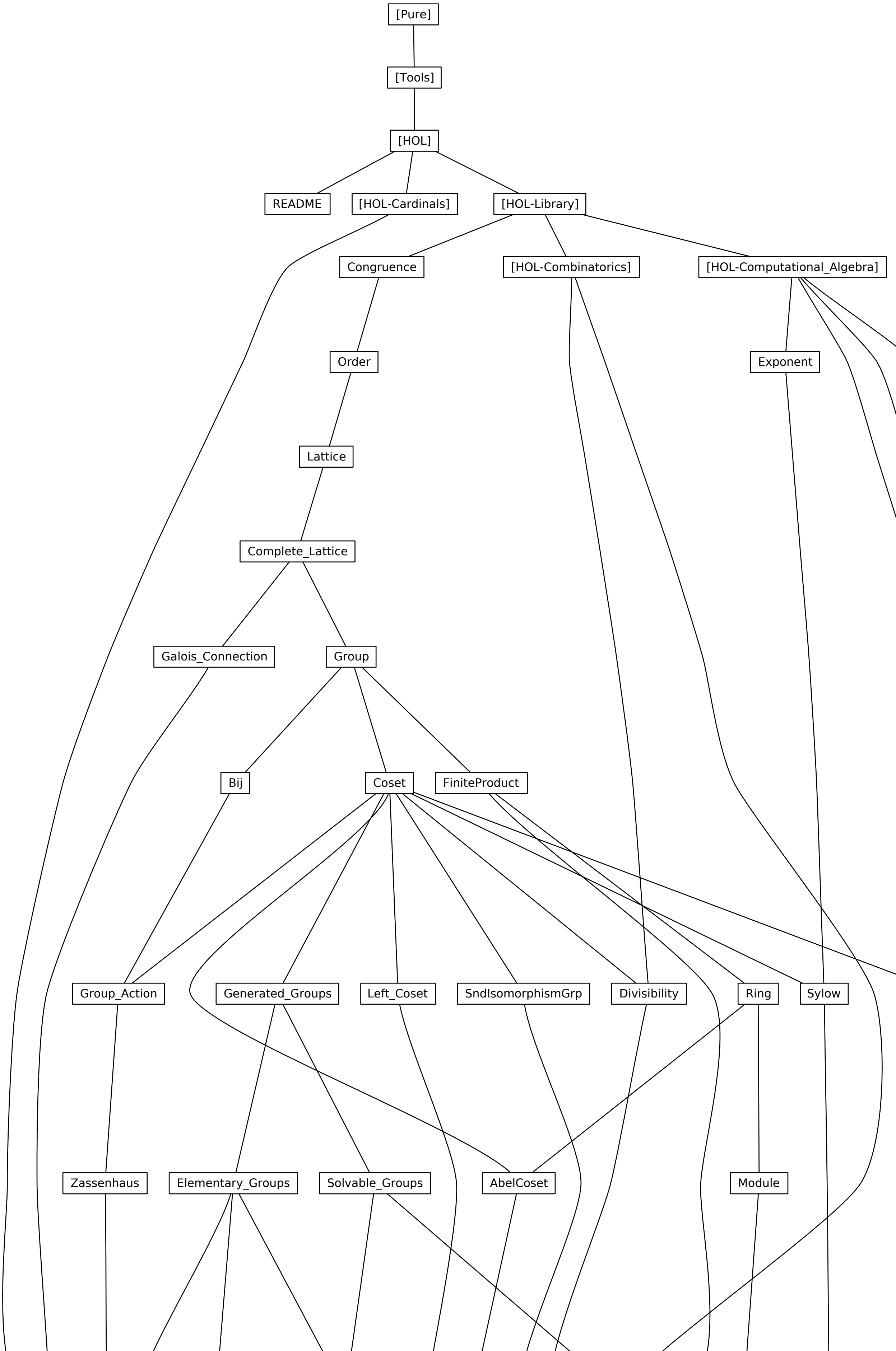
# The Isabelle/HOL Algebra Library

Clemens Ballarin (Editor)

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

March 13, 2025

## Contents


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

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

# 1 Objects

## 1.1 Structure with Carrier Set.

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

lemma funcset_carrier:
  "⟦ f ∈ carrier X → carrier Y; x ∈ carrier X ⟧ ⟹ f x ∈ carrier Y"
  by (fact funcset_mem)

lemma funcset_carrier':
  "⟦ f ∈ carrier A → carrier A; x ∈ carrier A ⟧ ⟹ f x ∈ carrier A"
  by (fact funcset_mem)
```

## 1.2 Structure with Carrier and Equivalence Relation eq

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

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

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

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

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

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

definition
```

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

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

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

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

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

lemma equivalenceI:
  fixes P :: "'a ⇒ 'a ⇒ bool" and E :: "'a set"
  assumes refl: "⋀x.     ⟦ x ∈ E ⟧ ⟹ P x x"
    and    sym: "⋀x y.   ⟦ x ∈ E; y ∈ E ⟧ ⟹ P x y ⟹ P y x"
    and  trans: "⋀x y z. ⟦ x ∈ E; y ∈ E; z ∈ E ⟧ ⟹ P x y ⟹ P y z
⟹ P x z"
  shows "equivalence ⦇ carrier = E, eq = P ⦈"
  unfolding equivalence_def using assms
  by (metis eq_object.select_convs(1) partial_object.select_convs(1))

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

lemma equivalence_subset:
  assumes "equivalence L" "A ⊆ carrier L"
  shows "equivalence (L⦇ carrier := A ⦈)"
proof -
  interpret L: equivalence L
    by (simp add: assms)
  show ?thesis
    by (unfold_locales, simp_all add: L.sym assms rev_subsetD, meson L.trans
assms(2) contra_subsetD)
qed
```

```
lemma elemI:
  fixes R (structure)
  assumes "a' ∈ A" "a .= a'"
  shows "a .∈ A"
  unfolding elem_def using assms by auto

lemma (in equivalence) elem_exact:
  assumes "a ∈ carrier S" "a ∈ A"
  shows "a .∈ A"
  unfolding elem_def using assms by auto

lemma elemE:
  fixes S (structure)
  assumes "a .∈ A"
    and "⋀a'. ⟦a' ∈ A; a .= a'⟧ ⟹ P"
  shows "P"
  using assms unfolding elem_def by auto

lemma (in equivalence) elem_cong_l [trans]:
  assumes "a ∈ carrier S"  "a' ∈ carrier S" "A ⊆ carrier S"
    and "a' .= a" "a .∈ A"
  shows "a' .∈ A"
  using assms by (meson elem_def trans subsetCE)

lemma (in equivalence) elem_subsetD:
  assumes "A ⊆ B" "a .∈ A"
  shows "a .∈ B"
  using assms by (fast intro: elemI elim: elemE dest: subsetD)

lemma (in equivalence) mem_imp_elem [simp, intro]:
  assumes "x ∈ carrier S"
  shows "x ∈ A ⟹ x .∈ A"
  using assms unfolding elem_def by blast

lemma set_eqI:
  fixes R (structure)
  assumes "⋀a. a ∈ A ⟹ a .∈ B"
    and   "⋀b. b ∈ B ⟹ b .∈ A"
  shows "A {.=} B"
  using assms unfolding set_eq_def by auto

lemma set_eqI2:
  fixes R (structure)
  assumes "⋀a. a ∈ A ⟹ ∃b ∈ B. a .= b"
    and   "⋀b. b ∈ B ⟹ ∃a ∈ A. b .= a"
```

```
  shows "A {.=} B"
  using assms by (simp add: set_eqI elem_def)

lemma set_eqD1:
  fixes R (structure)
  assumes "A {.=} A'" and "a ∈ A"
  shows "∃a'∈A'. a .= a'"
  using assms by (simp add: set_eq_def elem_def)

lemma set_eqD2:
  fixes R (structure)
  assumes "A {.=} A'" and "a' ∈ A'"
  shows "∃a∈A. a' .= a"
  using assms by (simp add: set_eq_def elem_def)

lemma set_eqE:
  fixes R (structure)
  assumes "A {.=} B"
    and "⟦ ∀a ∈ A. a .∈ B; ∀b ∈ B. b .∈ A ⟧ ⟹ P"
  shows "P"
  using assms unfolding set_eq_def by blast

lemma set_eqE2:
  fixes R (structure)
  assumes "A {.=} B"
    and "⟦ ∀a ∈ A. ∃b ∈ B. a .= b; ∀b ∈ B. ∃a ∈ A. b .= a ⟧ ⟹ P"
  shows "P"
  using assms unfolding set_eq_def by (simp add: elem_def)

lemma set_eqE':
  fixes R (structure)
  assumes "A {.=} B" "a ∈ A" "b ∈ B"
    and "⋀a' b'. ⟦ a' ∈ A; b' ∈ B ⟧ ⟹ b .= a' ⟹  a .= b' ⟹ P"
  shows "P"
  using assms by (meson set_eqE2)

lemma (in equivalence) eq_elem_cong_r [trans]:
  assumes "A ⊆ carrier S" "A' ⊆ carrier S" "A {.=} A'"
  shows "⟦ a ∈ carrier S ⟧ ⟹ a .∈ A ⟹ a .∈ A'"
  using assms by (meson elemE elem_cong_l set_eqE subset_eq)

lemma (in equivalence) set_eq_sym [sym]:
  assumes "A ⊆ carrier S" "B ⊆ carrier S"
  shows "A {.=} B ⟹ B {.=} A"
  using assms unfolding set_eq_def elem_def by auto

lemma (in equivalence) equal_set_eq_trans [trans]:
  "⟦ A = B; B {.=} C ⟧ ⟹ A {.=} C"
  by simp
```

```
lemma (in equivalence) set_eq_equal_trans [trans]:
  "⟦ A {.=} B; B = C ⟧ ⟹ A {.=} C"
  by simp

lemma (in equivalence) set_eq_trans_aux:
  assumes "A ⊆ carrier S" "B ⊆ carrier S" "C ⊆ carrier S"
    and "A {.=} B" "B {.=} C"
  shows "⋀a. a ∈ A ⟹ a .∈ C"
  using assms by (simp add: eq_elem_cong_r subset_iff)

corollary (in equivalence) set_eq_trans [trans]:
  assumes "A ⊆ carrier S" "B ⊆ carrier S" "C ⊆ carrier S"
    and "A {.=} B" " B {.=} C"
  shows "A {.=} C"
proof (intro set_eqI)
  show "⋀a. a ∈ A ⟹ a .∈ C" using set_eq_trans_aux assms by blast

next
  show "⋀b. b ∈ C ⟹ b .∈ A" using set_eq_trans_aux set_eq_sym assms
by blast
qed

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"
  unfolding eq_is_closed_def eq_closure_of_def elem_def
  using S
  by (blast dest: closed sym)

lemma (in equivalence) closure_of_eq:
  assumes "A ⊆ carrier S" "x ∈ closure_of A"
  shows "⟦ x' ∈ carrier S; x .= x' ⟧ ⟹ x' ∈ closure_of A"
  using assms elem_cong_l sym unfolding eq_closure_of_def by blast

lemma (in equivalence) is_closed_eq [dest]:
  assumes "is_closed A" "x ∈ A"
  shows "⟦ x .= x'; x' ∈ carrier S ⟧ ⟹ x' ∈ A"
  using assms closure_of_eq [where A = A] unfolding eq_is_closed_def
by simp

corollary (in equivalence) is_closed_eq_rev [dest]:
  assumes "is_closed A" "x' ∈ A"
  shows "⟦ x .= x'; x ∈ carrier S ⟧ ⟹ x ∈ A"
  using sym is_closed_eq assms by (meson contra_subsetD eq_is_closed_def)

lemma closure_of_closed [simp, intro]:
  fixes S (structure)
```

```
  shows "closure_of A ⊆ carrier S"
  unfolding eq_closure_of_def by auto

lemma closure_of_memI:
  fixes S (structure)
  assumes "a .∈ A" "a ∈ carrier S"
  shows "a ∈ closure_of A"
  by (simp add: assms eq_closure_of_def)

lemma closure_ofI2:
  fixes S (structure)
  assumes "a .= a'" and "a' ∈ A" and "a ∈ carrier S"
  shows "a ∈ closure_of A"
  by (meson assms closure_of_memI elem_def)

lemma closure_of_memE:
  fixes S (structure)
  assumes "a ∈ closure_of A"
    and "⟦a ∈ carrier S; a .∈ A⟧ ⟹ P"
  shows "P"
  using eq_closure_of_def assms by fastforce

lemma closure_ofE2:
  fixes S (structure)
  assumes "a ∈ closure_of A"
    and "⋀a'. ⟦a ∈ carrier S; a' ∈ A; a .= a'⟧ ⟹ P"
  shows "P"
  using assms by (meson closure_of_memE elemE)


lemma (in partition) equivalence_from_partition:
  "equivalence ⦇ carrier = A, eq = (λx y. y ∈ (THE b. b ∈ B ∧ x ∈ b))⦈"
    unfolding partition_def equivalence_def
proof (auto)
  let ?f = "λx. THE b. b ∈ B ∧ x ∈ b"
  show "⋀x. x ∈ A ⟹ x ∈ ?f x"
    using unique_class by (metis (mono_tags, lifting) theI')
  show "⋀x y. ⟦ x ∈ A; y ∈ A ⟧ ⟹ y ∈ ?f x ⟹ x ∈ ?f y"
    using unique_class by (metis (mono_tags, lifting) the_equality)
  show "⋀x y z. ⟦ x ∈ A; y ∈ A; z ∈ A ⟧ ⟹ y ∈ ?f x ⟹ z ∈ ?f y ⟹
z ∈ ?f x"
    using unique_class by (metis (mono_tags, lifting) the_equality)
qed

lemma (in partition) partition_coverture: "⋃B = A"
  by (meson Sup_le_iff UnionI unique_class incl subsetI subset_antisym)

lemma (in partition) disjoint_union:
  assumes "b1 ∈ B" "b2 ∈ B"
```

```
    and "b1 ∩ b2 ≠ {}"
  shows "b1 = b2"
proof (rule ccontr)
  assume "b1 ≠ b2"
  obtain a where "a ∈ A" "a ∈ b1" "a ∈ b2"
    using assms(2-3) incl by blast
  thus False using unique_class ‹b1 ≠ b2› assms(1) assms(2) by blast
qed

lemma partitionI:
  fixes A :: "'a set" and B :: "('a set) set"
  assumes "⋃B = A"
    and "⋀b1 b2. ⟦ b1 ∈ B; b2 ∈ B ⟧ ⟹ b1 ∩ b2 ≠ {} ⟹ b1 = b2"
  shows "partition A B"
proof
  show "⋀a. a ∈ A ⟹ ∃!b. b ∈ B ∧ a ∈ b"
  proof (rule ccontr)
    fix a assume "a ∈ A" "∄!b. b ∈ B ∧ a ∈ b"
    then obtain b1 b2 where "b1 ∈ B" "a ∈ b1"
                        and "b2 ∈ B" "a ∈ b2" "b1 ≠ b2" using assms(1)
by blast
    thus False using assms(2) by blast
  qed
next
  show "⋀b. b ∈ B ⟹ b ⊆ A" using assms(1) by blast
qed

lemma (in partition) remove_elem:
  assumes "b ∈ B"
  shows "partition (A - b) (B - {b})"
proof
  show "⋀a. a ∈ A - b ⟹ ∃!b'. b' ∈ B - {b} ∧ a ∈ b'"
    using unique_class by fastforce
next
  show "⋀b'. b' ∈ B - {b} ⟹ b' ⊆ A - b"
    using assms unique_class incl partition_axioms partition_coverture
by fastforce
qed

lemma disjoint_sum:
  "⟦ finite B; finite A; partition A B⟧ ⟹ (∑b∈B. ∑a∈b. f a) = (∑a∈A.
f a)"
proof (induct arbitrary: A set: finite)
  case empty thus ?case using partition.unique_class by fastforce
next
  case (insert b B')
  have "(∑b∈(insert b B'). ∑a∈b. f a) = (∑a∈b. f a) + (∑b∈B'. ∑a∈b.
f a)"
    by (simp add: insert.hyps(1) insert.hyps(2))
```

```
  also have "... = (∑a∈b. f a) + (∑a∈(A - b). f a)"
    using partition.remove_elem[of A "insert b B'" b] insert.hyps insert.prems
    by (metis Diff_insert_absorb finite_Diff insert_iff)
  finally show "(∑b∈(insert b B'). ∑a∈b. f a) = (∑a∈A. f a)"
    using partition.remove_elem[of A "insert b B'" b] insert.prems
    by (metis add.commute insert_iff partition.incl sum.subset_diff)
qed

lemma (in partition) disjoint_sum:
  assumes "finite A"
  shows "(∑b∈B. ∑a∈b. f a) = (∑a∈A. f a)"
proof -
  have "finite B"
    by (simp add: assms finite_UnionD partition_coverture)
  thus ?thesis using disjoint_sum assms partition_axioms by blast
qed

lemma (in equivalence) set_eq_insert_aux:
  assumes "A ⊆ carrier S"
    and "x ∈ carrier S" "x' ∈ carrier S" "x .= x'"
    and "y ∈ insert x A"
  shows "y .∈ insert x' A"
  by (metis assms(1) assms(4) assms(5) contra_subsetD elemI elem_exact
insert_iff)

corollary (in equivalence) set_eq_insert:
  assumes "A ⊆ carrier S"
    and "x ∈ carrier S" "x' ∈ carrier S" "x .= x'"
  shows "insert x A {.=} insert x' A"
  by (meson set_eqI assms set_eq_insert_aux sym equivalence_axioms)

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}"
  using assms set_eq_insert by simp

lemma (in equivalence) closure_inclusion:
  assumes "A ⊆ B"
  shows "closure_of A ⊆ closure_of B"
  unfolding eq_closure_of_def using assms elem_subsetD by auto

lemma (in equivalence) classes_small:
  assumes "is_closed B"
    and "A ⊆ B"
  shows "closure_of A ⊆ B"
  by (metis assms closure_inclusion eq_is_closed_def)

lemma (in equivalence) classes_eq:
```

```
  assumes "A ⊆ carrier S"
  shows "A {.=} closure_of A"
  using assms by (blast intro: set_eqI elem_exact closure_of_memI elim:
closure_of_memE)

lemma (in equivalence) complete_classes:
  assumes "is_closed A"
  shows "A = closure_of A"
  using assms by (simp add: eq_is_closed_def)

lemma (in equivalence) closure_idem_weak:
  "closure_of (closure_of A) {.=} closure_of A"
  by (simp add: classes_eq set_eq_sym)

lemma (in equivalence) closure_idem_strong:
  assumes "A ⊆ carrier S"
  shows "closure_of (closure_of A) = closure_of A"
  using assms closure_of_eq complete_classes is_closedI by auto

lemma (in equivalence) classes_consistent:
  assumes "A ⊆ carrier S"
  shows "is_closed (closure_of A)"
  using closure_idem_strong by (simp add: assms eq_is_closed_def)

lemma (in equivalence) classes_coverture:
  "⋃classes = carrier S"
proof
  show "⋃classes ⊆ carrier S"
    unfolding eq_classes_def eq_class_of_def by blast
next
  show "carrier S ⊆ ⋃classes" unfolding eq_classes_def eq_class_of_def
  proof
    fix x assume "x ∈ carrier S"
    hence "x ∈ {y ∈ carrier S. x .= y}" using refl by simp
    thus "x ∈ ⋃{{y ∈ carrier S. x .= y} |x. x ∈ carrier S}" by blast
  qed
qed

lemma (in equivalence) disjoint_union:
  assumes "class1 ∈ classes" "class2 ∈ classes"
    and "class1 ∩ class2 ≠ {}"
    shows "class1 = class2"
proof -
  obtain x y where x: "x ∈ carrier S" "class1 = class_of x"
               and y: "y ∈ carrier S" "class2 = class_of y"
    using assms(1-2) unfolding eq_classes_def by blast
  obtain z   where z: "z ∈ carrier S" "z ∈ class1 ∩ class2"
    using assms classes_coverture by fastforce
  hence "x .= z ∧ y .= z" using x y unfolding eq_class_of_def by blast
```

```
  hence "x .= y" using x y z trans sym by meson
  hence "class_of x = class_of y"
    unfolding eq_class_of_def using local.sym trans x y by blast
  thus ?thesis using x y by simp
qed

lemma (in equivalence) partition_from_equivalence:
  "partition (carrier S) classes"
proof (intro partitionI)
  show "⋃classes = carrier S" using classes_coverture by simp
next
  show "⋀class1 class2. ⟦ class1 ∈ classes; class2 ∈ classes ⟧ ⟹
                          class1 ∩ class2 ≠ {} ⟹ class1 = class2"
    using disjoint_union by simp
qed

lemma (in equivalence) disjoint_sum:
  assumes "finite (carrier S)"
  shows "(∑c∈classes. ∑x∈c. f x) = (∑x∈(carrier S). f x)"
proof -
  have "finite classes"
    unfolding eq_classes_def using assms by auto
  thus ?thesis using disjoint_sum assms partition_from_equivalence by
blast
qed

end


theory Order
  imports
    Congruence
begin
```

# 2 Orders

## 2.1 Partial Orders

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

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

lemma inv_gorder_inv:
  "inv_gorder (inv_gorder L) = L"
```

```
  by simp

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

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

### 2.1.1 The order relation

```
context weak_partial_order
begin

lemma le_cong_l [intro, trans]:
  "⟦x .= y; y ⊑ z; x ∈ carrier L; y ∈ carrier L; z ∈ carrier L⟧ ⟹ x
⊑ z"
  by (auto intro: le_cong [THEN iffD2])

lemma le_cong_r [intro, trans]:
  "⟦x ⊑ y; y .= z; x ∈ carrier L; y ∈ carrier L; z ∈ carrier L⟧ ⟹ x
⊑ z"
  by (auto intro: le_cong [THEN iffD1])

lemma weak_refl [intro, simp]: "⟦x .= y; x ∈ carrier L; y ∈ carrier
L⟧ ⟹ x ⊑ y"
  by (simp add: le_cong_l)

end

lemma weak_llessI:
  fixes R (structure)
  assumes "x ⊑ y" and "¬(x .= y)"
  shows "x ⊏ y"
  using assms unfolding lless_def by simp

lemma lless_imp_le:
  fixes R (structure)
  assumes "x ⊏ y"
```

```
  shows "x ⊑ y"
  using assms unfolding lless_def by simp

lemma weak_lless_imp_not_eq:
  fixes R (structure)
  assumes "x ⊏ y"
  shows "¬ (x .= y)"
  using assms unfolding lless_def by simp

lemma weak_llessE:
  fixes R (structure)
  assumes p: "x ⊏ y" and e: "⟦x ⊑ y; ¬ (x .= y)⟧ ⟹ P"
  shows "P"
  using p by (blast dest: lless_imp_le weak_lless_imp_not_eq e)

lemma (in weak_partial_order) lless_cong_l [trans]:
  assumes xx': "x .= x'"
    and xy: "x' ⊏ y"
    and carr: "x ∈ carrier L" "x' ∈ carrier L" "y ∈ carrier L"
  shows "x ⊏ y"
  using assms unfolding lless_def by (auto intro: trans sym)

lemma (in weak_partial_order) lless_cong_r [trans]:
  assumes xy: "x ⊏ y"
    and  yy': "y .= y'"
    and carr: "x ∈ carrier L" "y ∈ carrier L" "y' ∈ carrier L"
  shows "x ⊏ y'"
  using assms unfolding lless_def by (auto intro: trans sym)


lemma (in weak_partial_order) lless_antisym:
  assumes "a ∈ carrier L" "b ∈ carrier L"
    and "a ⊏ b" "b ⊏ a"
  shows "P"
  using assms
  by (elim weak_llessE) auto

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"
  using assms unfolding lless_def by (blast dest: le_trans intro: sym)

lemma weak_partial_order_subset:
  assumes "weak_partial_order L" "A ⊆ carrier L"
  shows "weak_partial_order (L⦇ carrier := A ⦈)"
proof -
  interpret L: weak_partial_order L
    by (simp add: assms)
```

```
  interpret equivalence "(L⦇ carrier := A ⦈)"
    by (simp add: L.equivalence_axioms assms(2) equivalence_subset)
  show ?thesis
    apply (unfold_locales, simp_all)
    using assms(2) apply auto[1]
    using assms(2) apply auto[1]
    apply (meson L.le_trans assms(2) contra_subsetD)
    apply (meson L.le_cong assms(2) subsetCE)
  done
qed
```

### 2.1.2 Upper and lower bounds of a set

```
definition
  Upper :: "[_, 'a set] => 'a set"
  where "Upper L A = {u. (∀x. x ∈ A ∩ carrier L ⟶ x ⊑L u)} ∩ carrier
L"

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

lemma Lower_dual [simp]:
  "Lower (inv_gorder L) A = Upper L A"
  by (simp add:Upper_def Lower_def)

lemma Upper_dual [simp]:
  "Upper (inv_gorder L) A = Lower L A"
  by (simp add:Upper_def Lower_def)

lemma (in weak_partial_order) equivalence_dual: "equivalence (inv_gorder
L)"
  by (rule equivalence.intro) (auto simp: intro: sym trans)

lemma  (in weak_partial_order) dual_weak_order: "weak_partial_order (inv_gorder
L)"
  by intro_locales (auto simp add: weak_partial_order_axioms_def le_cong
intro: equivalence_dual le_trans)

lemma (in weak_partial_order) dual_eq_iff [simp]: "A {.=}inv_gorder L
A' ⟷ A {.=} A'"
  by (auto simp: set_eq_def elem_def)

lemma dual_weak_order_iff:
  "weak_partial_order (inv_gorder A) ⟷ weak_partial_order A"
proof
  assume "weak_partial_order (inv_gorder A)"
  then interpret dpo: weak_partial_order "inv_gorder A"
```

```
  rewrites "carrier (inv_gorder A) = carrier A"
  and   "le (inv_gorder A)      = (λ x y. le A y x)"
  and   "eq (inv_gorder A)      = eq A"
    by (simp_all)
  show "weak_partial_order A"
    by (unfold_locales, auto intro: dpo.sym dpo.trans dpo.le_trans)
next
  assume "weak_partial_order A"
  thus "weak_partial_order (inv_gorder A)"
    by (metis weak_partial_order.dual_weak_order)
qed

lemma Upper_closed [iff]:
  "Upper L A ⊆ carrier L"
  by (unfold Upper_def) clarify

lemma Upper_memD [dest]:
  fixes L (structure)
  shows "⟦u ∈ Upper L A; x ∈ A; A ⊆ carrier L⟧ ⟹ x ⊑ u ∧ u ∈ carrier
L"
  by (unfold Upper_def) blast

lemma (in weak_partial_order) Upper_elemD [dest]:
  "⟦u .∈ Upper L A; u ∈ carrier L; x ∈ A; A ⊆ carrier L⟧ ⟹ x ⊑ u"
  unfolding Upper_def elem_def
  by (blast dest: sym)

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

lemma (in weak_partial_order) Upper_elemI:
  "⟦!! y. y ∈ A ⟹ y ⊑ x; x ∈ carrier L⟧ ⟹ x .∈ Upper L A"
  unfolding Upper_def by blast

lemma Upper_antimono:
  "A ⊆ B ⟹ Upper L B ⊆ Upper L A"
  by (unfold Upper_def) blast

lemma (in weak_partial_order) Upper_is_closed [simp]:
  "A ⊆ carrier L ⟹ is_closed (Upper L A)"
  by (rule is_closedI) (blast intro: Upper_memI)+

lemma (in weak_partial_order) Upper_mem_cong:
  assumes  "a' ∈ carrier L" "A ⊆ carrier L" "a .= a'" "a ∈ Upper L A"
  shows "a' ∈ Upper L A"
  by (metis assms Upper_closed Upper_is_closed closure_of_eq complete_classes)
```

```
lemma (in weak_partial_order) Upper_semi_cong:
  assumes "A ⊆ carrier L" "A {.=} A'"
  shows "Upper L A ⊆ Upper L A'"
  unfolding Upper_def
   by clarsimp (meson assms equivalence.refl equivalence_axioms le_cong
set_eqD2 subset_eq)

lemma (in weak_partial_order) Upper_cong:
  assumes "A ⊆ carrier L" "A' ⊆ carrier L" "A {.=} A'"
  shows "Upper L A = Upper L A'"
  using assms by (simp add: Upper_semi_cong set_eq_sym subset_antisym)

lemma Lower_closed [intro!, simp]:
  "Lower L A ⊆ carrier L"
  by (unfold Lower_def) clarify

lemma Lower_memD [dest]:
  fixes L (structure)
  shows "⟦l ∈ Lower L A; x ∈ A; A ⊆ carrier L⟧ ⟹ l ⊑ x ∧ l ∈ carrier
L"
  by (unfold Lower_def) blast

lemma Lower_memI:
  fixes L (structure)
  shows "⟦!! y. y ∈ A ⟹ x ⊑ y; x ∈ carrier L⟧ ⟹ x ∈ Lower L A"
  by (unfold Lower_def) blast

lemma Lower_antimono:
  "A ⊆ B ⟹ Lower L B ⊆ Lower L A"
  by (unfold Lower_def) blast

lemma (in weak_partial_order) Lower_is_closed [simp]:
  "A ⊆ carrier L ⟹ is_closed (Lower L A)"
  by (rule is_closedI) (blast intro: Lower_memI dest: sym)+

lemma (in weak_partial_order) Lower_mem_cong:
  assumes "a' ∈ carrier L"  "A ⊆ carrier L" "a .= a'" "a ∈ Lower L A"
  shows "a' ∈ Lower L A"
  by (meson assms Lower_closed Lower_is_closed is_closed_eq subsetCE)

lemma (in weak_partial_order) Lower_cong:
  assumes "A ⊆ carrier L" "A' ⊆ carrier L" "A {.=} A'"
  shows "Lower L A = Lower L A'"
  unfolding Upper_dual [symmetric]
  by (rule weak_partial_order.Upper_cong [OF dual_weak_order]) (simp_all
add: assms)
```

Jacobson: Theorem 8.1

```
lemma Lower_empty [simp]:
```

```
  "Lower L {} = carrier L"
  by (unfold Lower_def) simp

lemma Upper_empty [simp]:
  "Upper L {} = carrier L"
  by (unfold Upper_def) simp
```

### 2.1.3 Least and greatest, as predicate

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

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

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

