

The Isabelle/HOL Algebra Library

Clemens Ballarin (Editor)

With contributions by Jesús Aransay, Clemens Ballarin, Stephan
Hohe, Florian Kammüller and Lawrence C Paulson
May 22, 2012

Contents

1	Objects	6
1.1	Structure with Carrier Set.	6
1.2	Structure with Carrier and Equivalence Relation <code>eq</code>	6
2	Orders and Lattices	11
2.1	Partial Orders	11
2.1.1	The order relation	11
2.1.2	Upper and lower bounds of a set	13
2.1.3	Least and greatest, as predicate	15
2.2	Lattices	17
2.2.1	Supremum	18
2.2.2	Infimum	20
2.3	Total Orders	21
2.4	Complete Lattices	22
2.5	Orders and Lattices where <code>eq</code> is the Equality	23
2.6	Examples	26
2.6.1	The Powerset of a Set is a Complete Lattice	26
3	Monoids and Groups	26
3.1	Definitions	26
3.2	Groups	29
3.3	Cancellation Laws and Basic Properties	30
3.4	Subgroups	31
3.5	Direct Products	32
3.6	Homomorphisms and Isomorphisms	33
3.7	Commutative Structures	35
3.8	The Lattice of Subgroups of a Group	36
3.9	Product Operator for Commutative Monoids	37
3.9.1	Inductive Definition of a Relation for Products over Sets	37

3.9.2	Products over Finite Sets	40
4	Cosets and Quotient Groups	43
4.1	Basic Properties of Cosets	44
4.2	Normal subgroups	47
4.3	More Properties of Cosets	47
4.3.1	Set of Inverses of an <code>r_coset</code>	48
4.3.2	Theorems for <code><#></code> with <code>#></code> or <code><#</code>	48
4.3.3	An Equivalence Relation	49
4.3.4	Two Distinct Right Cosets are Disjoint	49
4.4	Further lemmas for <code>r_congruent</code>	50
4.5	Order of a Group and Lagrange's Theorem	50
4.6	Quotient Groups: Factorization of a Group	51
4.7	The First Isomorphism Theorem	52
5	Sylow's Theorem	53
5.1	The Combinatorial Argument Underlying the First Sylow Theorem	53
5.2	Main Part of the Proof	57
5.3	Discharging the Assumptions of <code>syLOW_central</code>	57
5.3.1	Introduction and Destruct Rules for <code>H</code>	58
5.4	Equal Cardinalities of <code>M</code> and the Set of Cosets	59
5.4.1	The Opposite Injection	59
5.5	Sylow's Theorem	60
6	Bijections of a Set, Permutation and Automorphism Groups	61
6.1	Bijections Form a Group	61
6.2	Automorphisms Form a Group	62
7	Divisibility in monoids and rings	63
8	Factorial Monoids	63
8.1	Monoids with Cancellation Law	63
8.2	Products of Units in Monoids	64
8.3	Divisibility and Association	64
8.3.1	Function definitions	64
8.3.2	Divisibility	65
8.3.3	Association	67
8.3.4	Division and associativity	68
8.3.5	Multiplication and associativity	69
8.3.6	Units	69
8.3.7	Proper factors	70
8.4	Irreducible Elements and Primes	72
8.4.1	Irreducible elements	72

8.4.2	Prime elements	73
8.5	Factorization and Factorial Monoids	74
8.5.1	Function definitions	74
8.5.2	Comparing lists of elements	75
8.5.3	Properties of lists of elements	77
8.5.4	Factorization in irreducible elements	77
8.5.5	Essentially equal factorizations	79
8.5.6	Factorial monoids and wfactors	82
8.6	Factorizations as Multisets	82
8.6.1	Comparing multisets	83
8.6.2	Interpreting multisets as factorizations	84
8.6.3	Multiplication on multisets	84
8.6.4	Divisibility on multisets	85
8.7	Irreducible Elements are Prime	85
8.8	Greatest Common Divisors and Lowest Common Multiples	86
8.8.1	Definitions	86
8.8.2	Connections to <code>Lattice.thy</code>	87
8.8.3	Existence of gcd and lcm	87
8.9	Conditions for Factoriality	88
8.9.1	Gcd condition	88
8.9.2	Divisor chain condition	90
8.9.3	Primeness condition	90
8.9.4	Application to factorial monoids	90
8.10	Factoriality Theorems	92
9	The Algebraic Hierarchy of Rings	92
9.1	Abelian Groups	92
9.2	Basic Properties	93
9.3	Rings: Basic Definitions	97
9.4	Rings	97
9.4.1	Normaliser for Rings	98
9.4.2	Sums over Finite Sets	100
9.5	Integral Domains	100
9.6	Fields	101
9.7	Morphisms	101
9.8	More Lifting from Groups to Abelian Groups	103
9.8.1	Definitions	103
9.8.2	Cosets	105
9.8.3	Subgroups	106
9.8.4	Additive subgroups are normal	107
9.8.5	Congruence Relation	109
9.8.6	Factorization	110
9.8.7	The First Isomorphism Theorem	111
9.8.8	Homomorphisms	112

9.8.9	Cosets	113
9.8.10	Addition of Subgroups	115
10	Ideals	115
10.1	Definitions	115
10.1.1	General definition	115
10.1.2	Ideals Generated by a Subset of carrier R	116
10.1.3	Principal Ideals	116
10.1.4	Maximal Ideals	116
10.1.5	Prime Ideals	116
10.2	Special Ideals	117
10.3	General Ideal Properties	117
10.4	Intersection of Ideals	117
10.5	Addition of Ideals	118
10.6	Ideals generated by a subset of carrier R	118
10.7	Union of Ideals	120
10.8	Properties of Principal Ideals	120
10.9	Prime Ideals	120
10.10	Maximal Ideals	121
10.11	Derived Theorems	121
11	Homomorphisms of Non-Commutative Rings	122
11.1	The Kernel of a Ring Homomorphism	123
11.2	Cosets	123
12	Quotient Rings	124
12.1	Multiplication on Cosets	124
12.2	Quotient Ring Definition	124
12.3	Factorization over General Ideals	125
12.4	Factorization over Prime Ideals	125
12.5	Factorization over Maximal Ideals	125
13	The Ring of Integers	126
13.1	Some properties of int	126
13.2	\mathcal{Z} : The Set of Integers as Algebraic Structure	126
13.3	Interpretations	126
13.4	Generated Ideals of \mathcal{Z}	128
13.5	Ideals and Divisibility	128
13.6	Ideals and the Modulus	128
13.7	Factorization	129
14	Modules over an Abelian Group	130
14.1	Definitions	130
14.2	Basic Properties of Algebras	131

15 Univariate Polynomials 132

15.1 The Constructor for Univariate Polynomials 132

15.2 Effect of Operations on Coefficients 134

15.3 Polynomials Form a Ring. 135

15.4 Polynomials Form a Commutative Ring. 137

15.5 Polynomials over a commutative ring for a commutative ring 137

15.6 Polynomials Form an Algebra 137

15.7 Further Lemmas Involving Monomials 138

15.8 The Degree Function 140

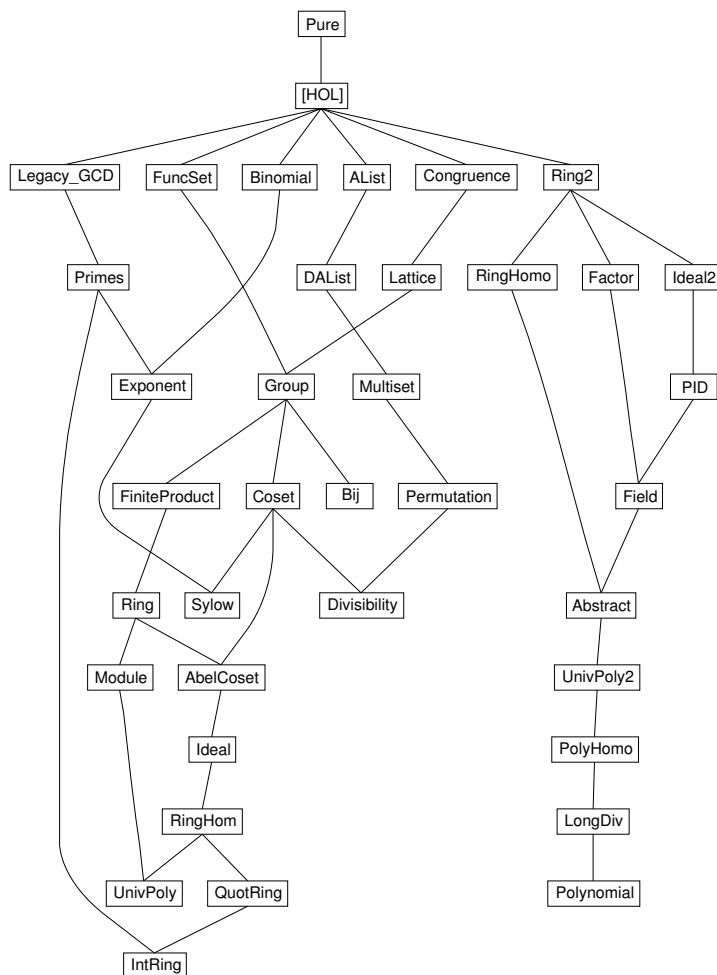
15.9 Polynomials over Integral Domains 142

15.10 The Evaluation Homomorphism and Universal Property . . . 143

15.11 The long division algorithm: some previous facts. 147

15.12 The long division proof for commutative rings 148

15.13 Sample Application of Evaluation Homomorphism 149



```
theory Congruence
imports Main
begin
```

1 Objects

1.1 Structure with Carrier Set.

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

1.2 Structure with Carrier and Equivalence Relation eq

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

definition

```
elem :: "'a ⇒ 'a ⇒ 'a set ⇒ bool" (infixl ".∈ι" 50)
where "x .∈S A ↔ (∃y ∈ A. x .=S y)"
```

definition

```
set_eq :: "'a set ⇒ 'a set ⇒ bool" (infixl "{.=}ι" 50)
where "A {.=}S B ↔ ((∀x ∈ A. x .∈S B) ∧ (∀x ∈ B. x .∈S A))"
```

definition

```
eq_class_of :: "'a ⇒ 'a ⇒ 'a set" ("class'_ofι")
where "class_ofS x = {y ∈ carrier S. x .=S y}"
```

definition

```
eq_closure_of :: "'a set ⇒ 'a set" ("closure'_ofι")
where "closure_ofS A = {y ∈ carrier S. y .∈S A}"
```

definition

```
eq_is_closed :: "'a set ⇒ bool" ("is'_closedι")
where "is_closedS A ↔ A ⊆ carrier S ∧ closure_ofS A = A"
```

abbreviation

```
not_eq :: "'a ⇒ 'a ⇒ bool" (infixl ".≠ι" 50)
where "x .≠S y == ~(x .=S y)"
```

abbreviation

```
not_elem :: "'a ⇒ 'a set ⇒ bool" (infixl ".∉ι" 50)
where "x .∉S A == ~(x .∈S A)"
```

abbreviation

```
set_not_eq :: "'a set ⇒ 'a set ⇒ bool" (infixl "{.≠}ι" 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 elemI:
  fixes R (structure)
    assumes "a' ∈ A" and "a .= a'"
    shows "a ∈ A"
  <proof>

```

```

lemma (in equivalence) elem_exact:
  assumes "a ∈ carrier S" and "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 cong: "a' .= a"
    and a: "a ∈ A"
    and carr: "a ∈ carrier S" "a' ∈ carrier S"
    and Acarr: "A ⊆ carrier S"
  shows "a' ∈ A"
  <proof>

```

```

lemma (in equivalence) elem_subsetD:
  assumes "A ⊆ B"
    and aA: "a ∈ A"
  shows "a ∈ B"
  <proof>

```

```

lemma (in equivalence) mem_imp_elem [simp, intro]:
  "[ | x ∈ A; x ∈ carrier S | ] ==> x ∈ A"
  <proof>

```

```

lemma set_eqI:

```

```

fixes R (structure)
assumes ltr: " $\bigwedge a. a \in A \implies a \cdot \in B$ "
and rtl: " $\bigwedge b. b \in B \implies b \cdot \in A$ "
shows "A  $\{.\} B$ "
<proof>

```

```

lemma set_eqI2:
fixes R (structure)
assumes ltr: " $\bigwedge a b. a \in A \implies \exists b \in B. a \cdot = b$ "
and rtl: " $\bigwedge b. b \in B \implies \exists a \in A. b \cdot = a$ "
shows "A  $\{.\} B$ "
<proof>

```

```

lemma set_eqD1:
fixes R (structure)
assumes AA': "A  $\{.\} A'$ "
and "a  $\in A$ "
shows " $\exists a' \in A'. a \cdot = a'$ "
<proof>

```

```

lemma set_eqD2:
fixes R (structure)
assumes AA': "A  $\{.\} A'$ "
and "a'  $\in A'$ "
shows " $\exists a \in A. a' \cdot = a$ "
<proof>

```

```

lemma set_eqE:
fixes R (structure)
assumes AB: "A  $\{.\} B$ "
and r: " $\llbracket \forall a \in A. a \cdot \in B; \forall b \in B. b \cdot \in A \rrbracket \implies P$ "
shows "P"
<proof>

```

```

lemma set_eqE2:
fixes R (structure)
assumes AB: "A  $\{.\} B$ "
and r: " $\llbracket \forall a \in A. (\exists b \in B. a \cdot = b); \forall b \in B. (\exists a \in A. b \cdot = a) \rrbracket \implies P$ "
shows "P"
<proof>

```

```

lemma set_eqE':
fixes R (structure)
assumes AB: "A  $\{.\} B$ "
and aA: "a  $\in A$ " and bB: "b  $\in B$ "
and r: " $\llbracket a' \cdot b'. \llbracket a' \in A; b \cdot = a'; b' \in B; a \cdot = b' \rrbracket \implies P$ "
shows "P"
<proof>

```

```

lemma (in equivalence) eq_elem_cong_r [trans]:
  assumes a: "a .∈ A"
    and cong: "A {.=} A'"
    and carr: "a ∈ carrier S"
    and Carr: "A ⊆ carrier S" "A' ⊆ carrier S"
  shows "a .∈ A'"
<proof>

```

```

lemma (in equivalence) set_eq_sym [sym]:
  assumes "A {.=} B"
    and "A ⊆ carrier S" "B ⊆ carrier S"
  shows "B {.=} A"
<proof>

```

```

lemma (in equivalence) equal_set_eq_trans [trans]:
  assumes AB: "A = B" and BC: "B {.=} C"
  shows "A {.=} C"
<proof>

```

```

lemma (in equivalence) set_eq_equal_trans [trans]:
  assumes AB: "A {.=} B" and BC: "B = C"
  shows "A {.=} C"
<proof>

```

```

lemma (in equivalence) set_eq_trans [trans]:
  assumes AB: "A {.=} B" and BC: "B {.=} C"
    and carr: "A ⊆ carrier S" "B ⊆ carrier S" "C ⊆ carrier S"
  shows "A {.=} C"
<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) 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:
  "[| x .= x'; A  $\subseteq$  carrier S; x  $\in$  closure_of A; x  $\in$  carrier S; x'  $\in$  carrier
  S |] ==> x'  $\in$  closure_of A"
  <proof>
```

```
lemma (in equivalence) is_closed_eq [dest]:
  "[| x .= x'; x  $\in$  A; is_closed A; x  $\in$  carrier S; x'  $\in$  carrier S |] ==>
  x'  $\in$  A"
  <proof>
```

```
lemma (in equivalence) is_closed_eq_rev [dest]:
  "[| x .= x'; x'  $\in$  A; is_closed A; x  $\in$  carrier S; x'  $\in$  carrier S |]
  ==> x  $\in$  A"
  <proof>
```

```
lemma closure_of_closed [simp, intro]:
  fixes S (structure)
  shows "closure_of A  $\subseteq$  carrier S"
  <proof>
```

```
lemma closure_of_memI:
  fixes S (structure)
  assumes "a  $\in$  A"
  and "a  $\in$  carrier S"
  shows "a  $\in$  closure_of A"
  <proof>
```

```
lemma closure_ofI2:
  fixes S (structure)
  assumes "a .= a'"
  and "a'  $\in$  A"
  and "a  $\in$  carrier S"
  shows "a  $\in$  closure_of A"
  <proof>
```

```
lemma closure_of_memE:
  fixes S (structure)
  assumes p: "a  $\in$  closure_of A"
  and r: "[| a  $\in$  carrier S; a  $\in$  A |]  $\implies$  P"
  shows "P"
  <proof>
```

```
lemma closure_ofE2:
  fixes S (structure)
  assumes p: "a  $\in$  closure_of A"
  and r: " $\bigwedge$ a'. [| a  $\in$  carrier S; a'  $\in$  A; a .= a' |]  $\implies$  P"
  shows "P"
```

