirProd_list (map (λi. R Quot (I i)) [0..< Suc n])) (canonical_proj_ext I n) = (⋂i ≤ n. I i)"
⟨*proof*⟩

lemma (in cring**)** canonical_proj_ext_is_surj:
assumes "⋀i. i ≤ n ⟹ ideal (I i) R" **and** "⋀i j. ⟦ i ≤ n; j ≤ n ⟧ ⟹ i ≠ j ⟹ I i <+> I j = carrier R"
shows "(canonical_proj_ext I n) ‘ carrier R = carrier (RDirProd_list (map (λi. R Quot (I i)) [0..< Suc n]))"
⟨*proof*⟩

theorem (in cring**)** chinese_remainder:
assumes "⋀i. i ≤ n ⟹ ideal (I i) R" **and** "⋀i j. ⟦ i ≤ n; j ≤ n ⟧ ⟹ i ≠ j ⟹ I i <+> I j = carrier R"
shows "R Quot (⋂i ≤ n. I i) ≃ RDirProd_list (map (λi. R Quot (I i)) [0..< Suc n])"
⟨*proof*⟩

end

theory Generated_Rings
imports Subrings
begin

43 Generated Rings

inductive_set
generate_ring :: "('a, 'b) ring_scheme ⇒ 'a set ⇒ 'a set"
for R **and** H **where**
one: "$\mathbf{1}_R$ ∈ generate_ring R H"
| incl: "h ∈ H ⟹ h ∈ generate_ring R H"
| a_inv: "h ∈ generate_ring R H ⟹ $\ominus_R$ h ∈ generate_ring R H"

```
  | eng_add : "⟦ h1 ∈ generate_ring R H; h2 ∈ generate_ring R H ⟧ ⟹
h1 ⊕R h2 ∈ generate_ring R H"
  | eng_mult: "⟦ h1 ∈ generate_ring R H; h2 ∈ generate_ring R H ⟧ ⟹
h1 ⊗R h2 ∈ generate_ring R H"
```

43.1 Basic Properties of Generated Rings - First Part

lemma (in ring**)** generate_ring_in_carrier:
  assumes "H ⊆ carrier R"
  shows "h ∈ generate_ring R H ⟹ h ∈ carrier R"
  ⟨*proof*⟩

lemma (in ring**)** generate_ring_incl:
  assumes "H ⊆ carrier R"
  shows "generate_ring R H ⊆ carrier R"
  ⟨*proof*⟩

lemma (in ring**)** zero_in_generate: "0R ∈ generate_ring R H"
  ⟨*proof*⟩

lemma (in ring**)** generate_ring_is_subring:
  assumes "H ⊆ carrier R"
  shows "subring (generate_ring R H) R"
  ⟨*proof*⟩

lemma (in ring**)** generate_ring_is_ring:
  assumes "H ⊆ carrier R"
  shows "ring (R (| carrier := generate_ring R H |))"
  ⟨*proof*⟩

lemma (in ring**)** generate_ring_min_subring1:
  assumes "H ⊆ carrier R" **and** "subring E R" "H ⊆ E"
  shows "generate_ring R H ⊆ E"
⟨*proof*⟩

lemma (in ring**)** generate_ringI:
  assumes "H ⊆ carrier R"
    and "subring E R" "H ⊆ E"
    and "⋀K. ⟦ subring K R; H ⊆ K ⟧ ⟹ E ⊆ K"
  shows "E = generate_ring R H"
⟨*proof*⟩

lemma (in ring**)** generate_ringE:
  assumes "H ⊆ carrier R" **and** "E = generate_ring R H"
  shows "subring E R" **and** "H ⊆ E" **and** "⋀K. ⟦ subring K R; H ⊆ K ⟧ ⟹
E ⊆ K"
⟨*proof*⟩

lemma (in ring**)** generate_ring_min_subring2:

assumes "H ⊆ carrier R"
shows "generate_ring R H = ⋂{K. subring K R ∧ H ⊆ K}"
⟨*proof*⟩

lemma (in ring**)** mono_generate_ring:
assumes "I ⊆ J" **and** "J ⊆ carrier R"
shows "generate_ring R I ⊆ generate_ring R J"
⟨*proof*⟩

lemma (in ring**)** subring_gen_incl :
assumes "subring H R"
and "subring K R"
and "I ⊆ H"
and "I ⊆ K"
shows "generate_ring (R⦇carrier := K⦈) I ⊆ generate_ring (R⦇carrier := H⦈) I"
⟨*proof*⟩

lemma (in ring**)** subring_gen_equality:
assumes "subring H R" "K ⊆ H"
shows "generate_ring R K = generate_ring (R ⦇ carrier := H ⦈) K"
⟨*proof*⟩

end

theory Generated_Fields
imports Generated_Rings Subrings Multiplicative_Group
begin

inductive_set
generate_field :: "('a, 'b) ring_scheme ⇒ 'a set ⇒ 'a set"
for R **and** H **where**
one : "$\mathbf{1}_R$ ∈ generate_field R H"
| incl : "h ∈ H ⟹ h ∈ generate_field R H"
| a_inv: "h ∈ generate_field R H ⟹ $\ominus_R$ h ∈ generate_field R H"
| m_inv: "⟦ h ∈ generate_field R H; h ≠ $\mathbf{0}_R$ ⟧ ⟹ inv_R h ∈ generate_field R H"
| eng_add : "⟦ h1 ∈ generate_field R H; h2 ∈ generate_field R H ⟧ ⟹ h1 $\oplus_R$ h2 ∈ generate_field R H"
| eng_mult: "⟦ h1 ∈ generate_field R H; h2 ∈ generate_field R H ⟧ ⟹ h1 $\otimes_R$ h2 ∈ generate_field R H"

43.2 Basic Properties of Generated Rings - First Part

lemma (in field**)** generate_field_in_carrier:
assumes "H ⊆ carrier R"
shows "h ∈ generate_field R H ⟹ h ∈ carrier R"
⟨*proof*⟩

lemma (in field**)** generate_field_incl:
assumes "H ⊆ carrier R"
shows "generate_field R H ⊆ carrier R"
⟨*proof*⟩

lemma (in field**)** zero_in_generate: "$\mathbf{0}_R$ ∈ generate_field R H"
⟨*proof*⟩

lemma (in field**)** generate_field_is_subfield:
assumes "H ⊆ carrier R"
shows "subfield (generate_field R H) R"
⟨*proof*⟩

lemma (in field**)** generate_field_is_add_subgroup:
assumes "H ⊆ carrier R"
shows "subgroup (generate_field R H) (add_monoid R)"
⟨*proof*⟩

lemma (in field**)** generate_field_is_field :
assumes "H ⊆ carrier R"
shows "field (R (| carrier := generate_field R H |))"
⟨*proof*⟩