```
lemma least_dual [simp]:
  "least (inv_gorder L) x A = greatest L x A"
  by (simp add:least_def greatest_def)

lemma greatest_dual [simp]:
  "greatest (inv_gorder L) x A = least L x A"
  by (simp add:least_def greatest_def)

lemma least_closed [intro, simp]:
  "least L l A ⟹ l ∈ carrier L"
  by (unfold least_def) fast

lemma least_mem:
  "least L l A ⟹ l ∈ A"
  by (unfold least_def) fast

lemma (in weak_partial_order) weak_least_unique:
  "⟦least L x A; least L y A⟧ ⟹ x .= y"
  by (unfold least_def) blast

lemma least_le:
  fixes L (structure)
  shows "⟦least L x A; a ∈ A⟧ ⟹ x ⊑ a"
  by (unfold least_def) fast

lemma (in weak_partial_order) least_cong:
  "⟦x .= x'; x ∈ carrier L; x' ∈ carrier L; is_closed A⟧ ⟹ least L x
A = least L x' A"
  unfolding least_def
  by (meson is_closed_eq is_closed_eq_rev le_cong local.refl subset_iff)
```

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

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

```
lemma (in weak_partial_order) least_Upper_cong_l:
  assumes "x .= x'"
    and "x ∈ carrier L" "x' ∈ carrier L"
    and "A ⊆ carrier L"
  shows "least L x (Upper L A) = least L x' (Upper L A)"
  apply (rule least_cong) using assms by auto

lemma (in weak_partial_order) least_Upper_cong_r:
  assumes "A ⊆ carrier L" "A' ⊆ carrier L" "A {.=} A'"
  shows "least L x (Upper L A) = least L x (Upper L A')"
  using Upper_cong assms by auto

lemma least_UpperI:
  fixes L (structure)
  assumes above: "!! x. x ∈ A ⟹ x ⊑ s"
    and below: "!! y. y ∈ Upper L A ⟹ s ⊑ y"
    and L: "A ⊆ carrier L"  "s ∈ carrier L"
  shows "least L s (Upper L A)"
proof -
  have "Upper L A ⊆ carrier L" by simp
  moreover from above L have "s ∈ Upper L A" by (simp add: Upper_def)
  moreover from below have "∀x ∈ Upper L A. s ⊑ x" by fast
  ultimately show ?thesis by (simp add: least_def)
qed

lemma least_Upper_above:
  fixes L (structure)
  shows "⟦least L s (Upper L A); x ∈ A; A ⊆ carrier L⟧ ⟹ x ⊑ s"
  by (unfold least_def) blast

lemma greatest_closed [intro, simp]:
  "greatest L l A ⟹ l ∈ carrier L"
  by (unfold greatest_def) fast

lemma greatest_mem:
  "greatest L l A ⟹ l ∈ A"
  by (unfold greatest_def) fast

lemma (in weak_partial_order) weak_greatest_unique:
  "⟦greatest L x A; greatest L y A⟧ ⟹ x .= y"
  by (unfold greatest_def) blast

lemma greatest_le:
  fixes L (structure)
```

```
  shows "⟦greatest L x A; a ∈ A⟧ ⟹ a ⊑ x"
  by (unfold greatest_def) fast

lemma (in weak_partial_order) greatest_cong:
  "⟦x .= x'; x ∈ carrier L; x' ∈ carrier L; is_closed A⟧ ⟹
  greatest L x A = greatest L x' A"
  unfolding greatest_def
  by (meson is_closed_eq_rev le_cong_r local.sym subset_eq)

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

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

```
lemma (in weak_partial_order) greatest_Lower_cong_l:
  assumes "x .= x'"
    and "x ∈ carrier L" "x' ∈ carrier L"
  shows "greatest L x (Lower L A) = greatest L x' (Lower L A)"
proof -
  have "∀A. is_closed (Lower L (A ∩ carrier L))"
    by simp
  then show ?thesis
    by (simp add: Lower_def assms greatest_cong)
qed

lemma (in weak_partial_order) greatest_Lower_cong_r:
  assumes "A ⊆ carrier L" "A' ⊆ carrier L" "A {.=} A'"
  shows "greatest L x (Lower L A) = greatest L x (Lower L A')"
  using Lower_cong assms by auto

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 -
  have "Lower L A ⊆ carrier L" by simp
  moreover from below L have "i ∈ Lower L A" by (simp add: Lower_def)
  moreover from above have "∀x ∈ Lower L A. x ⊑ i" by fast
  ultimately show ?thesis by (simp add: greatest_def)
qed

lemma greatest_Lower_below:
  fixes L (structure)
  shows "⟦greatest L i (Lower L A); x ∈ A; A ⊆ carrier L⟧ ⟹ i ⊑ x"
  by (unfold greatest_def) blast
```

### 2.1.4 Intervals

```
definition
  at_least_at_most :: "('a, 'c) gorder_scheme ⇒ 'a => 'a => 'a set"
    (‹(‹indent=1 notation=‹mixfix interval›› ⦃_.._⦄ı)›)
  where "⦃l..u⦄_A = {x ∈ carrier A. l ⊑_A x ∧ x ⊑_A u}"

context weak_partial_order
begin

  lemma at_least_at_most_upper [dest]:
    "x ∈ ⦃a..b⦄ ⟹ x ⊑ b"
    by (simp add: at_least_at_most_def)

  lemma at_least_at_most_lower [dest]:
    "x ∈ ⦃a..b⦄ ⟹ a ⊑ x"
    by (simp add: at_least_at_most_def)

  lemma at_least_at_most_closed: "⦃a..b⦄ ⊆ carrier L"
    by (auto simp add: at_least_at_most_def)

  lemma at_least_at_most_member [intro]:
    "⟦x ∈ carrier L; a ⊑ x; x ⊑ b⟧ ⟹ x ∈ ⦃a..b⦄"
    by (simp add: at_least_at_most_def)

end
```

### 2.1.5 Isotone functions

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

lemma isotoneI [intro?]:
  fixes f :: "'a ⇒ 'b"
  assumes "weak_partial_order L1"
          "weak_partial_order L2"
          "(⋀x y. ⟦x ∈ carrier L1; y ∈ carrier L1; x ⊑_L1 y⟧
                   ⟹ f x ⊑_L2 f y)"
  shows "isotone L1 L2 f"
  using assms by (auto simp add:isotone_def)

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

lemma use_iso1:
```

```
  "⟦isotone A A f; x ∈ carrier A; y ∈ carrier A; x ⊑A y⟧ ⟹
   f x ⊑A f y"
  by (simp add: isotone_def)
```

**lemma** use_iso2:
```
  "⟦isotone A B f; x ∈ carrier A; y ∈ carrier A; x ⊑A y⟧ ⟹
   f x ⊑B f y"
  by (simp add: isotone_def)
```

**lemma** iso_compose:
```
  "⟦f ∈ carrier A → carrier B; isotone A B f; g ∈ carrier B → carrier
C; isotone B C g⟧ ⟹
   isotone A C (g ∘ f)"
  by (simp add: isotone_def, safe, metis Pi_iff)
```

**lemma (in** weak_partial_order) inv_isotone [simp]:
```
  "isotone (inv_gorder A) (inv_gorder B) f = isotone A B f"
  by (auto simp add:isotone_def dual_weak_order dual_weak_order_iff)
```

### 2.1.6 Idempotent functions

**definition** idempotent ::
```
  "('a, 'b) gorder_scheme ⇒ ('a ⇒ 'a) ⇒ bool"
    (‹(‹open_block notation=‹prefix Idem› ›Idemı)›)
  where "IdemL f ≡ ∀x∈carrier L. f (f x) .=L f x"
```

**lemma (in** weak_partial_order) idempotent:
```
  "⟦Idem f; x ∈ carrier L⟧ ⟹ f (f x) .= f x"
  by (auto simp add: idempotent_def)
```

### 2.1.7 Order embeddings

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

**lemma** order_emb_isotone: "order_emb A B f ⟹ isotone A B f"
```
  by (auto simp add: isotone_def order_emb_def)
```

### 2.1.8 Commuting functions

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

## 2.2 Partial orders where eq is the Equality

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

declare weak_le_antisym [rule del]

lemma le_antisym [intro]:
  "⟦x ⊑ y; y ⊑ x; x ∈ carrier L; y ∈ carrier L⟧ ⟹ x = y"
  using weak_le_antisym unfolding eq_is_equal .

lemma lless_eq:
  "x ⊏ y ⟷ x ⊑ y ∧ x ≠ y"
  unfolding lless_def by (simp add: eq_is_equal)

lemma set_eq_is_eq: "A {.=} B ⟷ A = B"
  by (auto simp add: set_eq_def elem_def eq_is_equal)

end

lemma (in partial_order) dual_order:
  "partial_order (inv_gorder L)"
proof -
  interpret dwo: weak_partial_order "inv_gorder L"
    by (metis dual_weak_order)
  show ?thesis
    by (unfold_locales, simp add:eq_is_equal)
qed

lemma dual_order_iff:
  "partial_order (inv_gorder A) ⟷ partial_order A"
proof
  assume assm:"partial_order (inv_gorder A)"
  then interpret po: partial_order "inv_gorder A"
  rewrites "carrier (inv_gorder A) = carrier A"
  and   "le (inv_gorder A)      = (λ x y. le A y x)"
  and   "eq (inv_gorder A)      = eq A"
    by (simp_all)
  show "partial_order A"
    apply (unfold_locales, simp_all add: po.sym)
    apply (metis po.trans)
    apply (metis po.weak_le_antisym, metis po.le_trans)
    apply (metis (full_types) po.eq_is_equal, metis po.eq_is_equal)
  done
next
  assume "partial_order A"
  thus "partial_order (inv_gorder A)"
    by (metis partial_order.dual_order)
qed
```

Least and greatest, as predicate

```
lemma (in partial_order) least_unique:
  "⟦least L x A; least L y A⟧ ⟹ x = y"
  using weak_least_unique unfolding eq_is_equal .

lemma (in partial_order) greatest_unique:
  "⟦greatest L x A; greatest L y A⟧ ⟹ x = y"
  using weak_greatest_unique unfolding eq_is_equal .
```

## 2.3 Bounded Orders

```
definition
  top :: "_ => 'a" ("⊤ı") where
  "⊤L = (SOME x. greatest L x (carrier L))"

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

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

lemma bottom_least: "least L ⊥ (carrier L)"
proof -
  obtain x where "least L x (carrier L)"
    by (metis bottom_exists)

  thus ?thesis
    by (auto intro:someI2 simp add: bottom_def)
qed

lemma bottom_closed [simp, intro]:
  "⊥ ∈ carrier L"
  by (metis bottom_least least_mem)

lemma bottom_lower [simp, intro]:
  "x ∈ carrier L ⟹ ⊥ ⊑ x"
  by (metis bottom_least least_le)

end

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

lemma top_greatest: "greatest L ⊤ (carrier L)"
```

```
proof -
  obtain x where "greatest L x (carrier L)"
    by (metis top_exists)

  thus ?thesis
    by (auto intro:someI2 simp add: top_def)
qed

lemma top_closed [simp, intro]:
  "⊤ ∈ carrier L"
  by (metis greatest_mem top_greatest)

lemma top_higher [simp, intro]:
  "x ∈ carrier L ⟹ x ⊑ ⊤"
  by (metis greatest_le top_greatest)

end
```

## 2.4 Total Orders

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

Introduction rule: the usual definition of total order

```
lemma (in weak_partial_order) weak_total_orderI:
  assumes total: "!!x y. ⟦x ∈ carrier L; y ∈ carrier L⟧ ⟹ x ⊑ y ∨ y
⊑ x"
  shows "weak_total_order L"
  by unfold_locales (rule total)
```

## 2.5 Total orders where `eq` is the Equality

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

sublocale total_order < weak?: weak_total_order
  by unfold_locales (rule total_order_total)
```

Introduction rule: the usual definition of total order

```
lemma (in partial_order) total_orderI:
  assumes total: "!!x y. ⟦x ∈ carrier L; y ∈ carrier L⟧ ⟹ x ⊑ y ∨ y
⊑ x"
  shows "total_order L"
  by unfold_locales (rule total)

end
```

**theory** Lattice
**imports** Order
**begin**

# 3 Lattices

## 3.1 Supremum and infimum

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

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

**definition** supr ::
  "('a, 'b) gorder_scheme ⇒ 'c set ⇒ ('c ⇒ 'a) ⇒ 'a "
  **where** "supr L A f = $\bigsqcup_L$(f ` A)"

**definition** infi ::
  "('a, 'b) gorder_scheme ⇒ 'c set ⇒ ('c ⇒ 'a) ⇒ 'a "
  **where** "infi L A f = $\bigsqcap_L$(f ` A)"

**syntax**
  "_inf1"     :: "('a, 'b) gorder_scheme ⇒ pttrns ⇒ 'a ⇒ 'a"
    (‹(‹indent=3 notation=‹binder IINF››IINFı _./ _)› [0, 10] 10)
  "_inf"      :: "('a, 'b) gorder_scheme ⇒ pttrn ⇒ 'c set ⇒ 'a ⇒ 'a"
    (‹(‹indent=3 notation=‹binder IINF››IINFı _:_./ _)› [0, 0, 10] 10)
  "_sup1"     :: "('a, 'b) gorder_scheme ⇒ pttrns ⇒ 'a ⇒ 'a"
    (‹(‹indent=3 notation=‹binder SSUP››SSUPı _./ _)› [0, 10] 10)
  "_sup"      :: "('a, 'b) gorder_scheme ⇒ pttrn ⇒ 'c set ⇒ 'a ⇒ 'a"
    (‹(‹indent=3 notation=‹binder SSUP››SSUPı _:_./ _)› [0, 0, 10] 10)
**syntax_consts**
  "_inf1" "_inf" == infi **and**
  "_sup1" "_sup" == supr
**translations**
  "$\mathrm{IINF}_L$ x. B"     == "CONST infi L CONST UNIV (%x. B)"
  "$\mathrm{IINF}_L$ x:A. B"   == "CONST infi L A (%x. B)"
  "$\mathrm{SSUP}_L$ x. B"     == "CONST supr L CONST UNIV (%x. B)"
  "$\mathrm{SSUP}_L$ x:A. B"   == "CONST supr L A (%x. B)"

**definition**
  join :: "[_, 'a, 'a] => 'a" (**infixl** ‹⊔ı› 65)
  **where** "x $\sqcup_L$ y = $\bigsqcup_L${x, y}"

**definition**

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

**definition**

```
  LEAST_FP :: "('a, 'b) gorder_scheme ⇒ ('a ⇒ 'a) ⇒ 'a" (‹LFPı›) where
  "LEAST_FP L f = ⨅L {u ∈ carrier L. f u ⊑L u}"     — least fixed point
```

**definition**

```
  GREATEST_FP:: "('a, 'b) gorder_scheme ⇒ ('a ⇒ 'a) ⇒ 'a" (‹GFPı›)
where
  "GREATEST_FP L f = ⨆L {u ∈ carrier L. u ⊑L f u}"     — greatest fixed
point
```

## 3.2 Dual operators

```
lemma sup_dual [simp]:
  "⨆inv_gorder LA = ⨅LA"
  by (simp add: sup_def inf_def)

lemma inf_dual [simp]:
  "⨅inv_gorder LA = ⨆LA"
  by (simp add: sup_def inf_def)

lemma join_dual [simp]:
  "p ⊔inv_gorder L q = p ⊓L q"
  by (simp add:join_def meet_def)

lemma meet_dual [simp]:
  "p ⊓inv_gorder L q = p ⊔L q"
  by (simp add:join_def meet_def)

lemma top_dual [simp]:
  "⊤inv_gorder L = ⊥L"
  by (simp add: top_def bottom_def)

lemma bottom_dual [simp]:
  "⊥inv_gorder L = ⊤L"
  by (simp add: top_def bottom_def)

lemma LFP_dual [simp]:
  "LEAST_FP (inv_gorder L) f = GREATEST_FP L f"
  by (simp add:LEAST_FP_def GREATEST_FP_def)

lemma GFP_dual [simp]:
  "GREATEST_FP (inv_gorder L) f = LEAST_FP L f"
  by (simp add:LEAST_FP_def GREATEST_FP_def)
```

## 3.3 Lattices

```
locale weak_upper_semilattice = weak_partial_order +
```

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

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

locale weak_lattice = weak_upper_semilattice + weak_lower_semilattice

lemma (in weak_lattice) dual_weak_lattice:
  "weak_lattice (inv_gorder L)"
proof -
  interpret dual: weak_partial_order "inv_gorder L"
    by (metis dual_weak_order)
  show ?thesis
  proof qed (simp_all add: inf_of_two_exists sup_of_two_exists)
qed
```

### 3.3.1 Supremum

```
lemma (in weak_upper_semilattice) joinI:
  "[| !!l. least L l (Upper L {x, y}) ==> P l; x ∈ carrier L; y ∈ carrier
L |]
  ==> P (x ⊔ y)"
proof (unfold join_def sup_def)
  assume L: "x ∈ carrier L"  "y ∈ carrier L"
    and P: "!!l. least L l (Upper L {x, y}) ==> P l"
  with sup_of_two_exists obtain s where "least L s (Upper L {x, y})"
by fast
  with L show "P (SOME l. least L l (Upper L {x, y}))"
    by (fast intro: someI2 P)
qed

lemma (in weak_upper_semilattice) join_closed [simp]:
  "[| x ∈ carrier L; y ∈ carrier L |] ==> x ⊔ y ∈ carrier L"
  by (rule joinI) (rule least_closed)

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 (rule joinI, rule joinI)
  fix a b
  from xx' carr
      have seq: "{x, y} {.=} {x', y}" by (rule set_eq_pairI)

  assume leasta: "least L a (Upper L {x, y})"
```

```
  assume "least L b (Upper L {x', y})"
  with carr
      have leastb: "least L b (Upper L {x, y})"
      by (simp add: least_Upper_cong_r[OF _ _ seq])

  from leasta leastb
      show "a .= b" by (rule weak_least_unique)
qed (rule carr)+

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 (rule joinI, rule joinI)
  fix a b
  have "{x, y} = {y, x}" by fast
  also from carr yy'
      have "{y, x} {.=} {y', x}" by (intro set_eq_pairI)
  also have "{y', x} = {x, y'}" by fast
  finally
      have seq: "{x, y} {.=} {x, y'}" .

  assume leasta: "least L a (Upper L {x, y})"
  assume "least L b (Upper L {x, y'})"
  with carr
      have leastb: "least L b (Upper L {x, y})"
      by (simp add: least_Upper_cong_r[OF _ _ seq])

  from leasta leastb
      show "a .= b" by (rule weak_least_unique)
qed (rule carr)+

lemma (in weak_partial_order) sup_of_singletonI:
  "x ∈ carrier L ==> least L x (Upper L {x})"
  by (rule least_UpperI) auto

lemma (in weak_partial_order) weak_sup_of_singleton [simp]:
  "x ∈ carrier L ==> ⨆{x} .= x"
  unfolding sup_def
  by (rule someI2) (auto intro: weak_least_unique sup_of_singletonI)

lemma (in weak_partial_order) sup_of_singleton_closed [simp]:
  "x ∈ carrier L ⟹ ⨆{x} ∈ carrier L"
  unfolding sup_def
  by (rule someI2) (auto intro: sup_of_singletonI)
```

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 (unfold sup_def)
  assume L: "x ∈ carrier L"  "A ⊆ carrier L"
    and P: "!!l. least L l (Upper L (insert x A)) ==> P l"
    and least_a: "least L a (Upper L A)"
  from L least_a have La: "a ∈ carrier L" by simp
  from L sup_of_two_exists least_a
  obtain s where least_s: "least L s (Upper L {a, x})" by blast
  show "P (SOME l. least L l (Upper L (insert x A)))"
  proof (rule someI2)
    show "least L s (Upper L (insert x A))"
    proof (rule least_UpperI)
      fix z
      assume "z ∈ insert x A"
      then show "z ⊑ s"
      proof
        assume "z = x" then show ?thesis
          by (simp add: least_Upper_above [OF least_s] L La)
      next
        assume "z ∈ A"
        with L least_s least_a show ?thesis
          by (rule_tac le_trans [where y = a]) (auto dest: least_Upper_above)
      qed
    next
      fix y
      assume y: "y ∈ Upper L (insert x A)"
      show "s ⊑ y"
      proof (rule least_le [OF least_s], rule Upper_memI)
        fix z
        assume z: "z ∈ {a, x}"
        then show "z ⊑ y"
        proof
          have y': "y ∈ Upper L A"
            by (meson Upper_antimono in_mono subset_insertI y)
          assume "z = a"
          with y' least_a show ?thesis by (fast dest: least_le)
        next
          assume "z ∈ {x}"
          with y L show ?thesis by blast
        qed
      qed (rule Upper_closed [THEN subsetD, OF y])
    next
      from L show "insert x A ⊆ carrier L" by simp
      from least_s show "s ∈ carrier L" by simp
    qed
  qed (rule P)
qed
```

```
lemma (in weak_upper_semilattice) finite_sup_least:
  "[| finite A; A ⊆ carrier L; A ≠ {} |] ==> least L (⨆A) (Upper L A)"
proof (induct set: finite)
  case empty
  then show ?case by simp
next
  case (insert x A)
  show ?case
  proof (cases "A = {}")
    case True
    with insert show ?thesis
      by simp (simp add: least_cong [OF weak_sup_of_singleton] sup_of_singletonI)

  next
    case False
    with insert have "least L (⨆A) (Upper L A)" by simp
    with _ show ?thesis
      by (rule sup_insertI) (simp_all add: insert [simplified])
  qed
qed

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 (cases "A = {}")
  case True with P and xA show ?thesis
    by (simp add: finite_sup_least)
next
  case False with P and xA show ?thesis
    by (simp add: sup_insertI finite_sup_least)
qed

lemma (in weak_upper_semilattice) finite_sup_closed [simp]:
  "[| finite A; A ⊆ carrier L; A ≠ {} |] ==> ⨆A ∈ carrier L"
proof (induct set: finite)
  case empty then show ?case by simp
next
  case insert then show ?case
    by - (rule finite_sup_insertI, simp_all)
qed

lemma (in weak_upper_semilattice) join_left:
  "[| x ∈ carrier L; y ∈ carrier L |] ==> x ⊑ x ⊔ y"
  by (rule joinI [folded join_def]) (blast dest: least_mem)

lemma (in weak_upper_semilattice) join_right:
  "[| x ∈ carrier L; y ∈ carrier L |] ==> y ⊑ x ⊔ y"
  by (rule joinI [folded join_def]) (blast dest: least_mem)
```

```
lemma (in weak_upper_semilattice) sup_of_two_least:
  "[| x ∈ carrier L; y ∈ carrier L |] ==> least L (⨆{x, y}) (Upper L
{x, y})"
proof (unfold sup_def)
  assume L: "x ∈ carrier L"  "y ∈ carrier L"
  with sup_of_two_exists obtain s where "least L s (Upper L {x, y})"
by fast
  with L show "least L (SOME z. least L z (Upper L {x, y})) (Upper L
{x, y})"
  by (fast intro: someI2 weak_least_unique)
qed

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 (rule joinI [OF _ x y])
  fix s
  assume "least L s (Upper L {x, y})"
  with sub z show "s ⊑ z" by (fast elim: least_le intro: Upper_memI)
qed

lemma (in weak_lattice) weak_le_iff_meet:
  assumes "x ∈ carrier L" "y ∈ carrier L"
  shows "x ⊑ y ⟷ (x ⊔ y) .= y"
  by (meson assms(1) assms(2) join_closed join_le join_left join_right
le_cong_r local.le_refl weak_le_antisym)

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 (rule finite_sup_insertI)
  — The textbook argument in Jacobson I, p 457
  fix s
  assume sup: "least L s (Upper L {x, y, z})"
  show "x ⊔ (y ⊔ z) .= s"
  proof (rule weak_le_antisym)
    from sup L show "x ⊔ (y ⊔ z) ⊑ s"
      by (fastforce intro!: join_le elim: least_Upper_above)
  next
    from sup L show "s ⊑ x ⊔ (y ⊔ z)"
    by (erule_tac least_le)
      (blast intro!: Upper_memI intro: le_trans join_left join_right join_closed)
  qed (simp_all add: L least_closed [OF sup])
qed (simp_all add: L)
```

Commutativity holds for `=`.

```
lemma join_comm:
  fixes L (structure)
  shows "x ⊔ y = y ⊔ x"
  by (unfold join_def) (simp add: insert_commute)

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 -

  have "(x ⊔ y) ⊔ z = z ⊔ (x ⊔ y)" by (simp only: join_comm)
  also from L have "... .= ⨆{z, x, y}" by (simp add: weak_join_assoc_lemma)
  also from L have "... = ⨆{x, y, z}" by (simp add: insert_commute)
  also from L have "... .= x ⊔ (y ⊔ z)" by (simp add: weak_join_assoc_lemma
[symmetric])
  finally show ?thesis by (simp add: L)
qed
```

### 3.3.2 Infimum

```
lemma (in weak_lower_semilattice) meetI:
  "[| !!i. greatest L i (Lower L {x, y}) ==> P i;
  x ∈ carrier L; y ∈ carrier L |]
  ==> P (x ⊓ y)"
proof (unfold meet_def inf_def)
  assume L: "x ∈ carrier L"  "y ∈ carrier L"
    and P: "!!g. greatest L g (Lower L {x, y}) ==> P g"
  with inf_of_two_exists obtain i where "greatest L i (Lower L {x, y})"
by fast
  with L show "P (SOME g. greatest L g (Lower L {x, y}))"
  by (fast intro: someI2 weak_greatest_unique P)
qed

lemma (in weak_lower_semilattice) meet_closed [simp]:
  "[| x ∈ carrier L; y ∈ carrier L |] ==> x ⊓ y ∈ carrier L"
  by (rule meetI) (rule greatest_closed)

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 (rule meetI, rule meetI)
  fix a b
  from xx' carr
      have seq: "{x, y} {.=} {x', y}" by (rule set_eq_pairI)

  assume greatesta: "greatest L a (Lower L {x, y})"
  assume "greatest L b (Lower L {x', y})"
  with carr
```

```
    have greatestb: "greatest L b (Lower L {x, y})"
    by (simp add: greatest_Lower_cong_r[OF _ _ seq])

  from greatesta greatestb
    show "a .= b" by (rule weak_greatest_unique)
qed (rule carr)+

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 (rule meetI, rule meetI)
  fix a b
  have "{x, y} = {y, x}" by fast
  also from carr yy'
    have "{y, x} {.=} {y', x}" by (intro set_eq_pairI)
  also have "{y', x} = {x, y'}" by fast
  finally
    have seq: "{x, y} {.=} {x, y'}" .

  assume greatesta: "greatest L a (Lower L {x, y})"
  assume "greatest L b (Lower L {x, y'})"
  with carr
    have greatestb: "greatest L b (Lower L {x, y})"
    by (simp add: greatest_Lower_cong_r[OF _ _ seq])

  from greatesta greatestb
    show "a .= b" by (rule weak_greatest_unique)
qed (rule carr)+

lemma (in weak_partial_order) inf_of_singletonI:
  "x ∈ carrier L ==> greatest L x (Lower L {x})"
  by (rule greatest_LowerI) auto

lemma (in weak_partial_order) weak_inf_of_singleton [simp]:
  "x ∈ carrier L ==> ⨅{x} .= x"
  unfolding inf_def
  by (rule someI2) (auto intro: weak_greatest_unique inf_of_singletonI)

lemma (in weak_partial_order) inf_of_singleton_closed:
  "x ∈ carrier L ==> ⨅{x} ∈ carrier L"
  unfolding inf_def
  by (rule someI2) (auto intro: inf_of_singletonI)
```

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 (unfold inf_def)
  assume L: "x ∈ carrier L"  "A ⊆ carrier L"
    and P: "!!g. greatest L g (Lower L (insert x A)) ==> P g"
    and greatest_a: "greatest L a (Lower L A)"
  from L greatest_a have La: "a ∈ carrier L" by simp
  from L inf_of_two_exists greatest_a
  obtain i where greatest_i: "greatest L i (Lower L {a, x})" by blast
  show "P (SOME g. greatest L g (Lower L (insert x A)))"
  proof (rule someI2)
    show "greatest L i (Lower L (insert x A))"
    proof (rule greatest_LowerI)
      fix z
      assume "z ∈ insert x A"
      then show "i ⊑ z"
      proof
        assume "z = x" then show ?thesis
          by (simp add: greatest_Lower_below [OF greatest_i] L La)
      next
        assume "z ∈ A"
        with L greatest_i greatest_a show ?thesis
          by (rule_tac le_trans [where y = a]) (auto dest: greatest_Lower_below)
      qed
    next
      fix y
      assume y: "y ∈ Lower L (insert x A)"
      show "y ⊑ i"
      proof (rule greatest_le [OF greatest_i], rule Lower_memI)
        fix z
        assume z: "z ∈ {a, x}"
        then show "y ⊑ z"
        proof
          have y': "y ∈ Lower L A"
            by (meson Lower_antimono in_mono subset_insertI y)
          assume "z = a"
          with y' greatest_a show ?thesis by (fast dest: greatest_le)
        next
          assume "z ∈ {x}"
          with y L show ?thesis by blast
        qed
      qed (rule Lower_closed [THEN subsetD, OF y])
    next
      from L show "insert x A ⊆ carrier L" by simp
      from greatest_i show "i ∈ carrier L" by simp
    qed
  qed (rule P)
qed

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

```
L A)"
proof (induct set: finite)
  case empty then show ?case by simp
next
  case (insert x A)
  show ?case
  proof (cases "A = {}")
    case True
    with insert show ?thesis
      by simp (simp add: greatest_cong [OF weak_inf_of_singleton]
        inf_of_singleton_closed inf_of_singletonI)
  next
    case False
    from insert show ?thesis
    proof (rule_tac inf_insertI)
      from False insert show "greatest L (⨅A) (Lower L A)" by simp
    qed simp_all
  qed
qed

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 (cases "A = {}")
  case True with P and xA show ?thesis
    by (simp add: finite_inf_greatest)
next
  case False with P and xA show ?thesis
    by (simp add: inf_insertI finite_inf_greatest)
qed

lemma (in weak_lower_semilattice) finite_inf_closed [simp]:
  "[| finite A; A ⊆ carrier L; A ≠ {} |] ==> ⨅A ∈ carrier L"
proof (induct set: finite)
  case empty then show ?case by simp
next
  case insert then show ?case
    by (rule_tac finite_inf_insertI) (simp_all)
qed

lemma (in weak_lower_semilattice) meet_left:
  "[| x ∈ carrier L; y ∈ carrier L |] ==> x ⊓ y ⊑ x"
  by (rule meetI [folded meet_def]) (blast dest: greatest_mem)

lemma (in weak_lower_semilattice) meet_right:
  "[| x ∈ carrier L; y ∈ carrier L |] ==> x ⊓ y ⊑ y"
  by (rule meetI [folded meet_def]) (blast dest: greatest_mem)
```

```
lemma (in weak_lower_semilattice) inf_of_two_greatest:
  "[| x ∈ carrier L; y ∈ carrier L |] ==>
  greatest L (⨅{x, y}) (Lower L {x, y})"
proof (unfold inf_def)
  assume L: "x ∈ carrier L"  "y ∈ carrier L"
  with inf_of_two_exists obtain s where "greatest L s (Lower L {x, y})"
by fast
  with L
  show "greatest L (SOME z. greatest L z (Lower L {x, y})) (Lower L {x,
y})"
  by (fast intro: someI2 weak_greatest_unique)
qed

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 (rule meetI [OF _ x y])
  fix i
  assume "greatest L i (Lower L {x, y})"
  with sub z show "z ⊑ i" by (fast elim: greatest_le intro: Lower_memI)
qed

lemma (in weak_lattice) weak_le_iff_join:
  assumes "x ∈ carrier L" "y ∈ carrier L"
  shows "x ⊑ y ⟷ x .= (x ⊓ y)"
  by (meson assms(1) assms(2) local.le_refl local.le_trans meet_closed
meet_le meet_left meet_right weak_le_antisym weak_refl)

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 (rule finite_inf_insertI)
```

The textbook argument in Jacobson I, p 457

```
  fix i
  assume inf: "greatest L i (Lower L {x, y, z})"
  show "x ⊓ (y ⊓ z) .= i"
  proof (rule weak_le_antisym)
    from inf L show "i ⊑ x ⊓ (y ⊓ z)"
      by (fastforce intro!: meet_le elim: greatest_Lower_below)
  next
    from inf L show "x ⊓ (y ⊓ z) ⊑ i"
    by (erule_tac greatest_le)
      (blast intro!: Lower_memI intro: le_trans meet_left meet_right meet_closed)
  qed (simp_all add: L greatest_closed [OF inf])
qed (simp_all add: L)
```

```
lemma meet_comm:
  fixes L (structure)
  shows "x ⊓ y = y ⊓ x"
  by (unfold meet_def) (simp add: insert_commute)

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 -

  have "(x ⊓ y) ⊓ z = z ⊓ (x ⊓ y)" by (simp only: meet_comm)
  also from L have "... .= ⨅ {z, x, y}" by (simp add: weak_meet_assoc_lemma)
  also from L have "... = ⨅ {x, y, z}" by (simp add: insert_commute)
  also from L have "... .= x ⊓ (y ⊓ z)" by (simp add: weak_meet_assoc_lemma
[symmetric])
  finally show ?thesis by (simp add: L)
qed
```

Total orders are lattices.