<proof>

end

```
theory Lattice
imports Congruence
begin
```

2 Orders and Lattices

2.1 Partial Orders

```
record 'a gorder = "'a eq_object" +
  le :: "'a, 'a] => bool" (infixl "⊆" 50)

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 "⊆L" 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 |] ==>
```

$x \sqsubseteq z$
<proof>

lemma weak_refl [intro, simp]: " $\llbracket x .= y; x \in \text{carrier } L; y \in \text{carrier } L \rrbracket \implies x \sqsubseteq y$ "
<proof>

end

lemma weak_llessI:
 fixes R (structure)
 assumes " $x \sqsubseteq y$ " and " $\neg(x .= y)$ "
 shows " $x \sqsubset y$ "
<proof>

lemma lless_imp_le:
 fixes R (structure)
 assumes " $x \sqsubset y$ "
 shows " $x \sqsubseteq y$ "
<proof>

lemma weak_lless_imp_not_eq:
 fixes R (structure)
 assumes " $x \sqsubset y$ "
 shows " $\neg(x .= y)$ "
<proof>

lemma weak_llessE:
 fixes R (structure)
 assumes p: " $x \sqsubset y$ " and e: " $\llbracket x \sqsubseteq y; \neg(x .= y) \rrbracket \implies P$ "
 shows "P"
<proof>

lemma (in weak_partial_order) lless_cong_l [trans]:
 assumes xx': " $x .= x'$ "
 and xy: " $x' \sqsubset y$ "
 and carr: " $x \in \text{carrier } L$ " " $x' \in \text{carrier } L$ " " $y \in \text{carrier } L$ "
 shows " $x \sqsubset y$ "
<proof>

lemma (in weak_partial_order) lless_cong_r [trans]:
 assumes xy: " $x \sqsubset y$ "
 and yy': " $y .= y'$ "
 and carr: " $x \in \text{carrier } L$ " " $y \in \text{carrier } L$ " " $y' \in \text{carrier } L$ "
 shows " $x \sqsubset y'$ "
<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⟩

```

2.1.2 Upper and lower bounds of a set

definition

```

Upper :: "[_, 'a set] => 'a set"
where "Upper L A = {u. (ALL x. x ∈ A ∩ carrier L --> x ⊆L u)} ∩ carrier L"

```

definition

```

Lower :: "[_, 'a set] => 'a set"
where "Lower L A = {l. (ALL x. x ∈ A ∩ carrier L --> l ⊆L x)} ∩ carrier L"

```

```

lemma Upper_closed [intro!, simp]:
  "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'carr: "a' ∈ carrier L" and Acarr: "A ⊆ carrier L"
    and aa': "a .= a'"
    and aelem: "a ∈ Upper L A"
  shows "a' ∈ Upper L A"
  ⟨proof⟩

lemma (in weak_partial_order) Upper_cong:
  assumes Acarr: "A ⊆ carrier L" and A'carr: "A' ⊆ carrier L"
    and AA': "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'carr: "a' ∈ carrier L" and Acarr: "A ⊆ carrier L"
    and aa': "a .= a'"
    and aelem: "a ∈ Lower L A"
  shows "a' ∈ Lower L A"

```

<proof>

```
lemma (in weak_partial_order) Lower_cong:
  assumes Acarr: "A  $\subseteq$  carrier L" and A'carr: "A'  $\subseteq$  carrier L"
    and AA': "A  $\{.\} A'$ "
  shows "Lower L A = Lower L A'"
<proof>
```

2.1.3 Least and greatest, as predicate

definition

```
least :: "[_, 'a, 'a set] => bool"
where "least L l A  $\longleftrightarrow$  A  $\subseteq$  carrier L & l  $\in$  A & (ALL x : A. l  $\sqsubseteq_L$  x)"
```

definition

```
greatest :: "[_, 'a, 'a set] => bool"
where "greatest L g A  $\longleftrightarrow$  A  $\subseteq$  carrier L & g  $\in$  A & (ALL x : A. x  $\sqsubseteq_L$  g)"
```

Could weaken these to $l \in \text{carrier } L \wedge l \in A$ and $g \in \text{carrier } L \wedge g \in A$.

```
lemma least_closed [intro, simp]:
  "least L l A  $\implies$  l  $\in$  carrier L"
<proof>
```

```
lemma least_mem:
  "least L l A  $\implies$  l  $\in$  A"
<proof>
```

```
lemma (in weak_partial_order) weak_least_unique:
  "[| least L x A; least L y A |]  $\implies$  x  $\{.\}$  y"
<proof>
```

```
lemma least_le:
  fixes L (structure)
  shows "[| least L x A; a  $\in$  A |]  $\implies$  x  $\sqsubseteq$  a"
<proof>
```

```
lemma (in weak_partial_order) least_cong:
  "[| x  $\{.\}$  x'; x  $\in$  carrier L; x'  $\in$  carrier L; is_closed A |]  $\implies$  least
L x A = least L x' A"
<proof>
```

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  $\in$  carrier L" "x'  $\in$  carrier L"
    and "A  $\subseteq$  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 Acarrs: "A  $\subseteq$  carrier L" "A'  $\subseteq$  carrier L"
    and AA': "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  $\in$  A ==> x  $\sqsubseteq$  s"
    and below: "!! y. y  $\in$  Upper L A ==> s  $\sqsubseteq$  y"
    and L: "A  $\subseteq$  carrier L" "s  $\in$  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  $\in$  A; A  $\subseteq$  carrier L |] ==> x  $\sqsubseteq$  s"
<proof>
```

```
lemma greatest_closed [intro, simp]:
  "greatest L l A ==> l  $\in$  carrier L"
<proof>
```

```
lemma greatest_mem:
  "greatest L l A ==> l  $\in$  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  $\in$  A |] ==> a  $\sqsubseteq$  x"
<proof>
```

```
lemma (in weak_partial_order) greatest_cong:
  "[| x  $\{.\} x'$ ; x  $\in$  carrier L; x'  $\in$  carrier L; is_closed A |] ==>
  greatest L x A = greatest L x' A"
<proof>
```

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  $\in$  carrier L" "x'  $\in$  carrier L"
    and "A  $\subseteq$  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 Acarrs: "A ⊆ carrier L" "A' ⊆ carrier L"
    and AA': "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>
```

Supremum and infimum

```
definition
  sup :: "[_, 'a set] => 'a" ("⊔" [90] 90)
  where "⊔LA = (SOME x. least L x (Upper L A))"
```

```
definition
  inf :: "[_, 'a set] => 'a" ("⊓" [90] 90)
  where "⊓LA = (SOME x. greatest L x (Lower L A))"
```

```
definition
  join :: "[_, 'a, 'a] => 'a" (infixl "⊔" 65)
  where "x ⊔L y = ⊔L{x, y}"
```

```
definition
  meet :: "[_, 'a, 'a] => 'a" (infixl "⊓" 70)
  where "x ⊓L y = ⊓L{x, y}"
```

2.2 Lattices

```
locale weak_upper_semilattice = weak_partial_order +
  assumes sup_of_two_exists:
    "[| x ∈ carrier L; y ∈ carrier L |] ==> EX 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 |] ==> EX s. greatest L s (Lower
```

L {x, y})"

locale weak_lattice = weak_upper_semilattice + weak_lower_semilattice

2.2.1 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_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>

2.2.2 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_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>

```

2.3 Total Orders

```

local 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>

```

Total orders are lattices.

```

sublocale weak_total_order < weak: weak_lattice
<proof>

```

2.4 Complete Lattices

```

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

```

Introduction rule: the usual definition of complete lattice

```

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

```

definition

```

top :: "_ => 'a" ("⊤")
where "⊤L = sup L (carrier L)"

```

definition

```

bottom :: "_ => 'a" ("⊥")
where "⊥L = inf L (carrier L)"

```

```

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) top_closed [simp, intro]:
  "⊤ ∈ carrier L"
  <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) bottom_closed [simp, intro]:
  "⊥ ∈ carrier L"
⟨proof⟩

```

Jacobson: Theorem 8.1

```

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

```

```

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

```

```

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

```

2.5 Orders and Lattices where eq is the Equality

```

locale partial_order = weak_partial_order +
  assumes eq_is_equal: "op .= = op ="
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 lless_asym:
  assumes "a ∈ carrier L" "b ∈ carrier L"
  and "a ⊆ b" "b ⊆ a"

```

```

shows "P"
  ⟨proof⟩

end

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⟩

Lattices

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

sublocale lower_semilattice < weak: weak_lower_semilattice
  ⟨proof⟩

locale lattice = upper_semilattice + lower_semilattice

Supremum

declare (in partial_order) weak_sup_of_singleton [simp del]

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

lemma (in upper_semilattice) join_assoc_lemma:
  assumes L: "x ∈ carrier L" "y ∈ carrier L" "z ∈ carrier L"
  shows "x  $\sqcup$  (y  $\sqcup$  z) =  $\sqcup$ {x, y, z}"
  ⟨proof⟩

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

```

Infimum

declare (in partial_order) weak_inf_of_singleton [simp del]

lemma (in partial_order) inf_of_singleton [simp]:
 "x ∈ carrier L ==> $\bigcap \{x\} = x$ "
 ⟨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 \sqcap (y \sqcap z) = $\bigcap \{x, y, z\}$ "
 ⟨proof⟩

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

Total Orders

locale total_order = partial_order +
 assumes total_order_total: "[| x ∈ carrier L; y ∈ carrier L |] ==>
 x \sqsubseteq y | y \sqsubseteq 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 \sqsubseteq
 y | y \sqsubseteq x"
 shows "total_order L"
 ⟨proof⟩

Total orders are lattices.

sublocale total_order < weak: lattice
 ⟨proof⟩

Complete lattices

locale complete_lattice = lattice +
 assumes sup_exists:
 "[| A \subseteq carrier L |] ==> EX s. least L s (Upper L A)"
 and inf_exists:
 "[| A \subseteq carrier L |] ==> EX i. greatest L i (Lower L A)"

sublocale complete_lattice < weak: weak_complete_lattice
 ⟨proof⟩

Introduction rule: the usual definition of complete lattice

```

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

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

```

2.6 Examples

2.6.1 The Powerset of a Set is a Complete Lattice

```

theorem powerset_is_complete_lattice:
  "complete_lattice (| carrier = Pow A, eq = op =, le = op ⊆ |)"
  (is "complete_lattice ?L")
  <proof>

```

Another example, that of the lattice of subgroups of a group, can be found in Group theory (Section 3.8).

end

```

theory Group
imports Lattice "~/src/HOL/Library/FuncSet"
begin

```

3 Monoids and Groups

3.1 Definitions

Definitions follow [2].

```

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

```

definition

```

m_inv :: "('a, 'b) monoid_scheme => 'a => 'a" ("inv" [81] 80)
where "invG x = (THE y. y ∈ carrier G & x ⊗G y = 1G & y ⊗G x = 1G>)"

```

definition

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

consts

```
pow :: "[('a, 'm) monoid_scheme, 'a, 'b::semiring_1] => 'a" (infixr
"^(^)" 75)
```

```
overloading nat_pow == "pow :: [_, 'a, nat] => 'a"
```

```
begin
```

```
  definition "nat_pow G a n = nat_rec 1G (%u b. b ⊗G a) n"
```

```
end
```

```
overloading int_pow == "pow :: [_, 'a, int] => 'a"
```

```
begin
```

```
  definition "int_pow G a z =
```

```
    (let p = nat_rec 1G (%u b. b ⊗G a)
```

```
      in if z < 0 then invG (p (nat (-z))) else p (nat z))"
```

```
end
```

```
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) inv_unique:
 assumes eq: " $y \otimes x = 1$ " " $x \otimes y' = 1$ "
 and G: " $x \in \text{carrier } G$ " " $y \in \text{carrier } G$ " " $y' \in \text{carrier } G$ "
 shows " $y = y'$ "
<proof>

lemma (in monoid) Units_m_closed [intro, simp]:
 assumes x: " $x \in \text{Units } G$ " and y: " $y \in \text{Units } G$ "
 shows " $x \otimes y \in \text{Units } G$ "
<proof>

lemma (in monoid) Units_one_closed [intro, simp]:
 " $1 \in \text{Units } G$ "
<proof>

lemma (in monoid) Units_inv_closed [intro, simp]:
 " $x \in \text{Units } G \implies \text{inv } x \in \text{carrier } G$ "
<proof>

lemma (in monoid) Units_l_inv_ex:
 " $x \in \text{Units } G \implies \exists y \in \text{carrier } G. y \otimes x = 1$ "
<proof>

lemma (in monoid) Units_r_inv_ex:
 " $x \in \text{Units } G \implies \exists y \in \text{carrier } G. x \otimes y = 1$ "
<proof>

lemma (in monoid) Units_l_inv [simp]:
 " $x \in \text{Units } G \implies \text{inv } x \otimes x = 1$ "
<proof>

lemma (in monoid) Units_r_inv [simp]:
 " $x \in \text{Units } G \implies x \otimes \text{inv } x = 1$ "
<proof>

lemma (in monoid) Units_inv_Units [intro, simp]:
 " $x \in \text{Units } G \implies \text{inv } x \in \text{Units } G$ "
<proof>

lemma (in monoid) Units_l_cancel [simp]:
 " $[| x \in \text{Units } G; y \in \text{carrier } G; z \in \text{carrier } G |] \implies$
 $(x \otimes y = x \otimes z) = (y = z)$ "
<proof>

lemma (in monoid) Units_inv_inv [simp]:
 " $x \in \text{Units } G \implies \text{inv } (\text{inv } x) = x$ "
<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  $\otimes$  y = 1"
  and G: "x  $\in$  Units G" "y  $\in$  Units G"
  shows "y  $\otimes$  x = 1"
  <proof>
```

Power

```
lemma (in monoid) nat_pow_closed [intro, simp]:
  "x  $\in$  carrier G ==> x ( $\wedge$ ) (n::nat)  $\in$  carrier G"
  <proof>
```

```
lemma (in monoid) nat_pow_0 [simp]:
  "x ( $\wedge$ ) (0::nat) = 1"
  <proof>
```

```
lemma (in monoid) nat_pow_Suc [simp]:
  "x ( $\wedge$ ) (Suc n) = x ( $\wedge$ ) n  $\otimes$  x"
  <proof>
```

```
lemma (in monoid) nat_pow_one [simp]:
  "1 ( $\wedge$ ) (n::nat) = 1"
  <proof>
```

```
lemma (in monoid) nat_pow_mult:
  "x  $\in$  carrier G ==> x ( $\wedge$ ) (n::nat)  $\otimes$  x ( $\wedge$ ) m = x ( $\wedge$ ) (n + m)"
  <proof>
```

```
lemma (in monoid) nat_pow_pow:
  "x  $\in$  carrier G ==> (x ( $\wedge$ ) n) ( $\wedge$ ) m = x ( $\wedge$ ) (n * m::nat)"
  <proof>
```

3.2 Groups

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

```
locale group = monoid +
  assumes Units: "carrier G  $\leq$  Units G"
```

```
lemma (in group) is_group: "group G" <proof>
```

```
theorem groupI:
  fixes G (structure)
  assumes m_closed [simp]:
    "!!x y. [| x  $\in$  carrier G; y  $\in$  carrier G |] ==> x  $\otimes$  y  $\in$  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⟩

```

3.3 Cancellation Laws and Basic Properties

```

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

```

```

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

```

```

lemma (in group) r_cancel [simp]:
  "[| x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |] ==>
  (y ⊗ x = z ⊗ x) = (y = z)"

```

<proof>

lemma (in group) inv_one [simp]:

"inv 1 = 1"

<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>

Power

lemma (in group) int_pow_def2:

"a (^) (z::int) = (if z < 0 then inv (a (^) (nat (-z))) else a (^) (nat z))"

<proof>

lemma (in group) int_pow_0 [simp]:

"x (^) (0::int) = 1"

<proof>

lemma (in group) int_pow_one [simp]:

"1 (^) (z::int) = 1"

<proof>

3.4 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 subgroup_imp_subset:
  "subgroup H G ⇒ H ⊆ carrier G"
  <proof>
```

```
lemma (in subgroup) subgroup_is_group [intro]:
  assumes "group G"
  shows "group (G(carrier := H))"
  <proof>
```

Since H is nonempty, it contains some element x . Since it is closed under inverse, it contains $\text{inv } x$. Since it is closed under product, it contains $x \otimes \text{inv } x = 1$.