lemma (in field**)** generate_field_min_subfield1:
assumes "H ⊆ carrier R"
and "subfield E R" "H ⊆ E"
shows "generate_field R H ⊆ E"
⟨*proof*⟩

lemma (in field**)** generate_fieldI:
assumes "H ⊆ carrier R"
and "subfield E R" "H ⊆ E"
and "⋀K. ⟦ subfield K R; H ⊆ K ⟧ ⟹ E ⊆ K"
shows "E = generate_field R H"
⟨*proof*⟩

lemma (in field**)** generate_fieldE:
assumes "H ⊆ carrier R" **and** "E = generate_field R H"
shows "subfield E R" **and** "H ⊆ E" **and** "⋀K. ⟦ subfield K R; H ⊆ K ⟧
⟹ E ⊆ K"
⟨*proof*⟩

lemma (in field**)** generate_field_min_subfield2:
assumes "H ⊆ carrier R"
shows "generate_field R H = ⋂{K. subfield K R ∧ H ⊆ K}"
⟨*proof*⟩

lemma (in field**)** mono_generate_field:

assumes "I ⊆ J" **and** "J ⊆ carrier R"
shows "generate_field R I ⊆ generate_field R J"
⟨*proof*⟩

lemma (**in** field) subfield_gen_incl :
assumes "subfield H R"
and "subfield K R"
and "I ⊆ H"
and "I ⊆ K"
shows "generate_field (R⦇carrier := K⦈) I ⊆ generate_field (R⦇carrier := H⦈) I"
⟨*proof*⟩

lemma (**in** field) subfield_gen_equality:
assumes "subfield H R" "K ⊆ H"
shows "generate_field R K = generate_field (R ⦇ carrier := H ⦈) K"
⟨*proof*⟩

end

44 Product and Sum Groups

theory Product_Groups
imports Elementary_Groups "HOL-Library.Equipollence"

begin

44.1 Product of a Family of Groups

definition product_group:: "'a set ⇒ ('a ⇒ ('b, 'c) monoid_scheme) ⇒ ('a ⇒ 'b) monoid"
where "product_group I G ≡ ⦇carrier = (Π$_E$ i∈I. carrier (G i)),
monoid.mult = (λx y. (λi∈I. x i ⊗$_{G\ i}$ y i)),
one = (λi∈I. **1**$_{G\ i}$)⦈"

lemma carrier_product_group [simp]: "carrier(product_group I G) = (Π$_E$ i∈I. carrier (G i))"
⟨*proof*⟩

lemma one_product_group [simp]: "one(product_group I G) = (λi∈I. one (G i))"
⟨*proof*⟩

lemma mult_product_group [simp]: "(⊗$_{\text{product_group I G}}$) = (λx y. λi∈I. x i ⊗$_{G\ i}$ y i)"
⟨*proof*⟩

lemma product_group [simp]:
assumes "⋀i. i ∈ I ⟹ group (G i)" **shows** "group (product_group I G)"
⟨*proof*⟩

lemma inv_product_group [simp]:
assumes "f ∈ (Π$_E$ i∈I. carrier (G i))" "⋀i. i ∈ I ⟹ group (G i)"
shows "inv$_{\text{product_group I G}}$ f = (λi∈I. inv$_{\text{G i}}$ f i)"
⟨*proof*⟩

lemma trivial_product_group: "trivial_group(product_group I G) ⟷ (∀i ∈ I. trivial_group(G i))"
(is "?lhs = ?rhs")
⟨*proof*⟩

lemma PiE_subgroup_product_group:
assumes "⋀i. i ∈ I ⟹ group (G i)"
shows "subgroup (PiE I H) (product_group I G) ⟷ (∀i ∈ I. subgroup (H i) (G i))"
(is "?lhs = ?rhs")
⟨*proof*⟩

lemma product_group_subgroup_generated:
assumes "⋀i. i ∈ I ⟹ subgroup (H i) (G i)" **and** gp: "⋀i. i ∈ I ⟹ group (G i)"
shows "product_group I (λi. subgroup_generated (G i) (H i))
= subgroup_generated (product_group I G) (PiE I H)"
⟨*proof*⟩

lemma finite_product_group:
assumes "⋀i. i ∈ I ⟹ group (G i)"
shows
"finite (carrier (product_group I G)) ⟷
finite {i. i ∈ I ∧ ~ trivial_group(G i)} ∧ (∀i ∈ I. finite(carrier(G i)))"
⟨*proof*⟩

44.2 Sum of a Family of Groups

definition sum_group :: "'a set ⇒ ('a ⇒ ('b, 'c) monoid_scheme) ⇒ ('a ⇒ 'b) monoid"
where "sum_group I G ≡
subgroup_generated
(product_group I G)
{x ∈ Π$_E$ i∈I. carrier (G i). finite {i ∈ I. x i ≠ **1**$_{\text{G i}}$}}"

lemma subgroup_sum_group:

assumes "⋀i. i ∈ I ⟹ group (G i)"
shows "subgroup {x ∈ Π_E i∈I. carrier (G i). finite {i ∈ I. x i ≠ $\mathbf{1}_{G\ i}$}}
(product_group I G)"
⟨*proof*⟩

lemma carrier_sum_group:
assumes "⋀i. i ∈ I ⟹ group (G i)"
shows "carrier(sum_group I G) = {x ∈ Π_E i∈I. carrier (G i). finite
{i ∈ I. x i ≠ $\mathbf{1}_{G\ i}$}}"
⟨*proof*⟩

lemma one_sum_group [simp]: "$\mathbf{1}_{sum_group\ I\ G}$ = (λi∈I. $\mathbf{1}_{G\ i}$)"
⟨*proof*⟩

lemma mult_sum_group [simp]: "($\otimes_{sum_group\ I\ G}$) = (λx y. (λi∈I. x i $\otimes_{G\ i}$
y i))"
⟨*proof*⟩

lemma sum_group [simp]:
assumes "⋀i. i ∈ I ⟹ group (G i)" **shows** "group (sum_group I G)"
⟨*proof*⟩

lemma inv_sum_group [simp]:
assumes "⋀i. i ∈ I ⟹ group (G i)" **and** x: "x ∈ carrier (sum_group
I G)"
shows "m_inv (sum_group I G) x = (λi∈I. m_inv (G i) (x i))"
⟨*proof*⟩

thm group.subgroups_Inter
theorem subgroup_Inter:
assumes subgr: "(⋀H. H ∈ A ⟹ subgroup H G)"
and not_empty: "A ≠ {}"
shows "subgroup (⋂A) G"
⟨*proof*⟩

thm group.subgroups_Inter_pair
lemma subgroup_Int:
assumes "subgroup I G" "subgroup J G"
shows "subgroup (I ∩ J) G" ⟨*proof*⟩