```
sublocale weak_total_order ⊆ weak?: weak_lattice
proof
  fix x y
  assume L: "x ∈ carrier L"  "y ∈ carrier L"
  show "∃s. least L s (Upper L {x, y})"
  proof -
    note total L
    moreover
    {
      assume "x ⊑ y"
      with L have "least L y (Upper L {x, y})"
        by (rule_tac least_UpperI) auto
    }
    moreover
    {
      assume "y ⊑ x"
      with L have "least L x (Upper L {x, y})"
        by (rule_tac least_UpperI) auto
    }
    ultimately show ?thesis by blast
  qed
next
  fix x y
  assume L: "x ∈ carrier L"  "y ∈ carrier L"
  show "∃i. greatest L i (Lower L {x, y})"
  proof -
    note total L
    moreover
    {
      assume "y ⊑ x"
```

```
      with L have "greatest L y (Lower L {x, y})"
        by (rule_tac greatest_LowerI) auto
    }
    moreover
    {
      assume "x ⊑ y"
      with L have "greatest L x (Lower L {x, y})"
        by (rule_tac greatest_LowerI) auto
    }
    ultimately show ?thesis by blast
  qed
qed
```

## 3.4 Weak Bounded Lattices

```
locale weak_bounded_lattice =
  weak_lattice +
  weak_partial_order_bottom +
  weak_partial_order_top
begin

lemma bottom_meet: "x ∈ carrier L ⟹ ⊥ ⊓ x .= ⊥"
  by (metis bottom_least least_def meet_closed meet_left weak_le_antisym)

lemma bottom_join: "x ∈ carrier L ⟹ ⊥ ⊔ x .= x"
  by (metis bottom_least join_closed join_le join_right le_refl least_def
weak_le_antisym)

lemma bottom_weak_eq:
  "⟦ b ∈ carrier L; ⋀ x. x ∈ carrier L ⟹ b ⊑ x ⟧ ⟹ b .= ⊥"
  by (metis bottom_closed bottom_lower weak_le_antisym)

lemma top_join: "x ∈ carrier L ⟹ ⊤ ⊔ x .= ⊤"
  by (metis join_closed join_left top_closed top_higher weak_le_antisym)

lemma top_meet: "x ∈ carrier L ⟹ ⊤ ⊓ x .= x"
  by (metis le_refl meet_closed meet_le meet_right top_closed top_higher
weak_le_antisym)

lemma top_weak_eq:  "⟦ t ∈ carrier L; ⋀ x. x ∈ carrier L ⟹ x ⊑ t
⟧ ⟹ t .= ⊤"
  by (metis top_closed top_higher weak_le_antisym)

end

sublocale weak_bounded_lattice ⊆ weak_partial_order ..
```

## 3.5 Lattices where eq is the Equality

```
locale upper_semilattice = partial_order +
```

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

sublocale upper_semilattice ⊆ weak?: weak_upper_semilattice
  by unfold_locales (rule sup_of_two_exists)

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

sublocale lower_semilattice ⊆ weak?: weak_lower_semilattice
  by unfold_locales (rule inf_of_two_exists)

locale lattice = upper_semilattice + lower_semilattice

sublocale lattice ⊆ weak_lattice ..

lemma (in lattice) dual_lattice:
  "lattice (inv_gorder L)"
proof -
  interpret dual: weak_lattice "inv_gorder L"
    by (metis dual_weak_lattice)

  show ?thesis
    apply (unfold_locales)
    apply (simp_all add: inf_of_two_exists sup_of_two_exists)
    apply (rule eq_is_equal)
  done
qed

lemma (in lattice) le_iff_join:
  assumes "x ∈ carrier L" "y ∈ carrier L"
  shows "x ⊑ y ⟷ x = (x ⊓ y)"
  by (simp add: assms(1) assms(2) eq_is_equal weak_le_iff_join)

lemma (in lattice) le_iff_meet:
  assumes "x ∈ carrier L" "y ∈ carrier L"
  shows "x ⊑ y ⟷ (x ⊔ y) = y"
  by (simp add: assms eq_is_equal weak_le_iff_meet)
```

Total orders are lattices.

```
sublocale total_order ⊆ weak?: lattice
  by standard (auto intro: weak.weak.sup_of_two_exists weak.weak.inf_of_two_exists)
```

Functions that preserve joins and meets

```
definition join_pres :: "('a, 'c) gorder_scheme ⇒ ('b, 'd) gorder_scheme
⇒ ('a ⇒ 'b) ⇒ bool" where
```

```
"join_pres X Y f ≡ lattice X ∧ lattice Y ∧ (∀ x ∈ carrier X. ∀ y ∈ carrier
X. f (x ⊔X y) = f x ⊔Y f y)"

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

lemma join_pres_isotone:
  assumes "f ∈ carrier X → carrier Y" "join_pres X Y f"
  shows "isotone X Y f"
proof (rule isotoneI)
  show "weak_partial_order X" "weak_partial_order Y"
    using assms unfolding join_pres_def lattice_def upper_semilattice_def
lower_semilattice_def
    by (meson partial_order.axioms(1))+
  show "⋀x y. ⟦x ∈ carrier X; y ∈ carrier X; x ⊑X y⟧ ⟹ f x ⊑Y f y"
    by (metis (no_types, lifting) PiE assms join_pres_def lattice.le_iff_meet)
qed

lemma meet_pres_isotone:
  assumes "f ∈ carrier X → carrier Y" "meet_pres X Y f"
  shows "isotone X Y f"
proof (rule isotoneI)
  show "weak_partial_order X" "weak_partial_order Y"
    using assms unfolding meet_pres_def lattice_def upper_semilattice_def
lower_semilattice_def
    by (meson partial_order.axioms(1))+
  show "⋀x y. ⟦x ∈ carrier X; y ∈ carrier X; x ⊑X y⟧ ⟹ f x ⊑Y f y"
    by (metis (no_types, lifting) PiE assms lattice.le_iff_join meet_pres_def)
qed
```

## 3.6 Bounded Lattices

```
locale bounded_lattice =
  lattice +
  weak_partial_order_bottom +
  weak_partial_order_top

sublocale bounded_lattice ⊆ weak_bounded_lattice ..

context bounded_lattice
begin

lemma bottom_eq:
  "⟦ b ∈ carrier L; ⋀ x. x ∈ carrier L ⟹ b ⊑ x ⟧ ⟹ b = ⊥"
  by (metis bottom_closed bottom_lower le_antisym)

lemma top_eq:  "⟦ t ∈ carrier L; ⋀ x. x ∈ carrier L ⟹ x ⊑ t ⟧ ⟹
```

```
t = ⊤"
  by (metis le_antisym top_closed top_higher)

end

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

end


theory Complete_Lattice
imports Lattice
begin
```

# 4 Complete Lattices

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

sublocale weak_complete_lattice ⊆ weak_lattice
proof
  fix x y
  assume a: "x ∈ carrier L" "y ∈ carrier L"
  thus "∃s. is_lub L s {x, y}"
    by (rule_tac sup_exists[of "{x, y}"], auto)
  from a show "∃s. is_glb L s {x, y}"
    by (rule_tac inf_exists[of "{x, y}"], auto)
qed
```

Introduction rule: the usual definition of complete lattice

```
lemma (in weak_partial_order) weak_complete_latticeI:
  assumes sup_exists:
    "!!A. [| A ⊆ carrier L |] ==> ∃s. least L s (Upper L A)"
    and inf_exists:
    "!!A. [| A ⊆ carrier L |] ==> ∃i. greatest L i (Lower L A)"
  shows "weak_complete_lattice L"
  by standard (auto intro: sup_exists inf_exists)

lemma (in weak_complete_lattice) dual_weak_complete_lattice:
  "weak_complete_lattice (inv_gorder L)"
proof -
  interpret dual: weak_lattice "inv_gorder L"
    by (metis dual_weak_lattice)
  show ?thesis
    by (unfold_locales) (simp_all add:inf_exists sup_exists)
```

```
qed

lemma (in weak_complete_lattice) supI:
  "[| !!l. least L l (Upper L A) ==> P l; A ⊆ carrier L |]
  ==> P (⨆A)"
proof (unfold sup_def)
  assume L: "A ⊆ carrier L"
    and P: "!!l. least L l (Upper L A) ==> P l"
  with sup_exists obtain s where "least L s (Upper L A)" by blast
  with L show "P (SOME l. least L l (Upper L A))"
  by (fast intro: someI2 weak_least_unique P)
qed

lemma (in weak_complete_lattice) sup_closed [simp]:
  "A ⊆ carrier L ==> ⨆A ∈ carrier L"
  by (rule supI) simp_all

lemma (in weak_complete_lattice) sup_cong:
  assumes "A ⊆ carrier L" "B ⊆ carrier L" "A {.=} B"
  shows "⨆ A .= ⨆ B"
proof -
  have "⋀ x. is_lub L x A ⟷ is_lub L x B"
    by (rule least_Upper_cong_r, simp_all add: assms)
  moreover have "⨆ B ∈ carrier L"
    by (simp add: assms(2))
  ultimately show ?thesis
    by (simp add: sup_def)
qed

sublocale weak_complete_lattice ⊆ weak_bounded_lattice
  apply (unfold_locales)
  apply (metis Upper_empty empty_subsetI sup_exists)
  apply (metis Lower_empty empty_subsetI inf_exists)
done

lemma (in weak_complete_lattice) infI:
  "[| !!i. greatest L i (Lower L A) ==> P i; A ⊆ carrier L |]
  ==> P (⨅A)"
proof (unfold inf_def)
  assume L: "A ⊆ carrier L"
    and P: "!!l. greatest L l (Lower L A) ==> P l"
  with inf_exists obtain s where "greatest L s (Lower L A)" by blast
  with L show "P (SOME l. greatest L l (Lower L A))"
  by (fast intro: someI2 weak_greatest_unique P)
qed

lemma (in weak_complete_lattice) inf_closed [simp]:
  "A ⊆ carrier L ==> ⨅A ∈ carrier L"
  by (rule infI) simp_all
```

```
lemma (in weak_complete_lattice) inf_cong:
  assumes "A ⊆ carrier L" "B ⊆ carrier L" "A {.=} B"
  shows "⨅ A .= ⨅ B"
proof -
  have "⋀ x. is_glb L x A ⟷ is_glb L x B"
    by (rule greatest_Lower_cong_r, simp_all add: assms)
  moreover have "⨅ B ∈ carrier L"
    by (simp add: assms(2))
  ultimately show ?thesis
    by (simp add: inf_def)
qed

theorem (in weak_partial_order) weak_complete_lattice_criterion1:
  assumes top_exists: "∃g. greatest L g (carrier L)"
    and inf_exists:
      "⋀A. [| A ⊆ carrier L; A ≠ {} |] ==> ∃i. greatest L i (Lower L
A)"
  shows "weak_complete_lattice L"
proof (rule weak_complete_latticeI)
  from top_exists obtain top where top: "greatest L top (carrier L)"
..
  fix A
  assume L: "A ⊆ carrier L"
  let ?B = "Upper L A"
  from L top have "top ∈ ?B" by (fast intro!: Upper_memI intro: greatest_le)
  then have B_non_empty: "?B ≠ {}" by fast
  have B_L: "?B ⊆ carrier L" by simp
  from inf_exists [OF B_L B_non_empty]
  obtain b where b_inf_B: "greatest L b (Lower L ?B)" ..
  then have bcarr: "b ∈ carrier L"
    by auto
  have "least L b (Upper L A)"
  proof (rule least_UpperI)
    show "⋀x. x ∈ A ⟹ x ⊑ b"
      by (meson L Lower_memI Upper_memD b_inf_B greatest_le subsetD)
    show "⋀y. y ∈ Upper L A ⟹ b ⊑ y"
      by (meson B_L b_inf_B greatest_Lower_below)
  qed (use bcarr L in auto)
  then show "∃s. least L s (Upper L A)" ..
next
  fix A
  assume L: "A ⊆ carrier L"
  show "∃i. greatest L i (Lower L A)"
    by (metis L Lower_empty inf_exists top_exists)
qed
```

Supremum

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

```
lemma (in partial_order) sup_of_singleton [simp]:
  "x ∈ carrier L ==> ⨆{x} = x"
  using weak_sup_of_singleton unfolding eq_is_equal .

lemma (in upper_semilattice) join_assoc_lemma:
  assumes L: "x ∈ carrier L"  "y ∈ carrier L"  "z ∈ carrier L"
  shows "x ⊔ (y ⊔ z) = ⨆{x, y, z}"
  using weak_join_assoc_lemma L unfolding eq_is_equal .

lemma (in upper_semilattice) join_assoc:
  assumes L: "x ∈ carrier L"  "y ∈ carrier L"  "z ∈ carrier L"
  shows "(x ⊔ y) ⊔ z = x ⊔ (y ⊔ z)"
  using weak_join_assoc L unfolding eq_is_equal .
```

Infimum

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

lemma (in partial_order) inf_of_singleton [simp]:
  "x ∈ carrier L ==> ⨅{x} = x"
  using weak_inf_of_singleton unfolding eq_is_equal .
```

Condition on `A`: infimum exists.

```
lemma (in lower_semilattice) meet_assoc_lemma:
  assumes L: "x ∈ carrier L"  "y ∈ carrier L"  "z ∈ carrier L"
  shows "x ⊓ (y ⊓ z) = ⨅{x, y, z}"
  using weak_meet_assoc_lemma L unfolding eq_is_equal .

lemma (in lower_semilattice) meet_assoc:
  assumes L: "x ∈ carrier L"  "y ∈ carrier L"  "z ∈ carrier L"
  shows "(x ⊓ y) ⊓ z = x ⊓ (y ⊓ z)"
  using weak_meet_assoc L unfolding eq_is_equal .
```

## 4.1 Infimum Laws

```
context weak_complete_lattice
begin

lemma inf_glb:
  assumes "A ⊆ carrier L"
  shows "greatest L (⨅A) (Lower L A)"
proof -
  obtain i where "greatest L i (Lower L A)"
    by (metis assms inf_exists)
  thus ?thesis
    by (metis inf_def someI_ex)
qed

lemma inf_lower:
```

```
  assumes "A ⊆ carrier L" "x ∈ A"
  shows "⨅A ⊑ x"
  by (metis assms greatest_Lower_below inf_glb)

lemma inf_greatest:
  assumes "A ⊆ carrier L" "z ∈ carrier L"
          "(⋀x. x ∈ A ⟹ z ⊑ x)"
  shows "z ⊑ ⨅A"
  by (metis Lower_memI assms greatest_le inf_glb)

lemma weak_inf_empty [simp]: "⨅{} .= ⊤"
  by (metis Lower_empty empty_subsetI inf_glb top_greatest weak_greatest_unique)

lemma weak_inf_carrier [simp]: "⨅carrier L .= ⊥"
  by (metis bottom_weak_eq inf_closed inf_lower subset_refl)

lemma weak_inf_insert [simp]:
  assumes "a ∈ carrier L" "A ⊆ carrier L"
  shows "⨅insert a A .= a ⊓ ⨅A"
proof (rule weak_le_antisym)
  show "⨅insert a A ⊑ a ⊓ ⨅A"
    by (simp add: assms inf_lower local.inf_greatest meet_le)
  show aA: "a ⊓ ⨅A ∈ carrier L"
    using assms by simp
  show "a ⊓ ⨅A ⊑ ⨅insert a A"
    apply (rule inf_greatest)
    using assms apply (simp_all add: aA)
    by (meson aA inf_closed inf_lower local.le_trans meet_left meet_right
subsetCE)
  show "⨅insert a A ∈ carrier L"
    using assms by (force intro: le_trans inf_closed meet_right meet_left
inf_lower)
qed
```

### 4.2 Supremum Laws

```
lemma sup_lub:
  assumes "A ⊆ carrier L"
  shows "least L (⨆A) (Upper L A)"
    by (metis Upper_is_closed assms least_closed least_cong supI sup_closed
sup_exists weak_least_unique)

lemma sup_upper:
  assumes "A ⊆ carrier L" "x ∈ A"
  shows "x ⊑ ⨆A"
  by (metis assms least_Upper_above supI)

lemma sup_least:
  assumes "A ⊆ carrier L" "z ∈ carrier L"
```

```
        "(⋀x. x ∈ A ⟹ x ⊑ z)"
  shows "⨆A ⊑ z"
  by (metis Upper_memI assms least_le sup_lub)

lemma weak_sup_empty [simp]: "⨆{} .= ⊥"
  by (metis Upper_empty bottom_least empty_subsetI sup_lub weak_least_unique)

lemma weak_sup_carrier [simp]: "⨆carrier L .= ⊤"
  by (metis Lower_closed Lower_empty sup_closed sup_upper top_closed top_higher
weak_le_antisym)

lemma weak_sup_insert [simp]:
  assumes "a ∈ carrier L" "A ⊆ carrier L"
  shows "⨆insert a A .= a ⊔ ⨆A"
proof (rule weak_le_antisym)
  show aA: "a ⊔ ⨆A ∈ carrier L"
    using assms by simp
  show "⨆insert a A ⊑ a ⊔ ⨆A"
    apply (rule sup_least)
    using assms apply (simp_all add: aA)
    by (meson aA join_left join_right local.le_trans subsetCE sup_closed
sup_upper)
  show "a ⊔ ⨆A ⊑ ⨆insert a A"
    by (simp add: assms join_le local.sup_least sup_upper)
  show "⨆insert a A ∈ carrier L"
    using assms by (force intro: le_trans inf_closed meet_right meet_left
inf_lower)
qed

end
```

## 4.3 Fixed points of a lattice

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

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

lemma (in weak_partial_order)
  use_fps: "x ∈ fps L f ⟹ f x .= x"
  by (simp add: fps_def)

lemma fps_carrier [simp]:
  "fps L f ⊆ carrier L"
  by (auto simp add: fps_def)

lemma (in weak_complete_lattice) fps_sup_image:
  assumes "f ∈ carrier L → carrier L" "A ⊆ fps L f"
  shows "⨆ (f ` A) .= ⨆ A"
proof -
```

```
  from assms(2) have AL: "A ⊆ carrier L"
    by (auto simp add: fps_def)
  show ?thesis
  proof (rule sup_cong, simp_all add: AL)
    from assms(1) AL show "f ` A ⊆ carrier L"
      by auto
    then have *: "⋀b. ⟦A ⊆ {x ∈ carrier L. f x .= x}; b ∈ A⟧ ⟹ ∃a∈f
` A. b .= a"
      by (meson AL assms(2) image_eqI local.sym subsetCE use_fps)
    from assms(2) show "f ` A {.=} A"
      by (auto simp add: fps_def intro: set_eqI2 [OF _ *])
  qed
qed

lemma (in weak_complete_lattice) fps_idem:
  assumes "f ∈ carrier L → carrier L" "Idem f"
  shows "fps L f {.=} f ` carrier L"
proof (rule set_eqI2)
  show "⋀a. a ∈ fps L f ⟹ ∃b∈f ` carrier L. a .= b"
    using assms by (force simp add: fps_def intro: local.sym)
  show "⋀b. b ∈ f ` carrier L ⟹ ∃a∈fps L f. b .= a"
    using assms by (force simp add: idempotent_def fps_def)
qed

context weak_complete_lattice
begin

lemma weak_sup_pre_fixed_point:
  assumes "f ∈ carrier L → carrier L" "isotone L L f" "A ⊆ fps L f"
  shows "(⨆L A) ⊑L f (⨆L A)"
proof (rule sup_least)
  from assms(3) show AL: "A ⊆ carrier L"
    by (auto simp add: fps_def)
  thus fA: "f (⨆A) ∈ carrier L"
    by (simp add: assms funcset_carrier[of f L L])
  fix x
  assume xA: "x ∈ A"
  hence "x ∈ fps L f"
    using assms subsetCE by blast
  hence "f x .=L x"
    by (auto simp add: fps_def)
  moreover have "f x ⊑L f (⨆LA)"
    by (meson AL assms(2) subsetCE sup_closed sup_upper use_iso1 xA)
  ultimately show "x ⊑L f (⨆LA)"
    by (meson AL fA assms(1) funcset_carrier le_cong local.refl subsetCE
xA)
qed

lemma weak_sup_post_fixed_point:
```

```
  assumes "f ∈ carrier L → carrier L" "isotone L L f" "A ⊆ fps L f"
  shows "f (⨅L A) ⊑L (⨅L A)"
proof (rule inf_greatest)
  from assms(3) show AL: "A ⊆ carrier L"
    by (auto simp add: fps_def)
  thus fA: "f (⨅A) ∈ carrier L"
    by (simp add: assms funcset_carrier[of f L L])
  fix x
  assume xA: "x ∈ A"
  hence "x ∈ fps L f"
    using assms subsetCE by blast
  hence "f x .=L x"
    by (auto simp add: fps_def)
  moreover have "f (⨅LA) ⊑L f x"
    by (meson AL assms(2) inf_closed inf_lower subsetCE use_iso1 xA)
  ultimately show "f (⨅LA) ⊑L x"
    by (meson AL assms(1) fA funcset_carrier le_cong_r subsetCE xA)
qed
```

### 4.3.1 Least fixed points

```
lemma LFP_closed [intro, simp]:
  "LFP f ∈ carrier L"
  by (metis (lifting) LEAST_FP_def inf_closed mem_Collect_eq subsetI)

lemma LFP_lowerbound:
  assumes "x ∈ carrier L" "f x ⊑ x"
  shows "LFP f ⊑ x"
  by (auto intro:inf_lower assms simp add:LEAST_FP_def)

lemma LFP_greatest:
  assumes "x ∈ carrier L"
          "(⋀u. ⟦ u ∈ carrier L; f u ⊑ u ⟧ ⟹ x ⊑ u)"
  shows "x ⊑ LFP f"
  by (auto simp add:LEAST_FP_def intro:inf_greatest assms)

lemma LFP_lemma2:
  assumes "Mono f" "f ∈ carrier L → carrier L"
  shows "f (LFP f) ⊑ LFP f"
proof (rule LFP_greatest)
  have f: "⋀x. x ∈ carrier L ⟹ f x ∈ carrier L"
    using assms by (auto simp add: Pi_def)
  with assms show "f (LFP f) ∈ carrier L"
    by blast
  show "⋀u. ⟦u ∈ carrier L; f u ⊑ u⟧ ⟹ f (LFP f) ⊑ u"
    by (meson LFP_closed LFP_lowerbound assms(1) f local.le_trans use_iso1)
qed

lemma LFP_lemma3:
```

```
  assumes "Mono f" "f ∈ carrier L → carrier L"
  shows "LFP f ⊑ f (LFP f)"
  using assms by (simp add: Pi_def) (metis LFP_closed LFP_lemma2 LFP_lowerbound
assms(2) use_iso2)

lemma LFP_weak_unfold:
  "⟦ Mono f; f ∈ carrier L → carrier L ⟧ ⟹ LFP f .= f (LFP f)"
  by (auto intro: LFP_lemma2 LFP_lemma3 funcset_mem)

lemma LFP_fixed_point [intro]:
  assumes "Mono f" "f ∈ carrier L → carrier L"
  shows "LFP f ∈ fps L f"
proof -
  have "f (LFP f) ∈ carrier L"
    using assms(2) by blast
  with assms show ?thesis
    by (simp add: LFP_weak_unfold fps_def local.sym)
qed

lemma LFP_least_fixed_point:
  assumes "Mono f" "f ∈ carrier L → carrier L" "x ∈ fps L f"
  shows "LFP f ⊑ x"
  using assms by (force intro: LFP_lowerbound simp add: fps_def)

lemma LFP_idem:
  assumes "f ∈ carrier L → carrier L" "Mono f" "Idem f"
  shows "LFP f .= (f ⊥)"
proof (rule weak_le_antisym)
  from assms(1) show fb: "f ⊥ ∈ carrier L"
    by (rule funcset_mem, simp)
  from assms show mf: "LFP f ∈ carrier L"
    by blast
  show "LFP f ⊑ f ⊥"
  proof -
    have "f (f ⊥) .= f ⊥"
      by (auto simp add: fps_def fb assms(3) idempotent)
    moreover have "f (f ⊥) ∈ carrier L"
      by (rule funcset_mem[of f "carrier L"], simp_all add: assms fb)
    ultimately show ?thesis
      by (auto intro: LFP_lowerbound simp add: fb)
  qed
  show "f ⊥ ⊑ LFP f"
  proof -
    have "f ⊥ ⊑ f (LFP f)"
      by (auto intro: use_iso1[of _ f] simp add: assms)
    moreover have "... .= LFP f"
      using assms(1) assms(2) fps_def by force
    moreover from assms(1) have "f (LFP f) ∈ carrier L"
      by (auto)
```

```
    ultimately show ?thesis
      using fb by blast
  qed
qed
```

### 4.3.2 Greatest fixed points

```
lemma GFP_closed [intro, simp]:
  "GFP f ∈ carrier L"
  by (auto intro:sup_closed simp add:GREATEST_FP_def)

lemma GFP_upperbound:
  assumes "x ∈ carrier L" "x ⊑ f x"
  shows "x ⊑ GFP f"
  by (auto intro:sup_upper assms simp add:GREATEST_FP_def)

lemma GFP_least:
  assumes "x ∈ carrier L"
          "(⋀u. ⟦ u ∈ carrier L; u ⊑ f u ⟧ ⟹ u ⊑ x)"
  shows "GFP f ⊑ x"
  by (auto simp add:GREATEST_FP_def intro:sup_least assms)

lemma GFP_lemma2:
  assumes "Mono f" "f ∈ carrier L → carrier L"
  shows "GFP f ⊑ f (GFP f)"
proof (rule GFP_least)
  have f: "⋀x. x ∈ carrier L ⟹ f x ∈ carrier L"
    using assms by (auto simp add: Pi_def)
  with assms show "f (GFP f) ∈ carrier L"
    by blast
  show "⋀u. ⟦u ∈ carrier L; u ⊑ f u⟧ ⟹ u ⊑ f (GFP f)"
    by (meson GFP_closed GFP_upperbound le_trans assms(1) f local.le_trans
use_iso1)
qed

lemma GFP_lemma3:
  assumes "Mono f" "f ∈ carrier L → carrier L"
  shows "f (GFP f) ⊑ GFP f"
  by (metis GFP_closed GFP_lemma2 GFP_upperbound assms funcset_mem use_iso2)

lemma GFP_weak_unfold:
  "⟦ Mono f; f ∈ carrier L → carrier L ⟧ ⟹ GFP f .= f (GFP f)"
  by (auto intro: GFP_lemma2 GFP_lemma3 funcset_mem)

lemma (in weak_complete_lattice) GFP_fixed_point [intro]:
  assumes "Mono f" "f ∈ carrier L → carrier L"
  shows "GFP f ∈ fps L f"
  using assms
proof -
```