```
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>
```

```
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>
```

3.5 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 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>

```

3.6 Homomorphisms 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 (in group) hom_compose:
  "[|h ∈ hom G H; i ∈ hom H I|] ==> compose (carrier G) i h ∈ hom G I"
<proof>

```

definition

```
iso :: "_ => _ => ('a => 'b) set" (infixr "≅" 60)
where "G ≅ H = {h. h ∈ hom G H & bij_betw h (carrier G) (carrier H)}"
```

```
lemma iso_refl: "(%x. x) ∈ G ≅ G"
⟨proof⟩
```

```
lemma (in group) iso_sym:
  "h ∈ G ≅ H ==> inv_into (carrier G) h ∈ H ≅ G"
⟨proof⟩
```

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

```
lemma DirProd_commute_iso:
  shows "(λ(x,y). (y,x)) ∈ (G ×× H) ≅ (H ×× G)"
⟨proof⟩
```

```
lemma DirProd_assoc_iso:
  shows "(λ(x,y,z). (x,(y,z))) ∈ (G ×× H ×× I) ≅ (G ×× (H ×× I))"
⟨proof⟩
```

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: "h ∈ hom G H"
```

```
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 [simp]:
  "h 1 ∈ carrier H"
⟨proof⟩
```

```
lemma (in group_hom) hom_one [simp]:
  "h 1 = 1H"
⟨proof⟩
```

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

```

lemma (in group_hom) hom_inv [simp]:
  "x ∈ carrier G ==> h (inv x) = invH (h x)"
  <proof>

```

3.7 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) nat_pow_distr:
  "[| x ∈ carrier G; y ∈ carrier G |] ==>
  (x ⊗ y) (^) (n::nat) = x (^) n ⊗ y (^) n"
  <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 (in comm_group) inv_mult:
  "[| x ∈ carrier G; y ∈ carrier G |] ==> inv (x ⊗ y) = inv x ⊗ inv y"
  <proof>

```

3.8 The Lattice of Subgroups of a Group

```

theorem (in group) subgroups_partial_order:
  "partial_order (| carrier = {H. subgroup H G}, eq = op =, le = op ⊆
|)"
  <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) is_monoid [intro, simp]:
  "monoid G"
  <proof>

```

```

lemma (in group) subgroup_inv_equality:
  "[| subgroup H G; x ∈ H |] ==> m_inv (G (| carrier := H |)) x = inv
x"
  <proof>

```

```

theorem (in group) subgroups_Inter:
  assumes subgr: "(!!H. H ∈ A ==> subgroup H G)"
    and not_empty: "A ≠ {}"
  shows "subgroup (⋂ A) G"
  <proof>

theorem (in group) subgroups_complete_lattice:
  "complete_lattice (| carrier = {H. subgroup H G}, eq = op =, le = op
  ⊆ |)"
  (is "complete_lattice ?L")
  <proof>

end

```

```

theory FiniteProduct
imports Group
begin

```

3.9 Product Operator for Commutative Monoids

3.9.1 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 $x \in \text{carrier } G$. 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 ; e ∈ D; !!x y. [| x ∈ A; y ∈ D |] ==>
  f x y ∈ D
  |] ==> 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 |] ==>
  EX x. (A, x) ∈ foldSetD D f e"
⟨proof⟩
```

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) foldSetD_imp_finite [simp]:
  "(A, x) ∈ foldSetD D f e ==> finite A"
⟨proof⟩
```

```
lemma (in LCD) finite_imp_foldSetD:
  "[| finite A; A ⊆ B; e ∈ D |] ==> EX x. (A, x) ∈ foldSetD D f e"
⟨proof⟩
```

```
lemma (in LCD) foldSetD_determ_aux:
  "e ∈ D ==> ∀A x. A ⊆ B & card A < n --> (A, x) ∈ foldSetD D f e -->
  (∀y. (A, y) ∈ foldSetD D f e --> 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) =
 (EX y. (A, y) ∈ foldSetD D f e & v = f x y)"
<proof>

lemma (in LCD) foldD_insert:
 "[| finite A; x ~: A; x ∈ B; e ∈ D; A ⊆ B |] ==>
 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 $f :: 'a \Rightarrow 'a \Rightarrow 'a$ instead of $'b \Rightarrow 'a \Rightarrow 'a$.

```
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>
```

3.9.2 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 o f) 1G A
   else undefined)"
```

syntax

```
"_finprod" :: "index => idt => 'a set => 'b => 'b"
  ("(3⊗ __:_. _)" [1000, 0, 51, 10] 10)
syntax (xsymbols)
```

```

    "_finprod" :: "index => idt => 'a set => 'b => 'b"
                ("( $\bigotimes_{i \in I} \cdot$ )" [1000, 0, 51, 10] 10)
syntax (HTML output)
    "_finprod" :: "index => idt => 'a set => 'b => 'b"
                ("( $\bigotimes_{i \in I} \cdot$ )" [1000, 0, 51, 10] 10)
translations
    " $\bigotimes_{i:A} b$ " == "CONST finprod  $\diamond z$  (%i. b) A"
    — Beware of argument permutation!

lemma (in comm_monoid) finprod_empty [simp]:
    "finprod G f {} = 1"
    <proof>

declare funcsetI [intro]
    funcset_mem [dest]

context comm_monoid begin

lemma finprod_insert [simp]:
    "[| finite F; a  $\notin$  F; f  $\in$  F  $\rightarrow$  carrier G; f a  $\in$  carrier G |] ==>
    finprod G f (insert a F) = f a  $\otimes$  finprod G f F"
    <proof>

lemma finprod_one [simp]:
    "finite A ==> ( $\bigotimes_{i:A} 1$ ) = 1"
    <proof>

lemma finprod_closed [simp]:
    fixes A
    assumes fin: "finite A" and f: "f  $\in$  A  $\rightarrow$  carrier G"
    shows "finprod G f A  $\in$  carrier G"
    <proof>

lemma funcset_Int_left [simp, intro]:
    "[| f  $\in$  A  $\rightarrow$  C; f  $\in$  B  $\rightarrow$  C |] ==> f  $\in$  A Int B  $\rightarrow$  C"
    <proof>

lemma funcset_Un_left [iff]:
    "(f  $\in$  A Un B  $\rightarrow$  C) = (f  $\in$  A  $\rightarrow$  C & f  $\in$  B  $\rightarrow$  C)"
    <proof>

lemma finprod_Un_Int:
    "[| finite A; finite B; g  $\in$  A  $\rightarrow$  carrier G; g  $\in$  B  $\rightarrow$  carrier G |] ==>
    finprod G g (A Un B)  $\otimes$  finprod G g (A Int B) =
    finprod G g A  $\otimes$  finprod G g B"
    — The reversed orientation looks more natural, but LOOPS as a simp rule!
    <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:
"[] finite A; f ∈ A -> carrier G; g ∈ A -> carrier G [] ==>
  finprod G (%x. 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  $g \in B \rightarrow \text{carrier } G$  cannot be shown. Adding Pi_def to the
simpset is often useful. For this reason, comm_monoid.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_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_mult [simp]:
"[] f ∈ {...n} -> carrier G; g ∈ {...n} -> carrier G [] ==>

```

```

    finprod G (%i. f i ⊗ g i) {..n::nat} =
    finprod G f {..n} ⊗ finprod G g {..n}"
  ⟨proof⟩

```

```

lemma finprod_reindex:
  assumes fin: "finite A"
  shows "f : (h ' A) → carrier G ==>
        inj_on h A ==> finprod G f (h ' A) = finprod G (%x. f (h x)) A"
  ⟨proof⟩

```

```

lemma finprod_const:
  assumes fin [simp]: "finite A"
  and a [simp]: "a : carrier G"
  shows "finprod G (%x. 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⟩

```

end

end

```

theory Coset
imports Group
begin

```

4 Cosets and Quotient Groups

```

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

```

```

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

```

```

definition
  RCOSSETS  :: "[_, 'a set] ⇒ ('a set)set"    ("rcosetsₐ _" [81] 80)
  where "rcosetsₐ H = (⋃a∈carrier G. {H #>ₐ a})"

```

definition

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

definition

```
SET_INV :: "[_, 'a set] ⇒ 'a set" ("set'_invι _" [81] 80)
  where "set_invG H = (⋃h∈H. {invG 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"
```

4.1 Basic Properties of Cosets

lemma (in group) coset_mult_assoc:

```
"[| M ⊆ carrier G; g ∈ carrier G; h ∈ carrier G |]
  ==> (M #> g) #> h = M #> (g ⊗ h)"
```

<proof>

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

<proof>

lemma (in group) coset_mult_inv1:

```
"[| M #> (x ⊗ (inv y)) = M; x ∈ carrier G ; y ∈ carrier G;
  M ⊆ carrier G |] ==> M #> x = M #> y"
```

<proof>

lemma (in group) coset_mult_inv2:

```
"[| M #> x = M #> y; x ∈ carrier G; y ∈ carrier G; M ⊆ carrier
G |]
```

```
  ==> M #> (x ⊗ (inv y)) = M "
```

<proof>

lemma (in group) coset_join1:

```
"[| H #> x = H; x ∈ carrier G; subgroup H G |] ==> x ∈ H"
```

<proof>

lemma (in group) solve_equation:

```
"[subgroup H G; x ∈ H; y ∈ H] ==> ∃h∈H. y = h ⊗ x"
```

<proof>

lemma (in group) repr_independence:

" $[y \in H \#> x; x \in \text{carrier } G; \text{subgroup } H \ G] \implies H \#> x = H \#> y$ "
 <proof>

lemma (in group) coset_join2:

" $[x \in \text{carrier } G; \text{subgroup } H \ G; x \in H] \implies H \#> x = H$ "

— Alternative proof is to put $x = 1$ in repr_independence.

<proof>

lemma (in monoid) r_coset_subset_G:

" $[H \subseteq \text{carrier } G; x \in \text{carrier } G] \implies H \#> x \subseteq \text{carrier } G$ "

<proof>

lemma (in group) rcosI:

" $[h \in H; H \subseteq \text{carrier } G; x \in \text{carrier } G] \implies h \otimes x \in H \#> x$ "

<proof>

lemma (in group) rcosetsI:

" $[H \subseteq \text{carrier } G; x \in \text{carrier } G] \implies H \#> x \in \text{rcosets } H$ "

<proof>

Really needed?

lemma (in group) transpose_inv:

" $[x \otimes y = z; x \in \text{carrier } G; y \in \text{carrier } G; z \in \text{carrier } G] \implies (\text{inv } x) \otimes z = y$ "

<proof>

lemma (in group) rcos_self: " $[x \in \text{carrier } G; \text{subgroup } H \ G] \implies x \in H \#> x$ "

<proof>

Opposite of "repr_independence"

lemma (in group) repr_independenceD:

assumes "subgroup H G"

assumes ycarr: "y ∈ carrier G"

and repr: "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"

assumes acarr: "a ∈ carrier G"

and a': "a' ∈ H #> a"

shows "a' ∈ carrier G"

<proof>

lemma (in subgroup) rcos_const:

assumes "group G"

assumes hH: "h ∈ H"

shows "H #> h = H"
<proof>

Step one for lemma rcos_module

lemma (in subgroup) rcos_module_imp:
 assumes "group G"
 assumes xcarr: "x ∈ carrier G"
 and x'cos: "x' ∈ H #> x"
 shows "(x' ⊗ inv x) ∈ H"
<proof>

Step two for lemma rcos_module

lemma (in subgroup) rcos_module_rev:
 assumes "group G"
 assumes carr: "x ∈ carrier G" "x' ∈ carrier G"
 and xixH: "(x' ⊗ inv x) ∈ H"
 shows "x' ∈ H #> x"
<proof>

Module property of right cosets

lemma (in subgroup) rcos_module:
 assumes "group G"
 assumes carr: "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"
 assumes XH: "X ∈ rcosets H"
 shows "X ⊆ carrier G"
<proof>

Multiplication of general subsets

lemma (in monoid) set_mult_closed:
 assumes Acarr: "A ⊆ carrier G"
 and Bcarr: "B ⊆ carrier G"
 shows "A <#> B ⊆ carrier G"
<proof>

lemma (in comm_group) mult_subgroups:
 assumes subH: "subgroup H G"
 and subK: "subgroup K G"
 shows "subgroup (H <#> K) G"
<proof>

lemma (in subgroup) lcos_module_rev:
 assumes "group G"

```

assumes carr: "x ∈ carrier G" "x' ∈ carrier G"
and xixH: "(inv x ⊗ x') ∈ H"
shows "x' ∈ x <# H"
⟨proof⟩

```

4.2 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:
  "[x ∈ carrier G; h ∈ H] ⇒ (inv x) ⊗ h ⊗ x ∈ H"
⟨proof⟩

```

```

lemma (in normal) inv_op_closed2:
  "[x ∈ carrier G; h ∈ H] ⇒ x ⊗ h ⊗ (inv x) ∈ H"
⟨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⟩

```

4.3 More Properties of Cosets

```

lemma (in group) lcos_m_assoc:
  "[| M ⊆ carrier G; g ∈ carrier G; h ∈ carrier G |]
  ==> g <# (h <# M) = (g ⊗ h) <# M"
⟨proof⟩

```

```

lemma (in group) lcos_mult_one: "M ⊆ carrier G ==> 1 <# M = M"
⟨proof⟩

```

```

lemma (in group) l_coset_subset_G:
  "[| H ⊆ carrier G; x ∈ carrier G |] ==> x <# H ⊆ carrier G"
⟨proof⟩

```

```

lemma (in group) l_coset_swap:
  "[| y ∈ x <# H; x ∈ carrier G; subgroup H G |] ==> x ∈ y <# H"
⟨proof⟩

```

```

lemma (in group) l_coset_carrier:
  "[| y ∈ x <# H; x ∈ carrier G; subgroup H G |] ==> y ∈ carrier G"

```

<proof>

lemma (in group) l_repr_imp_subset:
 assumes y: "y ∈ x <# H" and x: "x ∈ carrier G" and sb: "subgroup H G"
 shows "y <# H ⊆ x <# H"
<proof>

lemma (in group) l_repr_independence:
 assumes y: "y ∈ x <# H" and x: "x ∈ carrier G" and sb: "subgroup H G"
 shows "x <# H = y <# H"
<proof>

lemma (in group) setmult_subset_G:
 "[H ⊆ carrier G; K ⊆ carrier G] ⇒ H <#> K ⊆ carrier G"
<proof>

lemma (in group) subgroup_mult_id: "subgroup H G ⇒ H <#> H = H"
<proof>

4.3.1 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>

4.3.2 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⟩

```

4.3.3 An Equivalence Relation

definition

```

r_congruent :: "[('a,'b)monoid_scheme, 'a set] ⇒ ('a*'a)set" ("rcongG
_")
where "rcongG H = {(x,y). x ∈ carrier G & y ∈ carrier G & invG 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⟩

```

4.3.4 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"
  assumes p: "a ∈ rcosets H" "b ∈ rcosets H" "a≠b"
  shows "a ∩ b = {}"
⟨proof⟩

```

4.4 Further lemmas for `r_congruent`

The relation is a congruence

```
lemma (in normal) congruent_rcong:
  shows "congruent2 (rcong H) (rcong H) ( $\lambda$ a b. a  $\otimes$  b  $<\#$  H)"
  <proof>
```

4.5 Order of a Group and Lagrange's Theorem

definition

```
order :: "('a, 'b) monoid_scheme  $\Rightarrow$  nat"
where "order S = card (carrier S)"
```

```
lemma (in group) rcosets_part_G:
  assumes "subgroup H G"
  shows " $\bigcup$  (rcosets H) = carrier G"
  <proof>
```

```
lemma (in group) cosets_finite:
  "[c  $\in$  rcosets H; H  $\subseteq$  carrier G; finite (carrier G)]  $\Longrightarrow$  finite c"
  <proof>
```

The next two lemmas support the proof of `card_cosets_equal`.

```
lemma (in group) inj_on_f:
  "[H  $\subseteq$  carrier G; a  $\in$  carrier G]  $\Longrightarrow$  inj_on ( $\lambda$ y. y  $\otimes$  inv a) (H  $\#$ > a)"
  <proof>
```

```
lemma (in group) inj_on_g:
  "[H  $\subseteq$  carrier G; a  $\in$  carrier G]  $\Longrightarrow$  inj_on ( $\lambda$ y. y  $\otimes$  a) H"
  <proof>
```

```
lemma (in group) card_cosets_equal:
  "[c  $\in$  rcosets H; H  $\subseteq$  carrier G; finite(carrier G)]
   $\Longrightarrow$  card c = card H"
  <proof>
```

```
lemma (in group) rcosets_subset_PowG:
  "subgroup H G  $\Longrightarrow$  rcosets H  $\subseteq$  Pow(carrier G)"
  <proof>
```

```
theorem (in group) lagrange:
  "[finite(carrier G); subgroup H G]
   $\Longrightarrow$  card(rcosets H) * card(H) = order(G)"
  <proof>
```