lemma sum_group_subgroup_generated:
assumes "⋀i. i ∈ I ⟹ group (G i)" **and** sg: "⋀i. i ∈ I ⟹ subgroup
(H i) (G i)"
shows "sum_group I (λi. subgroup_generated (G i) (H i)) = subgroup_generated
(sum_group I G) (PiE I H)"
⟨*proof*⟩

lemma iso_product_groupI:
  assumes iso: "⋀i. i ∈ I ⟹ G i ≅ H i"
    and G: "⋀i. i ∈ I ⟹ group (G i)" **and** H: "⋀i. i ∈ I ⟹ group (H i)"
  shows "product_group I G ≅ product_group I H" (**is** "?IG ≅ ?IH")
⟨*proof*⟩

lemma iso_sum_groupI:
  assumes iso: "⋀i. i ∈ I ⟹ G i ≅ H i"
    and G: "⋀i. i ∈ I ⟹ group (G i)" **and** H: "⋀i. i ∈ I ⟹ group (H i)"
  shows "sum_group I G ≅ sum_group I H" (**is** "?IG ≅ ?IH")
⟨*proof*⟩

end

45 Free Abelian Groups

theory Free_Abelian_Groups
  imports
    Product_Groups FiniteProduct "HOL-Cardinals.Cardinal_Arithmetic"
   "HOL-Library.Countable_Set" "HOL-Library.Poly_Mapping" "HOL-Library.Equipollence"

begin

lemma eqpoll_Fpow:
  assumes "infinite A" **shows** "Fpow A ≈ A"
  ⟨*proof*⟩

lemma infinite_iff_card_of_countable: "⟦countable B; infinite B⟧ ⟹ infinite A ⟷ (|B| ≤o |A|)"
  ⟨*proof*⟩

lemma iso_imp_eqpoll_carrier: "G ≅ H ⟹ carrier G ≈ carrier H"
  ⟨*proof*⟩

45.1 Generalised finite product

definition
  gfinprod :: "[('b, 'm) monoid_scheme, 'a ⇒ 'b, 'a set] ⇒ 'b"
  where "gfinprod G f A =
   (if finite {x ∈ A. f x ≠ 1_G} then finprod G f {x ∈ A. f x ≠ 1_G} else 1_G)"

context comm_monoid **begin**

lemma gfinprod_closed [simp]:

"f ∈ A → carrier G ⟹ gfinprod G f A ∈ carrier G"
⟨*proof*⟩

lemma gfinprod_cong:
"⟦A = B; f ∈ B → carrier G;
⋀i. i ∈ B =simp=> f i = g i⟧ ⟹ gfinprod G f A = gfinprod G g B"
⟨*proof*⟩

lemma gfinprod_eq_finprod [simp]: "⟦finite A; f ∈ A → carrier G⟧ ⟹
gfinprod G f A = finprod G f A"
⟨*proof*⟩

lemma gfinprod_insert [simp]:
assumes "finite {x ∈ A. f x ≠ **1**$_G$}" "f ∈ A → carrier G" "f i ∈ carrier
G"
shows "gfinprod G f (insert i A) = (if i ∈ A then gfinprod G f A else
f i ⊗ gfinprod G f A)"
⟨*proof*⟩

lemma gfinprod_distrib:
assumes fin: "finite {x ∈ A. f x ≠ **1**$_G$}" "finite {x ∈ A. g x ≠ **1**$_G$}"
and "f ∈ A → carrier G" "g ∈ A → carrier G"
shows "gfinprod G (λi. f i ⊗ g i) A = gfinprod G f A ⊗ gfinprod G
g A"
⟨*proof*⟩

lemma gfinprod_mono_neutral_cong_left:
assumes "A ⊆ B"
and 1: "⋀i. i ∈ B - A ⟹ h i = **1**"
and gh: "⋀x. x ∈ A ⟹ g x = h x"
and h: "h ∈ B → carrier G"
shows "gfinprod G g A = gfinprod G h B"
⟨*proof*⟩

lemma gfinprod_mono_neutral_cong_right:
assumes "A ⊆ B" "⋀i. i ∈ B - A ⟹ g i = **1**" "⋀x. x ∈ A ⟹ g x =
h x" "g ∈ B → carrier G"
shows "gfinprod G g B = gfinprod G h A"
⟨*proof*⟩

lemma gfinprod_mono_neutral_cong:
assumes [simp]: "finite B" "finite A"
and ∗: "⋀i. i ∈ B - A ⟹ h i = **1**" "⋀i. i ∈ A - B ⟹ g i = **1**"
and gh: "⋀x. x ∈ A ∩ B ⟹ g x = h x"
and g: "g ∈ A → carrier G"
and h: "h ∈ B → carrier G"
shows "gfinprod G g A = gfinprod G h B"
⟨*proof*⟩

end

lemma (in comm_group**)** hom_group_sum:
assumes hom: "⋀i. i ∈ I ⟹ f i ∈ hom (A i) G" **and** grp: "⋀i. i ∈ I ⟹ group (A i)"
shows "(λx. gfinprod G (λi. (f i) (x i)) I) ∈ hom (sum_group I A) G"
⟨*proof*⟩

45.2 Free Abelian groups on a set, using the "frag" type constructor.

definition free_Abelian_group :: "'a set ⇒ ('a ⇒$_0$ int) monoid"
where "free_Abelian_group S = ⦇carrier = {c. Poly_Mapping.keys c ⊆ S}, monoid.mult = (+), one = 0⦈"

lemma group_free_Abelian_group [simp]: "group (free_Abelian_group S)"
⟨*proof*⟩

lemma carrier_free_Abelian_group_iff [simp]:
shows "x ∈ carrier (free_Abelian_group S) ⟷ Poly_Mapping.keys x ⊆ S"
⟨*proof*⟩

lemma one_free_Abelian_group [simp]: "$\mathbf{1}_{\text{free_Abelian_group S}}$ = 0"
⟨*proof*⟩

lemma mult_free_Abelian_group [simp]: "x $\otimes_{\text{free_Abelian_group S}}$ y = x + y"
⟨*proof*⟩

lemma inv_free_Abelian_group [simp]: "Poly_Mapping.keys x ⊆ S ⟹ $\text{inv}_{\text{free_Abelian_group S}}$ x = -x"
⟨*proof*⟩

lemma abelian_free_Abelian_group: "comm_group(free_Abelian_group S)"
⟨*proof*⟩

lemma pow_free_Abelian_group [simp]:
fixes n::nat
shows "Group.pow (free_Abelian_group S) x n = frag_cmul (int n) x"
⟨*proof*⟩

lemma int_pow_free_Abelian_group [simp]:
fixes n::int
assumes "Poly_Mapping.keys x ⊆ S"
shows "Group.pow (free_Abelian_group S) x n = frag_cmul n x"
⟨*proof*⟩