```
  have "f (GFP f) ∈ carrier L"
    using assms(2) by blast
  with assms show ?thesis
    by (simp add: GFP_weak_unfold fps_def local.sym)
qed

lemma GFP_greatest_fixed_point:
  assumes "Mono f" "f ∈ carrier L → carrier L" "x ∈ fps L f"
  shows "x ⊑ GFP f"
  using assms
  by (rule_tac GFP_upperbound, auto simp add: fps_def, meson PiE local.sym
weak_refl)

lemma GFP_idem:
  assumes "f ∈ carrier L → carrier L" "Mono f" "Idem f"
  shows "GFP f .= (f ⊤)"
proof (rule weak_le_antisym)
  from assms(1) show fb: "f ⊤ ∈ carrier L"
    by (rule funcset_mem, simp)
  from assms show mf: "GFP f ∈ carrier L"
    by blast
  show "f ⊤ ⊑ GFP f"
  proof -
    have "f (f ⊤) .= f ⊤"
      by (auto simp add: fps_def fb assms(3) idempotent)
    moreover have "f (f ⊤) ∈ carrier L"
      by (rule funcset_mem[of f "carrier L"], simp_all add: assms fb)
    ultimately show ?thesis
      by (rule_tac GFP_upperbound, simp_all add: fb local.sym)
  qed
  show "GFP f ⊑ f ⊤"
  proof -
    have "GFP f ⊑ f (GFP f)"
      by (simp add: GFP_lemma2 assms(1) assms(2))
    moreover have "... ⊑ f ⊤"
      by (auto intro: use_iso1[of _ f] simp add: assms)
    moreover from assms(1) have "f (GFP f) ∈ carrier L"
      by (auto)
    ultimately show ?thesis
      using fb local.le_trans by blast
  qed
qed

end
```

## 4.4 Complete lattices where eq is the Equality

```
locale complete_lattice = partial_order +
  assumes sup_exists:
```

```
    "[| A ⊆ carrier L |] ==> ∃s. least L s (Upper L A)"
    and inf_exists:
    "[| A ⊆ carrier L |] ==> ∃i. greatest L i (Lower L A)"

sublocale complete_lattice ⊆ lattice
proof
  fix x y
  assume a: "x ∈ carrier L" "y ∈ carrier L"
  thus "∃s. is_lub L s {x, y}"
    by (rule_tac sup_exists[of "{x, y}"], auto)
  from a show "∃s. is_glb L s {x, y}"
    by (rule_tac inf_exists[of "{x, y}"], auto)
qed

sublocale complete_lattice ⊆ weak?: weak_complete_lattice
  by standard (auto intro: sup_exists inf_exists)

lemma complete_lattice_lattice [simp]:
  assumes "complete_lattice X"
  shows "lattice X"
proof -
  interpret c: complete_lattice X
    by (simp add: assms)
  show ?thesis
    by (unfold_locales)
qed
```

Introduction rule: the usual definition of complete lattice

```
lemma (in partial_order) complete_latticeI:
  assumes sup_exists:
    "!!A. [| A ⊆ carrier L |] ==> ∃s. least L s (Upper L A)"
    and inf_exists:
    "!!A. [| A ⊆ carrier L |] ==> ∃i. greatest L i (Lower L A)"
  shows "complete_lattice L"
  by standard (auto intro: sup_exists inf_exists)

theorem (in partial_order) complete_lattice_criterion1:
  assumes top_exists: "∃g. greatest L g (carrier L)"
    and inf_exists:
      "!!A. [| A ⊆ carrier L; A ≠ {} |] ==> ∃i. greatest L i (Lower L
A)"
  shows "complete_lattice L"
proof (rule complete_latticeI)
  from top_exists obtain top where top: "greatest L top (carrier L)"
..
  fix A
  assume L: "A ⊆ carrier L"
  let ?B = "Upper L A"
  from L top have "top ∈ ?B" by (fast intro!: Upper_memI intro: greatest_le)
```

```
  then have B_non_empty: "?B ≠ {}" by fast
  have B_L: "?B ⊆ carrier L" by simp
  from inf_exists [OF B_L B_non_empty]
  obtain b where b_inf_B: "greatest L b (Lower L ?B)" ..
  then have bcarr: "b ∈ carrier L"
    by blast
  have "least L b (Upper L A)"
  proof (rule least_UpperI)
    show "⋀x. x ∈ A ⟹ x ⊑ b"
      by (meson L Lower_memI Upper_memD b_inf_B greatest_le rev_subsetD)
    show "⋀y. y ∈ Upper L A ⟹ b ⊑ y"
      by (auto elim: greatest_Lower_below [OF b_inf_B])
  qed (use L bcarr in auto)
  then show "∃s. least L s (Upper L A)" ..
next
  fix A
  assume L: "A ⊆ carrier L"
  show "∃i. greatest L i (Lower L A)"
  proof (cases "A = {}")
    case True then show ?thesis
      by (simp add: top_exists)
  next
    case False with L show ?thesis
      by (rule inf_exists)
  qed
qed
```

## 4.5 Fixed points

```
context complete_lattice
begin

lemma LFP_unfold:
  "⟦ Mono f; f ∈ carrier L → carrier L ⟧ ⟹ LFP f = f (LFP f)"
  using eq_is_equal weak.LFP_weak_unfold by auto

lemma LFP_const:
  "t ∈ carrier L ⟹ LFP (λ x. t) = t"
  by (simp add: local.le_antisym weak.LFP_greatest weak.LFP_lowerbound)

lemma LFP_id:
  "LFP id = ⊥"
  by (simp add: local.le_antisym weak.LFP_lowerbound)

lemma GFP_unfold:
  "⟦ Mono f; f ∈ carrier L → carrier L ⟧ ⟹ GFP f = f (GFP f)"
  using eq_is_equal weak.GFP_weak_unfold by auto

lemma GFP_const:
```

```
  "t ∈ carrier L ⟹ GFP (λ x. t) = t"
  by (simp add: local.le_antisym weak.GFP_least weak.GFP_upperbound)

lemma GFP_id:
  "GFP id = ⊤"
  using weak.GFP_upperbound by auto

end
```

## 4.6 Interval complete lattices

```
context weak_complete_lattice
begin

  lemma at_least_at_most_Sup: "⟦ a ∈ carrier L; b ∈ carrier L; a ⊑ b
⟧ ⟹ ⨆ ⦃a..b⦄ .= b"
    by (rule weak_le_antisym [OF sup_least sup_upper]) (auto simp add:
at_least_at_most_closed)

  lemma at_least_at_most_Inf: "⟦ a ∈ carrier L; b ∈ carrier L; a ⊑ b
⟧ ⟹ ⨅ ⦃a..b⦄ .= a"
    by (rule weak_le_antisym [OF inf_lower inf_greatest]) (auto simp add:
at_least_at_most_closed)

end

lemma weak_complete_lattice_interval:
  assumes "weak_complete_lattice L" "a ∈ carrier L" "b ∈ carrier L" "a
⊑L b"
  shows "weak_complete_lattice (L ⦇ carrier := ⦃a..b⦄L ⦈)"
proof -
  interpret L: weak_complete_lattice L
    by (simp add: assms)
  interpret weak_partial_order "L ⦇ carrier := ⦃a..b⦄L ⦈"
  proof -
    have "⦃a..b⦄L ⊆ carrier L"
      by (auto simp add: at_least_at_most_def)
    thus "weak_partial_order (L⦇carrier := ⦃a..b⦄L⦈)"
      by (simp add: L.weak_partial_order_axioms weak_partial_order_subset)
  qed

  show ?thesis
  proof
    fix A
    assume a: "A ⊆ carrier (L⦇carrier := ⦃a..b⦄L⦈)"
    show "∃s. is_lub (L⦇carrier := ⦃a..b⦄L⦈) s A"
    proof (cases "A = {}")
      case True
      thus ?thesis
```

```
        by (rule_tac x="a" in exI, auto simp add: least_def assms)
    next
      case False
      show ?thesis
      proof (intro exI least_UpperI, simp_all)
        show b:"⋀ x. x ∈ A ⟹ x ⊑L ⨆LA"
          using a by (auto intro: L.sup_upper, meson L.at_least_at_most_closed
L.sup_upper subset_trans)
        show "⋀y. y ∈ Upper (L⦇carrier := ⦃a..b⦄L⦈) A ⟹ ⨆LA ⊑L y"
          using a L.at_least_at_most_closed by (rule_tac L.sup_least,
auto intro: funcset_mem simp add: Upper_def)
        from a show *: "A ⊆ ⦃a..b⦄L"
          by auto
        show "⨆LA ∈ ⦃a..b⦄L"
        proof (rule_tac L.at_least_at_most_member)
          show 1: "⨆LA ∈ carrier L"
            by (meson L.at_least_at_most_closed L.sup_closed subset_trans
*)
          show "a ⊑L ⨆LA"
            by (meson "*" False L.at_least_at_most_closed L.at_least_at_most_lower
L.le_trans L.sup_upper 1 all_not_in_conv assms(2) subsetD subset_trans)
          show "⨆LA ⊑L b"
          proof (rule L.sup_least)
            show "A ⊆ carrier L" "⋀x. x ∈ A ⟹ x ⊑L b"
              using * L.at_least_at_most_closed by blast+
          qed (simp add: assms)
        qed
      qed
    qed
    show "∃s. is_glb (L⦇carrier := ⦃a..b⦄L⦈) s A"
    proof (cases "A = {}")
      case True
      thus ?thesis
        by (rule_tac x="b" in exI, auto simp add: greatest_def assms)
    next
      case False
      show ?thesis
      proof (rule_tac x="⨅L A" in exI, rule greatest_LowerI, simp_all)
        show b:"⋀x. x ∈ A ⟹ ⨅LA ⊑L x"
          using a L.at_least_at_most_closed by (force intro!: L.inf_lower)
        show "⋀y. y ∈ Lower (L⦇carrier := ⦃a..b⦄L⦈) A ⟹ y ⊑L ⨅LA"
           using a L.at_least_at_most_closed by (rule_tac L.inf_greatest,
auto intro: funcset_carrier' simp add: Lower_def)
        from a show *: "A ⊆ ⦃a..b⦄L"
          by auto
        show "⨅LA ∈ ⦃a..b⦄L"
        proof (rule_tac L.at_least_at_most_member)
          show 1: "⨅LA ∈ carrier L"
            by (meson "*" L.at_least_at_most_closed L.inf_closed subset_trans)
```

```
          show "a ⊑L ⨅LA"
            by (meson "*" L.at_least_at_most_closed L.at_least_at_most_lower
L.inf_greatest assms(2) subsetD subset_trans)
          show "⨅LA ⊑L b"
            by (meson * 1 False L.at_least_at_most_closed L.at_least_at_most_upper
L.inf_lower L.le_trans all_not_in_conv assms(3) subsetD subset_trans)
        qed
      qed
    qed
  qed
qed
```

## 4.7 Knaster-Tarski theorem and variants

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

```
theorem Knaster_Tarski:
  assumes "weak_complete_lattice L" and f: "f ∈ carrier L → carrier
L" and "isotone L L f"
  shows "weak_complete_lattice (fpl L f)" (is "weak_complete_lattice ?L'")
proof -
  interpret L: weak_complete_lattice L
    by (simp add: assms)
  interpret weak_partial_order ?L'
  proof -
    have "{x ∈ carrier L. f x .=L x} ⊆ carrier L"
      by (auto)
    thus "weak_partial_order ?L'"
      by (simp add: L.weak_partial_order_axioms weak_partial_order_subset)
  qed
  show ?thesis
  proof (unfold_locales, simp_all)
    fix A
    assume A: "A ⊆ fps L f"
    show "∃s. is_lub (fpl L f) s A"
    proof
      from A have AL: "A ⊆ carrier L"
        by (meson fps_carrier subset_eq)

      let ?w = "⨆L A"
      have w: "f (⨆LA) ∈ carrier L"
        by (rule funcset_mem[of f "carrier L"], simp_all add: AL assms(2))

      have pf_w: "(⨆L A) ⊑L f (⨆L A)"
        by (simp add: A L.weak_sup_pre_fixed_point assms(2) assms(3))

      have f_top_chain: "f ` ⦃?w..⊤L⦄L ⊆ ⦃?w..⊤L⦄L"
      proof (auto simp add: at_least_at_most_def)
        fix x
        assume b: "x ∈ carrier L" "⨆LA ⊑L x"
```

```
        from b show fx: "f x ∈ carrier L"
          using assms(2) by blast
        show "⨆LA ⊑L f x"
        proof -
          have "?w ⊑L f ?w"
          proof (rule_tac L.sup_least, simp_all add: AL w)
            fix y
            assume c: "y ∈ A"
            hence y: "y ∈ fps L f"
              using A subsetCE by blast
            with assms have "y .=L f y"
            proof -
              from y have "y ∈ carrier L"
                by (simp add: fps_def)
              moreover hence "f y ∈ carrier L"
                by (rule_tac funcset_mem[of f "carrier L"], simp_all add:
assms)
              ultimately show ?thesis using y
                by (rule_tac L.sym, simp_all add: L.use_fps)
            qed
            moreover have "y ⊑L ⨆LA"
              by (simp add: AL L.sup_upper c(1))
            ultimately show "y ⊑L f (⨆LA)"
              by (meson fps_def AL funcset_mem L.refl L.weak_complete_lattice_axioms
assms(2) assms(3) c(1) isotone_def rev_subsetD weak_complete_lattice.sup_closed
weak_partial_order.le_cong)
          qed
          thus ?thesis
            by (meson AL funcset_mem L.le_trans L.sup_closed assms(2)
assms(3) b(1) b(2) use_iso2)
        qed

        show "f x ⊑L ⊤L"
          by (simp add: fx)
      qed

      let ?L' = "L⦇ carrier := ⦃?w..⊤L⦄L ⦈"

      interpret L': weak_complete_lattice ?L'
        by (auto intro: weak_complete_lattice_interval simp add: L.weak_complete_lattice_ax
AL)

      let ?L'' = "L⦇ carrier := fps L f ⦈"

      show "is_lub ?L'' (LFP?L' f) A"
      proof (rule least_UpperI, simp_all)
        fix x
        assume x: "x ∈ Upper ?L'' A"
        have "LFP?L' f ⊑?L' x"
```

```
        proof (rule L'.LFP_lowerbound, simp_all)
          show "x ∈ ⦃⨆LA..⊤L⦄L"
            using x by (auto simp add: Upper_def A AL L.at_least_at_most_member
L.sup_least rev_subsetD)
          with x show "f x ⊑L x"
            by (simp add: Upper_def) (meson L.at_least_at_most_closed
L.use_fps L.weak_refl subsetD f_top_chain imageI)
        qed
        thus " LFP?L' f ⊑L x"
          by (simp)
      next
        fix x
        assume xA: "x ∈ A"
        show "x ⊑L LFP?L' f"
        proof -
          have "LFP?L' f ∈ carrier ?L'"
            by blast
          thus ?thesis
            by (simp, meson AL L.at_least_at_most_closed L.at_least_at_most_lower
L.le_trans L.sup_closed L.sup_upper xA subsetCE)
        qed
      next
        show "A ⊆ fps L f"
          by (simp add: A)
      next
        show "LFP?L' f ∈ fps L f"
        proof (auto simp add: fps_def)
          have "LFP?L' f ∈ carrier ?L'"
            by (rule L'.LFP_closed)
          thus c:"LFP?L' f ∈ carrier L"
             by (auto simp add: at_least_at_most_def)
          have "LFP?L' f .=?L' f (LFP?L' f)"
          proof (rule "L'.LFP_weak_unfold", simp_all)
            have "⋀x. ⟦x ∈ carrier L; ⨆LA ⊑L x⟧ ⟹ ⨆LA ⊑L f x"
              by (meson AL funcset_mem L.le_trans L.sup_closed assms(2)
assms(3) pf_w use_iso2)
            with f show "f ∈ ⦃⨆LA..⊤L⦄L → ⦃⨆LA..⊤L⦄L"
              by (auto simp add: Pi_def at_least_at_most_def)
            show "MonoL⦇carrier := ⦃⨆LA..⊤L⦄L⦈ f"
              using L'.weak_partial_order_axioms assms(3)
              by (auto simp add: isotone_def) (meson L.at_least_at_most_closed
subsetCE)
          qed
          thus "f (LFP?L' f) .=L LFP?L' f"
            by (simp add: L.equivalence_axioms funcset_carrier' c assms(2)
equivalence.sym)
        qed
      qed
    qed
```

```
    show "∃i. is_glb (L(|carrier := fps L f|)) i A"
    proof
      from A have AL: "A ⊆ carrier L"
        by (meson fps_carrier subset_eq)

      let ?w = "⨅L A"
      have w: "f (⨅LA) ∈ carrier L"
        by (simp add: AL funcset_carrier' assms(2))

      have pf_w: "f (⨅L A) ⊑L (⨅L A)"
        by (simp add: A L.weak_sup_post_fixed_point assms(2) assms(3))

      have f_bot_chain: "f ` ⦃⊥L..?w⦄L ⊆ ⦃⊥L..?w⦄L"
      proof (auto simp add: at_least_at_most_def)
        fix x
        assume b: "x ∈ carrier L" "x ⊑L ⨅LA"
        from b show fx: "f x ∈ carrier L"
          using assms(2) by blast
        show "f x ⊑L ⨅LA"
        proof -
          have "f ?w ⊑L ?w"
          proof (rule_tac L.inf_greatest, simp_all add: AL w)
            fix y
            assume c: "y ∈ A"
            with assms have "y .=L f y"
              by (metis (no_types, lifting) A funcset_carrier'[OF assms(2)]
L.sym fps_def mem_Collect_eq subset_eq)
            moreover have "⨅LA ⊑L y"
              by (simp add: AL L.inf_lower c)
            ultimately show "f (⨅LA) ⊑L y"
              by (meson AL L.inf_closed L.le_trans c pf_w rev_subsetD
w)
          qed
          thus ?thesis
            by (meson AL L.inf_closed L.le_trans assms(3) b(1) b(2) fx
use_iso2 w)
        qed
        show "⊥L ⊑L f x"
          by (simp add: fx)
      qed

      let ?L' = "L(| carrier := ⦃⊥L..?w⦄L |)"

      interpret L': weak_complete_lattice ?L'
        by (auto intro!: weak_complete_lattice_interval simp add: L.weak_complete_lattice_a
AL)

      let ?L'' = "L(| carrier := fps L f |)"
```

```
      show "is_glb ?L'' (GFP?L' f) A"
      proof (rule greatest_LowerI, simp_all)
        fix x
        assume "x ∈ Lower ?L'' A"
        then have x: "∀y. y ∈ A ∧ y ∈ fps L f ⟶ x ⊑L y" "x ∈ fps
L f"
          by (auto simp add: Lower_def)
        have "x ⊑?L' GFP?L' f"
          unfolding Lower_def
        proof (rule_tac L'.GFP_upperbound; simp)
          show "x ∈ ⦃⊥L..⨅LA⦄L"
            by (meson x A AL L.at_least_at_most_member L.bottom_lower
L.inf_greatest contra_subsetD fps_carrier)
          show "x ⊑L f x"
            using x by (simp add: funcset_carrier' L.sym assms(2) fps_def)
        qed
        thus "x ⊑L GFP?L' f"
          by (simp)
      next
        fix x
        assume xA: "x ∈ A"
        show "GFP?L' f ⊑L x"
        proof -
          have "GFP?L' f ∈ carrier ?L'"
            by blast
          thus ?thesis
            by (simp, meson AL L.at_least_at_most_closed L.at_least_at_most_upper
L.inf_closed L.inf_lower L.le_trans subsetCE xA)
        qed
      next
        show "A ⊆ fps L f"
          by (simp add: A)
      next
        show "GFP?L' f ∈ fps L f"
        proof (auto simp add: fps_def)
          have "GFP?L' f ∈ carrier ?L'"
            by (rule L'.GFP_closed)
          thus c:"GFP?L' f ∈ carrier L"
             by (auto simp add: at_least_at_most_def)
          have "GFP?L' f .=?L' f (GFP?L' f)"
          proof (rule "L'.GFP_weak_unfold", simp_all)
            have "⋀x. ⟦x ∈ carrier L; x ⊑L ⨅LA⟧ ⟹ f x ⊑L ⨅LA"
              by (meson AL funcset_carrier L.inf_closed L.le_trans assms(2)
assms(3) pf_w use_iso2)
            with assms(2) show "f ∈ ⦃⊥L..?w⦄L → ⦃⊥L..?w⦄L"
              by (auto simp add: Pi_def at_least_at_most_def)
            have "⋀x y. ⟦x ∈ ⦃⊥L..⨅LA⦄L; y ∈ ⦃⊥L..⨅LA⦄L; x ⊑L y⟧ ⟹
f x ⊑L f y"
              by (meson L.at_least_at_most_closed subsetD use_iso1  assms(3))
```

```
          with L'.weak_partial_order_axioms show "MonoL(|carrier := {|⊥L..?w|}L|)
f"
            by (auto simp add: isotone_def)
        qed
        thus "f (GFP?L' f) .=L GFP?L' f"
          by (simp add: L.equivalence_axioms funcset_carrier' c assms(2)
equivalence.sym)
      qed
    qed
  qed
 qed
qed

theorem Knaster_Tarski_top:
  assumes "weak_complete_lattice L" "isotone L L f" "f ∈ carrier L →
carrier L"
  shows "⊤fpl L f .=L GFPL f"
proof -
  interpret L: weak_complete_lattice L
    by (simp add: assms)
  interpret L': weak_complete_lattice "fpl L f"
    by (rule Knaster_Tarski, simp_all add: assms)
  show ?thesis
  proof (rule L.weak_le_antisym, simp_all)
    show "⊤fpl L f ⊑L GFPL f"
      by (rule L.GFP_greatest_fixed_point, simp_all add: assms L'.top_closed[simplified])
    show "GFPL f ⊑L ⊤fpl L f"
    proof -
      have "GFPL f ∈ fps L f"
        by (rule L.GFP_fixed_point, simp_all add: assms)
      hence "GFPL f ∈ carrier (fpl L f)"
        by simp
      hence "GFPL f ⊑fpl L f ⊤fpl L f"
        by (rule L'.top_higher)
      thus ?thesis
        by simp
    qed
    show "⊤fpl L f ∈ carrier L"
    proof -
      have "carrier (fpl L f) ⊆ carrier L"
        by (auto simp add: fps_def)
      with L'.top_closed show ?thesis
        by blast
    qed
  qed
qed

theorem Knaster_Tarski_bottom:
```

```
  assumes "weak_complete_lattice L" "isotone L L f" "f ∈ carrier L →
carrier L"
  shows "⊥fpl L f .=L LFPL f"
proof -
  interpret L: weak_complete_lattice L
    by (simp add: assms)
  interpret L': weak_complete_lattice "fpl L f"
    by (rule Knaster_Tarski, simp_all add: assms)
  show ?thesis
  proof (rule L.weak_le_antisym, simp_all)
    show "LFPL f ⊑L ⊥fpl L f"
      by (rule L.LFP_least_fixed_point, simp_all add: assms L'.bottom_closed[simplified])
    show "⊥fpl L f ⊑L LFPL f"
    proof -
      have "LFPL f ∈ fps L f"
        by (rule L.LFP_fixed_point, simp_all add: assms)
      hence "LFPL f ∈ carrier (fpl L f)"
        by simp
      hence "⊥fpl L f ⊑fpl L f LFPL f"
        by (rule L'.bottom_lower)
      thus ?thesis
        by simp
    qed
    show "⊥fpl L f ∈ carrier L"
    proof -
      have "carrier (fpl L f) ⊆ carrier L"
        by (auto simp add: fps_def)
      with L'.bottom_closed show ?thesis
        by blast
    qed
  qed
qed
```

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

```
theorem Knaster_Tarski_idem:
  assumes "complete_lattice L" "f ∈ carrier L → carrier L" "isotone
L L f" "idempotent L f"
  shows "complete_lattice (L(|carrier := f ` carrier L|))"
proof -
  interpret L: complete_lattice L
    by (simp add: assms)
  have "fps L f = f ` carrier L"
    using L.weak.fps_idem[OF assms(2) assms(4)]
    by (simp add: L.set_eq_is_eq)
  then interpret L': weak_complete_lattice "(L(|carrier := f ` carrier
L|))"
    by (metis Knaster_Tarski L.weak.weak_complete_lattice_axioms assms(2)
assms(3))
```

```
  show ?thesis
    using L'.sup_exists L'.inf_exists
    by (unfold_locales, auto simp add: L.eq_is_equal)
qed

theorem Knaster_Tarski_idem_extremes:
  assumes "weak_complete_lattice L" "isotone L L f" "idempotent L f"
"f ∈ carrier L → carrier L"
  shows "⊤fpl L f .=L f (⊤L)" "⊥fpl L f .=L f (⊥L)"
proof -
  interpret L: weak_complete_lattice "L"
    by (simp_all add: assms)
  interpret L': weak_complete_lattice "fpl L f"
    by (rule Knaster_Tarski, simp_all add: assms)
  have FA: "fps L f ⊆ carrier L"
    by (auto simp add: fps_def)
  show "⊤fpl L f .=L f (⊤L)"
  proof -
    from FA have "⊤fpl L f ∈ carrier L"
    proof -
      have "⊤fpl L f ∈ fps L f"
        using L'.top_closed by auto
      thus ?thesis
        using FA by blast
    qed
    moreover with assms have "f ⊤L ∈ carrier L"
      by (auto)

    ultimately show ?thesis
      using L.trans[OF Knaster_Tarski_top[of L f] L.GFP_idem[of f]]
      by (simp_all add: assms)
  qed
  show "⊥fpl L f .=L f (⊥L)"
  proof -
    from FA have "⊥fpl L f ∈ carrier L"
    proof -
      have "⊥fpl L f ∈ fps L f"
        using L'.bottom_closed by auto
      thus ?thesis
        using FA by blast
    qed
    moreover with assms have "f ⊥L ∈ carrier L"
      by (auto)

    ultimately show ?thesis
      using L.trans[OF Knaster_Tarski_bottom[of L f] L.LFP_idem[of f]]
      by (simp_all add: assms)
  qed
qed
```

```
theorem Knaster_Tarski_idem_inf_eq:
  assumes "weak_complete_lattice L" "isotone L L f" "idempotent L f"
"f ∈ carrier L → carrier L"
          "A ⊆ fps L f"
  shows "⨅fpl L f A .=L f (⨅L A)"
proof -
  interpret L: weak_complete_lattice "L"
    by (simp_all add: assms)
  interpret L': weak_complete_lattice "fpl L f"
    by (rule Knaster_Tarski, simp_all add: assms)
  have FA: "fps L f ⊆ carrier L"
    by (auto simp add: fps_def)
  have A: "A ⊆ carrier L"
    using FA assms(5) by blast
  have fA: "f (⨅LA) ∈ fps L f"
    by (metis (no_types, lifting) A L.idempotent L.inf_closed PiE assms(3)
assms(4) fps_def mem_Collect_eq)
  have infA: "⨅fpl L fA ∈ fps L f"
    by (rule L'.inf_closed[simplified], simp add: assms)
  show ?thesis
  proof (rule L.weak_le_antisym)
    show ic: "⨅fpl L fA ∈ carrier L"
      using FA infA by blast
    show fc: "f (⨅LA) ∈ carrier L"
      using FA fA by blast
    show "f (⨅LA) ⊑L ⨅fpl L fA"
    proof -
      have "⋀x. x ∈ A ⟹ f (⨅LA) ⊑L x"
        by (meson A FA L.inf_closed L.inf_lower L.le_trans L.weak_sup_post_fixed_point
assms(2) assms(4) assms(5) fA subsetCE)
      hence "f (⨅LA) ⊑fpl L f ⨅fpl L fA"
        by (rule_tac L'.inf_greatest, simp_all add: fA assms(3,5))
      thus ?thesis
        by (simp)
    qed
    show "⨅fpl L fA ⊑L f (⨅LA)"
    proof -
      have *: "⨅fpl L fA ∈ carrier L"
        using FA infA by blast
      have "⋀x. x ∈ A ⟹ ⨅fpl L fA ⊑fpl L f x"
        by (rule L'.inf_lower, simp_all add: assms)
      hence "⨅fpl L fA ⊑L (⨅LA)"
        by (rule_tac L.inf_greatest, simp_all add: A *)
      hence 1:"f(⨅fpl L fA) ⊑L f(⨅LA)"
        by (metis (no_types, lifting) A FA L.inf_closed assms(2) infA
subsetCE use_iso1)
      have 2:"⨅fpl L fA ⊑L f (⨅fpl L fA)"
        by (metis (no_types, lifting) FA L.sym L.use_fps L.weak_complete_lattice_axioms
```

```
PiE assms(4) infA subsetCE weak_complete_lattice_def weak_partial_order.weak_refl)
      show ?thesis
        using FA fA infA by (auto intro!: L.le_trans[OF 2 1] ic fc, metis
FA PiE assms(4) subsetCE)
    qed
  qed
qed
```

## 4.8 Examples

### 4.8.1 The Powerset of a Set is a Complete Lattice

```
theorem powerset_is_complete_lattice:
  "complete_lattice (|carrier = Pow A, eq = (=), le = (⊆)|)"
  (is "complete_lattice ?L")
proof (rule partial_order.complete_latticeI)
  show "partial_order ?L"
    by standard auto
next
  fix B
  assume "B ⊆ carrier ?L"
  then have "least ?L (⋃ B) (Upper ?L B)"
    by (fastforce intro!: least_UpperI simp: Upper_def)
  then show "∃s. least ?L s (Upper ?L B)" ..
next
  fix B
  assume "B ⊆ carrier ?L"
  then have "greatest ?L (⋂ B ∩ A) (Lower ?L B)"
```

⋂ B is not the infimum of B: ⋂ {} = UNIV which is in general bigger than A!