4.6 Quotient Groups: Factorization of a Group

definition

```
FactGroup :: "[('a,'b) monoid_scheme, 'a set] => ('a set) monoid" (in-
fixl "Mod" 65)
  — Actually defined for groups rather than monoids
  where "FactGroup G H = (carrier = rcosetsG 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 mult_FactGroup [simp]: "X ⊗(G Mod H) X' = X <#>G X'"
<proof>
```

lemma (in normal) inv_FactGroup:

```
"X ∈ carrier (G Mod H) => invG Mod H X = set_inv X"
<proof>
```

The coset map is a homomorphism from G to the quotient group $G \text{ Mod } H$

lemma (in normal) r_coset_hom_Mod:

```
"(λa. H #> a) ∈ hom G (G Mod H)"
<proof>
```

4.7 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 = 1H}"
```

```
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 (in group_hom) FactGroup_nonempty:
  assumes X: "X ∈ carrier (G Mod kernel G H h)"
  shows "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:
  "[[g ∈ carrier G; g' ∈ carrier G; h g = h g']]
  ⇒ 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⟩
```

<proof>

If h is a homomorphism from G onto H , then the quotient group $G \text{ Mod } \text{kernel } G \ H \ h$ is isomorphic to H .

theorem (in group_hom) FactGroup_iso:

"h ' carrier G = carrier H

==> ($\lambda X. \text{the_elem } (h'X) \in (G \text{ Mod } (\text{kernel } G \ H \ h)) \cong H$)"

<proof>

end

theory Exponent

imports Main "~~/src/HOL/Old_Number_Theory/Primes" "~~/src/HOL/Library/Binomial"

begin

5 Sylow's Theorem

5.1 The Combinatorial Argument Underlying the First Sylow Theorem

definition

exponent :: "nat => nat => nat"

where "exponent p s = (if prime p then (GREATEST r. p^r dvd s) else 0)"

Prime Theorems

lemma prime_imp_one_less: "prime p ==> Suc 0 < p"

<proof>

lemma prime_iff:

"(prime p) = (Suc 0 < p & ($\forall a \ b. \ p \ \text{dvd} \ a*b \ \rightarrow (p \ \text{dvd} \ a) \ \vee (p \ \text{dvd} \ b)$))"

<proof>

lemma zero_less_prime_power: "prime p ==> 0 < p^a"

<proof>

lemma zero_less_card_empty: "[| finite S; S ≠ {} |] ==> 0 < card(S)"

<proof>

lemma prime_dvd_cases:

"[| p*k dvd m*n; prime p |]"

==> ($\exists x. \ k \ \text{dvd} \ x*n \ \& \ m = p*x$) | ($\exists y. \ k \ \text{dvd} \ m*y \ \& \ n = p*y$)"

<proof>

```

lemma prime_power_dvd_cases [rule_format (no_asm)]: "prime p
  ==>  $\forall m n. p^c \text{ dvd } m * n \text{ -->}$ 
    ( $\forall a b. a + b = \text{Suc } c \text{ --> } p^a \text{ dvd } m \mid p^b \text{ dvd } n)$ "
<proof>

```

```

lemma div_combine:
  "[| prime p;  $\sim (p \text{ } ^ ( \text{Suc } r) \text{ dvd } n)$ ;  $p^{(a+r)} \text{ dvd } n * k$  |]
  ==>  $p^a \text{ dvd } k$ "
<proof>

```

```

lemma Suc_le_power: "Suc 0 < p ==> Suc n <= p^n"
<proof>

```

```

lemma power_dvd_bound: "[| p^n dvd a; Suc 0 < p; a > 0 |] ==> n < a"
<proof>

```

Exponent Theorems

```

lemma exponent_ge [rule_format]:
  "[| p^k dvd n; prime p; 0 < n |] ==> k <= exponent p n"
<proof>

```

```

lemma power_exponent_dvd: "s > 0 ==> (p ^ exponent p s) dvd s"
<proof>

```

```

lemma power_Suc_exponent_Not_dvd:
  "[| (p * p ^ exponent p s) dvd s; prime p |] ==> s = 0"
<proof>

```

```

lemma exponent_power_eq [simp]: "prime p ==> exponent p (p^a) = a"
<proof>

```

```

lemma exponent_equalityI:
  "!r::nat. (p^r dvd a) = (p^r dvd b) ==> exponent p a = exponent p b"
<proof>

```

```

lemma exponent_eq_0 [simp]: " $\neg$  prime p ==> exponent p s = 0"
<proof>

```

```

lemma exponent_mult_add1: "[| a > 0; b > 0 |]
  ==> (exponent p a) + (exponent p b) <= exponent p (a * b)"
<proof>

```

lemma exponent_mult_add2: "[| a > 0; b > 0 |]
 ==> exponent p (a * b) <= (exponent p a) + (exponent p b)"
 <proof>

lemma exponent_mult_add: "[| a > 0; b > 0 |]
 ==> exponent p (a * b) = (exponent p a) + (exponent p b)"
 <proof>

lemma not_divides_exponent_0: "~ (p dvd n) ==> exponent p n = 0"
 <proof>

lemma exponent_1_eq_0 [simp]: "exponent p (Suc 0) = 0"
 <proof>

Main Combinatorial Argument

lemma le_extend_mult: "[| c > 0; a <= b |] ==> a <= b * (c::nat)"
 <proof>

lemma p_fac_forw_lemma:
 "[| (m::nat) > 0; k > 0; k < p^a; (p^r) dvd (p^a)* m - k |] ==> r <= a"
 <proof>

lemma p_fac_forw: "[| (m::nat) > 0; k>0; k < p^a; (p^r) dvd (p^a)* m - k |]
 ==> (p^r) dvd (p^a) - k"
 <proof>

lemma r_le_a_forw:
 "[| (k::nat) > 0; k < p^a; p>0; (p^r) dvd (p^a) - k |] ==> r <= a"
 <proof>

lemma p_fac_backw: "[| m>0; k>0; (p::nat)≠0; k < p^a; (p^r) dvd p^a - k |]
 ==> (p^r) dvd (p^a)*m - k"
 <proof>

lemma exponent_p_a_m_k_equation: "[| m>0; k>0; (p::nat)≠0; k < p^a |]
 ==> exponent p (p^a * m - k) = exponent p (p^a - k)"
 <proof>

Suc rules that we have to delete from the simpset

lemmas bad_Sucs = binomial_Suc_Suc mult_Suc mult_Suc_right

```

lemma p_not_div_choose_lemma [rule_format]:
  "[|  $\forall i. \text{Suc } i < K \rightarrow \text{exponent } p (\text{Suc } i) = \text{exponent } p (\text{Suc}(j+i))$  |]
  ==>  $k < K \rightarrow \text{exponent } p ((j+k) \text{ choose } k) = 0$ "
  <proof>

lemma p_not_div_choose:
  "[|  $k < K; k \leq n;$ 
   $\forall j. 0 < j \ \& \ j < K \rightarrow \text{exponent } p (n - k + (K - j)) = \text{exponent } p (K - j)$  |]
  ==>  $\text{exponent } p (n \text{ choose } k) = 0$ "
  <proof>

lemma const_p_fac_right:
  " $m > 0 \Rightarrow \text{exponent } p ((p^a * m - \text{Suc } 0) \text{ choose } (p^a - \text{Suc } 0)) = 0$ "
  <proof>

lemma const_p_fac:
  " $m > 0 \Rightarrow \text{exponent } p (((p^a) * m) \text{ choose } p^a) = \text{exponent } p m$ "
  <proof>

end

theory Sylow
imports Coset Exponent
begin

See also [3].

The combinatorial argument is in theory Exponent

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  $\subseteq$  carrier(G) & card(s) = p^a}"
  and "RelM == {(N1,N2). N1  $\in$  calM & N2  $\in$  calM &
  ( $\exists g \in \text{carrier}(G). N1 = (N2 \#> g)$  )}"

lemma (in sylow) RelM_refl_on: "refl_on calM RelM"
  <proof>

lemma (in sylow) RelM_sym: "sym RelM"
  <proof>

lemma (in sylow) RelM_trans: "trans RelM"

```

<proof>

lemma (in sylow) RelM_equiv: "equiv calM RelM"
<proof>

lemma (in sylow) M_subset_calM_prep: "M' ∈ calM // RelM ==> M' ⊆ calM"
<proof>

5.2 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(exponent p m) dvd card(M))"
 and M1_in_M: "M1 ∈ M"
 defines "H == {g. g ∈ carrier G & M1 #> g = M1}"

lemma (in sylow_central) M_subset_calM: "M ⊆ calM"
<proof>

lemma (in sylow_central) card_M1: "card(M1) = p^a"
<proof>

lemma card_nonempty: "0 < card(S) ==> S ≠ {}"
<proof>

lemma (in sylow_central) exists_x_in_M1: "∃x. x ∈ M1"
<proof>

lemma (in sylow_central) M1_subset_G [simp]: "M1 ⊆ carrier G"
<proof>

lemma (in sylow_central) M1_inj_H: "∃f ∈ H → M1. inj_on f H"
<proof>

5.3 Discharging the Assumptions of sylow_central

lemma (in sylow) EmptyNotInEquivSet: "{} ∉ calM // RelM"
<proof>

lemma (in sylow) existsM1inM: "M ∈ calM // RelM ==> ∃M1. M1 ∈ M"
<proof>

lemma (in sylow) zero_less_o_G: "0 < order(G)"
<proof>

lemma (in sylow) zero_less_m: "m > 0"
<proof>

lemma (in sylow) card_calM: "card(calM) = (p^a) * m choose p^a"

<proof>

lemma (in sylow) zero_less_card_calM: "card calM > 0"

<proof>

lemma (in sylow) max_p_div_calM:

"~ (p ^ Suc(exponent p m) dvd card(calM))"

<proof>

lemma (in sylow) finite_calM: "finite calM"

<proof>

lemma (in sylow) lemma_A1:

" $\exists M \in \text{calM} // \text{RelM. } \sim (p \wedge \text{Suc}(\text{exponent } p \ m) \text{ dvd } \text{card}(M))$ "

<proof>

5.3.1 Introduction and Destruct Rules for H

lemma (in sylow_central) H_I: "[|g ∈ carrier G; M1 #> g = M1|] ==> g ∈ H"

<proof>

lemma (in sylow_central) H_into_carrier_G: "x ∈ H ==> x ∈ carrier G"

<proof>

lemma (in sylow_central) in_H_imp_eq: "g : H ==> M1 #> g = M1"

<proof>

lemma (in sylow_central) H_m_closed: "[| x∈H; y∈H|] ==> x ⊗ y ∈ H"

<proof>

lemma (in sylow_central) H_not_empty: "H ≠ {}"

<proof>

lemma (in sylow_central) H_is_subgroup: "subgroup H G"

<proof>

lemma (in sylow_central) rcosetGM1g_subset_G:

"[| g ∈ carrier G; x ∈ M1 #> g |] ==> x ∈ carrier G"

<proof>

lemma (in sylow_central) finite_M1: "finite M1"

<proof>

lemma (in sylow_central) finite_rcosetGM1g: "g∈carrier G ==> finite (M1 #> g)"

<proof>

lemma (in sylow_central) M1_cardeq_rcosetGM1g:
 "g ∈ carrier G ==> card(M1 #> g) = card(M1)"
 <proof>

lemma (in sylow_central) M1_RelM_rcosetGM1g:
 "g ∈ carrier G ==> (M1, M1 #> g) ∈ RelM"
 <proof>

5.4 Equal Cardinalities of M and the Set of Cosets

Injections between M and $\text{rcosets}_G H$ show that their cardinalities are equal.

lemma ElemClassEquiv:
 "[| equiv A r; C ∈ A // r |] ==> ∀x ∈ C. ∀y ∈ C. (x,y) ∈ r"
 <proof>

lemma (in sylow_central) M_elem_map:
 "M2 ∈ M ==> ∃g. g ∈ carrier G & M1 #> g = M2"
 <proof>

lemmas (in sylow_central) M_elem_map_carrier =
 M_elem_map [THEN someI_ex, THEN conjunct1]

lemmas (in sylow_central) M_elem_map_eq =
 M_elem_map [THEN someI_ex, THEN conjunct2]

lemma (in sylow_central) M_funcset_rcosets_H:
 "(%x:M. H #> (SOME g. g ∈ carrier G & M1 #> g = x)) ∈ M → rcosets
 H"
 <proof>

lemma (in sylow_central) inj_M_GmodH: "∃f ∈ M → rcosets H. inj_on f M"
 <proof>

5.4.1 The Opposite Injection

lemma (in sylow_central) H_elem_map:
 "H1 ∈ rcosets H ==> ∃g. g ∈ carrier G & H #> g = H1"
 <proof>

lemmas (in sylow_central) H_elem_map_carrier =
 H_elem_map [THEN someI_ex, THEN conjunct1]

lemmas (in sylow_central) H_elem_map_eq =
 H_elem_map [THEN someI_ex, THEN conjunct2]

lemma EquivElemClass:
 "[|equiv A r; M ∈ A//r; M1 ∈ M; (M1,M2) ∈ r |] ==> M2 ∈ M"
 <proof>

```

lemma (in sylow_central) rcosets_H_funcset_M:
  "(\C \in rcosets H. M1 #> (@g. g \in carrier G \wedge H #> g = C)) \in rcosets
H \to M"
<proof>

```

close to a duplicate of inj_M_GmodH

```

lemma (in sylow_central) inj_GmodH_M:
  "\g \in rcosets H \to M. inj_on g (rcosets H)"
<proof>

```

```

lemma (in sylow_central) calM_subset_PowG: "calM \subseteq Pow(carrier G)"
<proof>

```

```

lemma (in sylow_central) finite_M: "finite M"
<proof>

```

```

lemma (in sylow_central) cardMeqIndexH: "card(M) = card(rcosets H)"
<proof>

```

```

lemma (in sylow_central) index_lem: "card(M) * card(H) = order(G)"
<proof>

```

```

lemma (in sylow_central) lemma_leq1: "p^a \le card(H)"
<proof>

```

```

lemma (in sylow_central) lemma_leq2: "card(H) \le p^a"
<proof>

```

```

lemma (in sylow_central) card_H_eq: "card(H) = p^a"
<proof>

```

```

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_axioms G rather than simply to group G.

```

lemma sylow_eq: "sylow G p a m = (group G & sylow_axioms G p a m)"
<proof>

```

5.5 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
```

6 Bijections of a Set, Permutation and Automorphism Groups

definition

```
Bij :: "'a set  $\Rightarrow$  ('a  $\Rightarrow$  'a) set"
  — Only extensional functions, since otherwise we get too many.
  where "Bij S = extensional S  $\cap$  {f. bij_betw f S S}"
```

definition

```
BijGroup :: "'a set  $\Rightarrow$  ('a  $\Rightarrow$  'a) monoid"
  where "BijGroup S =
    (carrier = Bij S,
     mult =  $\lambda$ g  $\in$  Bij S.  $\lambda$ f  $\in$  Bij S. compose S g f,
     one =  $\lambda$ x  $\in$  S. x)"
```

```
declare Id_compose [simp] compose_Id [simp]
```

```
lemma Bij_imp_extensional: "f  $\in$  Bij S  $\implies$  f  $\in$  extensional S"
  <proof>
```

```
lemma Bij_imp_funcset: "f  $\in$  Bij S  $\implies$  f  $\in$  S  $\rightarrow$  S"
  <proof>
```

6.1 Bijections Form a Group

```
lemma restrict_inv_into_Bij: "f  $\in$  Bij S  $\implies$  ( $\lambda$ x  $\in$  S. (inv_into S f)
x)  $\in$  Bij S"
  <proof>
```

```
lemma id_Bij: "( $\lambda$ x  $\in$  S. x)  $\in$  Bij S "
  <proof>
```

```
lemma compose_Bij: "[[x  $\in$  Bij S; y  $\in$  Bij S]]  $\implies$  compose S x y  $\in$  Bij S"
  <proof>
```