lemma frag_of_in_free_Abelian_group [simp]:

```
   "frag_of x ∈ carrier(free_Abelian_group S) ⟷ x ∈ S"
```
⟨*proof*⟩

lemma free_Abelian_group_induct:
```
  assumes major: "Poly_Mapping.keys x ⊆ S"
      and minor: "P(0)"
           "⋀x y. ⟦Poly_Mapping.keys x ⊆ S; Poly_Mapping.keys y ⊆ S;
P x; P y⟧ ⟹ P(x-y)"
           "⋀a. a ∈ S ⟹ P(frag_of a)"
         shows "P x"
```
⟨*proof*⟩

lemma sum_closed_free_Abelian_group:
```
    "(⋀i. i ∈ I ⟹ x i ∈ carrier (free_Abelian_group S)) ⟹ sum x I
∈ carrier (free_Abelian_group S)"
```
⟨*proof*⟩

lemma (**in** comm_group) free_Abelian_group_universal:
```
  fixes f :: "'c ⇒ 'a"
  assumes "f ‘ S ⊆ carrier G"
  obtains h where "h ∈ hom (free_Abelian_group S) G" "⋀x. x ∈ S ⟹
h(frag_of x) = f x"
```
⟨*proof*⟩

lemma eqpoll_free_Abelian_group_infinite:
```
  assumes "infinite A" shows "carrier(free_Abelian_group A) ≈ A"
```
⟨*proof*⟩

proposition (**in** comm_group) eqpoll_homomorphisms_from_free_Abelian_group:
```
   "{f. f ∈ extensional (carrier(free_Abelian_group S)) ∧ f ∈ hom (free_Abelian_group
S) G}
    ≈ (S →E carrier G)"  (is "?lhs ≈ ?rhs")
```
⟨*proof*⟩

lemma hom_frag_minus:
```
  assumes "h ∈ hom (free_Abelian_group S) (free_Abelian_group T)" "Poly_Mapping.keys
a ⊆ S"
  shows "h (-a) = - (h a)"
```
⟨*proof*⟩

lemma hom_frag_add:
```
  assumes "h ∈ hom (free_Abelian_group S) (free_Abelian_group T)" "Poly_Mapping.keys
a ⊆ S" "Poly_Mapping.keys b ⊆ S"
  shows "h (a+b) = h a + h b"
```
⟨*proof*⟩

lemma hom_frag_diff:
```
  assumes "h ∈ hom (free_Abelian_group S) (free_Abelian_group T)" "Poly_Mapping.keys
```

a ⊆ S" "Poly_Mapping.keys b ⊆ S"
shows "h (a-b) = h a - h b"
⟨*proof*⟩

proposition isomorphic_free_Abelian_groups:
"free_Abelian_group S ≅ free_Abelian_group T ⟷ S ≈ T" (**is** "(?FS ≅ ?FT) = ?rhs")
⟨*proof*⟩

lemma isomorphic_group_integer_free_Abelian_group_singleton:
"integer_group ≅ free_Abelian_group {x}"
⟨*proof*⟩

lemma group_hom_free_Abelian_groups_id:
"id ∈ hom (free_Abelian_group S) (free_Abelian_group T) ⟷ S ⊆ T"
⟨*proof*⟩

proposition iso_free_Abelian_group_sum:
assumes "pairwise (λi j. disjnt (S i) (S j)) I"
shows "(λf. sum' f I) ∈ iso (sum_group I (λi. free_Abelian_group(S i))) (free_Abelian_group (⋃(S ' I)))"
(**is** "?h ∈ iso ?G ?H")
⟨*proof*⟩

lemma isomorphic_free_Abelian_group_Union:
"pairwise disjnt I ⟹ free_Abelian_group(⋃ I) ≅ sum_group I free_Abelian_group"
⟨*proof*⟩

lemma isomorphic_sum_integer_group:
"sum_group I (λi. integer_group) ≅ free_Abelian_group I"
⟨*proof*⟩

end

theory Solvable_Groups
imports Generated_Groups

begin

46 Solvable Groups

46.1 Definitions

inductive solvable_seq :: "('a, 'b) monoid_scheme ⇒ 'a set ⇒ bool"
for G **where**
unity: "solvable_seq G { $\mathbf{1}_G$ }"
| extension: "⟦ solvable_seq G K; K ◁ (G ⦇ carrier := H ⦈); subgroup

```
H G;
                    comm_group ((G (| carrier := H |)) Mod K) ⟧ ⟹ solvable_seq
G H"

definition solvable :: "('a, 'b) monoid_scheme ⇒ bool"
  where "solvable G ⟷ solvable_seq G (carrier G)"
```

46.2 Solvable Groups and Derived Subgroups

We show that a group G is solvable iff the subgroup (derived G ‘n) (carrier G) is trivial for a sufficiently large n.

```
lemma (in group) solvable_imp_subgroup:
  assumes "solvable_seq G H" shows "subgroup H G"
  ⟨proof⟩

lemma (in group) augment_solvable_seq:
  assumes "subgroup H G" and "solvable_seq G (derived G H)" shows "solvable_seq
G H"
  ⟨proof⟩

theorem (in group) trivial_derived_seq_imp_solvable:
  assumes "subgroup H G" and "((derived G) ^^ n) H = { 1 }" shows "solvable_seq
G H"
  ⟨proof⟩

theorem (in group) solvable_imp_trivial_derived_seq:
  assumes "solvable_seq G H" shows "∃n. (derived G ^^ n) H = { 1 }"
  ⟨proof⟩

theorem (in group) solvable_iff_trivial_derived_seq:
  "solvable G ⟷ (∃n. (derived G ^^ n) (carrier G) = { 1 })"
  ⟨proof⟩

corollary (in group) solvable_subgroup:
  assumes "subgroup H G" and "solvable G" shows "solvable_seq G H"
⟨proof⟩
```