```
    by (fastforce intro!: greatest_LowerI simp: Lower_def)
  then show "∃i. greatest ?L i (Lower ?L B)" ..
qed
```

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

## 4.9 Limit preserving functions

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

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

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

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

lemma weak_sup_pres:
  "sup_pres X Y f ⟹ weak_sup_pres X Y f"
  by (simp add: sup_pres_def weak_sup_pres_def)

lemma weak_inf_pres:
  "inf_pres X Y f ⟹ weak_inf_pres X Y f"
  by (simp add: inf_pres_def weak_inf_pres_def)

lemma sup_pres_is_join_pres:
  assumes "weak_sup_pres X Y f"
  shows "join_pres X Y f"
  using assms by (auto simp: join_pres_def weak_sup_pres_def join_def)

lemma inf_pres_is_meet_pres:
  assumes "weak_inf_pres X Y f"
  shows "meet_pres X Y f"
  using assms by (auto simp: meet_pres_def weak_inf_pres_def meet_def)

end


theory Galois_Connection
  imports Complete_Lattice
begin
```

# 5 Galois connections

## 5.1 Definition and basic properties

```
record ('a, 'b, 'c, 'd) galcon =
  orderA :: "('a, 'c) gorder_scheme" ("𝒳ı")
  orderB :: "('b, 'd) gorder_scheme" ("𝒴ı")
  lower  :: "'a ⇒ 'b" ("π*ı")
  upper  :: "'b ⇒ 'a" ("π⇩*ı")

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

abbreviation "inv_galcon G ≡ ⦇ orderA = inv_gorder 𝒴G, orderB = inv_gorder
𝒳G, lower = upper G, upper = lower G ⦈"
```

```
definition comp_galcon :: "('b, 'c) galois ⇒ ('a, 'b) galois ⇒ ('a, 'c)
galois" (infixr ‹∘g› 85)
  where "G ∘g F = ⦇ orderA = orderA F, orderB = orderB G, lower = lower
G ∘ lower F, upper = upper F ∘ upper G ⦈"

definition id_galcon :: "'a gorder ⇒ ('a, 'a) galois" (‹Ig›) where
"Ig(A) = ⦇ orderA = A, orderB = A, lower = id, upper = id ⦈"
```

## 5.2 Well-typed connections

```
locale connection =
  fixes G (structure)
  assumes is_order_A: "partial_order 𝒳"
  and is_order_B: "partial_order 𝒴"
  and lower_closure: "π* ∈ carrier 𝒳 → carrier 𝒴"
  and upper_closure: "π⇩* ∈ carrier 𝒴 → carrier 𝒳"
begin

  lemma lower_closed: "x ∈ carrier 𝒳 ⟹ π* x ∈ carrier 𝒴"
    using lower_closure by auto

  lemma upper_closed: "y ∈ carrier 𝒴 ⟹ π⇩* y ∈ carrier 𝒳"
    using upper_closure by auto

end
```

## 5.3 Galois connections

```
locale galois_connection = connection +
  assumes galois_property: "⟦x ∈ carrier 𝒳; y ∈ carrier 𝒴⟧ ⟹ π* x
⊑𝒴 y ⟷ x ⊑𝒳 π⇩* y"
begin

  lemma is_weak_order_A: "weak_partial_order 𝒳"
  proof -
    interpret po: partial_order 𝒳
      by (metis is_order_A)
    show ?thesis ..
  qed

  lemma is_weak_order_B: "weak_partial_order 𝒴"
  proof -
    interpret po: partial_order 𝒴
      by (metis is_order_B)
    show ?thesis ..
  qed

  lemma right: "⟦x ∈ carrier 𝒳; y ∈ carrier 𝒴; π* x ⊑𝒴 y⟧ ⟹ x ⊑𝒳
π⇩* y"
```

```
    by (metis galois_property)

  lemma left: "⟦x ∈ carrier 𝒳; y ∈ carrier 𝒴; x ⊑𝒳 π* y⟧ ⟹ π^* x
⊑𝒴 y"
    by (metis galois_property)

  lemma deflation: "y ∈ carrier 𝒴 ⟹ π^* (π* y) ⊑𝒴 y"
    by (metis Pi_iff is_weak_order_A left upper_closure weak_partial_order.le_refl)

  lemma inflation: "x ∈ carrier 𝒳 ⟹ x ⊑𝒳 π* (π^* x)"
    by (metis (no_types, lifting) PiE galois_connection.right galois_connection_axioms
is_weak_order_B lower_closure weak_partial_order.le_refl)

  lemma lower_iso: "isotone 𝒳 𝒴 π^*"
  proof (auto simp add:isotone_def)
    show "weak_partial_order 𝒳"
      by (metis is_weak_order_A)
    show "weak_partial_order 𝒴"
      by (metis is_weak_order_B)
    fix x y
    assume a: "x ∈ carrier 𝒳" "y ∈ carrier 𝒳" "x ⊑𝒳 y"
    have b: "π^* y ∈ carrier 𝒴"
      using a(2) lower_closure by blast
    then have "π* (π^* y) ∈ carrier 𝒳"
      using upper_closure by blast
    then have "x ⊑𝒳 π* (π^* y)"
      by (meson a inflation is_weak_order_A weak_partial_order.le_trans)
    thus "π^* x ⊑𝒴 π^* y"
      by (meson b a(1) Pi_iff galois_property lower_closure upper_closure)
  qed

  lemma upper_iso: "isotone 𝒴 𝒳 π*"
    apply (auto simp add:isotone_def)
    apply (metis is_weak_order_B)
    apply (metis is_weak_order_A)
    apply (metis (no_types, lifting) Pi_mem deflation is_weak_order_B
lower_closure right upper_closure weak_partial_order.le_trans)
  done

  lemma lower_comp: "x ∈ carrier 𝒳 ⟹ π^* (π* (π^* x)) = π^* x"
    by (meson deflation funcset_mem inflation is_order_B lower_closure
lower_iso partial_order.le_antisym upper_closure use_iso2)

  lemma lower_comp': "x ∈ carrier 𝒳 ⟹ (π^* ∘ π* ∘ π^*) x = π^* x"
    by (simp add: lower_comp)

  lemma upper_comp: "y ∈ carrier 𝒴 ⟹ π* (π^* (π* y)) = π* y"
  proof -
    assume a1: "y ∈ carrier 𝒴"
```

```
    hence f1: "π⇩* y ∈ carrier 𝒳" using upper_closure by blast
    have f2: "π⇧* (π⇩* y) ⊑⇘𝒴⇙ y" using a1 deflation by blast
    have f3: "π⇩* (π⇧* (π⇩* y)) ∈ carrier 𝒳"
      using f1 lower_closure upper_closure by auto
    have "π⇧* (π⇩* y) ∈ carrier 𝒴" using f1 lower_closure by blast
    thus "π⇩* (π⇧* (π⇩* y)) = π⇩* y"
      by (meson a1 f1 f2 f3 inflation is_order_A partial_order.le_antisym
upper_iso use_iso2)
  qed

  lemma upper_comp': "y ∈ carrier 𝒴 ⟹ (π⇩* ∘ π⇧* ∘ π⇩*) y = π⇩* y"
    by (simp add: upper_comp)

  lemma adjoint_idem1: "idempotent 𝒴 (π⇧* ∘ π⇩*)"
    by (simp add: idempotent_def is_order_B partial_order.eq_is_equal
upper_comp)

  lemma adjoint_idem2: "idempotent 𝒳 (π⇩* ∘ π⇧*)"
    by (simp add: idempotent_def is_order_A partial_order.eq_is_equal
lower_comp)

  lemma fg_iso: "isotone 𝒴 𝒴 (π⇧* ∘ π⇩*)"
    by (metis iso_compose lower_closure lower_iso upper_closure upper_iso)

  lemma gf_iso: "isotone 𝒳 𝒳 (π⇩* ∘ π⇧*)"
    by (metis iso_compose lower_closure lower_iso upper_closure upper_iso)

  lemma semi_inverse1: "x ∈ carrier 𝒳 ⟹ π⇧* x = π⇧* (π⇩* (π⇧* x))"
    by (metis lower_comp)

  lemma semi_inverse2: "x ∈ carrier 𝒴 ⟹ π⇩* x = π⇩* (π⇧* (π⇩* x))"
    by (metis upper_comp)

  theorem lower_by_complete_lattice:
    assumes "complete_lattice 𝒴" "x ∈ carrier 𝒳"
    shows "π⇧*(x) = ⨅⇘𝒴⇙ { y ∈ carrier 𝒴. x ⊑⇘𝒳⇙ π⇩*(y) }"
  proof -
    interpret Y: complete_lattice 𝒴
      by (simp add: assms)

    show ?thesis
    proof (rule Y.le_antisym)
      show x: "π⇧* x ∈ carrier 𝒴"
        using assms(2) lower_closure by blast
      show "π⇧* x ⊑⇘𝒴⇙ ⨅⇘𝒴⇙{y ∈ carrier 𝒴. x ⊑⇘𝒳⇙ π⇩* y}"
      proof (rule Y.weak.inf_greatest)
        show "{y ∈ carrier 𝒴. x ⊑⇘𝒳⇙ π⇩* y} ⊆ carrier 𝒴"
          by auto
        show "π⇧* x ∈ carrier 𝒴" by (fact x)
```

```
        fix z
        assume "z ∈ {y ∈ carrier 𝒴. x ⊑𝒳 π⇩* y}"
        thus "π^* x ⊑𝒴 z"
          using assms(2) left by auto
      qed
      show "⨅𝒴{y ∈ carrier 𝒴. x ⊑𝒳 π⇩* y} ⊑𝒴 π^* x"
      proof (rule Y.weak.inf_lower)
        show "{y ∈ carrier 𝒴. x ⊑𝒳 π⇩* y} ⊆ carrier 𝒴"
          by auto
        show "π^* x ∈ {y ∈ carrier 𝒴. x ⊑𝒳 π⇩* y}"
        proof (auto)
          show "π^* x ∈ carrier 𝒴" by (fact x)
          show "x ⊑𝒳 π⇩* (π^* x)"
            using assms(2) inflation by blast
        qed
      qed
      show "⨅𝒴{y ∈ carrier 𝒴. x ⊑𝒳 π⇩* y} ∈ carrier 𝒴"
       by (auto intro: Y.weak.inf_closed)
    qed
  qed

  theorem upper_by_complete_lattice:
    assumes "complete_lattice 𝒳" "y ∈ carrier 𝒴"
    shows "π⇩*(y) = ⨆𝒳 { x ∈ carrier 𝒳. π^*(x) ⊑𝒴 y }"
  proof -
    interpret X: complete_lattice 𝒳
      by (simp add: assms)
    show ?thesis
    proof (rule X.le_antisym)
      show y: "π⇩* y ∈ carrier 𝒳"
        using assms(2) upper_closure by blast
      show "π⇩* y ⊑𝒳 ⨆𝒳{x ∈ carrier 𝒳. π^* x ⊑𝒴 y}"
      proof (rule X.weak.sup_upper)
        show "{x ∈ carrier 𝒳. π^* x ⊑𝒴 y} ⊆ carrier 𝒳"
          by auto
        show "π⇩* y ∈ {x ∈ carrier 𝒳. π^* x ⊑𝒴 y}"
        proof (auto)
          show "π⇩* y ∈ carrier 𝒳" by (fact y)
          show "π^* (π⇩* y) ⊑𝒴 y"
            by (simp add: assms(2) deflation)
        qed
      qed
      show "⨆𝒳{x ∈ carrier 𝒳. π^* x ⊑𝒴 y} ⊑𝒳 π⇩* y"
      proof (rule X.weak.sup_least)
        show "{x ∈ carrier 𝒳. π^* x ⊑𝒴 y} ⊆ carrier 𝒳"
          by auto
        show "π⇩* y ∈ carrier 𝒳" by (fact y)
        fix z
        assume "z ∈ {x ∈ carrier 𝒳. π^* x ⊑𝒴 y}"
```

```
        thus "z ⊑𝒳 π∗ y"
          by (simp add: assms(2) right)
      qed
      show "⨆𝒳{x ∈ carrier 𝒳. π* x ⊑𝒴 y} ∈ carrier 𝒳"
       by (auto intro: X.weak.sup_closed)
    qed
  qed

end

lemma dual_galois [simp]: " galois_connection ⦇ orderA = inv_gorder B,
orderB = inv_gorder A, lower = f, upper = g ⦈
                          = galois_connection ⦇ orderA = A, orderB = B,
lower = g, upper = f ⦈"
  by (auto simp add: galois_connection_def galois_connection_axioms_def
connection_def dual_order_iff)

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

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

lemma lower_adjoint_dual [simp]: "lower_adjoint (inv_gorder A) (inv_gorder
B) f = upper_adjoint B A f"
  by (simp add: lower_adjoint_def upper_adjoint_def)

lemma upper_adjoint_dual [simp]: "upper_adjoint (inv_gorder A) (inv_gorder
B) f = lower_adjoint B A f"
  by (simp add: lower_adjoint_def upper_adjoint_def)

lemma lower_type: "lower_adjoint A B f ⟹ f ∈ carrier A → carrier
B"
  by (auto simp add:lower_adjoint_def galois_connection_def galois_connection_axioms_def
connection_def)

lemma upper_type: "upper_adjoint A B g ⟹ g ∈ carrier B → carrier
A"
  by (auto simp add:upper_adjoint_def galois_connection_def galois_connection_axioms_def
connection_def)
```

## 5.4 Composition of Galois connections

```
lemma id_galois: "partial_order A ⟹ galois_connection (I_g(A))"
  by (simp add: id_galcon_def galois_connection_def galois_connection_axioms_def
```

```
connection_def)

lemma comp_galcon_closed:
  assumes "galois_connection G" "galois_connection F" "𝒴F = 𝒳G"
  shows "galois_connection (G ∘g F)"
proof -
  interpret F: galois_connection F
    by (simp add: assms)
  interpret G: galois_connection G
    by (simp add: assms)

  have "partial_order 𝒳G ∘g F"
    by (simp add: F.is_order_A comp_galcon_def)
  moreover have "partial_order 𝒴G ∘g F"
    by (simp add: G.is_order_B comp_galcon_def)
  moreover have "π*G ∘ π*F ∈ carrier 𝒳F → carrier 𝒴G"
    using F.lower_closure G.lower_closure assms(3) by auto
  moreover have "π*F ∘ π*G ∈ carrier 𝒴G → carrier 𝒳F"
    using F.upper_closure G.upper_closure assms(3) by auto
  moreover
  have "⋀ x y. ⟦x ∈ carrier 𝒳F; y ∈ carrier 𝒴G ⟧ ⟹
              (π*G (π*F x) ⊑𝒴G y) = (x ⊑𝒳F π*F (π*G y))"
    by (metis F.galois_property F.lower_closure G.galois_property G.upper_closure
assms(3) Pi_iff)
  ultimately show ?thesis
    by (simp add: comp_galcon_def galois_connection_def galois_connection_axioms_def
connection_def)
qed

lemma comp_galcon_right_unit [simp]: "F ∘g Ig(𝒳F) = F"
  by (simp add: comp_galcon_def id_galcon_def)

lemma comp_galcon_left_unit [simp]: "Ig(𝒴F) ∘g F = F"
  by (simp add: comp_galcon_def id_galcon_def)

lemma galois_connectionI:
  assumes
    "partial_order A" "partial_order B"
    "L ∈ carrier A → carrier B" "R ∈ carrier B → carrier A"
    "isotone A B L" "isotone B A R"
    "⋀ x y. ⟦ x ∈ carrier A; y ∈ carrier B ⟧ ⟹ L x ⊑B y ⟷ x ⊑A R
y"
  shows "galois_connection ⦇ orderA = A, orderB = B, lower = L, upper
= R ⦈"
  using assms by (simp add: galois_connection_def connection_def galois_connection_axioms_d

lemma galois_connectionI':
  assumes
    "partial_order A" "partial_order B"
```

```
    "L ∈ carrier A → carrier B" "R ∈ carrier B → carrier A"
    "isotone A B L" "isotone B A R"
    "⋀ X. X ∈ carrier(B) ⟹ L(R(X)) ⊑B X"
    "⋀ X. X ∈ carrier(A) ⟹ X ⊑A R(L(X))"
  shows "galois_connection ⦇ orderA = A, orderB = B, lower = L, upper
= R ⦈"
  using assms
  by (auto simp add: galois_connection_def connection_def galois_connection_axioms_def,
(meson PiE isotone_def weak_partial_order.le_trans)+)
```

## 5.5 Retracts

```
locale retract = galois_connection +
  assumes retract_property: "x ∈ carrier 𝒳 ⟹ π⇩* (π⇧* x) ⊑𝒳 x"
begin
  lemma retract_inverse: "x ∈ carrier 𝒳 ⟹ π⇩* (π⇧* x) = x"
    by (meson funcset_mem inflation is_order_A lower_closure partial_order.le_antisym
retract_axioms retract_axioms_def retract_def upper_closure)

  lemma retract_injective: "inj_on π⇧* (carrier 𝒳)"
    by (metis inj_onI retract_inverse)
end

theorem comp_retract_closed:
  assumes "retract G" "retract F" "𝒴F = 𝒳G"
  shows "retract (G ∘g F)"
proof -
  interpret f: retract F
    by (simp add: assms)
  interpret g: retract G
    by (simp add: assms)
  interpret gf: galois_connection "(G ∘g F)"
    by (simp add: assms(1) assms(2) assms(3) comp_galcon_closed retract.axioms(1))
  show ?thesis
  proof
    fix x
    assume "x ∈ carrier 𝒳G ∘g F"
    thus "le 𝒳G ∘g F (π⇩*G ∘g F (π⇧*G ∘g F x)) x"
      using assms(3) f.inflation f.lower_closed f.retract_inverse g.retract_inverse
by (auto simp add: comp_galcon_def)
  qed
qed
```

## 5.6 Coretracts

```
locale coretract = galois_connection +
  assumes coretract_property: "y ∈ carrier 𝒴 ⟹ y ⊑𝒴 π⇧* (π⇩* y)"
begin
  lemma coretract_inverse: "y ∈ carrier 𝒴 ⟹ π⇧* (π⇩* y) = y"
```

```
    by (meson coretract_axioms coretract_axioms_def coretract_def deflation
funcset_mem is_order_B lower_closure partial_order.le_antisym upper_closure)

  lemma retract_injective: "inj_on π∗ (carrier 𝒴)"
    by (metis coretract_inverse inj_onI)
end

theorem comp_coretract_closed:
  assumes "coretract G" "coretract F" "𝒴F = 𝒳G"
  shows "coretract (G ∘g F)"
proof -
  interpret f: coretract F
    by (simp add: assms)
  interpret g: coretract G
    by (simp add: assms)
  interpret gf: galois_connection "(G ∘g F)"
    by (simp add: assms(1) assms(2) assms(3) comp_galcon_closed coretract.axioms(1))
  show ?thesis
  proof
    fix y
    assume "y ∈ carrier 𝒴G ∘g F"
    thus "le 𝒴G ∘g F y (π*G ∘g F (π∗G ∘g F y))"
      by (simp add: comp_galcon_def assms(3) f.coretract_inverse g.coretract_property
g.upper_closed)
  qed
qed
```

## 5.7 Galois Bijections

```
locale galois_bijection = connection +
  assumes lower_iso: "isotone 𝒳 𝒴 π*"
  and upper_iso: "isotone 𝒴 𝒳 π∗"
  and lower_inv_eq: "x ∈ carrier 𝒳 ⟹ π∗ (π* x) = x"
  and upper_inv_eq: "y ∈ carrier 𝒴 ⟹ π* (π∗ y) = y"
begin

  lemma lower_bij: "bij_betw π* (carrier 𝒳) (carrier 𝒴)"
    by (rule bij_betwI[where g="π∗"], auto intro: upper_inv_eq lower_inv_eq
upper_closed lower_closed)

  lemma upper_bij: "bij_betw π∗ (carrier 𝒴) (carrier 𝒳)"
    by (rule bij_betwI[where g="π*"], auto intro: upper_inv_eq lower_inv_eq
upper_closed lower_closed)

sublocale gal_bij_conn: galois_connection
  apply (unfold_locales, auto)
  using lower_closed lower_inv_eq upper_iso use_iso2 apply fastforce
  using lower_iso upper_closed upper_inv_eq use_iso2 apply fastforce
done
```

```
sublocale gal_bij_ret: retract
  by (unfold_locales, simp add: gal_bij_conn.is_weak_order_A lower_inv_eq
weak_partial_order.le_refl)

sublocale gal_bij_coret: coretract
  by (unfold_locales, simp add: gal_bij_conn.is_weak_order_B upper_inv_eq
weak_partial_order.le_refl)

end

theorem comp_galois_bijection_closed:
  assumes "galois_bijection G" "galois_bijection F" "𝒴F = 𝒳G"
  shows "galois_bijection (G ∘g F)"
proof -
  interpret f: galois_bijection F
    by (simp add: assms)
  interpret g: galois_bijection G
    by (simp add: assms)
  interpret gf: galois_connection "(G ∘g F)"
    by (simp add: assms(3) comp_galcon_closed f.gal_bij_conn.galois_connection_axioms
g.gal_bij_conn.galois_connection_axioms galois_connection.axioms(1))
  show ?thesis
  proof
    show "isotone 𝒳G ∘g F 𝒴G ∘g F π*G ∘g F"
      by (simp add: comp_galcon_def, metis comp_galcon_def galcon.select_convs(1)
galcon.select_convs(2) galcon.select_convs(3) gf.lower_iso)
    show "isotone 𝒴G ∘g F 𝒳G ∘g F π*G ∘g F"
      by (simp add: gf.upper_iso)
    fix x
    assume "x ∈ carrier 𝒳G ∘g F"
    thus "π*G ∘g F (π*G ∘g F x) = x"
      using assms(3) f.lower_closed f.lower_inv_eq g.lower_inv_eq by (auto
simp add: comp_galcon_def)
  next
    fix y
    assume "y ∈ carrier 𝒴G ∘g F"
    thus "π*G ∘g F (π*G ∘g F y) = y"
      by (simp add: comp_galcon_def assms(3) f.upper_inv_eq g.upper_closed
g.upper_inv_eq)
  qed
qed

end


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

# 6 Monoids and Groups

## 6.1 Definitions

Definitions follow [3].

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

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

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

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

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"
  by (fast intro!: monoid.intro intro: assms)
```

```
lemma (in monoid) Units_closed [dest]:
  "x ∈ Units G ==> x ∈ carrier G"
  by (unfold Units_def) fast

lemma (in monoid) one_unique:
  assumes "u ∈ carrier G"
    and "⋀x. x ∈ carrier G ⟹ u ⊗ x = x"
  shows "u = 𝟭"
  using assms(2)[OF one_closed] r_one[OF assms(1)] by simp

lemma (in monoid) inv_unique:
  assumes eq: "y ⊗ x = 𝟭"  "x ⊗ y' = 𝟭"
    and G: "x ∈ carrier G"  "y ∈ carrier G"  "y' ∈ carrier G"
  shows "y = y'"
proof -
  from G eq have "y = y ⊗ (x ⊗ y')" by simp
  also from G have "... = (y ⊗ x) ⊗ y'" by (simp add: m_assoc)
  also from G eq have "... = y'" by simp
  finally show ?thesis .
qed

lemma (in monoid) Units_m_closed [simp, intro]:
  assumes x: "x ∈ Units G" and y: "y ∈ Units G"
  shows "x ⊗ y ∈ Units G"
proof -
  from x obtain x' where x: "x ∈ carrier G" "x' ∈ carrier G" and xinv:
"x ⊗ x' = 𝟭" "x' ⊗ x = 𝟭"
    unfolding Units_def by fast
  from y obtain y' where y: "y ∈ carrier G" "y' ∈ carrier G" and yinv:
"y ⊗ y' = 𝟭" "y' ⊗ y = 𝟭"
    unfolding Units_def by fast
  from x y xinv yinv have "y' ⊗ (x' ⊗ x) ⊗ y = 𝟭" by simp
  moreover from x y xinv yinv have "x ⊗ (y ⊗ y') ⊗ x' = 𝟭" by simp
  moreover note x y
  ultimately show ?thesis unfolding Units_def
    by simp (metis m_assoc m_closed)
qed

lemma (in monoid) Units_one_closed [intro, simp]:
  "𝟭 ∈ Units G"
  by (unfold Units_def) auto

lemma (in monoid) Units_inv_closed [intro, simp]:
  "x ∈ Units G ==> inv x ∈ carrier G"
  apply (simp add: Units_def m_inv_def)
  by (metis (mono_tags, lifting) inv_unique the_equality)

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

```
  by (unfold Units_def) auto

lemma (in monoid) Units_r_inv_ex:
  "x ∈ Units G ==> ∃y ∈ carrier G. x ⊗ y = 1"
  by (unfold Units_def) auto

lemma (in monoid) Units_l_inv [simp]:
  "x ∈ Units G ==> inv x ⊗ x = 1"
  apply (unfold Units_def m_inv_def, simp)
  by (metis (mono_tags, lifting) inv_unique the_equality)

lemma (in monoid) Units_r_inv [simp]:
  "x ∈ Units G ==> x ⊗ inv x = 1"
  by (metis (full_types) Units_closed Units_inv_closed Units_l_inv Units_r_inv_ex
inv_unique)

lemma (in monoid) inv_one [simp]:
  "inv 1 = 1"
  by (metis Units_one_closed Units_r_inv l_one monoid.Units_inv_closed
monoid_axioms)

lemma (in monoid) Units_inv_Units [intro, simp]:
  "x ∈ Units G ==> inv x ∈ Units G"
proof -
  assume x: "x ∈ Units G"
  show "inv x ∈ Units G"
    by (auto simp add: Units_def
      intro: Units_l_inv Units_r_inv x Units_closed [OF x])
qed

lemma (in monoid) Units_l_cancel [simp]:
  "[| x ∈ Units G; y ∈ carrier G; z ∈ carrier G |] ==>
   (x ⊗ y = x ⊗ z) = (y = z)"
proof
  assume eq: "x ⊗ y = x ⊗ z"
    and G: "x ∈ Units G"  "y ∈ carrier G"  "z ∈ carrier G"
  then have "(inv x ⊗ x) ⊗ y = (inv x ⊗ x) ⊗ z"
    by (simp add: m_assoc Units_closed del: Units_l_inv)
  with G show "y = z" by simp
next
  assume eq: "y = z"
    and G: "x ∈ Units G"  "y ∈ carrier G"  "z ∈ carrier G"
  then show "x ⊗ y = x ⊗ z" by simp
qed

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

```
  then have "inv x ⊗ inv (inv x) = inv x ⊗ x" by simp
  with x show ?thesis by (simp add: Units_closed del: Units_l_inv Units_r_inv)
qed

lemma (in monoid) inv_inj_on_Units:
  "inj_on (m_inv G) (Units G)"
proof (rule inj_onI)
  fix x y
  assume G: "x ∈ Units G"  "y ∈ Units G" and eq: "inv x = inv y"
  then have "inv (inv x) = inv (inv y)" by simp
  with G show "x = y" by simp
qed

lemma (in monoid) Units_inv_comm:
  assumes inv: "x ⊗ y = 1"
    and G: "x ∈ Units G"  "y ∈ Units G"
  shows "y ⊗ x = 1"
proof -
  from G have "x ⊗ y ⊗ x = x ⊗ 1" by (auto simp add: inv Units_closed)
  with G show ?thesis by (simp del: r_one add: m_assoc Units_closed)
qed

lemma (in monoid) carrier_not_empty: "carrier G ≠ {}"
by auto
```

## 6.2 Groups

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

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

lemma (in group) is_group [iff]: "group G" by (rule group_axioms)

lemma (in group) is_monoid [iff]: "monoid G"
  by (rule monoid_axioms)

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

```
  have l_cancel [simp]:
    "!!x y z. [| x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |] ==>
    (x ⊗ y = x ⊗ z) = (y = z)"
  proof
    fix x y z
    assume eq: "x ⊗ y = x ⊗ z"
      and G: "x ∈ carrier G"  "y ∈ carrier G"  "z ∈ carrier G"
    with l_inv_ex obtain x_inv where xG: "x_inv ∈ carrier G"
      and l_inv: "x_inv ⊗ x = 1" by fast
    from G eq xG have "(x_inv ⊗ x) ⊗ y = (x_inv ⊗ x) ⊗ z"
      by (simp add: m_assoc)
    with G show "y = z" by (simp add: l_inv)
  next
    fix x y z
    assume eq: "y = z"
      and G: "x ∈ carrier G"  "y ∈ carrier G"  "z ∈ carrier G"
    then show "x ⊗ y = x ⊗ z" by simp
  qed
  have r_one:
    "!!x. x ∈ carrier G ==> x ⊗ 1 = x"
  proof -
    fix x
    assume x: "x ∈ carrier G"
    with l_inv_ex obtain x_inv where xG: "x_inv ∈ carrier G"
      and l_inv: "x_inv ⊗ x = 1" by fast
    from x xG have "x_inv ⊗ (x ⊗ 1) = x_inv ⊗ x"
      by (simp add: m_assoc [symmetric] l_inv)
    with x xG show "x ⊗ 1 = x" by simp
  qed
  have inv_ex:
    "⋀x. x ∈ carrier G ⟹ ∃y ∈ carrier G. y ⊗ x = 1 ∧ x ⊗ y = 1"
  proof -
    fix x
    assume x: "x ∈ carrier G"
    with l_inv_ex obtain y where y: "y ∈ carrier G"
      and l_inv: "y ⊗ x = 1" by fast
    from x y have "y ⊗ (x ⊗ y) = y ⊗ 1"
      by (simp add: m_assoc [symmetric] l_inv r_one)
    with x y have r_inv: "x ⊗ y = 1"
      by simp
    from x y show "∃y ∈ carrier G. y ⊗ x = 1 ∧ x ⊗ y = 1"
      by (fast intro: l_inv r_inv)
  qed
  then have carrier_subset_Units: "carrier G ⊆ Units G"
    by (unfold Units_def) fast
  show ?thesis
    by standard (auto simp: r_one m_assoc carrier_subset_Units)
qed
```

```
lemma (in monoid) group_l_invI:
  assumes l_inv_ex:
    "!!x. x ∈ carrier G ==> ∃y ∈ carrier G. y ⊗ x = 1"
  shows "group G"
  by (rule groupI) (auto intro: m_assoc l_inv_ex)

lemma (in group) Units_eq [simp]:
  "Units G = carrier G"
proof
  show "Units G ⊆ carrier G" by fast
next
  show "carrier G ⊆ Units G" by (rule Units)
qed

lemma (in group) inv_closed [intro, simp]:
  "x ∈ carrier G ==> inv x ∈ carrier G"
  using Units_inv_closed by simp

lemma (in group) l_inv_ex [simp]:
  "x ∈ carrier G ==> ∃y ∈ carrier G. y ⊗ x = 1"
  using Units_l_inv_ex by simp

lemma (in group) r_inv_ex [simp]:
  "x ∈ carrier G ==> ∃y ∈ carrier G. x ⊗ y = 1"
  using Units_r_inv_ex by simp

lemma (in group) l_inv [simp]:
  "x ∈ carrier G ==> inv x ⊗ x = 1"
  by simp
```