```
lemma Bij_compose_restrict_eq:
  "f  $\in$  Bij S  $\implies$  compose S (restrict (inv_into S f) S) f = ( $\lambda$ x  $\in$  S.
x)"
  <proof>
```

theorem group_BijGroup: "group (BijGroup S)"
<proof>

6.2 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: " $\bigwedge x y. [x \in S; y \in S] \implies h(g x y) = g (h x) (h y)$ "
shows "[[h ∈ Bij S; g ∈ S → S → S; x ∈ S; y ∈ S]
 $\implies \text{inv_into } S h (g x y) = g (\text{inv_into } S h x) (\text{inv_into } S h y)$]"
<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:
" $[h \in \text{hom } G G; h \in \text{Bij (carrier } G)]$
 $\implies \text{restrict (inv_into (carrier } G) h) (carrier G) \in \text{hom } G G$ "
<proof>

lemma inv_BijGroup:
" $f \in \text{Bij } S \implies \text{m_inv (BijGroup } S) f = (\lambda x \in S. (\text{inv_into } S f) x)$ "
<proof>

lemma (in group) subgroup_auto:
"subgroup (auto G) (BijGroup (carrier G))"
<proof>

theorem (in group) AutoGroup: "group (AutoGroup G)"
<proof>

end

7 Divisibility in monoids and rings

```
theory Divisibility
imports "~~/src/HOL/Library/Permutation" Coset Group
begin
```

8 Factorial Monoids

8.1 Monoids with Cancellation Law

```
locale monoid_cancel = monoid +
  assumes l_cancel:
    "[c  $\otimes$  a = c  $\otimes$  b; a  $\in$  carrier G; b  $\in$  carrier G; c  $\in$  carrier
G]  $\implies$  a = b"
    and r_cancel:
    "[a  $\otimes$  c = b  $\otimes$  c; a  $\in$  carrier G; b  $\in$  carrier G; c  $\in$  carrier
G]  $\implies$  a = b"

lemma (in monoid) monoid_cancelI:
  assumes l_cancel:
    "[ $\bigwedge$  a b c. [c  $\otimes$  a = c  $\otimes$  b; a  $\in$  carrier G; b  $\in$  carrier G; c  $\in$ 
carrier G]  $\implies$  a = b]"
    and r_cancel:
    "[ $\bigwedge$  a b c. [a  $\otimes$  c = b  $\otimes$  c; a  $\in$  carrier G; b  $\in$  carrier G; c  $\in$ 
carrier G]  $\implies$  a = b]"
  shows "monoid_cancel G"
    <proof>

lemma (in monoid_cancel) is_monoid_cancel:
  "monoid_cancel G"
  <proof>

sublocale group  $\subseteq$  monoid_cancel
  <proof>

locale comm_monoid_cancel = monoid_cancel + comm_monoid

lemma comm_monoid_cancelI:
  fixes G (structure)
  assumes "comm_monoid G"
  assumes cancel:
    "[ $\bigwedge$  a b c. [a  $\otimes$  c = b  $\otimes$  c; a  $\in$  carrier G; b  $\in$  carrier G; c  $\in$ 
carrier G]  $\implies$  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 \subseteq comm_monoid_cancel

<proof>

8.2 Products of Units in Monoids

lemma (in monoid) Units_m_closed[simp, intro]:
 assumes h1unit: "h1 \in Units G" and h2unit: "h2 \in Units G"
 shows "h1 \otimes h2 \in Units G"

<proof>

lemma (in monoid) prod_unit_l:
 assumes abunit[simp]: "a \otimes b \in Units G" and aunit[simp]: "a \in Units G"

and carr[simp]: "a \in carrier G" "b \in carrier G"

shows "b \in Units G"

<proof>

lemma (in monoid) prod_unit_r:
 assumes abunit[simp]: "a \otimes b \in Units G" and bunit[simp]: "b \in Units G"

and carr[simp]: "a \in carrier G" "b \in carrier G"

shows "a \in Units G"

<proof>

lemma (in comm_monoid) unit_factor:
 assumes abunit: "a \otimes b \in Units G"
 and [simp]: "a \in carrier G" "b \in carrier G"
 shows "a \in Units G"

<proof>

8.3 Divisibility and Association

8.3.1 Function definitions

definition

factor :: "[_, 'a, 'a] \Rightarrow bool" (infix "divides_z" 65)
 where "a divides_G b \longleftrightarrow ($\exists c \in$ carrier G. b = a \otimes_G c)"

definition

associated :: "[_, 'a, 'a] \Rightarrow bool" (infix " \sim_z " 55)
 where "a \sim_G b \longleftrightarrow a divides_G b \wedge b divides_G a"

abbreviation

"division_rel G == (carrier = carrier G, eq = op \sim_G , le = op divides_G)"

definition

properfactor :: "[_, 'a, 'a] \Rightarrow bool"
 where "properfactor G a b \longleftrightarrow a divides_G b \wedge \neg (b divides_G a)"

definition

```
irreducible :: "[_, 'a] => bool"
  where "irreducible G a <math>\iff a \notin \text{Units } G \wedge (\forall b \in \text{carrier } G. \text{properfactor } G \ b \ a \longrightarrow b \in \text{Units } G)</math>"
```

definition

```
prime :: "[_, 'a] => bool" where
  "prime G p <math>\iff p \notin \text{Units } G \wedge (\forall a \in \text{carrier } G. \forall b \in \text{carrier } G. p \ \text{divides}_G \ (a \otimes_G b) \longrightarrow p \ \text{divides}_G \ a \vee p \ \text{divides}_G \ b)</math>"
```

8.3.2 Divisibility**lemma dividesI:**

```
fixes G (structure)
  assumes carr: "c ∈ carrier G"
    and p: "b = a ⊗ c"
  shows "a divides b"
  <math>\langle proof \rangle</math>
```

lemma dividesI' [intro]:

```
fixes G (structure)
  assumes p: "b = a ⊗ c"
    and carr: "c ∈ carrier G"
  shows "a divides b"
  <math>\langle proof \rangle</math>
```

lemma dividesD:

```
fixes G (structure)
  assumes "a divides b"
  shows "∃ c ∈ carrier G. b = a ⊗ c"
  <math>\langle proof \rangle</math>
```

lemma dividesE [elim]:

```
fixes G (structure)
  assumes d: "a divides b"
    and elim: "∧ c. [b = a ⊗ c; c ∈ carrier G] ⇒ P"
  shows "P"
  <math>\langle proof \rangle</math>
```

lemma (in monoid) divides_refl[simp, intro!]:

```
assumes carr: "a ∈ carrier G"
  shows "a divides a"
  <math>\langle proof \rangle</math>
```

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 ab: "a divides b"
    and carr: "a ∈ carrier G" "b ∈ 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" "b ∈ carrier G" "c ∈ carrier G"
  shows "a divides (b ⊗ c)"
  <proof>

lemma (in comm_monoid) divides_prod_l:
  assumes carr[intro]: "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier
G"
    and ab: "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>
```

8.3.3 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[simp]: "b ∈ carrier G"
  shows "a ~ b"
<proof>
```

```
lemma (in monoid) associatedI2':
  assumes a: "a = b ⊗ u"
  and uunit: "u ∈ Units G"
  and bcarr: "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 \otimes u; u \in Units G] \implies P"
  and carr: "a \in carrier G" "b \in carrier G"
shows "P"
<proof>

```

```

lemma (in monoid) associated_refl [simp, intro!]:
  assumes "a \in carrier G"
  shows "a ~ a"
<proof>

```

```

lemma (in monoid) associated_sym [sym]:
  assumes "a ~ b"
  and "a \in carrier G" "b \in carrier G"
  shows "b ~ a"
<proof>

```

```

lemma (in monoid) associated_trans [trans]:
  assumes "a ~ b" "b ~ c"
  and "a \in carrier G" "b \in carrier G" "c \in carrier G"
  shows "a ~ c"
<proof>

```

```

lemma (in monoid) division_equiv [intro, simp]:
  "equivalence (division_rel G)"
<proof>

```

8.3.4 Division and associativity

```

lemma divides_antisym:
  fixes G (structure)
  assumes "a divides b" "b divides a"
  and "a \in carrier G" "b \in carrier G"
  shows "a ~ b"
<proof>

```

```

lemma (in monoid) divides_cong_l [trans]:
  assumes xx': "x ~ x'"
  and xdvd y: "x' divides y"
  and carr [simp]: "x \in carrier G" "x' \in carrier G" "y \in carrier
G"
  shows "x divides y"
<proof>

```

```

lemma (in monoid) divides_cong_r [trans]:
  assumes xdvd y: "x divides y"
  and yy': "y ~ y'"
  and carr [simp]: "x \in carrier G" "y \in carrier G" "y' \in carrier G"
  shows "x divides y'"

```

<proof>

```
lemma (in monoid) division_weak_partial_order [simp, intro!]:
  "weak_partial_order (division_rel G)"
  <proof>
```

8.3.5 Multiplication and associativity

```
lemma (in monoid_cancel) mult_cong_r:
  assumes "b ~ b'"
  and carr: "a ∈ carrier G" "b ∈ carrier G" "b' ∈ carrier G"
  shows "a ⊗ b ~ a ⊗ b'"
  <proof>
```

```
lemma (in comm_monoid_cancel) mult_cong_l:
  assumes "a ~ a'"
  and carr: "a ∈ carrier G" "a' ∈ carrier G" "b ∈ carrier G"
  shows "a ⊗ b ~ a' ⊗ b"
  <proof>
```

```
lemma (in monoid_cancel) assoc_l_cancel:
  assumes carr: "a ∈ carrier G" "b ∈ carrier G" "b' ∈ carrier G"
  and "a ⊗ b ~ a ⊗ b'"
  shows "b ~ b'"
  <proof>
```

```
lemma (in comm_monoid_cancel) assoc_r_cancel:
  assumes "a ⊗ b ~ a' ⊗ b"
  and carr: "a ∈ carrier G" "a' ∈ carrier G" "b ∈ carrier G"
  shows "a ~ a'"
  <proof>
```

8.3.6 Units

```
lemma (in monoid_cancel) assoc_unit_l [trans]:
  assumes asc: "a ~ b" and bunit: "b ∈ Units G"
  and carr: "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 {.=}(division_rel G) {1}"
<proof>
```

```
lemma (in comm_monoid) Units_Lower:
  "Units G = Lower (division_rel G) (carrier G)"
<proof>
```

8.3.7 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" "p ∈ 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"
```

```

    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 dvds: "a divides b" "properfactor G b c"
    and carr: "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier G"
  shows "properfactor G a c"
  <proof>

lemma (in monoid) properfactor_trans2 [trans]:
  assumes dvds: "properfactor G a b" "b divides c"
    and carr: "a ∈ carrier G" "b ∈ carrier G" "c ∈ 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" "b ∈ 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" "b ∈ 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>

```

8.4 Irreducible Elements and Primes

8.4.1 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 irred: "irreducible G a"
    and aa': "a ~ a'"
    and carr[simp]: "a ∈ carrier G" "a' ∈ carrier G"
  shows "irreducible G a'"
⟨proof⟩

lemma (in monoid) irreducible_prod_rI:
  assumes airr: "irreducible G a"
    and bunit: "b ∈ Units G"
    and carr[simp]: "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⟩

```

8.4.2 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 pprime: "prime G p"
  and pp': "p ∼ p'"
  and carr[simp]: "p ∈ carrier G" "p' ∈ carrier G"
  shows "prime G p'"
  <proof>

```

8.5 Factorization and Factorial Monoids

8.5.1 Function definitions

definition

```

factors :: "[_, 'a list, 'a] ⇒ bool"
  where "factors G fs a ⟷ (∀x ∈ (set fs). irreducible G x) ∧ foldr
(op ⊗G) fs 1G = a"

```

definition

```

wfactors :: "[_, 'a list, 'a] ⇒ bool"
  where "wfactors G fs a ⟷ (∀x ∈ (set fs). irreducible G x) ∧ foldr
(op ⊗G) fs 1G ∼G a"

```

abbreviation

```

list_assoc :: "('a, _) monoid_scheme ⇒ 'a list ⇒ 'a list ⇒ bool" (in-
fix "[~]i" 44)
  where "list_assoc G == list_all2 (op ∼G)"

```

definition

```

essentially_equal :: "[_, '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'"

```

8.5.2 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  $\subseteq$  carrier G"
  shows "essentially_equal G as cs"
<proof>

```

8.5.3 Properties of lists of elements

Multiplication of factors in a list

```

lemma (in monoid) multlist_closed [simp, intro]:
  assumes ascarr: "set fs  $\subseteq$  carrier G"
  shows "foldr (op  $\otimes$ ) fs 1  $\in$  carrier G"
<proof>

```

```

lemma (in comm_monoid) multlist_dividesI :
  assumes "f  $\in$  set fs" and "f  $\in$  carrier G" and "set fs  $\subseteq$  carrier G"
  shows "f divides (foldr (op  $\otimes$ ) fs 1)"
<proof>

```

```

lemma (in comm_monoid_cancel) multlist_listassoc_cong:
  assumes "fs  $[\sim]$  fs'"
  and "set fs  $\subseteq$  carrier G" and "set fs'  $\subseteq$  carrier G"
  shows "foldr (op  $\otimes$ ) fs 1  $\sim$  foldr (op  $\otimes$ ) fs' 1"
<proof>

```

```

lemma (in comm_monoid) multlist_perm_cong:
  assumes prm: "as  $\langle \sim \rangle$  bs"
  and ascarr: "set as  $\subseteq$  carrier G"
  shows "foldr (op  $\otimes$ ) as 1 = foldr (op  $\otimes$ ) bs 1"
<proof>

```

```

lemma (in comm_monoid_cancel) multlist_ee_cong:
  assumes "essentially_equal G fs fs'"
  and "set fs  $\subseteq$  carrier G" and "set fs'  $\subseteq$  carrier G"
  shows "foldr (op  $\otimes$ ) fs 1  $\sim$  foldr (op  $\otimes$ ) fs' 1"
<proof>

```

8.5.4 Factorization in irreducible elements

```

lemma wfactorsI:
  fixes G (structure)
  assumes " $\forall f \in \text{set fs. irreducible } G \ f$ "
  and "foldr (op  $\otimes$ ) fs 1  $\sim$  a"
  shows "wfactors G fs a"
<proof>

```

```

lemma wfactorsE:
  fixes G (structure)
  assumes wf: "wfactors G fs a"
  and e: " $\llbracket \forall f \in \text{set fs. irreducible } G \ f; \text{foldr (op } \otimes \text{) fs 1 } \sim \text{ a} \rrbracket \implies$ "
  p"

```

```

    shows "P"
  <proof>

lemma (in monoid) factorsI:
  assumes "∀f∈set fs. irreducible G f"
    and "foldr (op ⊗) 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 (op ⊗) 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_1:
  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_1:
  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_1 [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>

```

8.5.5 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 uunit: "u ∈ Units G" and anunit: "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  $\subseteq$  carrier G"
    shows "f divides a"
  <proof>

```

```

lemma (in comm_monoid) wfactors_dividesI:
  assumes p: "wfactors G fs a"
    and fscarr: "set fs  $\subseteq$  carrier G" and acarr: "a  $\in$  carrier G"
    and f: "f  $\in$  set fs"
  shows "f divides a"
  <proof>

```

8.5.6 Factorial monoids and wfactors

```

lemma (in comm_monoid_cancel) factorial_monoidI:
  assumes wfactors_exists:
    " $\bigwedge a. a \in \text{carrier } G \implies \exists \text{fs. set fs} \subseteq \text{carrier } G \wedge \text{wfactors}$ 
  G fs a"
    and wfactors_unique:
    " $\bigwedge a \text{ fs fs}'. \llbracket a \in \text{carrier } G; \text{set fs} \subseteq \text{carrier } G; \text{set fs}' \subseteq \text{carrier}$ 
  G;
    wfactors G fs a; wfactors G fs' a  $\rrbracket \implies \text{essentially\_equal}$ 
  G fs fs'"
  shows "factorial_monoid G"
  <proof>

```

8.6 Factorizations as Multisets

Gives useful operations like intersection

abbreviation

```
"assocs G x == eq_closure_of (division_rel G) {x}"
```

definition

```
"fmset G as = multiset_of (map ( $\lambda a. \text{assocs } G a$ ) as)"
```

Helper lemmas

```

lemma (in monoid) assocs_repr_independence:
  assumes "y  $\in$  assocs G x"
    and "x  $\in$  carrier G"
  shows "assocs G x = assocs G y"
  <proof>

```

```

lemma (in monoid) assocs_self:
  assumes "x  $\in$  carrier G"
  shows "x  $\in$  assocs G x"
  <proof>

```