46.3 Short Exact Sequences

Even if we don’t talk about short exact sequences explicitly, we show that given an injective homomorphism from a group H to a group G, if H isn’t solvable the group G isn’t neither.

```
theorem (in group_hom) solvable_img_imp_solvable:
  assumes "subgroup K G" and "inj_on h K" and "solvable_seq H (h ‘ K)"
shows "solvable_seq G K"
⟨proof⟩

corollary (in group_hom) inj_hom_imp_solvable:
```

assumes "inj_on h (carrier G)" **and** "solvable H" **shows** "solvable G"
⟨*proof*⟩

theorem (**in** group_hom) solvable_imp_solvable_img:
assumes "solvable_seq G K" **shows** "solvable_seq H (h ‘ K)"
⟨*proof*⟩

corollary (**in** group_hom) surj_hom_imp_solvable:
assumes "h ‘ carrier G = carrier H" **and** "solvable G" **shows** "solvable H"
⟨*proof*⟩

lemma solvable_seq_condition:
assumes "group_hom G H f" "group_hom H K g" **and** "f ‘ I ⊆ J" **and** "kernel H K g ⊆ f ‘ I"
and "subgroup J H" **and** "solvable_seq G I" "solvable_seq K (g ‘ J)"
shows "solvable_seq H J"
⟨*proof*⟩

lemma solvable_condition:
assumes "group_hom G H f" "group_hom H K g"
and "g ‘ (carrier H) = carrier K" **and** "kernel H K g ⊆ f ‘ (carrier G)"
and "solvable G" "solvable K" **shows** "solvable H"
⟨*proof*⟩

end

theory Sym_Groups
imports
"HOL-Combinatorics.Cycles"
Solvable_Groups
begin

47 Symmetric Groups

47.1 Definitions

abbreviation inv' :: "('a ⇒ 'b) ⇒ ('b ⇒ 'a)"
where "inv' f ≡ Hilbert_Choice.inv f"

definition sym_group :: "nat ⇒ (nat ⇒ nat) monoid"
where "sym_group n = ⦇ carrier = { p. p permutes {1..n} }, mult = (∘), one = id ⦈"

definition alt_group :: "nat ⇒ (nat ⇒ nat) monoid"
where "alt_group n = (sym_group n) ⦇ carrier := { p. p permutes {1..n} ∧ evenperm p } ⦈"

definition sign_img :: "int monoid"
  where "sign_img = ⦇ carrier = { -1, 1 }, mult = (*), one = 1 ⦈"

47.2 Basic Properties

lemma sym_group_carrier: "p ∈ carrier (sym_group n) ⟷ p permutes {1..n}"
  ⟨*proof*⟩

lemma sym_group_mult: "mult (sym_group n) = (∘)"
  ⟨*proof*⟩

lemma sym_group_one: "one (sym_group n) = id"
  ⟨*proof*⟩

lemma sym_group_carrier': "p ∈ carrier (sym_group n) ⟹ permutation p"
  ⟨*proof*⟩

lemma alt_group_carrier: "p ∈ carrier (alt_group n) ⟷ p permutes {1..n} ∧ evenperm p"
  ⟨*proof*⟩

lemma alt_group_mult: "mult (alt_group n) = (∘)"
  ⟨*proof*⟩

lemma alt_group_one: "one (alt_group n) = id"
  ⟨*proof*⟩

lemma alt_group_carrier': "p ∈ carrier (alt_group n) ⟹ permutation p"
  ⟨*proof*⟩

lemma sym_group_is_group: "group (sym_group n)"
  ⟨*proof*⟩

lemma sign_img_is_group: "group sign_img"
  ⟨*proof*⟩

lemma sym_group_inv_closed:
  assumes "p ∈ carrier (sym_group n)" **shows** "inv' p ∈ carrier (sym_group n)"
  ⟨*proof*⟩

lemma alt_group_inv_closed:
  assumes "p ∈ carrier (alt_group n)" **shows** "inv' p ∈ carrier (alt_group n)"
  ⟨*proof*⟩

lemma sym_group_inv_equality [simp]:
assumes "p ∈ carrier (sym_group n)" **shows** "inv$_{\text{(sym_group n)}}$ p = inv' p"
⟨*proof*⟩

lemma sign_is_hom: "sign ∈ hom (sym_group n) sign_img"
⟨*proof*⟩

lemma sign_group_hom: "group_hom (sym_group n) sign_img sign"
⟨*proof*⟩

lemma sign_is_surj:
assumes "n ≥ 2" **shows** "sign ‘ (carrier (sym_group n)) = carrier sign_img"
⟨*proof*⟩

lemma alt_group_is_sign_kernel:
"carrier (alt_group n) = kernel (sym_group n) sign_img sign"
⟨*proof*⟩

lemma alt_group_is_subgroup: "subgroup (carrier (alt_group n)) (sym_group n)"
⟨*proof*⟩

lemma alt_group_is_group: "group (alt_group n)"
⟨*proof*⟩

lemma sign_iso:
assumes "n ≥ 2" **shows** "(sym_group n) Mod (carrier (alt_group n)) ≅ sign_img"
⟨*proof*⟩

lemma alt_group_inv_equality:
assumes "p ∈ carrier (alt_group n)" **shows** "inv$_{\text{(alt_group n)}}$ p = inv' p"
⟨*proof*⟩

lemma sym_group_card_carrier: "card (carrier (sym_group n)) = fact n"
⟨*proof*⟩

lemma alt_group_card_carrier:
assumes "n ≥ 2" **shows** "2 * card (carrier (alt_group n)) = fact n"
⟨*proof*⟩

47.3 Transposition Sequences

In order to prove that the Alternating Group can be generated by 3-cycles, we need a stronger decomposition of permutations as transposition sequences than the one proposed at Permutations.thy.

inductive swapidseq_ext :: "'a set ⇒ nat ⇒ ('a ⇒ 'a) ⇒ bool"

```
  where
    empty:  "swapidseq_ext {} 0 id"
  | single: "⟦ swapidseq_ext S n p; a ∉ S ⟧ ⟹ swapidseq_ext (insert
a S) n p"
  | comp:   "⟦ swapidseq_ext S n p; a ≠ b ⟧ ⟹
              swapidseq_ext (insert a (insert b S)) (Suc n) ((transpose
a b) ∘ p)"


lemma swapidseq_ext_finite:
  assumes "swapidseq_ext S n p" shows "finite S"
  ⟨proof⟩

lemma swapidseq_ext_zero:
  assumes "finite S" shows "swapidseq_ext S 0 id"
  ⟨proof⟩

lemma swapidseq_ext_imp_swapidseq:
  ‹swapidseq n p› if ‹swapidseq_ext S n p›
⟨proof⟩

lemma swapidseq_ext_zero_imp_id:
  assumes "swapidseq_ext S 0 p" shows "p = id"
⟨proof⟩

lemma swapidseq_ext_finite_expansion:
  assumes "finite B" and "swapidseq_ext A n p" shows "swapidseq_ext
(A ∪ B) n p"
  ⟨proof⟩

lemma swapidseq_ext_backwards:
  assumes "swapidseq_ext A (Suc n) p"
  shows "∃a b A' p'. a ≠ b ∧ A = (insert a (insert b A')) ∧
                     swapidseq_ext A' n p' ∧ p = (transpose a b) ∘ p'"
⟨proof⟩

lemma swapidseq_ext_backwards':
  assumes "swapidseq_ext A (Suc n) p"
  shows "∃a b A' p'. a ∈ A ∧ b ∈ A ∧ a ≠ b ∧ swapidseq_ext A n p' ∧
p = (transpose a b) ∘ p'"
  ⟨proof⟩

lemma swapidseq_ext_endswap:
  assumes "swapidseq_ext S n p" "a ≠ b"
  shows "swapidseq_ext (insert a (insert b S)) (Suc n) (p ∘ (transpose
a b))"
  ⟨proof⟩

lemma swapidseq_ext_extension:
```