## 6.3 Cancellation Laws and Basic Properties

```
lemma (in group) inv_eq_1_iff [simp]:
  assumes "x ∈ carrier G" shows "inv_G x = 1_G ⟷ x = 1_G"
proof -
  have "x = 1" if "inv x = 1"
  proof -
    have "inv x ⊗ x = 1"
      using assms l_inv by blast
    then show "x = 1"
      using that assms by simp
  qed
  then show ?thesis
    by auto
qed

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

```
lemma (in group) right_cancel [simp]:
  "[| x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |] ==>
   (y ⊗ x = z ⊗ x) = (y = z)"
  by (metis inv_closed m_assoc r_inv r_one)

lemma (in group) inv_inv [simp]:
  "x ∈ carrier G ==> inv (inv x) = x"
  using Units_inv_inv by simp

lemma (in group) inv_inj:
  "inj_on (m_inv G) (carrier G)"
  using inv_inj_on_Units by simp

lemma (in group) inv_mult_group:
  "[| x ∈ carrier G; y ∈ carrier G |] ==> inv (x ⊗ y) = inv y ⊗ inv x"
proof -
  assume G: "x ∈ carrier G"  "y ∈ carrier G"
  then have "inv (x ⊗ y) ⊗ (x ⊗ y) = (inv y ⊗ inv x) ⊗ (x ⊗ y)"
    by (simp add: m_assoc) (simp add: m_assoc [symmetric])
  with G show ?thesis by (simp del: l_inv Units_l_inv)
qed

lemma (in group) inv_comm:
  "[| x ⊗ y = 1; x ∈ carrier G; y ∈ carrier G |] ==> y ⊗ x = 1"
  by (rule Units_inv_comm) auto

lemma (in group) inv_equality:
     "[|y ⊗ x = 1; x ∈ carrier G; y ∈ carrier G|] ==> inv x = y"
  using inv_unique r_inv by blast

lemma (in group) inv_solve_left:
  "⟦ a ∈ carrier G; b ∈ carrier G; c ∈ carrier G ⟧ ⟹ a = inv b ⊗ c
⟷ c = b ⊗ a"
  by (metis inv_equality l_inv_ex l_one m_assoc r_inv)

lemma (in group) inv_solve_left':
  "⟦ a ∈ carrier G; b ∈ carrier G; c ∈ carrier G ⟧ ⟹ inv b ⊗ c = a
⟷ c = b ⊗ a"
  by (metis inv_equality l_inv_ex l_one m_assoc r_inv)

lemma (in group) inv_solve_right:
  "⟦ a ∈ carrier G; b ∈ carrier G; c ∈ carrier G ⟧ ⟹ a = b ⊗ inv c
⟷ b = a ⊗ c"
  by (metis inv_equality l_inv_ex l_one m_assoc r_inv)

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

```
  by (auto simp: m_assoc)
```

## 6.4 Power

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

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

lemma (in monoid) nat_pow_closed [intro, simp]:
  "x ∈ carrier G ==> x [^] (n::nat) ∈ carrier G"
  by (induct n) (simp_all add: nat_pow_def)

lemma (in monoid) nat_pow_0 [simp]:
  "x [^] (0::nat) = 1"
  by (simp add: nat_pow_def)

lemma (in monoid) nat_pow_Suc [simp]:
  "x [^] (Suc n) = x [^] n ⊗ x"
  by (simp add: nat_pow_def)

lemma (in monoid) nat_pow_one [simp]:
  "1 [^] (n::nat) = 1"
  by (induct n) simp_all

lemma (in monoid) nat_pow_mult:
  "x ∈ carrier G ==> x [^] (n::nat) ⊗ x [^] m = x [^] (n + m)"
  by (induct m) (simp_all add: m_assoc [THEN sym])

lemma (in monoid) nat_pow_comm:
  "x ∈ carrier G ⟹ (x [^] (n::nat)) ⊗ (x [^] (m :: nat)) = (x [^] m)
⊗ (x [^] n)"
  using nat_pow_mult[of x n m] nat_pow_mult[of x m n] by (simp add: add.commute)

lemma (in monoid) nat_pow_Suc2:
  "x ∈ carrier G ⟹ x [^] (Suc n) = x ⊗ (x [^] n)"
  using nat_pow_mult[of x 1 n] Suc_eq_plus1[of n]
  by (metis One_nat_def Suc_eq_plus1_left l_one nat.rec(1) nat_pow_Suc
nat_pow_def)

lemma (in monoid) nat_pow_pow:
  "x ∈ carrier G ==> (x [^] n) [^] m = x [^] (n * m::nat)"
  by (induct m) (simp, simp add: nat_pow_mult add.commute)

lemma (in monoid) nat_pow_consistent:
```

```
  "x [^] (n :: nat) = x [^](G (| carrier := H |)) n"
  unfolding nat_pow_def by simp

lemma nat_pow_0 [simp]: "x [^]G (0::nat) = 1G"
  by (simp add: nat_pow_def)

lemma nat_pow_Suc [simp]: "x [^]G (Suc n) = (x [^]G n)⊗G x"
  by (simp add: nat_pow_def)

lemma (in group) nat_pow_inv:
  assumes "x ∈ carrier G" shows "(inv x) [^] (i :: nat) = inv (x [^]
i)"
proof (induction i)
  case 0 thus ?case by simp
next
  case (Suc i)
  have "(inv x) [^] Suc i = ((inv x) [^] i) ⊗ inv x"
    by simp
  also have " ... = (inv (x [^] i)) ⊗ inv x"
    by (simp add: Suc.IH Suc.prems)
  also have " ... = inv (x ⊗ (x [^] i))"
    by (simp add: assms inv_mult_group)
  also have " ... = inv (x [^] (Suc i))"
    using assms nat_pow_Suc2 by auto
  finally show ?case .
qed

overloading int_pow == "pow :: [_, 'a, int] => 'a"
begin
  definition "int_pow G a z =
   (let p = rec_nat 1G (%u b. b ⊗G a)
    in if z < 0 then invG (p (nat (-z))) else p (nat z))"
end

lemma int_pow_int: "x [^]G (int n) = x [^]G n"
  by(simp add: int_pow_def nat_pow_def)

lemma pow_nat:
  assumes "i≥0"
  shows "x [^]G nat i = x [^]G i"
proof (cases i rule: int_cases)
  case (nonneg n)
  then show ?thesis
    by (simp add: int_pow_int)
next
  case (neg n)
  then show ?thesis
    using assms by linarith
qed
```

```
lemma int_pow_0 [simp]: "x [^]G (0::int) = 1G"
  by (simp add: int_pow_def)

lemma int_pow_def2: "a [^]G z =
   (if z < 0 then invG (a [^]G (nat (-z))) else a [^]G (nat z))"
  by (simp add: int_pow_def nat_pow_def)

lemma (in group) int_pow_one [simp]:
  "1 [^] (z::int) = 1"
  by (simp add: int_pow_def2)

lemma (in group) int_pow_closed [intro, simp]:
  "x ∈ carrier G ==> x [^] (i::int) ∈ carrier G"
  by (simp add: int_pow_def2)

lemma (in group) int_pow_1 [simp]:
  "x ∈ carrier G ⟹ x [^] (1::int) = x"
  by (simp add: int_pow_def2)

lemma (in group) int_pow_neg:
  "x ∈ carrier G ⟹ x [^] (-i::int) = inv (x [^] i)"
  by (simp add: int_pow_def2)

lemma (in group) int_pow_neg_int: "x ∈ carrier G ⟹ x [^] -(int n) =
inv (x [^] n)"
  by (simp add: int_pow_neg int_pow_int)

lemma (in group) int_pow_mult:
  assumes "x ∈ carrier G" shows "x [^] (i + j::int) = x [^] i ⊗ x [^]
j"
proof -
  have [simp]: "-i - j = -j - i" by simp
  show ?thesis
    by (auto simp: assms int_pow_def2 inv_solve_left inv_solve_right nat_add_distrib
[symmetric] nat_pow_mult)
qed

lemma (in group) int_pow_inv:
  "x ∈ carrier G ⟹ (inv x) [^] (i :: int) = inv (x [^] i)"
  by (metis int_pow_def2 nat_pow_inv)

lemma (in group) int_pow_pow:
  assumes "x ∈ carrier G"
  shows "(x [^] (n :: int)) [^] (m :: int) = x [^] (n * m :: int)"
proof (cases)
  assume n_ge: "n ≥ 0" thus ?thesis
  proof (cases)
    assume m_ge: "m ≥ 0" thus ?thesis
```

```
      using n_ge nat_pow_pow[OF assms, of "nat n" "nat m"] int_pow_def2
[where G=G]
      by (simp add: mult_less_0_iff nat_mult_distrib)
  next
    assume m_lt: "¬ m ≥ 0"
    with n_ge show ?thesis
      apply (simp add: int_pow_def2 mult_less_0_iff)
      by (metis assms mult_minus_right n_ge nat_mult_distrib nat_pow_pow)
  qed
next
  assume n_lt: "¬ n ≥ 0" thus ?thesis
  proof (cases)
    assume m_ge: "m ≥ 0"
    have "inv x [^] (nat m * nat (- n)) = inv x [^] nat (- (m * n))"
      by (metis (full_types) m_ge mult_minus_right nat_mult_distrib)
    with m_ge n_lt show ?thesis
      by (simp add: int_pow_def2 mult_less_0_iff assms mult.commute nat_pow_inv
nat_pow_pow)
  next
    assume m_lt: "¬ m ≥ 0" thus ?thesis
      using n_lt by (auto simp: int_pow_def2 mult_less_0_iff assms nat_mult_distrib_neg
nat_pow_inv nat_pow_pow)
  qed
qed

lemma (in group) int_pow_diff:
  "x ∈ carrier G ⟹ x [^] (n - m :: int) = x [^] n ⊗ inv (x [^] m)"
  by(simp only: diff_conv_add_uminus int_pow_mult int_pow_neg)

lemma (in group) inj_on_multc: "c ∈ carrier G ⟹ inj_on (λx. x ⊗ c)
(carrier G)"
  by(simp add: inj_on_def)

lemma (in group) inj_on_cmult: "c ∈ carrier G ⟹ inj_on (λx. c ⊗ x)
(carrier G)"
  by(simp add: inj_on_def)


lemma (in monoid) group_commutes_pow:
  fixes n::nat
  shows "⟦x ⊗ y = y ⊗ x; x ∈ carrier G; y ∈ carrier G⟧ ⟹ x [^] n ⊗
y = y ⊗ x [^] n"
  apply (induction n, auto)
  by (metis m_assoc nat_pow_closed)

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

```
proof (induct n)
  case (Suc n)
  have "x ⊗ (y [^] n ⊗ y) = y [^] n ⊗ x ⊗ y"
    by (simp add: eq group_commutes_pow m_assoc xy)
  then show ?case
    using assms Suc.hyps m_assoc by auto
qed auto

lemma (in group) int_pow_mult_distrib:
  assumes eq: "x ⊗ y = y ⊗ x" and xy: "x ∈ carrier G" "y ∈ carrier
G"
  shows "(x ⊗ y) [^] (i::int) = x [^] i ⊗ y [^] i"
proof (cases i rule: int_cases)
  case (nonneg n)
  then show ?thesis
    by (metis eq int_pow_int pow_mult_distrib xy)
next
  case (neg n)
  then show ?thesis
    unfolding neg
    apply (simp add: xy int_pow_neg_int del: of_nat_Suc)
    by (metis eq inv_mult_group local.nat_pow_Suc nat_pow_closed pow_mult_distrib
xy)
qed

lemma (in group) pow_eq_div2:
  fixes m n :: nat
  assumes x_car: "x ∈ carrier G"
  assumes pow_eq: "x [^] m = x [^] n"
  shows "x [^] (m - n) = 1"
proof (cases "m < n")
  case False
  have "1 ⊗ x [^] m = x [^] m" by (simp add: x_car)
  also have "... = x [^] (m - n) ⊗ x [^] n"
    using False by (simp add: nat_pow_mult x_car)
  also have "... = x [^] (m - n) ⊗ x [^] m"
    by (simp add: pow_eq)
  finally show ?thesis
    by (metis nat_pow_closed one_closed right_cancel x_car)
qed simp
```

## 6.5 Submonoids

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

```
lemma (in submonoid) is_submonoid:
  "submonoid H G" by (rule submonoid_axioms)

lemma (in submonoid) mem_carrier [simp]:
  "x ∈ H ⟹ x ∈ carrier G"
  using subset by blast

lemma (in submonoid) submonoid_is_monoid [intro]:
  assumes "monoid G"
  shows "monoid (G⦇carrier := H⦈)"
proof -
  interpret monoid G by fact
  show ?thesis
    by (simp add: monoid_def m_assoc)
qed

lemma submonoid_nonempty:
  "~ submonoid {} G"
  by (blast dest: submonoid.one_closed)

lemma (in submonoid) finite_monoid_imp_card_positive:
  "finite (carrier G) ==> 0 < card H"
proof (rule classical)
  assume "finite (carrier G)" and a: "~ 0 < card H"
  then have "finite H" by (blast intro: finite_subset [OF subset])
  with is_submonoid a have "submonoid {} G" by simp
  with submonoid_nonempty show ?thesis by contradiction
qed


lemma (in monoid) monoid_incl_imp_submonoid :
  assumes "H ⊆ carrier G"
and "monoid (G⦇carrier := H⦈)"
shows "submonoid H G"
proof (intro submonoid.intro[OF assms(1)])
  have ab_eq : "⋀ a b. a ∈ H ⟹ b ∈ H ⟹ a ⊗G⦇carrier := H⦈ b = a ⊗
b" using assms by simp
  have "⋀a b. a ∈ H ⟹ b ∈ H ⟹ a ⊗ b ∈ carrier (G⦇carrier := H⦈)
"
    using assms ab_eq unfolding group_def using monoid.m_closed by fastforce
  thus "⋀a b. a ∈ H ⟹ b ∈ H ⟹ a ⊗ b ∈ H" by simp
  show "1 ∈ H " using monoid.one_closed[OF assms(2)] assms by simp
qed

lemma (in monoid) inv_unique':
  assumes "x ∈ carrier G" "y ∈ carrier G"
  shows "⟦ x ⊗ y = 1; y ⊗ x = 1 ⟧ ⟹ y = inv x"
proof -
  assume "x ⊗ y = 1" and l_inv: "y ⊗ x = 1"
```

```
  hence unit: "x ∈ Units G"
    using assms unfolding Units_def by auto
  show "y = inv x"
    using inv_unique[OF l_inv Units_r_inv[OF unit] assms Units_inv_closed[OF
unit]] .
qed

lemma (in monoid) m_inv_monoid_consistent:
  assumes "x ∈ Units (G ⦇ carrier := H ⦈)" and "submonoid H G"
  shows "inv(G ⦇ carrier := H ⦈) x = inv x"
proof -
  have monoid: "monoid (G ⦇ carrier := H ⦈)"
    using submonoid.submonoid_is_monoid[OF assms(2) monoid_axioms] .
  obtain y where y: "y ∈ H" "x ⊗ y = 1" "y ⊗ x = 1"
    using assms(1) unfolding Units_def by auto
  have x: "x ∈ H" and in_carrier: "x ∈ carrier G" "y ∈ carrier G"
    using y(1) submonoid.subset[OF assms(2)] assms(1) unfolding Units_def
by auto
  show ?thesis
    using monoid.inv_unique'[OF monoid, of x y] x y
    using inv_unique'[OF in_carrier y(2-3)] by auto
qed
```

## 6.6 Subgroups

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

lemma (in subgroup) is_subgroup:
  "subgroup H G" by (rule subgroup_axioms)

declare (in subgroup) group.intro [intro]

lemma (in subgroup) mem_carrier [simp]:
  "x ∈ H ⟹ x ∈ carrier G"
  using subset by blast

lemma (in subgroup) subgroup_is_group [intro]:
  assumes "group G"
  shows "group (G⦇carrier := H⦈)"
proof -
  interpret group G by fact
  have "Group.monoid (G⦇carrier := H⦈)"
    by (simp add: monoid_axioms submonoid.intro submonoid.submonoid_is_monoid
subset)
```

```
  then show ?thesis
    by (rule monoid.group_l_invI) (auto intro: l_inv mem_carrier)
qed

lemma (in group) triv_subgroup: "subgroup {1} G"
  by (auto simp: subgroup_def)

lemma subgroup_is_submonoid:
  assumes "subgroup H G" shows "submonoid H G"
  using assms by (auto intro: submonoid.intro simp add: subgroup_def)

lemma (in group) subgroup_Units:
  assumes "subgroup H G" shows "H ⊆ Units (G ⦇ carrier := H ⦈)"
  using group.Units[OF subgroup.subgroup_is_group[OF assms group_axioms]]
by simp

lemma (in group) m_inv_consistent [simp]:
  assumes "subgroup H G" "x ∈ H"
  shows "inv(G ⦇ carrier := H ⦈) x = inv x"
  using assms m_inv_monoid_consistent[OF _ subgroup_is_submonoid] subgroup_Units[of
H] by auto

lemma (in group) int_pow_consistent:
  assumes "subgroup H G" "x ∈ H"
  shows "x [^] (n :: int) = x [^](G ⦇ carrier := H ⦈) n"
proof (cases)
  assume ge: "n ≥ 0"
  hence "x [^] n = x [^] (nat n)"
    using int_pow_def2 [of G] by auto
  also have " ... = x [^](G ⦇ carrier := H ⦈) (nat n)"
    using nat_pow_consistent by simp
  also have " ... = x [^](G ⦇ carrier := H ⦈) n"
    by (metis ge int_nat_eq int_pow_int)
  finally show ?thesis .
next
  assume "¬ n ≥ 0" hence lt: "n < 0" by simp
  hence "x [^] n = inv (x [^] (nat (- n)))"
    using int_pow_def2 [of G] by auto
  also have " ... = (inv x) [^] (nat (- n))"
    by (metis assms nat_pow_inv subgroup.mem_carrier)
  also have " ... = (inv(G ⦇ carrier := H ⦈) x) [^](G ⦇ carrier := H ⦈) (nat
(- n))"
    using m_inv_consistent[OF assms] nat_pow_consistent by auto
  also have " ... = inv(G ⦇ carrier := H ⦈) (x [^](G ⦇ carrier := H ⦈) (nat
(- n)))"
    using group.nat_pow_inv[OF subgroup.subgroup_is_group[OF assms(1)
is_group]] assms(2) by auto
  also have " ... = x [^](G ⦇ carrier := H ⦈) n"
```

```
    by (simp add: int_pow_def2 lt)
  finally show ?thesis .
qed
```

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

```
lemma (in group) one_in_subset:
  "⟦H ⊆ carrier G; H ≠ {}; ∀a ∈ H. inv a ∈ H; ∀a∈H. ∀b∈H. a ⊗ b ∈
H⟧
   ⟹ 1 ∈ H"
by force
```

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 (simp add: subgroup_def assms)
  show "1 ∈ H" by (rule one_in_subset) (auto simp only: assms)
qed

lemma (in group) subgroupE:
  assumes "subgroup H G"
  shows "H ⊆ carrier G"
    and "H ≠ {}"
    and "⋀a. a ∈ H ⟹ inv a ∈ H"
    and "⋀a b. ⟦ a ∈ H; b ∈ H ⟧ ⟹ a ⊗ b ∈ H"
  using assms unfolding subgroup_def[of H G] by auto

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"
  by (blast dest: subgroup.one_closed)

lemma (in subgroup) finite_imp_card_positive: "finite (carrier G) ⟹
0 < card H"
  using subset one_closed card_gt_0_iff finite_subset by blast

lemma (in subgroup) subgroup_is_submonoid :
  "submonoid H G"
  by (simp add: submonoid.intro subset)

lemma (in group) submonoid_subgroupI :
  assumes "submonoid H G"
    and "⋀a. a ∈ H ⟹ inv a ∈ H"
```

```
  shows "subgroup H G"
  by (metis assms subgroup_def submonoid_def)

lemma (in group) group_incl_imp_subgroup:
  assumes "H ⊆ carrier G"
    and "group (G⦇carrier := H⦈)"
  shows "subgroup H G"
proof (intro submonoid_subgroupI[OF monoid_incl_imp_submonoid[OF assms(1)]])
  show "monoid (G⦇carrier := H⦈)" using group_def assms by blast
  have ab_eq : "⋀ a b. a ∈ H ⟹ b ∈ H ⟹ a ⊗G⦇carrier := H⦈ b = a ⊗
b" using assms by simp
  fix a  assume aH : "a ∈ H"
  have " invG⦇carrier := H⦈ a ∈ carrier G"
    using assms aH group.inv_closed[OF assms(2)] by auto
  moreover have "1G⦇carrier := H⦈ = 1" using assms monoid.one_closed ab_eq
one_def by simp
  hence "a ⊗G⦇carrier := H⦈ invG⦇carrier := H⦈ a= 1"
    using assms ab_eq aH  group.r_inv[OF assms(2)] by simp
  hence "a ⊗ invG⦇carrier := H⦈ a= 1"
    using aH assms group.inv_closed[OF assms(2)] ab_eq by simp
  ultimately have "invG⦇carrier := H⦈ a = inv a"
    by (metis aH assms(1) contra_subsetD group.inv_inv is_group local.inv_equality)
  moreover have "invG⦇carrier := H⦈ a ∈ H"
    using aH group.inv_closed[OF assms(2)] by auto
  ultimately show "inv a ∈ H" by auto
qed
```

## 6.7 Direct Products

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

lemma DirProd_monoid:
  assumes "monoid G" and "monoid H"
  shows "monoid (G ×× H)"
proof -
  interpret G: monoid G by fact
  interpret H: monoid H by fact
  from assms
  show ?thesis by (unfold monoid_def DirProd_def, auto)
qed
```

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 -
  interpret G: group G by fact
  interpret H: group H by fact
  show ?thesis by (rule groupI)
     (auto intro: G.m_assoc H.m_assoc G.l_inv H.l_inv
           simp add: DirProd_def)
qed

lemma carrier_DirProd [simp]: "carrier (G ×× H) = carrier G × carrier
H"
  by (simp add: DirProd_def)

lemma one_DirProd [simp]: "𝟭G ×× H = (𝟭G, 𝟭H)"
  by (simp add: DirProd_def)

lemma mult_DirProd [simp]: "(g, h) ⊗(G ×× H) (g', h') = (g ⊗G g', h
⊗H h')"
  by (simp add: DirProd_def)

lemma mult_DirProd': "x ⊗(G ×× H) y = (fst x ⊗G fst y, snd x ⊗H snd
y)"
  by (subst mult_DirProd [symmetric]) simp

lemma DirProd_assoc: "(G ×× H ×× I) = (G ×× (H ×× I))"
  by auto

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 -
  interpret G: group G by fact
  interpret H: group H by fact
  interpret Prod: group "G ×× H"
    by (auto intro: DirProd_group group.intro group.axioms assms)
  show ?thesis by (simp add: Prod.inv_equality g h)
qed

lemma DirProd_subgroups :
  assumes "group G"
    and "subgroup H G"
    and "group K"
    and "subgroup I K"
  shows "subgroup (H × I) (G ×× K)"
proof (intro group.group_incl_imp_subgroup[OF DirProd_group[OF assms(1)assms(3)]])
  have "H ⊆ carrier G" "I ⊆ carrier K" using subgroup.subset assms by
```

```
blast+
  thus "(H × I) ⊆ carrier (G ×× K)" unfolding DirProd_def by auto
  have "Group.group ((G(|carrier := H|)) ×× (K(|carrier := I|)))"
    using DirProd_group[OF subgroup.subgroup_is_group[OF assms(2)assms(1)]
        subgroup.subgroup_is_group[OF assms(4)assms(3)]].
  moreover have "((G(|carrier := H|)) ×× (K(|carrier := I|))) = ((G ××
K)(|carrier := H × I|))"
    unfolding DirProd_def using assms by simp
  ultimately show "Group.group ((G ×× K)(|carrier := H × I|))" by simp
qed
```

## 6.8 Homomorphisms (mono and epi) and Isomorphisms

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

lemma homI:
  "⟦⋀x. x ∈ carrier G ⟹ h x ∈ carrier H;
    ⋀x y. ⟦x ∈ carrier G; y ∈ carrier G⟧ ⟹ h (x ⊗G y) = h x ⊗H h y⟧
⟹ h ∈ hom G H"
  by (auto simp: hom_def)

lemma hom_carrier: "h ∈ hom G H ⟹ h ` carrier G ⊆ carrier H"
  by (auto simp: hom_def)

lemma hom_in_carrier: "⟦h ∈ hom G H; x ∈ carrier G⟧ ⟹ h x ∈ carrier
H"
  by (auto simp: hom_def)

lemma hom_compose:
  "⟦ f ∈ hom G H; g ∈ hom H I ⟧ ⟹ g ∘ f ∈ hom G I"
  unfolding hom_def by (auto simp add: Pi_iff)

lemma (in group) hom_restrict:
  assumes "h ∈ hom G H" and "⋀g. g ∈ carrier G ⟹ h g = t g" shows
"t ∈ hom G H"
  using assms unfolding hom_def by (auto simp add: Pi_iff)

lemma (in group) hom_compose:
  "[|h ∈ hom G H; i ∈ hom H I|] ==> compose (carrier G) i h ∈ hom G I"
by (fastforce simp add: hom_def compose_def)