```

lemma (in monoid) assocs_repr_independenceD:
  assumes repr: "assocs G x = assocs G y"
    and ycarr: "y  $\in$  carrier G"

```

shows "y ∈ assoc G x"
<proof>

lemma (in comm_monoid) assoc_assoc:
 assumes "a ∈ assoc G b"
 and "b ∈ carrier G"
 shows "a ~ b"
<proof>

lemmas (in comm_monoid) assoc_eqD =
 assoc_repr_independenceD[THEN assoc_assoc]

8.6.1 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 (assoc G) as = map (assoc 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 monoid_cancel) fmset_ee_hlp_induct:
 assumes prm: "cas <~~> cbs"
 and cdef: "cas = map (assoc G) as" "cbs = map (assoc G) bs"
 shows "∀ as bs. (cas <~~> cbs ∧ cas = map (assoc G) as ∧
 cbs = map (assoc G) bs) → (∃ as'. as <~~> as' ∧ map
 (assoc G) as' = cbs)"
<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  $\subseteq$  carrier G" and "set bs  $\subseteq$  carrier G"
  shows "essentially_equal G as bs = (fmset G as = fmset G bs)"
<proof>
```

8.6.2 Interpreting multisets as factorizations

```
lemma (in monoid) mset_fmsetEx:
  assumes elems: " $\bigwedge X. X \in \text{set\_of } Cs \implies \exists x. P x \wedge X = \text{assoc } G x$ "
  shows " $\exists cs. (\forall c \in \text{set } cs. P c) \wedge \text{fmset } G cs = Cs$ "
<proof>
```

```
lemma (in monoid) mset_wfactorsEx:
  assumes elems: " $\bigwedge X. X \in \text{set\_of } Cs$ "
   $\implies \exists x. (x \in \text{carrier } G \wedge \text{irreducible } G x) \wedge X =$ 
  assoc G x"
  shows " $\exists c cs. c \in \text{carrier } G \wedge \text{set } cs \subseteq \text{carrier } G \wedge \text{wfactors } G cs c$ 
 $\wedge \text{fmset } G cs = Cs$ "
<proof>
```

8.6.3 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  $\otimes$  b)"
  and carr: "a  $\in$  carrier G" "b  $\in$  carrier G"
  "set as  $\subseteq$  carrier G" "set bs  $\subseteq$  carrier G" "set cs  $\subseteq$  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  $\otimes$  b)"
  and "set as  $\subseteq$  carrier G" "set bs  $\subseteq$  carrier G" "set cs  $\subseteq$  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  $\in$  carrier G" "b  $\in$  carrier G" "c  $\in$  carrier G"
  "set as  $\subseteq$  carrier G" "set bs  $\subseteq$  carrier G" "set cs  $\subseteq$  carrier
  G"
  and fs: "wfactors G as a" "wfactors G bs b" "wfactors G cs c"
  shows "c  $\sim$  a  $\otimes$  b"
<proof>
```

8.6.4 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 acar: "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 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 ∧ fmset G as ≠ fmset G bs"
<proof>
```

8.7 Irreducible Elements are Prime

```
lemma (in factorial_monoid) irreducible_is_prime:
  assumes pirr: "irreducible G p"
    and pcarr: "p ∈ carrier G"
  shows "prime G p"
<proof>
```

```

lemma (in factorial_monoid) factors_irreducible_is_prime:
  assumes pirr: "irreducible G p"
    and pcarr: "p ∈ carrier G"
  shows "prime G p"
  <proof>

```

8.8 Greatest Common Divisors and Lowest Common Multiples

8.8.1 Definitions

definition

```

isgcd :: "[('a,_) monoid_scheme, 'a, 'a, 'a] ⇒ bool" ("(_ gcdofz _ _)" [81,81,81] 80)
where "x gcdofG a b ⇔ x dividesG a ∧ x dividesG b ∧
      (∀y∈carrier G. (y dividesG a ∧ y dividesG b → y dividesG x))"

```

definition

```

islcm :: "[_, 'a, 'a, 'a] ⇒ bool" ("(_ lcmofz _ _)" [81,81,81] 80)
where "x lcmofG a b ⇔ a dividesG x ∧ b dividesG x ∧
      (∀y∈carrier G. (a dividesG y ∧ b dividesG y → x dividesG y))"

```

definition

```

somegcd :: "('a,_) monoid_scheme ⇒ 'a ⇒ 'a ⇒ 'a"
where "somegcd G a b = (SOME x. x ∈ carrier G ∧ x gcdofG a b)"

```

definition

```

somalcm :: "('a,_) monoid_scheme ⇒ 'a ⇒ 'a ⇒ 'a"
where "somalcm G a b = (SOME x. x ∈ carrier G ∧ x lcmofG a b)"

```

definition

```

"SomeGcd G A = 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}"

```

8.8.2 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>
```

8.8.3 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>
```

8.9 Conditions for Factoriality

8.9.1 Gcd condition

```
lemma (in gcd_condition_monoid) division_weak_lower_semilattice [simp]:
  shows "weak_lower_semilattice (division_rel G)"
  <proof>
```

```
lemma (in gcd_condition_monoid) gcdof_cong_l:
  assumes a'a: "a' ~ a"
    and agcd: "a gcdof b c"
    and a'carr: "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 carr: "a ∈ carrier G" "b ∈ carrier G"
  shows "somegcd G a b ∈ carrier G"
  <proof>
```

```
lemma (in gcd_condition_monoid) gcd_isgcd:
  assumes carr: "a ∈ carrier G" "b ∈ carrier G"
  shows "(somegcd G a b) gcdof a b"
  <proof>
```

```
lemma (in gcd_condition_monoid) gcd_exists:
  assumes carr: "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 carr: "a ∈ carrier G" "b ∈ carrier G"
  shows "(somegcd G a b) divides a"
  <proof>
```

```
lemma (in gcd_condition_monoid) gcd_divides_r:
  assumes carr: "a ∈ carrier G" "b ∈ carrier G"
  shows "(somegcd G a b) divides b"
  <proof>
```

```
lemma (in gcd_condition_monoid) gcd_divides:
  assumes sub: "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 xx': "x ~ x'"
    and carr: "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 carr: "x ∈ carrier G" "y ∈ carrier G" "y' ∈ carrier G"
 and yy': "y ~ y'"
 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∈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 bcarr: "b ∈ carrier G" and ccarr: "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 carr: "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: "ALL 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>
```

8.9.2 Divisor chain condition

```
lemma (in divisor_chain_condition_monoid) wfactors_exist:
  assumes acar: "a ∈ carrier G"
  shows "∃ as. set as ⊆ carrier G ∧ wfactors G as a"
<proof>
```

8.9.3 Primeness condition

```
lemma (in comm_monoid_cancel) multlist_prime_pos:
  assumes carr: "a ∈ carrier G" "set as ⊆ carrier G"
  and aprime: "prime G a"
  and "a divides (foldr (op ⊗) as 1)"
  shows "∃ i < length as. a divides (as!i)"
<proof>
```

```
lemma (in primeness_condition_monoid) wfactors_unique_hlp_induct:
  "∀ a as'. a ∈ carrier G ∧ set as ⊆ carrier G ∧ set as' ⊆ carrier G
∧
  wfactors G as a ∧ wfactors G as' a → essentially_equal G
as as'"
<proof>
```

```
lemma (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>
```

8.9.4 Application to factorial monoids

Number of factors for wellfoundedness

definition

```

factorcount :: "_ => 'a => nat" where
  "factorcount G a =
    (THE c. (ALL 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 "EX c. ALL 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[simp]: "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:

```

```

shows "primeness_condition_monoid G"
  <proof>

lemma (in factorial_monoid) gcd_condition [simp]:
  shows "gcd_condition_monoid G"
  <proof>

sublocale factorial_monoid  $\subseteq$  gcd_condition_monoid
  <proof>

lemma (in factorial_monoid) division_weak_lattice [simp]:
  shows "weak_lattice (division_rel G)"
  <proof>

```

8.10 Factoriality Theorems

```

theorem factorial_condition_one:
  shows "(divisor_chain_condition_monoid G  $\wedge$  primeness_condition_monoid
G) =
  factorial_monoid G"
  <proof>

```

```

theorem factorial_condition_two:
  shows "(divisor_chain_condition_monoid G  $\wedge$  gcd_condition_monoid G)
= factorial_monoid G"
  <proof>

```

end

```

theory Ring
imports FiniteProduct
uses ("ringsimp.ML")
begin

```

9 The Algebraic Hierarchy of Rings

9.1 Abelian Groups

```

record 'a ring = "'a monoid" +
  zero :: 'a ("0")
  add :: "'a, 'a] => 'a" (infixl " $\oplus$ " 65)

```

Derived operations.

```

definition
  a_inv :: "('a, 'm) ring_scheme, 'a ] => 'a" (" $\ominus$  _" [81] 80)
  where "a_inv R = m_inv (| carrier = carrier R, mult = add R, one =
zero R |)"

```

definition

```
a_minus :: "[('a, 'm) ring_scheme, 'a, 'a] => 'a" (infixl "⊖" 65)
where "[| x ∈ carrier R; y ∈ carrier R |] ==> x ⊖R y = x ⊕R (⊖R y)"
```

```
locale abelian_monoid =
```

```
  fixes G (structure)
```

```
  assumes a_comm_monoid:
```

```
    "comm_monoid (| carrier = carrier G, mult = add G, one = zero G |)"
```

definition

```
finsum :: "[('b, 'm) ring_scheme, 'a => 'b, 'a set] => 'b" where
"finsum G = finprod (| carrier = carrier G, mult = add G, one = zero
G |)"
```

syntax

```
"_finsum" :: "index => idt => 'a set => 'b => 'b"
  ("(3⊕__:_._)" [1000, 0, 51, 10] 10)
```

```
syntax (xsymbols)
```

```
"_finsum" :: "index => idt => 'a set => 'b => 'b"
  ("(3⊕__∈_. _)" [1000, 0, 51, 10] 10)
```

```
syntax (HTML output)
```

```
"_finsum" :: "index => idt => 'a set => 'b => 'b"
  ("(3⊕__∈_. _)" [1000, 0, 51, 10] 10)
```

translations

```
"⊕i:A b" == "CONST finsum ◊i (%i. b) A"
— Beware of argument permutation!
```

```
locale abelian_group = abelian_monoid +
```

```
  assumes a_comm_group:
```

```
    "comm_group (| carrier = carrier G, mult = add G, one = zero G |)"
```

9.2 Basic Properties

```
lemma abelian_monoidI:
```

```
  fixes R (structure)
```

```
  assumes a_closed:
```

```
    "!!x y. [| x ∈ carrier R; y ∈ carrier R |] ==> x ⊕ y ∈ carrier
R"
```

```
  and zero_closed: "0 ∈ carrier R"
```

```
  and a_assoc:
```

```
    "!!x y z. [| x ∈ carrier R; y ∈ carrier R; z ∈ carrier R |] ==>
(x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)"
```

```
  and l_zero: "!!x. x ∈ carrier R ==> 0 ⊕ x = x"
```

```
  and a_comm:
```

```
    "!!x y. [| x ∈ carrier R; y ∈ carrier R |] ==> x ⊕ y = y ⊕ x"
```

```
shows "abelian_monoid R"
```

```
<proof>
```

```

lemma abelian_groupI:
  fixes R (structure)
  assumes a_closed:
    "!!x y. [| x ∈ carrier R; y ∈ carrier R |] ==> x ⊕ y ∈ carrier
R"
  and zero_closed: "zero R ∈ carrier R"
  and a_assoc:
    "!!x y z. [| x ∈ carrier R; y ∈ carrier R; z ∈ carrier R |] ==>
(x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)"
  and a_comm:
    "!!x y. [| x ∈ carrier R; y ∈ carrier R |] ==> x ⊕ y = y ⊕ x"
  and l_zero: "!!x. x ∈ carrier R ==> 0 ⊕ x = x"
  and l_inv_ex: "!!x. x ∈ carrier R ==> EX y : carrier R. y ⊕ x = 0"
shows "abelian_group R"
  <proof>

```

```

lemma (in abelian_monoid) a_monoid:
  "monoid (| carrier = carrier G, mult = add G, one = zero G |)"
  <proof>

```

```

lemma (in abelian_group) a_group:
  "group (| carrier = carrier G, mult = add G, one = zero G |)"
  <proof>

```

```

lemmas monoid_record_simps = partial_object.simps monoid.simps

```

Transfer facts from multiplicative structures via interpretation.

```

sublocale abelian_monoid <
  add!: monoid "(| carrier = carrier G, mult = add G, one = zero G |)"
  where "carrier (| carrier = carrier G, mult = add G, one = zero G |)
= carrier G"
  and "mult (| carrier = carrier G, mult = add G, one = zero G |) =
add G"
  and "one (| carrier = carrier G, mult = add G, one = zero G |) = zero
G"
  <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 "(| carrier = carrier G, mult = add G, one = zero
G |)"
  where "carrier (| carrier = carrier G, mult = add G, one = zero G |)
= carrier G"
    and "mult (| carrier = carrier G, mult = add G, one = zero G |) =
add G"
    and "one (| carrier = carrier G, mult = add G, one = zero G |) = zero
G"
    and "finprod (| carrier = carrier G, mult = add G, one = zero G |)
= finsum G"
    <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_add = add.finprod_mult

```

```

lemmas finsum_cong = add.finprod_cong

```

Usually, if this rule causes a failed congruence proof error, the reason is that the premise $g \in B \rightarrow \text{carrier } G$ 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 "(| carrier = carrier G, mult = add G, one = zero G |)"
  where "carrier (| carrier = carrier G, mult = add G, one = zero G |)
= carrier G"

```

```

    and "mult (| carrier = carrier G, mult = add G, one = zero G |) =
add G"
    and "one (| carrier = carrier G, mult = add G, one = zero G |) = zero
G"
    and "m_inv (| carrier = carrier G, mult = add G, one = zero G |) =
a_inv G"
      <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 a_l_cancel = add.l_cancel
lemmas a_r_cancel = add.r_cancel
lemmas l_neg = add.l_inv [simp del]
lemmas r_neg = add.r_inv [simp del]
lemmas minus_zero = add.inv_one
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 "(| carrier = carrier G, mult = add G, one = zero G
|)"
  where "carrier (| carrier = carrier G, mult = add G, one = zero G |)
= carrier G"
    and "mult (| carrier = carrier G, mult = add G, one = zero G |) =
add G"
    and "one (| carrier = carrier G, mult = add G, one = zero G |) = zero
G"
    and "m_inv (| carrier = carrier G, mult = add G, one = zero G |) =
a_inv G"
    and "finprod (| carrier = carrier G, mult = add G, one = zero G |)
= finsum G"
      <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 (|carrier = carrier G, mult = add G, one = zero
G|)"

```

```

  shows "abelian_group G"
  ⟨proof⟩

```

9.3 Rings: Basic Definitions

```

locale ring = abelian_group 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"

```

```

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}"

```

9.4 Rings

```

lemma ringI:
  fixes R (structure)
  assumes abelian_group: "abelian_group R"
    and monoid: "monoid R"
    and l_distr: "!!x y z. [| x ∈ carrier R; y ∈ carrier R; z ∈ carrier
R |]
    ==> (x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
    and r_distr: "!!x y z. [| x ∈ carrier R; y ∈ carrier R; z ∈ carrier
R |]
    ==> z ⊗ (x ⊕ y) = z ⊗ x ⊕ z ⊗ y"
  shows "ring R"
  ⟨proof⟩

```

```

context ring begin

```

```

lemma is_abelian_group: "abelian_group R" ⟨proof⟩

```

```

lemma is_monoid: "monoid R"
  ⟨proof⟩

```

```

lemma is_ring: "ring R"
  ⟨proof⟩

```

```

end

```

```

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 (in cring) is_cring:
  "cring R" <proof>

```

9.4.1 Normaliser for Rings

```

lemma (in abelian_group) r_neg2:
  "[| x ∈ carrier G; y ∈ carrier G |] ==> x ⊕ (⊖ x ⊕ y) = y"
  <proof>

```

```

lemma (in abelian_group) r_neg1:
  "[| 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.

```

lemma l_null [simp]:
  "x ∈ carrier R ==> 0 ⊗ x = 0"
  <proof>

```

```

lemma r_null [simp]:
  "x ∈ carrier R ==> x ⊗ 0 = 0"
  <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 ∈ carrier G; y ∈ carrier G |] ==> x ⊖ y = x ⊕ ⊖ y"
  <proof>

```

Setup algebra method: compute distributive normal form in locale contexts

$\langle ML \rangle$

```
lemmas (in ring) ring_simps
  [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_simps
  [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 cring) nat_pow_zero:
  "(n::nat) ~ 0 ==> 0 (^) n = 0"
  <proof>
```

context ring 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⟩

```

9.4.2 Sums over Finite Sets

```

lemma (in ring) finsum_ldistr:
  "[| finite A; a ∈ carrier R; f ∈ A -> carrier R |] ==>
  finsum R f A ⊗ a = finsum R (%i. f i ⊗ a) A"
⟨proof⟩

```