assumes "swapidseq_ext A n p" **and** "swapidseq_ext B m q" **and** "A ∩ B = {}"
shows "swapidseq_ext (A ∪ B) (n + m) (p ∘ q)"
⟨*proof*⟩

lemma swapidseq_ext_of_cycles:
assumes "cycle cs" **shows** "swapidseq_ext (set cs) (length cs - 1) (cycle_of_list cs)"
⟨*proof*⟩

lemma cycle_decomp_imp_swapidseq_ext:
assumes "cycle_decomp S p" **shows** "∃n. swapidseq_ext S n p"
⟨*proof*⟩

lemma swapidseq_ext_of_permutation:
assumes "p permutes S" **and** "finite S" **shows** "∃n. swapidseq_ext S n p"
⟨*proof*⟩

lemma split_swapidseq_ext:
assumes "k ≤ n" **and** "swapidseq_ext S n p"
obtains q r U V **where** "swapidseq_ext U (n - k) q" **and** "swapidseq_ext V k r" **and** "p = q ∘ r" **and** "U ∪ V = S"
⟨*proof*⟩

47.4 Unsolvability of Symmetric Groups

We show that symmetric groups (sym_group n) are unsolvable for (5::'a) ≤ n.

abbreviation three_cycles :: "nat ⇒ (nat ⇒ nat) set"
where "three_cycles n ≡
{ cycle_of_list cs | cs. cycle cs ∧ length cs = 3 ∧ set cs ⊆ {1..n} }"

lemma stupid_lemma:
assumes "length cs = 3" **shows** "cs = [(cs ! 0), (cs ! 1), (cs ! 2)]"
⟨*proof*⟩

lemma three_cycles_incl: "three_cycles n ⊆ carrier (alt_group n)"
⟨*proof*⟩

lemma alt_group_carrier_as_three_cycles:
"carrier (alt_group n) = generate (alt_group n) (three_cycles n)"
⟨*proof*⟩

theorem derived_alt_group_const:
assumes "n ≥ 5" **shows** "derived (alt_group n) (carrier (alt_group n))

```
= carrier (alt_group n)"
```
⟨*proof*⟩

corollary alt_group_is_unsolvable:
  assumes "n ≥ 5" **shows** "¬ solvable (alt_group n)"
⟨*proof*⟩

corollary sym_group_is_unsolvable:
  assumes "n ≥ 5" **shows** "¬ solvable (sym_group n)"
⟨*proof*⟩

end

48 Exact Sequences

theory Exact_Sequence
  imports Elementary_Groups Solvable_Groups
begin

48.1 Definitions

inductive exact_seq :: "'a monoid list × ('a ⇒ 'a) list ⇒ bool" **where**
unity: " group_hom G1 G2 f ⟹ exact_seq ([G2, G1], [f])" |
extension: "⟦ exact_seq ((G # K # l), (g # q)); group H ; h ∈ hom G H
;
              kernel G H h = image g (carrier K) ⟧ ⟹ exact_seq (H #
G # K # l, h # g # q)"

inductive_simps exact_seq_end_iff [simp]: "exact_seq ([G,H], (g # q))"
inductive_simps exact_seq_cons_iff [simp]: "exact_seq ((G # K # H # l),
(g # h # q))"

abbreviation exact_seq_arrow ::
  "('a ⇒ 'a) ⇒ 'a monoid list × ('a ⇒ 'a) list ⇒ 'a monoid ⇒ 'a
monoid list × ('a ⇒ 'a) list"
  (‹(‹indent=3 notation=‹mixfix exact_seq››_ / ⟶ı _)› [1000, 60])
  where "exact_seq_arrow f t G ≡ (G # (fst t), f # (snd t))"

48.2 Basic Properties

lemma exact_seq_length1: "exact_seq t ⟹ length (fst t) = Suc (length
(snd t))"
  ⟨*proof*⟩

lemma exact_seq_length2: "exact_seq t ⟹ length (snd t) ≥ Suc 0"
  ⟨*proof*⟩

lemma dropped_seq_is_exact_seq:
  assumes "exact_seq (G, F)" **and** "(i :: nat) < length F"

shows "exact_seq (drop i G, drop i F)"
⟨*proof*⟩

lemma truncated_seq_is_exact_seq:
assumes "exact_seq (l, q)" **and** "length l ≥ 3"
shows "exact_seq (tl l, tl q)"
⟨*proof*⟩

lemma exact_seq_imp_exact_hom:
assumes "exact_seq (G1 # l,q) $\longrightarrow_{g1}$ G2 $\longrightarrow_{g2}$ G3"
shows "g1 ‘ (carrier G1) = kernel G2 G3 g2"
⟨*proof*⟩

lemma exact_seq_imp_exact_hom_arbitrary:
assumes "exact_seq (G, F)"
and "Suc i < length F"
shows "(F ! (Suc i)) ‘ (carrier (G ! (Suc (Suc i)))) = kernel (G ! (Suc i)) (G ! i) (F ! i)"
⟨*proof*⟩

lemma exact_seq_imp_group_hom :
assumes "exact_seq ((G # l, q)) $\longrightarrow_{g}$ H"
shows "group_hom G H g"
⟨*proof*⟩

lemma exact_seq_imp_group_hom_arbitrary:
assumes "exact_seq (G, F)" **and** "(i :: nat) < length F"
shows "group_hom (G ! (Suc i)) (G ! i) (F ! i)"
⟨*proof*⟩

48.3 Link Between Exact Sequences and Solvable Conditions

lemma exact_seq_solvable_imp :
assumes "exact_seq ([G1],[]) $\longrightarrow_{g1}$ G2 $\longrightarrow_{g2}$ G3"
and "inj_on g1 (carrier G1)"
and "g2 ‘ (carrier G2) = carrier G3"
shows "solvable G2 ⟹ (solvable G1) ∧ (solvable G3)"
⟨*proof*⟩

lemma exact_seq_solvable_recip :
assumes "exact_seq ([G1],[]) $\longrightarrow_{g1}$ G2 $\longrightarrow_{g2}$ G3"
and "inj_on g1 (carrier G1)"
and "g2 ‘ (carrier G2) = carrier G3"
shows "(solvable G1) ∧ (solvable G3) ⟹ solvable G2"
⟨*proof*⟩

proposition exact_seq_solvable_iff :
assumes "exact_seq ([G1],[]) $\longrightarrow_{g1}$ G2 $\longrightarrow_{g2}$ G3"
and "inj_on g1 (carrier G1)"

and "g2 ‘ (carrier G2) = carrier G3"
shows "(solvable G1) ∧ (solvable G3) ⟷ solvable G2"
⟨*proof*⟩