lemma (in group) restrict_hom_iff [simp]:
  "(λx. if x ∈ carrier G then f x else g x) ∈ hom G H ⟷ f ∈ hom G H"
  by (simp add: hom_def Pi_iff)
```

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

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

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

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

lemma isoI:
  "⟦h ∈ hom G H; bij_betw h (carrier G) (carrier H)⟧ ⟹ h ∈ iso G H"
  by (auto simp: iso_def)

lemma is_isoI: "h ∈ iso G H ⟹ G ≅ H"
  using is_iso_def by auto

lemma epi_iff_subset:
   "f ∈ epi G G' ⟷ f ∈ hom G G' ∧ carrier G' ⊆ f ` carrier G"
  by (auto simp: epi_def hom_def)

lemma iso_iff_mon_epi: "f ∈ iso G H ⟷ f ∈ mon G H ∧ f ∈ epi G H"
  by (auto simp: iso_def mon_def epi_def bij_betw_def)

lemma iso_set_refl: "(λx. x) ∈ iso G G"
  by (simp add: iso_def hom_def inj_on_def bij_betw_def Pi_def)

lemma id_iso: "id ∈ iso G G"
  by (simp add: iso_def hom_def inj_on_def bij_betw_def Pi_def)

corollary iso_refl [simp]: "G ≅ G"
  using iso_set_refl unfolding is_iso_def by auto

lemma iso_iff:
   "h ∈ iso G H ⟷ h ∈ hom G H ∧ h ` (carrier G) = carrier H ∧ inj_on
h (carrier G)"
  by (auto simp: iso_def hom_def bij_betw_def)

lemma iso_imp_homomorphism:
   "h ∈ iso G H ⟹ h ∈ hom G H"
  by (simp add: iso_iff)

lemma trivial_hom:
   "group H ⟹ (λx. one H) ∈ hom G H"
  by (auto simp: hom_def Group.group_def)

lemma (in group) hom_eq:
```

```
  assumes "f ∈ hom G H" "⋀x. x ∈ carrier G ⟹ f' x = f x"
  shows "f' ∈ hom G H"
  using assms by (auto simp: hom_def)

lemma (in group) iso_eq:
  assumes "f ∈ iso G H" "⋀x. x ∈ carrier G ⟹ f' x = f x"
  shows "f' ∈ iso G H"
  using assms  by (fastforce simp: iso_def inj_on_def bij_betw_def hom_eq
image_iff)

lemma (in group) iso_set_sym:
  assumes "h ∈ iso G H"
  shows "inv_into (carrier G) h ∈ iso H G"
proof -
  have h: "h ∈ hom G H" "bij_betw h (carrier G) (carrier H)"
    using assms by (auto simp add: iso_def bij_betw_inv_into)
  then have HG: "bij_betw (inv_into (carrier G) h) (carrier H) (carrier
G)"
    by (simp add: bij_betw_inv_into)
  have "inv_into (carrier G) h ∈ hom H G"
    unfolding hom_def
  proof safe
    show *: "⋀x. x ∈ carrier H ⟹ inv_into (carrier G) h x ∈ carrier
G"
      by (meson HG bij_betwE)
    show "inv_into (carrier G) h (x ⊗H y) = inv_into (carrier G) h x
⊗ inv_into (carrier G) h y"
      if "x ∈ carrier H" "y ∈ carrier H" for x y
    proof (rule inv_into_f_eq)
      show "inj_on h (carrier G)"
        using bij_betw_def h(2) by blast
      show "inv_into (carrier G) h x ⊗ inv_into (carrier G) h y ∈ carrier
G"
        by (simp add: * that)
      show "h (inv_into (carrier G) h x ⊗ inv_into (carrier G) h y) =
x ⊗H y"
        using h bij_betw_inv_into_right [of h] unfolding hom_def by (simp
add: "*" that)
    qed
  qed
  then show ?thesis
    by (simp add: Group.iso_def bij_betw_inv_into h)
qed

corollary (in group) iso_sym: "G ≅ H ⟹ H ≅ G"
  using iso_set_sym unfolding is_iso_def by auto

lemma iso_set_trans:
  "⟦h ∈ Group.iso G H; i ∈ Group.iso H I⟧ ⟹ i ∘ h ∈ Group.iso G I"
```

```
  by (force simp: iso_def hom_compose intro: bij_betw_trans)

corollary iso_trans [trans]: "⟦G ≅ H ; H ≅ I⟧ ⟹ G ≅ I"
  using iso_set_trans unfolding is_iso_def by blast

lemma iso_same_card: "G ≅ H ⟹ card (carrier G) = card (carrier H)"
  using bij_betw_same_card  unfolding is_iso_def iso_def by auto

lemma iso_finite: "G ≅ H ⟹ finite(carrier G) ⟷ finite(carrier H)"
  by (auto simp: is_iso_def iso_def bij_betw_finite)

lemma mon_compose:
   "⟦f ∈ mon G H; g ∈ mon H K⟧ ⟹ (g ∘ f) ∈ mon G K"
  by (auto simp: mon_def intro: hom_compose comp_inj_on inj_on_subset
[OF _ hom_carrier])

lemma mon_compose_rev:
   "⟦f ∈ hom G H; g ∈ hom H K; (g ∘ f) ∈ mon G K⟧ ⟹ f ∈ mon G H"
  using inj_on_imageI2 by (auto simp: mon_def)

lemma epi_compose:
   "⟦f ∈ epi G H; g ∈ epi H K⟧ ⟹ (g ∘ f) ∈ epi G K"
  using hom_compose by (force simp: epi_def hom_compose simp flip: image_image)

lemma epi_compose_rev:
   "⟦f ∈ hom G H; g ∈ hom H K; (g ∘ f) ∈ epi G K⟧ ⟹ g ∈ epi H K"
  by (fastforce simp: epi_def hom_def Pi_iff image_def set_eq_iff)

lemma iso_compose_rev:
   "⟦f ∈ hom G H; g ∈ hom H K; (g ∘ f) ∈ iso G K⟧ ⟹ f ∈ mon G H ∧ g
∈ epi H K"
  unfolding iso_iff_mon_epi using mon_compose_rev epi_compose_rev by blast

lemma epi_iso_compose_rev:
  assumes "f ∈ epi G H" "g ∈ hom H K" "(g ∘ f) ∈ iso G K"
  shows "f ∈ iso G H ∧ g ∈ iso H K"
proof
  show "f ∈ iso G H"
    by (metis (no_types, lifting) assms epi_def iso_compose_rev iso_iff_mon_epi
mem_Collect_eq)
  then have "f ∈ hom G H ∧ bij_betw f (carrier G) (carrier H)"
    using Group.iso_def ‹f ∈ Group.iso G H› by blast
  then have "bij_betw g (carrier H) (carrier K)"
    using Group.iso_def assms(3) bij_betw_comp_iff by blast
  then show "g ∈ iso H K"
    using Group.iso_def assms(2) by blast
qed

lemma mon_left_invertible:
```

```
   "⟦f ∈ hom G H; ⋀x. x ∈ carrier G ⟹ g(f x) = x⟧ ⟹ f ∈ mon G H"
  by (simp add: mon_def inj_on_def) metis

lemma epi_right_invertible:
   "⟦g ∈ hom H G; f ∈ carrier G → carrier H; ⋀x. x ∈ carrier G ⟹ g(f
x) = x⟧ ⟹ g ∈ epi H G"
  by (force simp: Pi_iff epi_iff_subset image_subset_iff_funcset subset_iff)

lemma (in monoid) hom_imp_img_monoid:
  assumes "h ∈ hom G H"
  shows "monoid (H ⦇ carrier := h ‘ (carrier G), one := h 𝟭G ⦈)" (is "monoid
?h_img")
proof (rule monoidI)
  show "𝟭?h_img ∈ carrier ?h_img"
    by auto
next
  fix x y z assume "x ∈ carrier ?h_img" "y ∈ carrier ?h_img" "z ∈ carrier
?h_img"
  then obtain g1 g2 g3
    where g1: "g1 ∈ carrier G" "x = h g1"
      and g2: "g2 ∈ carrier G" "y = h g2"
      and g3: "g3 ∈ carrier G" "z = h g3"
    using image_iff[where ?f = h and ?A = "carrier G"] by auto
  have aux_lemma:
    "⋀a b. ⟦ a ∈ carrier G; b ∈ carrier G ⟧ ⟹ h a ⊗(?h_img) h b = h
(a ⊗ b)"
    using assms unfolding hom_def by auto

  show "x ⊗(?h_img) 𝟭(?h_img) = x"
    using aux_lemma[OF g1(1) one_closed] g1(2) r_one[OF g1(1)] by simp

  show "𝟭(?h_img) ⊗(?h_img) x = x"
    using aux_lemma[OF one_closed g1(1)] g1(2) l_one[OF g1(1)] by simp

  have "x ⊗(?h_img) y = h (g1 ⊗ g2)"
    using aux_lemma g1 g2 by auto
  thus "x ⊗(?h_img) y ∈ carrier ?h_img"
    using g1(1) g2(1) by simp

  have "(x ⊗(?h_img) y) ⊗(?h_img) z = h ((g1 ⊗ g2) ⊗ g3)"
    using aux_lemma g1 g2 g3 by auto
  also have " ... = h (g1 ⊗ (g2 ⊗ g3))"
    using m_assoc[OF g1(1) g2(1) g3(1)] by simp
  also have " ... = x ⊗(?h_img) (y ⊗(?h_img) z)"
    using aux_lemma g1 g2 g3 by auto
  finally show "(x ⊗(?h_img) y) ⊗(?h_img) z = x ⊗(?h_img) (y ⊗(?h_img)
z)" .
qed
```

```
lemma (in group) hom_imp_img_group:
  assumes "h ∈ hom G H"
  shows "group (H ⦇ carrier := h ` (carrier G), one := h 𝟭G ⦈)" (is "group
?h_img")
proof -
  interpret monoid ?h_img
    using hom_imp_img_monoid[OF assms] .

  show ?thesis
  proof (unfold_locales)
    show "carrier ?h_img ⊆ Units ?h_img"
    proof (auto simp add: Units_def)
      have aux_lemma:
        "⋀g1 g2. ⟦ g1 ∈ carrier G; g2 ∈ carrier G ⟧ ⟹ h g1 ⊗H h g2
= h (g1 ⊗ g2)"
        using assms unfolding hom_def by auto

      fix g1 assume g1: "g1 ∈ carrier G"
      thus "∃g2 ∈ carrier G. (h g2) ⊗H (h g1) = h 𝟭 ∧ (h g1) ⊗H (h g2)
= h 𝟭"
        using aux_lemma[OF g1 inv_closed[OF g1]]
              aux_lemma[OF inv_closed[OF g1] g1]
              inv_closed by auto
    qed
  qed
qed

lemma (in group) iso_imp_group:
  assumes "G ≅ H" and "monoid H"
  shows "group H"
proof -
  obtain φ where phi: "φ ∈ iso G H" "inv_into (carrier G) φ ∈ iso H
G"
    using iso_set_sym assms unfolding is_iso_def by blast
  define ψ where psi_def: "ψ = inv_into (carrier G) φ"

  have surj: "φ ` (carrier G) = (carrier H)" "ψ ` (carrier H) = (carrier
G)"
   and inj: "inj_on φ (carrier G)" "inj_on ψ (carrier H)"
   and phi_hom: "⋀g1 g2. ⟦ g1 ∈ carrier G; g2 ∈ carrier G ⟧ ⟹ φ (g1
⊗ g2) = (φ g1) ⊗H (φ g2)"
   and psi_hom: "⋀h1 h2. ⟦ h1 ∈ carrier H; h2 ∈ carrier H ⟧ ⟹ ψ (h1
⊗H h2) = (ψ h1) ⊗ (ψ h2)"
   using phi psi_def unfolding iso_def bij_betw_def hom_def by auto

  have phi_one: "φ 𝟭 = 𝟭H"
  proof -
    have "(φ 𝟭) ⊗H 𝟭H = (φ 𝟭) ⊗H (φ 𝟭)"
      by (metis assms(2) image_eqI monoid.r_one one_closed phi_hom r_one
```

```
surj(1))
    thus ?thesis
      by (metis (no_types, opaque_lifting) Units_eq Units_one_closed assms(2)
f_inv_into_f imageI
          monoid.l_one monoid.one_closed phi_hom psi_def r_one surj)
  qed

  have "carrier H ⊆ Units H"
  proof
    fix h assume h: "h ∈ carrier H"
    let ?inv_h = "φ (inv (ψ h))"
    have "h ⊗H ?inv_h = φ (ψ h) ⊗H ?inv_h"
      by (simp add: f_inv_into_f h psi_def surj(1))
    also have " ... = φ ((ψ h) ⊗ inv (ψ h))"
      by (metis h imageI inv_closed phi_hom surj(2))
    also have " ... = φ 1"
      by (simp add: h inv_into_into psi_def surj(1))
    finally have 1: "h ⊗H ?inv_h = 1H"
      using phi_one by simp

    have "?inv_h ⊗H h = ?inv_h ⊗H φ (ψ h)"
      by (simp add: f_inv_into_f h psi_def surj(1))
    also have " ... = φ (inv (ψ h) ⊗ (ψ h))"
      by (metis h imageI inv_closed phi_hom surj(2))
    also have " ... = φ 1"
      by (simp add: h inv_into_into psi_def surj(1))
    finally have 2: "?inv_h ⊗H h = 1H"
      using phi_one by simp

    thus "h ∈ Units H" unfolding Units_def using 1 2 h surj by fastforce
  qed
  thus ?thesis unfolding group_def group_axioms_def using assms(2) by
simp
qed

corollary (in group) iso_imp_img_group:
  assumes "h ∈ iso G H"
  shows "group (H (| one := h 1 |))"
proof -
  let ?h_img = "H (| carrier := h ` (carrier G), one := h 1 |)"
  have "h ∈ iso G ?h_img"
    using assms unfolding iso_def hom_def bij_betw_def by auto
  hence "G ≅ ?h_img"
    unfolding is_iso_def by auto
  hence "group ?h_img"
    using iso_imp_group[of ?h_img] hom_imp_img_monoid[of h H] assms un-
folding iso_def by simp
  moreover have "carrier H = carrier ?h_img"
    using assms unfolding iso_def bij_betw_def by simp
```

```
  hence "H ⦇ one := h 1 ⦈ = ?h_img"
    by simp
  ultimately show ?thesis by simp
qed
```

### 6.8.1 HOL Light's concept of an isomorphism pair

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

lemma group_isomorphisms_sym: "group_isomorphisms G H f g ⟹ group_isomorphisms
H G g f"
  by (auto simp: group_isomorphisms_def)

lemma group_isomorphisms_imp_iso: "group_isomorphisms G H f g ⟹ f ∈
iso G H"
by (auto simp: iso_def inj_on_def image_def group_isomorphisms_def hom_def
bij_betw_def Pi_iff, metis+)

lemma (in group) iso_iff_group_isomorphisms:
  "f ∈ iso G H ⟷ (∃g. group_isomorphisms G H f g)"
proof safe
  show "∃g. group_isomorphisms G H f g" if "f ∈ Group.iso G H"
    unfolding group_isomorphisms_def
  proof (intro exI conjI)
    let ?g = "inv_into (carrier G) f"
    show "∀x∈carrier G. ?g (f x) = x"
      by (metis (no_types, lifting) Group.iso_def bij_betw_inv_into_left
mem_Collect_eq that)
    show "∀y∈carrier H. f (?g y) = y"
      by (metis (no_types, lifting) Group.iso_def bij_betw_inv_into_right
mem_Collect_eq that)
  qed (use Group.iso_def iso_set_sym that in ‹blast+›)
next
  fix g
  assume "group_isomorphisms G H f g"
  then show "f ∈ Group.iso G H"
    by (auto simp: iso_def group_isomorphisms_def hom_in_carrier intro:
bij_betw_byWitness)
qed
```

### 6.8.2 Involving direct products

```
lemma DirProd_commute_iso_set:
  shows "(λ(x,y). (y,x)) ∈ iso (G ×× H) (H ×× G)"
  by (auto simp add: iso_def hom_def inj_on_def bij_betw_def)
```

```
corollary DirProd_commute_iso :
"(G ×× H) ≅ (H ×× G)"
  using DirProd_commute_iso_set unfolding is_iso_def by blast

lemma DirProd_assoc_iso_set:
  shows "(λ(x,y,z). (x,(y,z))) ∈ iso (G ×× H ×× I) (G ×× (H ×× I))"
by (auto simp add: iso_def hom_def inj_on_def bij_betw_def)

lemma (in group) DirProd_iso_set_trans:
  assumes "g ∈ iso G G2"
    and "h ∈ iso H I"
  shows "(λ(x,y). (g x, h y)) ∈ iso (G ×× H) (G2 ×× I)"
proof-
  have "(λ(x,y). (g x, h y)) ∈ hom (G ×× H) (G2 ×× I)"
    using assms unfolding iso_def hom_def by auto
  moreover have " inj_on (λ(x,y). (g x, h y)) (carrier (G ×× H))"
    using assms unfolding iso_def DirProd_def bij_betw_def inj_on_def
by auto
  moreover have "(λ(x, y). (g x, h y)) ‘ carrier (G ×× H) = carrier
(G2 ×× I)"
    using assms unfolding iso_def bij_betw_def image_def DirProd_def by
fastforce
  ultimately show "(λ(x,y). (g x, h y)) ∈ iso (G ×× H) (G2 ×× I)"
    unfolding iso_def bij_betw_def by auto
qed

corollary (in group) DirProd_iso_trans :
  assumes "G ≅ G2" and "H ≅ I"
  shows "G ×× H ≅ G2 ×× I"
  using DirProd_iso_set_trans assms unfolding is_iso_def by blast

lemma hom_pairwise: "f ∈ hom G (DirProd H K) ⟷ (fst ∘ f) ∈ hom G H
∧ (snd ∘ f) ∈ hom G K"
  apply (auto simp: hom_def mult_DirProd’ dest: Pi_mem)
   apply (metis Product_Type.mem_Times_iff comp_eq_dest_lhs funcset_mem)
  by (metis mult_DirProd prod.collapse)

lemma hom_paired:
   "(λx. (f x,g x)) ∈ hom G (DirProd H K) ⟷ f ∈ hom G H ∧ g ∈ hom
G K"
  by (simp add: hom_pairwise o_def)

lemma hom_paired2:
  assumes "group G" "group H"
  shows "(λ(x,y). (f x,g y)) ∈ hom (DirProd G H) (DirProd G’ H’) ⟷
f ∈ hom G G’ ∧ g ∈ hom H H’"
  using assms
  by (fastforce simp: hom_def Pi_def dest!: group.is_monoid)
```

```
lemma iso_paired2:
  assumes "group G" "group H"
  shows "(λ(x,y). (f x,g y)) ∈ iso (DirProd G H) (DirProd G' H') ⟷
f ∈ iso G G' ∧ g ∈ iso H H'"
  using assms
  by (fastforce simp add: iso_def inj_on_def bij_betw_def hom_paired2
image_paired_Times
      times_eq_iff group_def monoid.carrier_not_empty)

lemma hom_of_fst:
  assumes "group H"
  shows "(f ∘ fst) ∈ hom (DirProd G H) K ⟷ f ∈ hom G K"
proof -
  interpret group H
    by (rule assms)
  show ?thesis
    using one_closed by (auto simp: hom_def Pi_def)
qed

lemma hom_of_snd:
  assumes "group G"
  shows "(f ∘ snd) ∈ hom (DirProd G H) K ⟷ f ∈ hom H K"
proof -
  interpret group G
    by (rule assms)
  show ?thesis
    using one_closed by (auto simp: hom_def Pi_def)
qed
```

## 6.9 The locale for a homomorphism between two groups

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

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

declare group_hom.homh [simp]

lemma (in group_hom) hom_mult [simp]:
  "[| x ∈ carrier G; y ∈ carrier G |] ==> h (x ⊗G y) = h x ⊗H h y"
proof -
  assume "x ∈ carrier G" "y ∈ carrier G"
  with homh [unfolded hom_def] show ?thesis by simp
qed

lemma (in group_hom) hom_closed [simp]:
```

```
  "x ∈ carrier G ==> h x ∈ carrier H"
proof -
  assume "x ∈ carrier G"
  with homh [unfolded hom_def] show ?thesis by auto
qed

lemma (in group_hom) one_closed: "h 𝟭 ∈ carrier H"
  by simp

lemma (in group_hom) hom_one [simp]: "h 𝟭 = 𝟭⇘H⇙"
proof -
  have "h 𝟭 ⊗⇘H⇙ 𝟭⇘H⇙ = h 𝟭 ⊗⇘H⇙ h 𝟭"
    by (simp add: hom_mult [symmetric] del: hom_mult)
  then show ?thesis
    by (metis H.Units_eq H.Units_l_cancel H.one_closed local.one_closed)
qed

lemma hom_one:
  assumes "h ∈ hom G H" "group G" "group H"
  shows "h (one G) = one H"
  apply (rule group_hom.hom_one)
  by (simp add: assms group_hom_axioms_def group_hom_def)

lemma hom_mult:
  "⟦h ∈ hom G H; x ∈ carrier G; y ∈ carrier G⟧ ⟹ h (x ⊗⇘G⇙ y) = h x ⊗⇘H⇙
h y"
  by (auto simp: hom_def)

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

lemma (in group_hom) hom_inv [simp]:
  assumes "x ∈ carrier G" shows "h (inv x) = inv⇘H⇙ (h x)"
proof -
  have "h x ⊗⇘H⇙ h (inv x) = h x ⊗⇘H⇙ inv⇘H⇙ (h x)"
    using assms by (simp flip: hom_mult)
  with assms show ?thesis by (simp del: H.r_inv H.Units_r_inv)
qed

lemma (in group) int_pow_is_hom:
  "x ∈ carrier G ⟹ (([^]) x) ∈ hom ⦇ carrier = UNIV, mult = (+), one
= 0::int ⦈ G "
  unfolding hom_def by (simp add: int_pow_mult)

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

  apply (rule subgroupI)
  apply (auto simp add: image_subsetI)
```

```
  apply (metis G.inv_closed hom_inv image_iff)
  by (metis G.monoid_axioms hom_mult image_eqI monoid.m_closed)

lemma (in group_hom) subgroup_img_is_subgroup:
  assumes "subgroup I G"
  shows "subgroup (h ` I) H"
proof -
  have "h ∈ hom (G ⦇ carrier := I ⦈) H"
    using G.subgroupE[OF assms] subgroup.mem_carrier[OF assms] homh
    unfolding hom_def by auto
  hence "group_hom (G ⦇ carrier := I ⦈) H h"
    using subgroup.subgroup_is_group[OF assms G.is_group] is_group
    unfolding group_hom_def group_hom_axioms_def by simp
  thus ?thesis
    using group_hom.img_is_subgroup[of "G ⦇ carrier := I ⦈" H h] by simp
qed

lemma (in subgroup) iso_subgroup:
  assumes "group G" "group F"
  assumes "φ ∈ iso G F"
  shows "subgroup (φ ` H) F"
  by (metis assms Group.iso_iff group_hom.intro group_hom_axioms_def group_hom.subgroup_img
subgroup_axioms)

lemma (in group_hom) induced_group_hom:
  assumes "subgroup I G"
  shows "group_hom (G ⦇ carrier := I ⦈) (H ⦇ carrier := h ` I ⦈) h"
proof -
  have "h ∈ hom (G ⦇ carrier := I ⦈) (H ⦇ carrier := h ` I ⦈)"
    using homh subgroup.mem_carrier[OF assms] unfolding hom_def by auto
  thus ?thesis
    unfolding group_hom_def group_hom_axioms_def
    using subgroup.subgroup_is_group[OF assms G.is_group]
          subgroup.subgroup_is_group[OF subgroup_img_is_subgroup[OF assms]
is_group] by simp
qed
```

An isomorphism restricts to an isomorphism of subgroups.

```
lemma iso_restrict:
  assumes "φ ∈ iso G F"
  assumes groups: "group G" "group F"
  assumes HG: "subgroup H G"
  shows "(restrict φ H) ∈ iso (G⦇carrier := H⦈) (F⦇carrier := φ ` H⦈)"
proof -
  have "⋀x y. ⟦x ∈ H; y ∈ H; x ⊗G y ∈ H⟧ ⟹ φ (x ⊗G y) = φ x ⊗F φ
y"
    by (meson assms hom_mult iso_imp_homomorphism subgroup.mem_carrier)
  moreover have "⋀x y. ⟦x ∈ H; y ∈ H; x ⊗G y ∉ H⟧ ⟹ φ x ⊗F φ y =
undefined"
```

```
    by (simp add: HG subgroup.m_closed)
  moreover have "⋀x y. ⟦x ∈ H; y ∈ H; φ x = φ y⟧ ⟹ x = y"
    by (smt (verit, ccfv_SIG) assms group.iso_iff_group_isomorphisms group_isomorphisms_def
subgroup.mem_carrier)
  ultimately show ?thesis
    by (auto simp: iso_def hom_def bij_betw_def inj_on_def)
qed

lemma (in group) canonical_inj_is_hom:
  assumes "subgroup H G"
  shows "group_hom (G ⦇ carrier := H ⦈) G id"
  unfolding group_hom_def group_hom_axioms_def hom_def
  using subgroup.subgroup_is_group[OF assms is_group]
        is_group subgroup.subset[OF assms] by auto

lemma (in group_hom) hom_nat_pow:
  "x ∈ carrier G ⟹ h (x [^] (n :: nat)) = (h x) [^]⇘H⇙ n"
  by (induction n) auto

lemma (in group_hom) hom_int_pow:
  "x ∈ carrier G ⟹ h (x [^] (n :: int)) = (h x) [^]⇘H⇙ n"
  using hom_nat_pow by (simp add: int_pow_def2)

lemma hom_nat_pow:
  "⟦h ∈ hom G H; x ∈ carrier G; group G; group H⟧ ⟹ h (x [^]⇘G⇙ (n ::
nat)) = (h x) [^]⇘H⇙ n"
  by (simp add: group_hom.hom_nat_pow group_hom_axioms_def group_hom_def)

lemma hom_int_pow:
  "⟦h ∈ hom G H; x ∈ carrier G; group G; group H⟧ ⟹ h (x [^]⇘G⇙ (n ::
int)) = (h x) [^]⇘H⇙ n"
  by (simp add: group_hom.hom_int_pow group_hom_axioms.intro group_hom_def)
```

## 6.10 Commutative Structures

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

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

lemma (in comm_monoid) m_lcomm:
  "⟦x ∈ carrier G; y ∈ carrier G; z ∈ carrier G⟧ ⟹
   x ⊗ (y ⊗ z) = y ⊗ (x ⊗ z)"
proof -
  assume xyz: "x ∈ carrier G"  "y ∈ carrier G"  "z ∈ carrier G"
  from xyz have "x ⊗ (y ⊗ z) = (x ⊗ y) ⊗ z" by (simp add: m_assoc)
  also from xyz have "... = (y ⊗ x) ⊗ z" by (simp add: m_comm)
  also from xyz have "... = y ⊗ (x ⊗ z)" by (simp add: m_assoc)
  finally show ?thesis .
```

```
qed

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"
  using l_one
    by (auto intro!: comm_monoid.intro comm_monoid_axioms.intro monoid.intro
             intro: assms simp: m_closed one_closed m_comm)

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"
  by (rule comm_monoidI) (auto intro: m_assoc m_comm)

lemma (in comm_monoid) submonoid_is_comm_monoid :
  assumes "submonoid H G"
  shows "comm_monoid (G⦇carrier := H⦈)"
proof (intro monoid.monoid_comm_monoidI)
  show "monoid (G⦇carrier := H⦈)"
    using submonoid.submonoid_is_monoid assms comm_monoid_axioms comm_monoid_def
by blast
  show "⋀x y. x ∈ carrier (G⦇carrier := H⦈) ⟹ y ∈ carrier (G⦇carrier
:= H⦈)
        ⟹ x ⊗G⦇carrier := H⦈ y = y ⊗G⦇carrier := H⦈ x"
    by simp (meson assms m_comm submonoid.mem_carrier)
qed

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"
  by standard (simp_all add: m_comm)

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"
  by (fast intro: group.group_comm_groupI groupI assms)

lemma comm_groupE:
  fixes G (structure)
  assumes "comm_group G"
  shows "⋀x y. ⟦ x ∈ carrier G; y ∈ carrier G ⟧ ⟹ x ⊗ y ∈ carrier
G"
    and "1 ∈ carrier G"
    and "⋀x y z. ⟦ x ∈ carrier G; y ∈ carrier G; z ∈ carrier G ⟧ ⟹
(x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
    and "⋀x y. ⟦ x ∈ carrier G; y ∈ carrier G ⟧ ⟹ x ⊗ y = y ⊗ x"
    and "⋀x. x ∈ carrier G ⟹ 1 ⊗ x = x"
    and "⋀x. x ∈ carrier G ⟹ ∃y ∈ carrier G. y ⊗ x = 1"
  apply (simp_all add: group.axioms assms comm_group.axioms comm_monoid.m_comm
comm_monoid.m_ac(1))
  by (simp_all add: Group.group.axioms(1) assms comm_group.axioms(2) monoid.m_closed
group.r_inv_ex)

lemma (in comm_group) inv_mult:
  "[| x ∈ carrier G; y ∈ carrier G |] ==> inv (x ⊗ y) = inv x ⊗ inv y"
  by (simp add: m_ac inv_mult_group)

lemma (in comm_monoid) nat_pow_distrib:
  fixes n::nat
  assumes "x ∈ carrier G" "y ∈ carrier G"
  shows "(x ⊗ y) [^] n = x [^] n ⊗ y [^] n"
  by (simp add: assms pow_mult_distrib m_comm)

lemma (in comm_group) int_pow_distrib:
  assumes "x ∈ carrier G" "y ∈ carrier G"
  shows "(x ⊗ y) [^] (i::int) = x [^] i ⊗ y [^] i"
  by (simp add: assms int_pow_mult_distrib m_comm)

lemma (in comm_monoid) hom_imp_img_comm_monoid:
  assumes "h ∈ hom G H"
  shows "comm_monoid (H ⦇ carrier := h ` (carrier G), one := h 1_G ⦈)"
```

```
(is "comm_monoid ?h_img")
proof (rule monoid.monoid_comm_monoidI)
  show "monoid ?h_img"
    using hom_imp_img_monoid[OF assms] .
next
  fix x y assume "x ∈ carrier ?h_img" "y ∈ carrier ?h_img"
  then obtain g1 g2
    where g1: "g1 ∈ carrier G" "x = h g1"
      and g2: "g2 ∈ carrier G" "y = h g2"
    by auto
  have "x ⊗(?h_img) y = h (g1 ⊗ g2)"
    using g1 g2 assms unfolding hom_def by auto
  also have " ... = h (g2 ⊗ g1)"
    using m_comm[OF g1(1) g2(1)] by simp
  also have " ... = y ⊗(?h_img) x"
    using g1 g2 assms unfolding hom_def by auto
  finally show "x ⊗(?h_img) y = y ⊗(?h_img) x" .
qed

lemma (in comm_group) hom_group_mult:
  assumes "f ∈ hom H G" "g ∈ hom H G"
 shows "(λx. f x ⊗G g x) ∈ hom H G"
    using assms by (auto simp: hom_def Pi_def m_ac)

lemma (in comm_group) hom_imp_img_comm_group:
  assumes "h ∈ hom G H"
  shows "comm_group (H ⦇ carrier := h ` (carrier G), one := h 𝟭G ⦈)"
  unfolding comm_group_def
  using hom_imp_img_group[OF assms] hom_imp_img_comm_monoid[OF assms]
by simp

lemma (in comm_group) iso_imp_img_comm_group:
  assumes "h ∈ iso G H"
  shows "comm_group (H ⦇ one := h 𝟭G ⦈)"
proof -
  let ?h_img = "H ⦇ carrier := h ` (carrier G), one := h 𝟭 ⦈"
  have "comm_group ?h_img"
    using hom_imp_img_comm_group[of h H] assms unfolding iso_def by auto
  moreover have "carrier H = carrier ?h_img"
    using assms unfolding iso_def bij_betw_def by simp
  hence "H ⦇ one := h 𝟭 ⦈ = ?h_img"
    by simp
  ultimately show ?thesis by simp
qed

lemma (in comm_group) iso_imp_comm_group:
  assumes "G ≅ H" "monoid H"
  shows "comm_group H"
proof -
```

```
  obtain h where h: "h ∈ iso G H"
    using assms(1) unfolding is_iso_def by auto
  hence comm_gr: "comm_group (H ⦇ one := h 𝟭 ⦈)"
    using iso_imp_img_comm_group[of h H] by simp
  hence "⋀x. x ∈ carrier H ⟹ h 𝟭 ⊗H x = x"
    using monoid.l_one[of "H ⦇ one := h 𝟭 ⦈"] unfolding comm_group_def
comm_monoid_def by simp
  moreover have "h 𝟭 ∈ carrier H"
    using h one_closed unfolding iso_def hom_def by auto
  ultimately have "h 𝟭 = 𝟭H"
    using monoid.one_unique[OF assms(2), of "h 𝟭"] by simp
  hence "H = H ⦇ one := h 𝟭 ⦈"
    by simp
  thus ?thesis
    using comm_gr by simp
qed


lemma (in group) incl_subgroup:
  assumes "subgroup J G"
    and "subgroup I (G⦇carrier:=J⦈)"
  shows "subgroup I G" unfolding subgroup_def
proof
  have H1: "I ⊆ carrier (G⦇carrier:=J⦈)" using assms(2) subgroup.subset
by blast
  also have H2: "...⊆J" by simp
  also  have "...⊆(carrier G)"  by (simp add: assms(1) subgroup.subset)
  finally have H: "I ⊆ carrier G" by simp
  have "(⋀x y. ⟦x ∈ I ; y ∈ I⟧ ⟹ x ⊗ y ∈ I)" using assms(2) by (auto
simp add: subgroup_def)
  thus  "I ⊆ carrier G ∧ (∀x y. x ∈ I ⟶ y ∈ I ⟶ x ⊗ y ∈ I)"  us-
ing H by blast
  have K: "𝟭 ∈ I" using assms(2) by (auto simp add: subgroup_def)
  have "(⋀x. x ∈ I ⟹ inv x ∈ I)" using assms  subgroup.m_inv_closed
H
    by (metis H1 H2 m_inv_consistent subsetCE)
  thus "𝟭 ∈ I ∧ (∀x. x ∈ I ⟶ inv x ∈ I)" using K by blast
qed


lemma (in group) subgroup_incl:
  assumes "subgroup I G" and "subgroup J G" and "I ⊆ J"
  shows "subgroup I (G ⦇ carrier := J ⦈)"
  using group.group_incl_imp_subgroup[of "G ⦇ carrier := J ⦈" I]
        assms(1-2)[THEN subgroup.subgroup_is_group[OF _ group_axioms]]
assms(3) by auto
```

### 6.11 The Lattice of Subgroups of a Group