```

lemma (in ring) finsum_rdistr:
  "[| finite A; a ∈ carrier R; f ∈ A -> carrier R |] ==>
  a ⊗ finsum R f A = finsum R (%i. a ⊗ f i) A"
⟨proof⟩

```

9.5 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

9.6 Fields

Field would not need to be derived from domain, the properties for domain follow from the assumptions of field

```
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 ≠ 1"
  and invex: "∧a. [[a ∈ carrier R; a ≠ 0]] ⇒ ∃b∈carrier R. a ⊗ b = 1"
  shows "field R"
```

<proof>

9.7 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 &
  (ALL 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 1R = 1S}}
```

```
lemma ring_hom_memI:
```

```
  fixes R (structure) and S (structure)
```

```
  assumes hom_closed: "!!x. x ∈ carrier R ==> h x ∈ carrier S"
```

```
  and hom_mult: "!!x y. [| x ∈ carrier R; y ∈ carrier R |] ==>
```

```
    h (x ⊗ y) = h x ⊗S h y"
```

```
  and hom_add: "!!x y. [| x ∈ carrier R; y ∈ carrier R |] ==>
```

```
    h (x ⊕ y) = h x ⊕S h y"
```

```
  and hom_one: "h 1 = 1S"
```

```
  shows "h ∈ ring_hom R S"
```

<proof>

```
lemma ring_hom_closed:
```

```
" [| h ∈ ring_hom R S; x ∈ carrier R |] ==> 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 |] ==>
  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 = 1S"
  ⟨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 = 0S"
  ⟨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]:
  "[| finite A; f ∈ A -> carrier R |] ==>
  h (finsum R f A) = finsum S (h o f) A"
  ⟨proof⟩

lemma (in ring_hom_cring) hom_finprod:
  "[| finite A; f ∈ A -> carrier R |] ==>
  h (finprod R f A) = finprod S (h o f) A"
  ⟨proof⟩

declare ring_hom_cring.hom_finprod [simp]

lemma id_ring_hom [simp]:
  "id ∈ ring_hom R R"
  ⟨proof⟩

```

end

```
theory AbelCoset
imports Coset Ring
begin
```

9.8 More Lifting from Groups to Abelian Groups

9.8.1 Definitions

Hiding $\langle + \rangle$ from Sum_Type until I come up with better syntax here

```
no_notation Sum_Type.Plus (infixr "<+>" 65)
```

definition

```
a_r_coset    :: "[_, 'a set, 'a]  $\Rightarrow$  'a set" (infixl "+>" 60)
  where "a_r_coset G = r_coset ( $\mid$ carrier = carrier G, mult = add G, one
= zero G)"
```

definition

```
a_l_coset    :: "[_, 'a, 'a set]  $\Rightarrow$  'a set" (infixl "<+>" 60)
  where "a_l_coset G = l_coset ( $\mid$ carrier = carrier G, mult = add G, one
= zero G)"
```

definition

```
A_RCOSSETS  :: "[_, 'a set]  $\Rightarrow$  ('a set)set" ("a'_rcosets" [81] 80)
  where "A_RCOSSETS G H = RCOSSETS ( $\mid$ carrier = carrier G, mult = add G,
one = zero G) H"
```

definition

```
set_add     :: "[_, 'a set, 'a set]  $\Rightarrow$  'a set" (infixl "<+>" 60)
  where "set_add G = set_mult ( $\mid$ carrier = carrier G, mult = add G, one
= zero G)"
```

definition

```
A_SET_INV   :: "[_, 'a set]  $\Rightarrow$  'a set" ("a'_set'_inv" [81] 80)
  where "A_SET_INV G H = SET_INV ( $\mid$ carrier = carrier G, mult = add G,
one = zero G) H"
```

definition

```
a_r_congruent :: "[('a,'b)ring_scheme, 'a set]  $\Rightarrow$  ('a*'a)set" ("racong")
  where "a_r_congruent G = r_congruent ( $\mid$ carrier = carrier G, mult = add
G, one = zero G)"
```

definition

```
A_FactGroup :: "[('a,'b) ring_scheme, 'a set]  $\Rightarrow$  ('a set) monoid" (in-
fixl "A'_Mod" 65)
```

— Actually defined for groups rather than monoids
where "A_FactGroup G H = FactGroup ((carrier = carrier G, mult = add G, one = zero 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 ((carrier = carrier G, mult = add G, one = zero G)
 ((carrier = carrier H, mult = add H, one = zero 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 (| carrier = carrier G, mult = add G, one = zero G |)
 (| carrier = carrier H, mult = add H, one = zero 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 ≡ $\bigcup_{h \in H} \{h \oplus a\}$ "
 <proof>

lemmas a_l_coset_defs =
 a_l_coset_def l_coset_def

lemma a_l_coset_def':
 fixes G (structure)
 shows "a <+ H ≡ $\bigcup_{h \in H} \{a \oplus h\}$ "
 <proof>

lemmas A_RCOSETS_defs =
 A_RCOSETS_def RCOSETS_def

lemma A_RCOSETS_def':
 fixes G (structure)
 shows "a_rcosets H ≡ $\bigcup_{a \in \text{carrier } G} \{H +> a\}$ "
 <proof>

lemmas set_add_defs =
 set_add_def set_mult_def

lemma set_add_def':
 fixes G (structure)

shows "H <+> K $\equiv \bigcup h \in H. \bigcup k \in K. \{h \oplus k\}$ "
<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 $\equiv \bigcup h \in H. \{\ominus h\}$ "
<proof>

9.8.2 Cosets

lemma (in abelian_group) a_coset_add_assoc:
 "[| M \subseteq carrier G; g \in carrier G; h \in carrier G |]
 ==> (M +> g) +> h = M +> (g \oplus h)"
<proof>

lemma (in abelian_group) a_coset_add_zero [simp]:
 "M \subseteq carrier G ==> M +> 0 = M"
<proof>

lemma (in abelian_group) a_coset_add_inv1:
 "[| M +> (x \oplus (\ominus y)) = M; x \in carrier G ; y \in carrier G;
 M \subseteq carrier G |] ==> M +> x = M +> y"
<proof>

lemma (in abelian_group) a_coset_add_inv2:
 "[| M +> x = M +> y; x \in carrier G; y \in carrier G; M \subseteq carrier
 G |]
 ==> M +> (x \oplus (\ominus y)) = M"
<proof>

lemma (in abelian_group) a_coset_join1:
 "[| H +> x = H; x \in carrier G; subgroup H (|carrier = carrier G,
 mult = add G, one = zero G|) |] ==> x \in H"
<proof>

lemma (in abelian_group) a_solve_equation:
 "[subgroup H (|carrier = carrier G, mult = add G, one = zero G|);
 x \in H; y \in H] $\implies \exists h \in H. y = h \oplus x$ "
<proof>

lemma (in abelian_group) a_repr_independence:
 "[y \in H +> x; x \in carrier G; subgroup H (|carrier = carrier G, mult
 = add G, one = zero G|)] $\implies H +> x = H +> y$ "
<proof>

lemma (in abelian_group) a_coset_join2:

```

    "[x ∈ carrier G; subgroup H (|carrier = carrier G, mult = add G,
one = zero 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>

```

9.8.3 Subgroups

```

locale additive_subgroup =
  fixes H and G (structure)
  assumes a_subgroup: "subgroup H (|carrier = carrier G, mult = add G,
one = zero 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 (|carrier = carrier G, mult = add G,
one = zero 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>

9.8.4 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 (carrier = carrier G, mult = add G, one
 = zero G)"

lemma (in abelian_subgroup) is_abelian_subgroup:
 shows "abelian_subgroup H G"
<proof>

lemma abelian_subgroupI:
 assumes a_normal: "normal H (carrier = carrier G, mult = add G, one
 = zero 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 (carrier = carrier G, mult = add
 G, one = zero G)"
 and a_subgroup: "subgroup H (carrier = carrier G, mult = add G,
 one = zero G)"
 shows "abelian_subgroup H G"
<proof>

lemma abelian_subgroupI3:
 fixes G (structure)
 assumes asg: "additive_subgroup H G"
 and ag: "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⟩

Alternative characterization of normal subgroups

lemma (in abelian_group) a_normal_inv_iff:
 "(N ◁ (|carrier = carrier G, mult = add G, one = zero G)) =
 (subgroup N (|carrier = carrier G, mult = add G, one = zero G) &
 (∀x ∈ carrier G. ∀h ∈ N. x ⊕ h ⊕ (⊖ x) ∈ N))"
 (is "_ = ?rhs")
 ⟨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 (|carrier = carrier G, mult
 = add G, one = zero G)|] ⇒ x ∈ y <+ H"
 ⟨proof⟩

lemma (in abelian_group) a_l_coset_carrier:
 "[| y ∈ x <+ H; x ∈ carrier G; subgroup H (|carrier = carrier G,
 mult = add G, one = zero G) |] ==> y ∈ carrier G"
 ⟨proof⟩

lemma (in abelian_group) a_l_repr_imp_subset:
 assumes y: "y ∈ x <+ H" and x: "x ∈ carrier G" and sb: "subgroup H
 (|carrier = carrier G, mult = add G, one = zero 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
 (|carrier = carrier G, mult = add G, one = zero 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 (carrier = carrier
G, mult = add G, one = zero 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>

9.8.5 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]"

```

$\implies hb \oplus a \in (\bigcup_{h \in H}. \{h \oplus b\})$
<proof>

lemma (in abelian_subgroup) a_rcos_disjoint:
 shows "[a ∈ a_rcosets H; b ∈ a_rcosets H; a ≠ b] \implies a ∩ b = {}"
<proof>

lemma (in abelian_subgroup) a_rcos_self:
 shows "x ∈ carrier G \implies x ∈ H +> x"
<proof>

lemma (in abelian_subgroup) a_rcosets_part_G:
 shows " \bigcup (a_rcosets H) = carrier G"
<proof>

lemma (in abelian_subgroup) a_cosets_finite:
 "[c ∈ a_rcosets H; H ⊆ carrier G; finite (carrier G)] \implies finite
 c"
<proof>

lemma (in abelian_group) a_card_cosets_equal:
 "[c ∈ a_rcosets H; H ⊆ carrier G; finite(carrier G)]
 \implies card c = card H"
<proof>

lemma (in abelian_group) rcosets_subset_PowG:
 "additive_subgroup H G \implies a_rcosets H ⊆ Pow(carrier G)"
<proof>

theorem (in abelian_group) a_lagrange:
 "[finite(carrier G); additive_subgroup H G]
 \implies card(a_rcosets H) * card(H) = order(G)"
<proof>

9.8.6 Factorization

lemmas A_FactGroup_defs = A_FactGroup_def FactGroup_def

lemma A_FactGroup_def':
 fixes G (structure)
 shows "G A_Mod H \equiv (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] \implies 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, it 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 ⊗(G A_Mod H) X' = X <+>G X'"
⟨proof⟩

```

```

lemma (in abelian_subgroup) a_inv_FactGroup:
  "X ∈ carrier (G A_Mod H) ⇒ invG A_Mod H X = a_set_inv X"
⟨proof⟩

```

The coset map is a homomorphism from G to the quotient group $G \text{ Mod } H$

```

lemma (in abelian_subgroup) a_r_coset_hom_A_Mod:
  "(λa. H +> a) ∈ hom (|carrier = carrier G, mult = add G, one = zero G)
  (G A_Mod H)"
⟨proof⟩

```

The isomorphism theorems have been omitted from lifting, at least for now

9.8.7 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 = 0S}"
⟨proof⟩
```

9.8.8 Homomorphisms

```
lemma abelian_group_homI:
  assumes "abelian_group G"
  assumes "abelian_group H"
  assumes a_group_hom: "group_hom (| carrier = carrier G, mult = add
G, one = zero G |)
                                (| carrier = carrier H, mult = add H,
one = zero 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 ⊕G y) = h x ⊕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 = 0H"
⟨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))
  (|carrier = carrier H, mult = add H, one = zero 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 $G \text{ Mod } \text{kernel } G \text{ H } h$ is isomorphic to H .

```
theorem (in abelian_group_hom) A_FactGroup_iso:
  "h ' carrier G = carrier H
  ⇒ (λX. the_elem (h'X)) ∈ (G A_Mod (a_kernel G H h)) ≅
  (| carrier = carrier H, mult = add H, one = zero H |)"
<proof>
```

9.8.9 Cosets

Not everything 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 acar: "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: "y ∈ H +> x"
    and xcarr: "x ∈ carrier G"
  shows "H +> x = H +> y"
⟨proof⟩

lemma (in abelian_subgroup) a_repr_independenceD:
  assumes ycarr: "y ∈ carrier G"
    and repr: "H +> x = H +> y"
  shows "y ∈ H +> x"
⟨proof⟩

lemma (in abelian_subgroup) a_rcosets_carrier:
  "X ∈ a_rcosets H ⇒ X ⊆ carrier G"

```

<proof>

9.8.10 Addition of Subgroups

```
lemma (in abelian_monoid) set_add_closed:
  assumes Acarr: "A  $\subseteq$  carrier G"
  and Bcarr: "B  $\subseteq$  carrier G"
  shows "A  $\langle + \rangle$  B  $\subseteq$  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  $\langle + \rangle$  K) G"
<proof>
```

end

```
theory Ideal
imports Ring AbelCoset
begin
```

10 Ideals

10.1 Definitions

10.1.1 General definition

```
locale ideal = additive_subgroup I R + ring R for I and R (structure) +
  assumes I_l_closed: "[[a  $\in$  I; x  $\in$  carrier R]]  $\implies$  x  $\otimes$  a  $\in$  I"
  and I_r_closed: "[[a  $\in$  I; x  $\in$  carrier R]]  $\implies$  a  $\otimes$  x  $\in$  I"
```

```
sublocale ideal  $\subseteq$  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 ((carrier = carrier R, mult = add R,
one = zero R))"
  and I_l_closed: " $\bigwedge$  a x. [[a  $\in$  I; x  $\in$  carrier R]]  $\implies$  x  $\otimes$  a  $\in$  I"
  and I_r_closed: " $\bigwedge$  a x. [[a  $\in$  I; x  $\in$  carrier R]]  $\implies$  a  $\otimes$  x  $\in$  I"
  shows "ideal I R"
<proof>
```

10.1.2 Ideals Generated by a Subset of carrier R

definition genideal :: "_ \Rightarrow 'a set \Rightarrow 'a set" ("Idl_z_" [80] 79)
 where "genideal R S = Inter {I. ideal I R \wedge S \subseteq I}"

10.1.3 Principal Ideals

locale principalideal = ideal +
 assumes generate: " $\exists i \in$ carrier R. I = Idl {i}"

lemma (in principalideal) is_principalideal: "principalideal I R"
 \langle proof \rangle

lemma principalidealI:
 fixes R (structure)
 assumes "ideal I R"
 and generate: " $\exists i \in$ carrier R. I = Idl {i}"
 shows "principalideal I R"
 \langle proof \rangle

10.1.4 Maximal Ideals

locale maximalideal = ideal +
 assumes I_notcarr: "carrier R \neq I"
 and I_maximal: "[ideal J R; I \subseteq J; J \subseteq carrier R] \implies J = I \vee J = carrier R"

lemma (in maximalideal) is_maximalideal: "maximalideal I R"
 \langle proof \rangle

lemma maximalidealI:
 fixes R
 assumes "ideal I R"
 and I_notcarr: "carrier R \neq I"
 and I_maximal: " $\bigwedge J. [$ ideal J R; I \subseteq J; J \subseteq carrier R] \implies J = I \vee J = carrier R"
 shows "maximalideal I R"
 \langle proof \rangle

10.1.5 Prime Ideals

locale primeideal = ideal + cring +
 assumes I_notcarr: "carrier R \neq I"
 and I_prime: "[a \in carrier R; b \in carrier R; a \otimes b \in I] \implies a \in I \vee b \in I"

lemma (in primeideal) is_primeideal: "primeideal I R"
 \langle proof \rangle

lemma primeidealI:

```

fixes R (structure)
assumes "ideal I R"
  and "cring R"
  and I_notcarr: "carrier R  $\neq$  I"
  and I_prime: " $\bigwedge$ a b.  $\llbracket$ a  $\in$  carrier R; b  $\in$  carrier R; a  $\otimes$  b  $\in$  I $\rrbracket$   $\implies$ 
a  $\in$  I  $\vee$  b  $\in$  I"
  shows "primeideal I R"
<proof>