lemma exact_seq_eq_triviality:
assumes "exact_seq ([E,D,C,B,A], [k,h,g,f])"
shows "trivial_group C ⟷ f ‘ carrier A = carrier B ∧ inj_on k (carrier D)" (**is** "_ = ?rhs")
⟨*proof*⟩

lemma exact_seq_imp_triviality:
"⟦exact_seq ([E,D,C,B,A], [k,h,g,f]); f ∈ iso A B; k ∈ iso D E⟧ ⟹ trivial_group C"
⟨*proof*⟩

lemma exact_seq_epi_eq_triviality:
"exact_seq ([D,C,B,A], [h,g,f]) ⟹ (f ‘ carrier A = carrier B) ⟷ trivial_homomorphism B C g"
⟨*proof*⟩

lemma exact_seq_mon_eq_triviality:
"exact_seq ([D,C,B,A], [h,g,f]) ⟹ inj_on h (carrier C) ⟷ trivial_homomorphism B C g"
⟨*proof*⟩

lemma exact_sequence_sum_lemma:
assumes "comm_group G" **and** h: "h ∈ iso A C" **and** k: "k ∈ iso B D"
and ex: "exact_seq ([D,G,A], [g,i])" "exact_seq ([C,G,B], [f,j])"
and fih: "⋀x. x ∈ carrier A ⟹ f(i x) = h x"
and gjk: "⋀x. x ∈ carrier B ⟹ g(j x) = k x"
shows "(λ(x, y). i x ⊗$_G$ j y) ∈ Group.iso (A ×× B) G ∧ (λz. (f z, g z)) ∈ Group.iso G (C ×× D)"
(**is** "?ij ∈ _ ∧ ?gf ∈ _")
⟨*proof*⟩

48.4 Splitting lemmas and Short exact sequences

Ported from HOL Light by LCP

definition short_exact_sequence
where "short_exact_sequence A B C f g ≡ ∃T1 T2 e1 e2. exact_seq ([T1,A,B,C,T2], [e1,f,g,e2]) ∧ trivial_group T1 ∧ trivial_group T2"

lemma short_exact_sequenceD:
assumes "short_exact_sequence A B C f g" **shows** "exact_seq ([A,B,C], [f,g]) ∧ f ∈ epi B A ∧ g ∈ mon C B"
⟨*proof*⟩

lemma short_exact_sequence_iff:

```
  "short_exact_sequence A B C f g ⟷ exact_seq ([A,B,C], [f,g]) ∧ f
∈ epi B A ∧ g ∈ mon C B"
```
⟨*proof*⟩

lemma very_short_exact_sequence:
```
  assumes "exact_seq ([D,C,B,A], [h,g,f])" "trivial_group A" "trivial_group
D"
  shows "g ∈ iso B C"
```
⟨*proof*⟩

lemma splitting_sublemma_gen:
```
  assumes ex: "exact_seq ([C,B,A], [g,f])" and fim: "f ‘ carrier A =
H"
      and "subgroup K B" and 1: "H ∩ K ⊆ {one B}" and eq: "set_mult
B H K = carrier B"
  shows "g ∈ iso (subgroup_generated B K) (subgroup_generated C(g ‘ carrier
B))"
```
⟨*proof*⟩

lemma splitting_sublemma:
```
  assumes ex: "short_exact_sequence C B A g f" and fim: "f ‘ carrier
A = H"
      and "subgroup K B" and 1: "H ∩ K ⊆ {one B}" and eq: "set_mult
B H K = carrier B"
    shows "f ∈ iso A (subgroup_generated B H)" (is ?f)
          "g ∈ iso (subgroup_generated B K) C" (is ?g)
```
⟨*proof*⟩

lemma splitting_lemma_left_gen:
```
  assumes ex: "exact_seq ([C,B,A], [g,f])" and f': "f' ∈ hom B A" and
iso: "(f' ∘ f) ∈ iso A A"
    and injf: "inj_on f (carrier A)" and surj: "g ‘ carrier B = carrier
C"
 obtains H K where "H ◁ B" "K ◁ B" "H ∩ K ⊆ {one B}" "set_mult B H K
= carrier B"
                  "f ∈ iso A (subgroup_generated B H)" "g ∈ iso (subgroup_generated
B K) C"
```
⟨*proof*⟩

lemma splitting_lemma_left:
```
  assumes ex: "exact_seq ([C,B,A], [g,f])" and f': "f' ∈ hom B A"
    and inv: "(⋀x. x ∈ carrier A ⟹ f'(f x) = x)"
    and injf: "inj_on f (carrier A)" and surj: "g ‘ carrier B = carrier
C"
 obtains H K where "H ◁ B" "K ◁ B" "H ∩ K ⊆ {one B}" "set_mult B H K
= carrier B"
                  "f ∈ iso A (subgroup_generated B H)" "g ∈ iso (subgroup_generated
B K) C"
```
⟨*proof*⟩

lemma splitting_lemma_right_gen:
assumes ex: "short_exact_sequence C B A g f" **and** g': "g' ∈ hom C B"
and iso: "(g ∘ g') ∈ iso C C"
obtains H K **where** "H ◁ B" "subgroup K B" "H ∩ K ⊆ {one B}" "set_mult B H K = carrier B"
"f ∈ iso A (subgroup_generated B H)" "g ∈ iso (subgroup_generated B K) C"
⟨*proof*⟩

lemma splitting_lemma_right:
assumes ex: "short_exact_sequence C B A g f" **and** g': "g' ∈ hom C B"
and gg': "⋀z. z ∈ carrier C ⟹ g(g' z) = z"
obtains H K **where** "H ◁ B" "subgroup K B" "H ∩ K ⊆ {one B}" "set_mult B H K = carrier B"
"f ∈ iso A (subgroup_generated B H)" "g ∈ iso (subgroup_generated B K) C"
⟨*proof*⟩

end
theory Left_Coset
imports Coset

begin

definition
LCOSETS :: "[_, 'a set] ⇒ ('a set)set"
(‹(‹open_block notation=‹prefix lcosets››lcosets↝ _)› [81] 80)
where "lcosets$_G$ H = (⋃a∈carrier G. {a <#$_G$ H})"

definition
LFactGroup :: "[('a,'b) monoid_scheme, 'a set] ⇒ ('a set) monoid" (**infixl** ‹LMod› 65)
— Actually defined for groups rather than monoids
where "LFactGroup G H = (⦇carrier = lcosets$_G$ H, mult = set_mult G, one = H⦈"

lemma (**in** group) lcos_self: "[| x ∈ carrier G; subgroup H G |] ==> x ∈ x <# H"
⟨*proof*⟩

Elements of a left coset are in the carrier

lemma (**in** subgroup) elemlcos_carrier:
assumes "group G" "a ∈ carrier G" "a' ∈ a <# H"
shows "a' ∈ carrier G"
⟨*proof*⟩