```

```

lemma primeidealI2:
fixes R (structure)
assumes "additive_subgroup I R"
  and "cring R"
  and I_l_closed: " $\bigwedge$ a x.  $\llbracket$ a  $\in$  I; x  $\in$  carrier R $\rrbracket$   $\implies$  x  $\otimes$  a  $\in$  I"
  and I_r_closed: " $\bigwedge$ a x.  $\llbracket$ a  $\in$  I; x  $\in$  carrier R $\rrbracket$   $\implies$  a  $\otimes$  x  $\in$  I"
  and I_notcarr: "carrier R  $\neq$  I"
  and I_prime: " $\bigwedge$ a b.  $\llbracket$ a  $\in$  carrier R; b  $\in$  carrier R; a  $\otimes$  b  $\in$  I $\rrbracket$   $\implies$ 
a  $\in$  I  $\vee$  b  $\in$  I"
  shows "primeideal I R"
<proof>

```

10.2 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>

```

10.3 General Ideal Properties

```

lemma (in ideal) one_imp_carrier:
  assumes I_one_closed: "1  $\in$  I"
  shows "I = carrier R"
<proof>

```

```

lemma (in ideal) Icarr:
  assumes iI: "i  $\in$  I"
  shows "i  $\in$  carrier R"
<proof>

```

10.4 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  $\cap$  J) R"
<proof>

```

The intersection of any Number of Ideals is again an Ideal in R

```

lemma (in ring) i_Intersect:
  assumes Sideals: " $\bigwedge I. I \in S \implies$  ideal I R"
    and notempty: "S  $\neq$  {}"
  shows "ideal (Inter S) R"
<proof>

```

10.5 Addition of Ideals

```

lemma (in ring) add_ideals:
  assumes idealI: "ideal I R"
    and idealJ: "ideal J R"
  shows "ideal (I  $\langle + \rangle$  J) R"
<proof>

```

10.6 Ideals generated by a subset of carrier R

genideal generates an ideal

```

lemma (in ring) genideal_ideal:
  assumes Scarr: "S  $\subseteq$  carrier R"
  shows "ideal (Idl S) R"
<proof>

```

```

lemma (in ring) genideal_self:
  assumes "S  $\subseteq$  carrier R"
  shows "S  $\subseteq$  Idl S"
<proof>

```

```

lemma (in ring) genideal_self':
  assumes carr: "i  $\in$  carrier R"
  shows "i  $\in$  Idl {i}"
<proof>

```

genideal generates the minimal ideal

```

lemma (in ring) genideal_minimal:
  assumes a: "ideal I R"
    and b: "S  $\subseteq$  I"
  shows "Idl S  $\subseteq$  I"
<proof>

```

Generated ideals and subsets

```

lemma (in ring) Idl_subset_ideal:
  assumes Ideal: "ideal I R"

```

```

    and Hcarr: "H  $\subseteq$  carrier R"
    shows "(Idl H  $\subseteq$  I) = (H  $\subseteq$  I)"
  <proof>

```

```

lemma (in ring) subset_Idl_subset:
  assumes Icarr: "I  $\subseteq$  carrier R"
  and HI: "H  $\subseteq$  I"
  shows "Idl H  $\subseteq$  Idl I"
  <proof>

```

```

lemma (in ring) Idl_subset_ideal':
  assumes acar: "a  $\in$  carrier R" and bcarr: "b  $\in$  carrier R"
  shows "(Idl {a}  $\subseteq$  Idl {b}) = (a  $\in$  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 :: "_  $\Rightarrow$  'a  $\Rightarrow$  'a set" ("PIdl  $\nu$  _" [80] 79)
  where "cgenideal R a = {x  $\otimes_R$  a | x. x  $\in$  carrier R}"

```

genhideal (?) really generates an ideal

```

lemma (in cring) cgenideal_ideal:
  assumes acar: "a  $\in$  carrier R"
  shows "ideal (PIdl a) R"
  <proof>

```

```

lemma (in ring) cgenideal_self:
  assumes icarr: "i  $\in$  carrier R"
  shows "i  $\in$  PIdl i"
  <proof>

```

cgenideal is minimal

```

lemma (in ring) cgenideal_minimal:
  assumes "ideal J R"
  assumes aJ: "a  $\in$  J"
  shows "PIdl a  $\subseteq$  J"
  <proof>

```

```

lemma (in cring) cgenideal_eq_genideal:
  assumes icarr: "i  $\in$  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 icarr: "i ∈ carrier R"
  shows "principalideal (PIdl i) R"
  <proof>
```

10.7 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>
```

10.8 Properties of Principal Ideals

0 generates the zero ideal

```
lemma (in ring) zero_genideal: "Idl {0} = {0}"
  <proof>
```

1 generates the unit ideal

```
lemma (in ring) one_genideal: "Idl {1} = carrier R"
  <proof>
```

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>
```

10.9 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>

```

10.10 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 acar: "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_is_prime:
  assumes "maximalideal I R"
  shows "primeideal I R"
  <proof>

```

10.11 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  $\neq$  {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

```

11 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  $\in$  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  $\subseteq$  ring: ring_hom_ring
  <proof>

```

```

sublocale ring_hom_ring  $\subseteq$  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 = 1S"
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 = 1S"
  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>

```

11.1 The Kernel of a Ring Homomorphism

— the kernel of a ring homomorphism is an ideal

```

lemma (in ring_hom_ring) kernel_is_ideal:
  shows "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 = 0S"
<proof>

```

11.2 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⟩

end

theory QuotRing
imports RingHom
begin

```

12 Quotient Rings

12.1 Multiplication on Cosets

```

definition rcoset_mult :: "[('a, _) ring_scheme, 'a set, 'a set, 'a set]
⇒ 'a set"
  ("[mod _:] _ ⊗ z _" [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:
  "x ∈ carrier R ⇒ y ∈ carrier R ⇒ [mod I:] (I +> x) ⊗ (I +> y)
= I +> (x ⊗ y)"
  ⟨proof⟩

```

12.2 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)"

```

12.3 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: "(op +> I) ∈ ring_hom R (R Quot I)"
⟨proof⟩
```

```
lemma (in ideal) rcos_ring_hom_ring: "ring_hom_ring R (R Quot I) (op
+> 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) (op +> I)"
⟨proof⟩
```

12.4 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) (op +> I)"
⟨proof⟩
```

12.5 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”

```
lemma (in maximalideal) quotient_is_field:
  assumes "cring R"
  shows "field (R Quot I)"
```

<proof>

end

```
theory IntRing
imports QuotRing Lattice Int "~/src/HOL/Old_Number_Theory/Primes"
begin
```

13 The Ring of Integers

13.1 Some properties of int

```
lemma dvd_eq_abseq:
  "(1 dvd k & k dvd 1) = (abs 1 = abs (k::int))"
<proof>
```

13.2 \mathcal{Z} : The Set of Integers as Algebraic Structure

abbreviation

```
int_ring :: "int ring" ("Z") where
  "int_ring == (| carrier = UNIV, mult = op *, one = 1, zero = 0, add
= op + |)"
```

```
lemma int_Zcarr [intro!, simp]:
  "k ∈ carrier Z"
<proof>
```

```
lemma int_is_cring:
  "cring Z"
<proof>
```

13.3 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 Z
  where "carrier Z = UNIV"
        and "mult Z x y = x * y"
        and "one Z = 1"
        and "pow Z x n = x^n"
<proof>
```

```
interpretation int: comm_monoid Z
  where "finprod Z f A = (if finite A then setprod f A else undefined)"
<proof>
```

```

interpretation int: abelian_monoid  $\mathcal{Z}$ 
  where 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 = (if finite A then setsum f A else
undefined)"
  <proof>

```

```

interpretation int: abelian_group  $\mathcal{Z}$ 

  where "carrier  $\mathcal{Z}$  = UNIV"
        and "zero  $\mathcal{Z}$  = 0"
        and "add  $\mathcal{Z}$  x y = x + y"
        and "finsum  $\mathcal{Z}$  f A = (if finite A then setsum f A else undefined)"
        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"  $\mathcal{Z}$ 
  where "carrier  $\mathcal{Z}$  = UNIV"
        and "zero  $\mathcal{Z}$  = 0"
        and "add  $\mathcal{Z}$  x y = x + y"
        and "finsum  $\mathcal{Z}$  f A = (if finite A then setsum f A else undefined)"
        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"
  "(ALL x : UNIV. P x) = (ALL x. P x)"
  "(EX x : UNIV. P x) = (EX x. P x)"
  "(True --> Q) = Q"
  "(True ==> PROP R) == PROP R"
  <proof>

```

```

interpretation int :
  partial_order "(| carrier = UNIV::int set, eq = op =, le = op ≤ |)"
  where "carrier (| carrier = UNIV::int set, eq = op =, le = op ≤ |)
= UNIV"
        and "le (| carrier = UNIV::int set, eq = op =, le = op ≤ |) x y =
(x ≤ y)"
        and "lless (| carrier = UNIV::int set, eq = op =, le = op ≤ |) x
y = (x < y)"
  <proof>

```

```

interpretation int :

```

```

lattice "(| carrier = UNIV::int set, eq = op =, le = op ≤ |)"
  where "join (| carrier = UNIV::int set, eq = op =, le = op ≤ |) x y
= max x y"
  and "meet (| carrier = UNIV::int set, eq = op =, le = op ≤ |) x y
= min x y"
⟨proof⟩

```

```

interpretation int :
  total_order "(| carrier = UNIV::int set, eq = op =, le = op ≤ |)"
  ⟨proof⟩

```

13.4 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: "prime (nat p)"
  shows "primeideal (Idl $\mathcal{Z}$  {p})  $\mathcal{Z}$ "
  ⟨proof⟩

```

13.5 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}) = (abs l = abs k)"
  ⟨proof⟩

```

13.6 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 "EX 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:
  shows "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:
  shows "(ZMod m a = ZMod m b) = (a mod m = b mod m)"
  ⟨proof⟩

13.7 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:
  shows "cring (ZFact k)"
  ⟨proof⟩

lemma ZFact_zero:
  "carrier (ZFact 0) = ( $\bigcup$ a. {a})"
  ⟨proof⟩

lemma ZFact_one:
  "carrier (ZFact 1) = {UNIV}"
  ⟨proof⟩

lemma ZFact_prime_is_domain:
  assumes pprime: "prime (nat p)"

```

```

    shows "domain (ZFact p)"
  <proof>

```

```

end

```

```

theory Module
imports Ring
begin

```

14 Modules over an Abelian Group

14.1 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>

```

14.2 Basic Properties of Algebras

```

lemma (in algebra) smult_l_null [simp]:

```

```

"x ∈ carrier M ==> 0 ⊙M x = 0M"
⟨proof⟩

lemma (in algebra) smult_r_null [simp]:
  "a ∈ carrier R ==> a ⊙M 0M = 0M"
⟨proof⟩

lemma (in algebra) smult_l_minus:
  "[| a ∈ carrier R; x ∈ carrier M |] ==> (⊖a) ⊙M x = ⊖M (a ⊙M x)"
⟨proof⟩

lemma (in algebra) smult_r_minus:
  "[| a ∈ carrier R; x ∈ carrier M |] ==> a ⊙M (⊖Mx) = ⊖M (a ⊙M x)"
⟨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 & (EX n. bound 0R n f)}"
```

definition UP :: "('a, 'm) ring_scheme => ('a, nat => 'a) up_ring"

```
where "UP R = (|
  carrier = up R,
  mult = (%p:up R. %q:up R. %n. ⊕Ri ∈ {...n}. p i ⊗R q (n-i)),
  one = (%i. if i=0 then 1R else 0R),
  zero = (%i. 0R),
  add = (%p:up R. %q:up R. %i. p i ⊕R q i),
  smult = (%a:carrier R. %p:up R. %i. a ⊗R p i),
  monom = (%a:carrier R. %n i. if i=n then a else 0R),
  coeff = (%p:up R. %n. p n) |)"
```

Properties of the set of polynomials up.

lemma mem_upI [intro]:

```
"[| !!n. f n ∈ carrier R; EX 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 ==> EX n. bound 0 n f" *<proof>*

lemma up_one_closed: "(%n. if n = 0 then 1 else 0) ∈ up R" *<proof>*

lemma up_smult_closed: "[| a ∈ carrier R; p ∈ up R |] ==> (%i. a ⊗ p i) ∈ up R" *<proof>*

lemma up_add_closed:

```
"[| p ∈ up R; q ∈ up R |] ==> (%i. p i ⊕ q i) ∈ up R"
<proof>
```

lemma up_a_inv_closed:

```
"p ∈ up R ==> (%i. ⊖ (p i)) ∈ up R"
<proof>
```

lemma up_minus_closed:

```

" [| p ∈ up R; q ∈ up R |] ==> (∑ i. p i ⊖ q i) ∈ up R"
⟨proof⟩

lemma up_mult_closed:
" [| p ∈ up R; q ∈ up R |] ==>
(∑ n. ⊕ 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 0P n = 0" <proof>

lemma coeff_one [simp]: "coeff P 1P 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]:
  "1P ∈ carrier P" <proof>

lemma UP_zero_closed [intro, simp]:
  "0P ∈ 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 "0P ⊕P p = p" <proof>

lemma UP_l_neg_ex:
  assumes R: "p ∈ carrier P"
  shows "EX q : carrier P. q ⊕P p = 0P"
<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 ⊕ 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 cring) cring:

"cring R" *<proof>*

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 (%a. 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 0R 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 ~≠ 0P" and R: "p ∈ carrier
P"
  shows "coeff P p 0 ~≠ 0"
  <proof>
```

lemma lcoeff_nonzero:

```
  assumes neq: "p ~≠ 0P" 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 0p = 0"
<proof>

```

```

lemma deg_one [simp]:
  "deg R 1p = 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  $\neq$  zero R"
  and integral: "!!a b. [| mult R a b = zero R; a  $\in$  carrier R;
    b  $\in$  carrier R |] ==> a = zero R | b = zero R"
shows "domain R"
  <proof>

context UP_domain
begin

lemma UP_one_not_zero:
  "1P  $\neq$  0P"
  <proof>

lemma UP_integral:
  "[| p  $\otimes_P$  q = 0P; p  $\in$  carrier P; q  $\in$  carrier P |] ==> p = 0P | q = 0P"
  <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  $\in$  carrier G"
  <proof>

context ring
begin

theorem diagonal_sum:
  "[| f  $\in$  {.. $n$  +  $m$ ::nat}  $\rightarrow$  carrier R; g  $\in$  {.. $n$  +  $m$ }  $\rightarrow$  carrier R |] ==>
  ( $\bigoplus_{k \in \{..n + m\}} \bigoplus_{i \in \{..k\}} f\ i \otimes g\ (k - i)$ ) =
  ( $\bigoplus_{k \in \{..n + m\}} \bigoplus_{i \in \{..n + m - k\}} f\ k \otimes g\ i$ )"
  <proof>

theorem cauchy_product:
  assumes bf: "bound 0 n f" and bg: "bound 0 m g"
  and Rf: "f  $\in$  {.. $n$ }  $\rightarrow$  carrier R" and Rg: "g  $\in$  {.. $m$ }  $\rightarrow$  carrier R"
  shows " $(\bigoplus_{k \in \{..n + m\}} \bigoplus_{i \in \{..k\}} f\ i \otimes g\ (k - i)) =$ 
   $(\bigoplus_{i \in \{..n\}} f\ i) \otimes (\bigoplus_{i \in \{..m\}} g\ i)$ "

```

<proof>

end

lemma (in UP_ring) const_ring_hom:
 "(%a. 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]:
 "[| finite A; f ∈ A -> carrier R |] ==>
 h (finsum R f A) = finsum S (h o 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 $\mathbf{1} = \mathbf{0}$.

```
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 "EX! Phi. Phi ∈ ring_hom P S ∩ extensional (carrier P) &
    Phi (monom P 1 1) = s &
    (ALL 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) = ⊖_R (coeff P q (deg R r))"
shows "deg R (p ⊕_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 ⊗_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 ≠ 0P"
  shows "∃ q r (k::nat). (q ∈ carrier P) ∧ (r ∈ carrier P) ∧ (lcoeff
g)(^)Rk ⊙P f = g ⊗P q ⊕P r ∧ (r = 0P | 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 1R 1 ⊖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 1R 1 ⊖P monom P a 0) = 1"
  ⟨proof⟩

```

```

lemma deg_nzero_nzero:
  assumes deg_p_nzero: "deg R p ≠ 0"
  shows "p ≠ 0P"
  ⟨proof⟩

```

```

lemma deg_monom_minus:

```

```

    assumes a: "a ∈ carrier R"
    and R_not_trivial: "carrier R ≠ {0}"
    shows "deg R (monom P 1R 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 1R 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 1R
  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 1R 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 1R 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 = op *, one = 1, zero = 0, add
= op + |)"
```

```
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
```

References

- [1] C. Ballarin. *Computer Algebra and Theorem Proving*. PhD thesis, University of Cambridge, 1999. <http://www4.in.tum.de/~ballarin/publications.html>.
- [2] N. Jacobson. *Basic Algebra I*. Freeman, 1985.

- [3] 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.