Step one for lemma rcos_module

lemma (in subgroup**)** lcos_module_imp:
  assumes "group G"
  assumes xcarr: "x ∈ carrier G"
      and x'cos: "x' ∈ x <# H"
  shows "(inv x ⊗ x') ∈ H"
⟨*proof*⟩

Left cosets are subsets of the carrier.

lemma (in subgroup**)** lcosets_carrier:
  assumes "group G"
  assumes XH: "X ∈ lcosets H"
  shows "X ⊆ carrier G"
⟨*proof*⟩

lemma (in group**)** lcosets_part_G:
  assumes "subgroup H G"
  shows "⋃(lcosets H) = carrier G"
⟨*proof*⟩

lemma (in group**)** lcosets_subset_PowG:
     "subgroup H G ⟹ lcosets H ⊆ Pow(carrier G)"
  ⟨*proof*⟩

lemma (in group**)** lcos_disjoint:
  assumes "subgroup H G"
  assumes p: "a ∈ lcosets H" "b ∈ lcosets H" "a≠b"
  shows "a ∩ b = {}"
⟨*proof*⟩

The next two lemmas support the proof of card_cosets_equal.

lemma (in group**)** inj_on_f':
    "⟦H ⊆ carrier G; a ∈ carrier G⟧ ⟹ inj_on (λy. y ⊗ inv a) (a <#
H)"
  ⟨*proof*⟩

lemma (in group**)** inj_on_f'':
    "⟦H ⊆ carrier G; a ∈ carrier G⟧ ⟹ inj_on (λy. inv a ⊗ y) (a <#
H)"
  ⟨*proof*⟩

lemma (in group**)** inj_on_g':
    "⟦H ⊆ carrier G; a ∈ carrier G⟧ ⟹ inj_on (λy. a ⊗ y) H"
  ⟨*proof*⟩

lemma (in group**)** l_card_cosets_equal:
  assumes "c ∈ lcosets H" **and** H: "H ⊆ carrier G" **and** fin: "finite(carrier
G)"
  shows "card H = card c"

⟨*proof*⟩

theorem (**in** group) l_lagrange:
  assumes "finite(carrier G)" "subgroup H G"
  shows "card(lcosets H) * card(H) = order(G)"
⟨*proof*⟩

end

theory SimpleGroups
imports Coset "HOL-Computational_Algebra.Primes"
begin

49 Simple Groups

locale simple_group = group +
  assumes order_gt_one: "order G > 1"
  assumes no_real_normal_subgroup: "⋀H. H ◁ G ⟹ (H = carrier G ∨ H
= {1})"

lemma (**in** simple_group) is_simple_group: "simple_group G"
  ⟨*proof*⟩

Simple groups are non-trivial.

lemma (**in** simple_group) simple_not_triv: "carrier G ≠ {1}"
  ⟨*proof*⟩

Every group of prime order is simple

lemma (**in** group) prime_order_simple:
  assumes prime: "prime (order G)"
  shows "simple_group G"
⟨*proof*⟩

Being simple is a property that is preserved by isomorphisms.

lemma (**in** simple_group) iso_simple:
  assumes H: "group H"
  assumes iso: "φ ∈ iso G H"
  shows "simple_group H"
⟨*proof*⟩

As a corollary of this: Factorizing a group by itself does not result in a simple group!

lemma (**in** group) self_factor_not_simple: "¬ simple_group (G Mod (carrier
G))"
⟨*proof*⟩

end

theory SndIsomorphismGrp
imports Coset
begin

50 The Second Isomorphism Theorem for Groups

This theory provides a proof of the second isomorphism theorems for groups. The theorems consist of several facts about normal subgroups.

The first lemma states that whenever we have a subgroup `S` and a normal subgroup `H` of a group `G`, their intersection is normal in `G`

locale second_isomorphism_grp = normal +
  fixes S:: "'a set"
  assumes subgrpS: "subgroup S G"

context second_isomorphism_grp
begin

interpretation groupS: group "G⦇carrier := S⦈"
  ⟨*proof*⟩

lemma normal_subgrp_intersection_normal:
  shows "S ∩ H ◁ (G⦇carrier := S⦈)"
⟨*proof*⟩

lemma normal_set_mult_subgroup:
  shows "subgroup (H <#> S) G"
⟨*proof*⟩

lemma H_contained_in_set_mult:
  shows "H ⊆ H <#> S"
⟨*proof*⟩

lemma S_contained_in_set_mult:
  shows "S ⊆ H <#> S"
⟨*proof*⟩

lemma normal_intersection_hom:
  shows "group_hom (G⦇carrier := S⦈) ((G⦇carrier := H <#> S⦈) Mod H) (λg.
H #> g)"
⟨*proof*⟩

lemma normal_intersection_hom_kernel:
  shows "kernel (G⦇carrier := S⦈) ((G⦇carrier := H <#> S⦈) Mod H) (λg.
H #> g) = H ∩ S"
⟨*proof*⟩

lemma normal_intersection_hom_surj:
  shows "(λg. H #> g) ‘ carrier (G⦇carrier := S⦈) = carrier ((G⦇carrier := H <#> S⦈) Mod H)"
⟨*proof*⟩

Finally we can prove the actual isomorphism theorem:

theorem normal_intersection_quotient_isom:
  shows "(λX. the_elem ((λg. H #> g) ‘ X)) ∈ iso ((G⦇carrier := S⦈) Mod (H ∩ S)) (((G⦇carrier := H <#> S⦈)) Mod H)"
⟨*proof*⟩

end

corollary (**in** group) normal_subgroup_set_mult_closed:
  assumes "M ◁ G" **and** "N ◁ G"
  shows "M <#> N ◁ G"
⟨*proof*⟩

end

theory Algebra
  imports Sylow Chinese_Remainder Zassenhaus Galois_Connection Generated_Fields Free_Abelian_Groups
    Divisibility Embedded_Algebras IntRing Sym_Groups Exact_Sequence Polynomials Algebraic_Closure
    Left_Coset SimpleGroups SndIsomorphismGrp
begin

end

References

[1] C. Ballarin. *Computer Algebra and Theorem Proving.* PhD thesis, University of Cambridge, 1999. Also Computer Laboratory Technical Report number 473.

[2] K. Conrad. Cyclicity of $(Z/(p))^x$. Expository paper from the author's website. http://www.math.uconn.edu/~kconrad/blurbs/grouptheory/cyclicFp.pdf.

[3] N. Jacobson. *Basic Algebra I.* Freeman, 1985.

[4] F. Kammüller and L. C. Paulson. A formal proof of sylow's theorem: An experiment in abstract algebra with Isabelle HOL. *J. Automated Reasoning*, 23:235–264, 1999.