```
theorem (in group) subgroups_partial_order:
  "partial_order (|carrier = {H. subgroup H G}, eq = (=), le = (⊆)|)"
  by standard simp_all

lemma (in group) subgroup_self:
  "subgroup (carrier G) G"
  by (rule subgroupI) auto

lemma (in group) subgroup_imp_group:
  "subgroup H G ==> group (G(|carrier := H|))"
  by (erule subgroup.subgroup_is_group) (rule group_axioms)

lemma (in group) subgroup_mult_equality:
  "⟦ subgroup H G; h1 ∈ H; h2 ∈ H ⟧ ⟹  h1 ⊗G (| carrier := H |) h2 = h1
⊗ h2"
  unfolding subgroup_def by simp

theorem (in group) subgroups_Inter:
  assumes subgr: "(⋀H. H ∈ A ⟹ subgroup H G)"
    and not_empty: "A ≠ {}"
  shows "subgroup (⋂A) G"
proof (rule subgroupI)
  from subgr [THEN subgroup.subset] and not_empty
  show "⋂A ⊆ carrier G" by blast
next
  from subgr [THEN subgroup.one_closed]
  show "⋂A ≠ {}" by blast
next
  fix x assume "x ∈ ⋂A"
  with subgr [THEN subgroup.m_inv_closed]
  show "inv x ∈ ⋂A" by blast
next
  fix x y assume "x ∈ ⋂A" "y ∈ ⋂A"
  with subgr [THEN subgroup.m_closed]
  show "x ⊗ y ∈ ⋂A" by blast
qed

lemma (in group) subgroups_Inter_pair :
  assumes "subgroup I G" "subgroup J G" shows "subgroup (I∩J) G"
  using subgroups_Inter[ where ?A = "{I,J}"] assms by auto

theorem (in group) subgroups_complete_lattice:
  "complete_lattice (|carrier = {H. subgroup H G}, eq = (=), le = (⊆)|)"
    (is "complete_lattice ?L")
proof (rule partial_order.complete_lattice_criterion1)
  show "partial_order ?L" by (rule subgroups_partial_order)
next
  have "greatest ?L (carrier G) (carrier ?L)"
```

```
    by (unfold greatest_def) (simp add: subgroup.subset subgroup_self)
  then show "∃G. greatest ?L G (carrier ?L)" ..
next
  fix A
  assume L: "A ⊆ carrier ?L" and non_empty: "A ≠ {}"
  then have Int_subgroup: "subgroup (⋂A) G"
    by (fastforce intro: subgroups_Inter)
  have "greatest ?L (⋂A) (Lower ?L A)" (is "greatest _ ?Int _")
  proof (rule greatest_LowerI)
    fix H
    assume H: "H ∈ A"
    with L have subgroupH: "subgroup H G" by auto
    from subgroupH have groupH: "group (G (|carrier := H|))" (is "group
?H")
      by (rule subgroup_imp_group)
    from groupH have monoidH: "monoid ?H"
      by (rule group.is_monoid)
    from H have Int_subset: "?Int ⊆ H" by fastforce
    then show "le ?L ?Int H" by simp
  next
    fix H
    assume H: "H ∈ Lower ?L A"
    with L Int_subgroup show "le ?L H ?Int"
      by (fastforce simp: Lower_def intro: Inter_greatest)
  next
    show "A ⊆ carrier ?L" by (rule L)
  next
    show "?Int ∈ carrier ?L" by simp (rule Int_subgroup)
  qed
  then show "∃I. greatest ?L I (Lower ?L A)" ..
qed
```

## 6.12 The units in any monoid give rise to a group

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

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

lemma (in monoid) units_group: "group (units_of G)"
proof -
  have "⋀x y z. [[x ∈ Units G; y ∈ Units G; z ∈ Units G]] ⟹ x ⊗ y ⊗
z = x ⊗ (y ⊗ z)"
    by (simp add: Units_closed m_assoc)
  moreover have "⋀x. x ∈ Units G ⟹ ∃y∈Units G. y ⊗ x = 1"
    using Units_l_inv by blast
  ultimately show ?thesis
```

```
    unfolding units_of_def
    by (force intro!: groupI)
qed

lemma (in comm_monoid) units_comm_group: "comm_group (units_of G)"
proof -
  have "⋀x y. ⟦x ∈ carrier (units_of G); y ∈ carrier (units_of G)⟧
             ⟹ x ⊗units_of G y = y ⊗units_of G x"
    by (simp add: Units_closed m_comm units_of_def)
  then show ?thesis
    by (rule group.group_comm_groupI [OF units_group]) auto
qed

lemma units_of_carrier: "carrier (units_of G) = Units G"
  by (auto simp: units_of_def)

lemma units_of_mult: "mult (units_of G) = mult G"
  by (auto simp: units_of_def)

lemma units_of_one: "one (units_of G) = one G"
  by (auto simp: units_of_def)

lemma (in monoid) units_of_inv:
  assumes "x ∈ Units G"
  shows "m_inv (units_of G) x = m_inv G x"
  by (simp add: assms group.inv_equality units_group units_of_carrier
units_of_mult units_of_one)

lemma units_of_units [simp] : "Units (units_of G) = Units G"
  unfolding units_of_def Units_def by force

lemma (in group) surj_const_mult: "a ∈ carrier G ⟹ (λx. a ⊗ x) ` carrier
G = carrier G"
  apply (auto simp add: image_def)
  by (metis inv_closed inv_solve_left m_closed)

lemma (in group) l_cancel_one [simp]: "x ∈ carrier G ⟹ a ∈ carrier
G ⟹ x ⊗ a = x ⟷ a = one G"
  by (metis Units_eq Units_l_cancel monoid.r_one monoid_axioms one_closed)

lemma (in group) r_cancel_one [simp]: "x ∈ carrier G ⟹ a ∈ carrier
G ⟹ a ⊗ x = x ⟷ a = one G"
  by (metis monoid.l_one monoid_axioms one_closed right_cancel)

lemma (in group) l_cancel_one' [simp]: "x ∈ carrier G ⟹ a ∈ carrier
G ⟹ x = x ⊗ a ⟷ a = one G"
  using l_cancel_one by fastforce

lemma (in group) r_cancel_one' [simp]: "x ∈ carrier G ⟹ a ∈ carrier
```

```
G ⟹ x = a ⊗ x ⟷ a = one G"
  using r_cancel_one by fastforce

declare pow_nat [simp]

end


theory FiniteProduct
imports Group
begin
```

## 6.13 Product Operator for Commutative Monoids

### 6.13.1 Inductive Definition of a Relation for Products over Sets

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

```
inductive_set
  foldSetD :: "['a set, 'b ⇒ 'a ⇒ 'a, 'a] ⇒ ('b set * 'a) set"
  for D :: "'a set" and f :: "'b ⇒ 'a ⇒ 'a" and e :: 'a
  where
    emptyI [intro]: "e ∈ D ⟹ ({}, e) ∈ foldSetD D f e"
  | insertI [intro]: "⟦x ∉ A; f x y ∈ D; (A, y) ∈ foldSetD D f e⟧ ⟹
                      (insert x A, f x y) ∈ foldSetD D f e"

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

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

lemma foldSetD_closed: "(A, z) ∈ foldSetD D f e ⟹ z ∈ D"
  by (erule foldSetD.cases) auto

lemma Diff1_foldSetD:
  "⟦(A - {x}, y) ∈ foldSetD D f e; x ∈ A; f x y ∈ D⟧ ⟹
   (A, f x y) ∈ foldSetD D f e"
  by (metis Diff_insert_absorb foldSetD.insertI mk_disjoint_insert)

lemma foldSetD_imp_finite [simp]: "(A, x) ∈ foldSetD D f e ⟹ finite
A"
  by (induct set: foldSetD) auto

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

```
  case empty then show ?case by auto
next
  case (insert x F)
  then obtain y where y: "(F, y) ∈ foldSetD D f e" by auto
  with insert have "y ∈ D" by (auto dest: foldSetD_closed)
  with y and insert have "(insert x F, f x y) ∈ foldSetD D f e"
    by (intro foldSetD.intros) auto
  then show ?case ..
qed

lemma foldSetD_backwards:
  assumes "A ≠ {}" "(A, z) ∈ foldSetD D f e"
  shows "∃x y. x ∈ A ∧ (A - { x }, y) ∈ foldSetD D f e ∧ z = f x y"
  using assms(2) by (cases) (simp add: assms(1), metis Diff_insert_absorb
insertI1)
```

### 6.13.2 Left-Commutative Operations

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

lemma (in LCD) foldSetD_closed [dest]: "(A, z) ∈ foldSetD D f e ⟹ z
∈ D"
  by (erule foldSetD.cases) auto

lemma (in LCD) Diff1_foldSetD:
  "⟦(A - {x}, y) ∈ foldSetD D f e; x ∈ A; A ⊆ B⟧ ⟹
  (A, f x y) ∈ foldSetD D f e"
  by (meson Diff1_foldSetD f_closed local.foldSetD_closed subsetCE)

lemma (in LCD) finite_imp_foldSetD:
  "⟦finite A; A ⊆ B; e ∈ D⟧ ⟹ ∃x. (A, x) ∈ foldSetD D f e"
proof (induct set: finite)
  case empty then show ?case by auto
next
  case (insert x F)
  then obtain y where y: "(F, y) ∈ foldSetD D f e" by auto
  with insert have "y ∈ D" by auto
  with y and insert have "(insert x F, f x y) ∈ foldSetD D f e"
    by (intro foldSetD.intros) auto
  then show ?case ..
qed
```

```
lemma (in LCD) foldSetD_determ_aux:
  assumes "e ∈ D" and A: "card A < n" "A ⊆ B" "(A, x) ∈ foldSetD D f
e" "(A, y) ∈ foldSetD D f e"
  shows "y = x"
  using A
proof (induction n arbitrary: A x y)
  case 0
  then show ?case
    by auto
next
  case (Suc n)
  then consider "card A = n" | "card A < n"
    by linarith
  then show ?case
  proof cases
    case 1
    show ?thesis
      using foldSetD.cases [OF ‹(A,x) ∈ foldSetD D (·) e›]
    proof cases
      case 1
      then show ?thesis
        using ‹(A,y) ∈ foldSetD D (·) e› by auto
    next
      case A': (2 x' A' y')
      show ?thesis
        using foldSetD.cases [OF ‹(A,y) ∈ foldSetD D (·) e›]
      proof cases
        case 1
        then show ?thesis
          using ‹(A,x) ∈ foldSetD D (·) e› by auto
      next
        case A'': (2 x'' A'' y'')
        show ?thesis
        proof (cases "x' = x''")
          case True
          show ?thesis
          proof (cases "y' = y''")
            case True
            then show ?thesis
              using A' A'' ‹x' = x''› by (blast elim!: equalityE)
          next
            case False
            then show ?thesis
              using A' A'' ‹x' = x''›
              by (metis ‹card A = n› Suc.IH Suc.prems(2) card_insert_disjoint
foldSetD_imp_finite insert_eq_iff insert_subset lessI)
          qed
        next
          case False
```

```
          then have *: "A' - {x''} = A'' - {x'}" "x'' ∈ A'" "x' ∈ A''"
            using A' A'' by fastforce+
          then have "A' = insert x'' A'' - {x'}"
            using ‹x' ∉ A'› by blast
          then have card: "card A' ≤ card A''"
            using A' A'' * by (metis card_Suc_Diff1 eq_refl foldSetD_imp_finite)
          obtain u where u: "(A' - {x''}, u) ∈ foldSetD D (·) e"
            using finite_imp_foldSetD [of "A' - {x''}"] A' Diff_insert
‹A ⊆ B› ‹e ∈ D› by fastforce
          have "y' = f x'' u"
            using Diff1_foldSetD [OF u] ‹x'' ∈ A'› ‹card A = n› A' Suc.IH
‹A ⊆ B› by auto
          then have "(A'' - {x'}, u) ∈ foldSetD D f e"
            using "*"(1) u by auto
          then have "y'' = f x' u"
            using A'' by (metis * ‹card A = n› A'(1) Diff1_foldSetD Suc.IH
‹A ⊆ B›
                card card_Suc_Diff1 card_insert_disjoint foldSetD_imp_finite
insert_subset le_imp_less_Suc)
          then show ?thesis
            using A' A''
            by (metis ‹A ⊆ B› ‹y' = x'' · u› insert_subset left_commute
local.foldSetD_closed u)
        qed
      qed
    qed
  next
    case 2 with Suc show ?thesis by blast
  qed
qed

lemma (in LCD) foldSetD_determ:
  "⟦(A, x) ∈ foldSetD D f e; (A, y) ∈ foldSetD D f e; e ∈ D; A ⊆ B⟧
  ⟹ y = x"
  by (blast intro: foldSetD_determ_aux [rule_format])

lemma (in LCD) foldD_equality:
  "⟦(A, y) ∈ foldSetD D f e; e ∈ D; A ⊆ B⟧ ⟹ foldD D f e A = y"
  by (unfold foldD_def) (blast intro: foldSetD_determ)

lemma foldD_empty [simp]:
  "e ∈ D ⟹ foldD D f e {} = e"
  by (unfold foldD_def) blast

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

```
  by (metis Diff_insert_absorb f_closed finite_Diff foldSetD.insertI foldSetD_determ
foldSetD_imp_finite insert_subset local.finite_imp_foldSetD local.foldSetD_closed)

lemma (in LCD) foldD_insert:
  assumes "finite A" "x ∉ A" "x ∈ B" "e ∈ D" "A ⊆ B"
  shows "foldD D f e (insert x A) = f x (foldD D f e A)"
proof -
  have "(THE v. ∃y. (A, y) ∈ foldSetD D (·) e ∧ v = x · y) = x · (THE
y. (A, y) ∈ foldSetD D (·) e)"
    by (rule the_equality) (use assms foldD_def foldD_equality foldD_def
finite_imp_foldSetD in ‹metis+›)
  then show ?thesis
    unfolding foldD_def using assms by (simp add: foldD_insert_aux)
qed

lemma (in LCD) foldD_closed [simp]:
  "⟦finite A; e ∈ D; A ⊆ B⟧ ⟹ foldD D f e A ∈ D"
proof (induct set: finite)
  case empty then show ?case by simp
next
  case insert then show ?case by (simp add: foldD_insert)
qed

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"
  by (induct set: finite) (auto simp add: left_commute foldD_insert)

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

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 (induction set: finite)
  case (insert x F)
  then show ?case
    by (simp add: foldD_insert foldD_commute Int_insert_left insert_absorb
Int_mono2)
qed simp

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"
  by (simp add: foldD_nest_Un_Int)
```

— Delete rules to do with `foldSetD` relation.

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

Commutative Monoids

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

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

lemma (in ACeD) left_commute:
  "⟦x ∈ D; y ∈ D; z ∈ D⟧ ⟹ x · (y · z) = y · (x · z)"
proof -
  assume D: "x ∈ D" "y ∈ D" "z ∈ D"
  then have "x · (y · z) = (y · z) · x" by (simp add: commute)
  also from D have "... = y · (z · x)" by (simp add: assoc)
  also from D have "z · x = x · z" by (simp add: commute)
  finally show ?thesis .
qed

lemmas (in ACeD) AC = assoc commute left_commute

lemma (in ACeD) left_ident [simp]: "x ∈ D ⟹ e · x = x"
proof -
  assume "x ∈ D"
  then have "x · e = x" by (rule ident)
  with ‹x ∈ D› show ?thesis by (simp add: commute)
qed

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 (induction set: finite)
  case empty
  then show ?case
    by(simp add: left_commute LCD.foldD_closed [OF LCD.intro [of D]])
next
```

```
  case (insert x F)
  then show ?case
    by(simp add: AC insert_absorb Int_insert_left Int_mono2
                 LCD.foldD_insert [OF LCD.intro [of D]]
                 LCD.foldD_closed [OF LCD.intro [of D]])
qed

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"
  by (simp add: foldD_Un_Int
    left_commute LCD.foldD_closed [OF LCD.intro [of D]])
```

### 6.13.3 Products over Finite Sets

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

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

lemma (in comm_monoid) finprod_empty [simp]:
  "finprod G f {} = 𝟭"
  by (simp add: finprod_def)

lemma (in comm_monoid) finprod_infinite[simp]:
  "¬ finite A ⟹ finprod G f A = 𝟭"
  by (simp add: finprod_def)

declare funcsetI [intro]
  funcset_mem [dest]

context comm_monoid begin

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

```
  have "finprod G f (insert a F) = foldD (carrier G) ((⊗) ∘ f) 1 (insert
a F)"
    by (simp add: finprod_def assms)
  also have "... = ((⊗) ∘ f) a (foldD (carrier G) ((⊗) ∘ f) 1 F)"
    by (rule LCD.foldD_insert [OF LCD.intro [of "insert a F"]])
      (use assms in ‹auto simp: m_lcomm Pi_iff›)
  also have "... = f a ⊗ finprod G f F"
    using ‹finite F› by (auto simp add: finprod_def)
  finally show ?thesis .
qed

lemma finprod_one_eqI: "(⋀x. x ∈ A ⟹ f x = 1) ⟹ finprod G f A =
1"
proof (induct A rule: infinite_finite_induct)
  case empty show ?case by simp
next
  case (insert a A)
  have "(λi. 1) ∈ A → carrier G" by auto
  with insert show ?case by simp
qed simp

lemma finprod_one [simp]: "(⨂i∈A. 1) = 1"
  by (simp add: finprod_one_eqI)

lemma finprod_closed [simp]:
  fixes A
  assumes f: "f ∈ A → carrier G"
  shows "finprod G f A ∈ carrier G"
using f
proof (induct A rule: infinite_finite_induct)
  case empty show ?case by simp
next
  case (insert a A)
  then have a: "f a ∈ carrier G" by fast
  from insert have A: "f ∈ A → carrier G" by fast
  from insert A a show ?case by simp
qed simp

lemma funcset_Int_left [simp, intro]:
  "⟦f ∈ A → C; f ∈ B → C⟧ ⟹ f ∈ A Int B → C"
  by fast

lemma funcset_Un_left [iff]:
  "(f ∈ A Un B → C) = (f ∈ A → C ∧ f ∈ B → C)"
  by fast

lemma finprod_Un_Int:
  "⟦finite A; finite B; g ∈ A → carrier G; g ∈ B → carrier G⟧ ⟹
     finprod G g (A Un B) ⊗ finprod G g (A Int B) =
```

```
    finprod G g A ⊗ finprod G g B"
— The reversed orientation looks more natural, but LOOPS as a simprule!
proof (induct set: finite)
  case empty then show ?case by simp
next
  case (insert a A)
  then have a: "g a ∈ carrier G" by fast
  from insert have A: "g ∈ A → carrier G" by fast
  from insert A a show ?case
    by (simp add: m_ac Int_insert_left insert_absorb Int_mono2)
qed

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"
  by (metis Pi_split_domain finprod_Un_Int finprod_closed finprod_empty
r_one)

lemma finprod_multf [simp]:
  "⟦f ∈ A → carrier G; g ∈ A → carrier G⟧ ⟹
   finprod G (λx. f x ⊗ g x) A = (finprod G f A ⊗ finprod G g A)"
proof (induct A rule: infinite_finite_induct)
  case empty show ?case by simp
next
  case (insert a A) then
  have fA: "f ∈ A → carrier G" by fast
  from insert have fa: "f a ∈ carrier G" by fast
  from insert have gA: "g ∈ A → carrier G" by fast
  from insert have ga: "g a ∈ carrier G" by fast
  from insert have fgA: "(%x. f x ⊗ g x) ∈ A → carrier G"
    by (simp add: Pi_def)
  show ?case
    by (simp add: insert fA fa gA ga fgA m_ac)
qed simp

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 -
  assume prems: "A = B" "g ∈ B → carrier G"
    "!!i. i ∈ B ⟹ f i = g i"
  show ?thesis
  proof (cases "finite B")
    case True
    then have "!!A. ⟦A = B; g ∈ B → carrier G;
      !!i. i ∈ B ⟹ f i = g i⟧ ⟹ finprod G f A = finprod G g B"
    proof induct
      case empty thus ?case by simp
```

```
    next
      case (insert x B)
      then have "finprod G f A = finprod G f (insert x B)" by simp
      also from insert have "... = f x ⊗ finprod G f B"
      proof (intro finprod_insert)
        show "finite B" by fact
      next
        show "x ∉ B" by fact
      next
        assume "x ∉ B" "!!i. i ∈ insert x B ⟹ f i = g i"
          "g ∈ insert x B → carrier G"
        thus "f ∈ B → carrier G" by fastforce
      next
        assume "x ∉ B" "!!i. i ∈ insert x B ⟹ f i = g i"
          "g ∈ insert x B → carrier G"
        thus "f x ∈ carrier G" by fastforce
      qed
      also from insert have "... = g x ⊗ finprod G g B" by fastforce
      also from insert have "... = finprod G g (insert x B)"
      by (intro finprod_insert [THEN sym]) auto
      finally show ?case .
    qed
    with prems show ?thesis by simp
  next
    case False with prems show ?thesis by simp
  qed
qed

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"

  by (rule finprod_cong') (auto simp add: simp_implies_def)
```

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

```
end

declare funcsetI [rule del]
  funcset_mem [rule del]

context comm_monoid begin

lemma finprod_0 [simp]:
  "f ∈ {0::nat} → carrier G ⟹ finprod G f {..0} = f 0"
  by (simp add: Pi_def)
```

```
lemma finprod_0':
  "f ∈ {..n} → carrier G ⟹ (f 0) ⊗ finprod G f {Suc 0..n} = finprod
G f {..n}"
proof -
  assume A: "f ∈ {.. n} → carrier G"
  hence "(f 0) ⊗ finprod G f {Suc 0..n} = finprod G f {..0} ⊗ finprod
G f {Suc 0..n}"
    using finprod_0[of f] by (simp add: funcset_mem)
  also have " ... = finprod G f ({..0} ∪ {Suc 0..n})"
    using finprod_Un_disjoint[of "{..0}" "{Suc 0..n}" f] A by (simp add:
funcset_mem)
  also have " ... = finprod G f {..n}"
    by (simp add: atLeastAtMost_insertL atMost_atLeast0)
  finally show ?thesis .
qed

lemma finprod_Suc [simp]:
  "f ∈ {..Suc n} → carrier G ⟹
   finprod G f {..Suc n} = (f (Suc n) ⊗ finprod G f {..n})"
by (simp add: Pi_def atMost_Suc)

lemma finprod_Suc2:
  "f ∈ {..Suc n} → carrier G ⟹
   finprod G f {..Suc n} = (finprod G (%i. f (Suc i)) {..n} ⊗ f 0)"
proof (induct n)
  case 0 thus ?case by (simp add: Pi_def)
next
  case Suc thus ?case by (simp add: m_assoc Pi_def)
qed

lemma finprod_Suc3:
  assumes "f ∈ {..n :: nat} → carrier G"
  shows "finprod G f {.. n} = (f n) ⊗ finprod G f {..< n}"
proof (cases "n = 0")
  case True thus ?thesis
   using assms atMost_Suc by simp
next
  case False
  then obtain k where "n = Suc k"
    using not0_implies_Suc by blast
  thus ?thesis
    using finprod_Suc[of f k] assms atMost_Suc lessThan_Suc_atMost by
simp
qed

lemma finprod_reindex:
  "f ∈ (h ‘ A) → carrier G ⟹
        inj_on h A ⟹ finprod G f (h ‘ A) = finprod G (λx. f (h x)) A"
proof (induct A rule: infinite_finite_induct)
```

```
  case (infinite A)
  hence "¬ finite (h ‘ A)"
    using finite_imageD by blast
  with ‹¬ finite A› show ?case by simp
qed (auto simp add: Pi_def)

lemma finprod_const:
  assumes a [simp]: "a ∈ carrier G"
    shows "finprod G (λx. a) A = a [^] card A"
proof (induct A rule: infinite_finite_induct)
  case (insert b A)
  show ?case
  proof (subst finprod_insert[OF insert(1-2)])
    show "a ⊗ (⨂x∈A. a) = a [^] card (insert b A)"
      by (insert insert, auto, subst m_comm, auto)
  qed auto
qed auto

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"
  using i_in_A finprod_insert [of "A - {i}" i "(λj. if i = j then f j
else 1)"]
    fin_A f_Pi finprod_one [of "A - {i}"]
    finprod_cong [of "A - {i}" "A - {i}" "(λj. if i = j then f j else
1)" "(λi. 1)"]
  unfolding Pi_def simp_implies_def by (force simp add: insert_absorb)

lemma finprod_singleton_swap:
  assumes i_in_A: "i ∈ A" and fin_A: "finite A" and f_Pi: "f ∈ A →
carrier G"
  shows "(⨂j∈A. if j = i then f j else 1) = f i"
  using finprod_singleton [OF assms] by (simp add: eq_commute)

lemma finprod_mono_neutral_cong_left:
  assumes "finite B"
    and "A ⊆ B"
    and 1: "⋀i. i ∈ B - A ⟹ h i = 1"
    and gh: "⋀x. x ∈ A ⟹ g x = h x"
    and h: "h ∈ B → carrier G"
  shows "finprod G g A = finprod G h B"
proof-
  have eq: "A ∪ (B - A) = B" using ‹A ⊆ B› by blast
  have d: "A ∩ (B - A) = {}" using ‹A ⊆ B› by blast
  from ‹finite B› ‹A ⊆ B› have f: "finite A" "finite (B - A)"
    by (auto intro: finite_subset)
  have "h ∈ A → carrier G" "h ∈ B - A → carrier G"
    using assms by (auto simp: image_subset_iff_funcset)
```

```
  moreover have "finprod G g A = finprod G h A ⊗ finprod G h (B - A)"
  proof -
    have "finprod G h (B - A) = 1"
      using "1" finprod_one_eqI by blast
    moreover have "finprod G g A = finprod G h A"
      using ‹h ∈ A → carrier G› finprod_cong' gh by blast
    ultimately show ?thesis
      by (simp add: ‹h ∈ A → carrier G›)
  qed
  ultimately show ?thesis
    by (simp add: finprod_Un_disjoint [OF f d, unfolded eq])
qed

lemma finprod_mono_neutral_cong_right:
  assumes "finite B"
    and "A ⊆ B" "⋀i. i ∈ B - A ⟹ g i = 1" "⋀x. x ∈ A ⟹ g x = h
x" "g ∈ B → carrier G"
  shows "finprod G g B = finprod G h A"
  using assms  by (auto intro!: finprod_mono_neutral_cong_left [symmetric])

lemma finprod_mono_neutral_cong:
  assumes [simp]: "finite B" "finite A"
    and *: "⋀i. i ∈ B - A ⟹ h i = 1" "⋀i. i ∈ A - B ⟹ g i = 1"
    and gh: "⋀x. x ∈ A ∩ B ⟹ g x = h x"
    and g: "g ∈ A → carrier G"
    and h: "h ∈ B → carrier G"
 shows "finprod G g A = finprod G h B"
proof-
  have "finprod G g A = finprod G g (A ∩ B)"
    by (rule finprod_mono_neutral_cong_right) (use assms in auto)
  also have "... = finprod G h (A ∩ B)"
    by (rule finprod_cong) (use assms in auto)
  also have "... = finprod G h B"
    by (rule finprod_mono_neutral_cong_left) (use assms in auto)
  finally show ?thesis .
qed

end



lemma (in comm_group) power_order_eq_one:
  assumes fin [simp]: "finite (carrier G)"
    and a [simp]: "a ∈ carrier G"
  shows "a [^] card(carrier G) = one G"
proof -
  have "(⨂x∈carrier G. x) = (⨂x∈carrier G. a ⊗ x)"
    by (subst (2) finprod_reindex [symmetric],
      auto simp add: Pi_def inj_on_cmult surj_const_mult)
```

```
  also have "... = (⨂x∈carrier G. a) ⊗ (⨂x∈carrier G. x)"
    by (auto simp add: finprod_multf Pi_def)
  also have "(⨂x∈carrier G. a) = a [^] card(carrier G)"
    by (auto simp add: finprod_const)
  finally show ?thesis
    by auto
qed

lemma (in comm_monoid) finprod_UN_disjoint:
  assumes
    "finite I" "⋀i. i ∈ I ⟹ finite (A i)" "pairwise (λi j. disjnt (A
i) (A j)) I"
    "⋀i x. i ∈ I ⟹ x ∈ A i ⟹ g x ∈ carrier G"
shows "finprod G g (⋃(A ‘ I)) = finprod G (λi. finprod G g (A i)) I"
  using assms
proof (induction set: finite)
  case empty
  then show ?case
    by force
next
  case (insert i I)
  then show ?case
    unfolding pairwise_def disjnt_def
    apply clarsimp
    apply (subst finprod_Un_disjoint)
         apply (fastforce intro!: funcsetI finprod_closed)+
    done
qed

lemma (in comm_monoid) finprod_Union_disjoint:
  "⟦finite C; ⋀A. A ∈ C ⟹ finite A ∧ (∀x∈A. f x ∈ carrier G); pairwise
disjnt C⟧ ⟹
    finprod G f (⋃C) = finprod G (finprod G f) C"
  by (frule finprod_UN_disjoint [of C id f]) auto

end


theory Coset
imports Group
begin
```

# 7 Cosets and Quotient Groups

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

definition
```

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

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

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

definition
  SET_INV :: "[_,'a set] ⇒ 'a set"
    (‹(‹open_block notation=‹prefix set_inv››set'_invı _)› [81] 80)
  where "set_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"

lemma (in comm_group) subgroup_imp_normal: "subgroup A G ⟹ A ◁ G"
  by (simp add: normal_def normal_axioms_def l_coset_def r_coset_def m_comm
subgroup.mem_carrier)

lemma l_coset_eq_set_mult:
  fixes G (structure)
  shows "x <# H = {x} <#> H"
  unfolding l_coset_def set_mult_def by simp

lemma r_coset_eq_set_mult:
  fixes G (structure)
  shows "H #> x = H <#> {x}"
  unfolding r_coset_def set_mult_def by simp

lemma (in subgroup) rcosets_non_empty:
  assumes "R ∈ rcosets H"
  shows "R ≠ {}"
proof -
  obtain g where "g ∈ carrier G" "R = H #> g"
    using assms unfolding RCOSETS_def by blast
  hence "𝟭 ⊗ g ∈ R"
    using one_closed unfolding r_coset_def by blast
  thus ?thesis by blast
```

```
qed

lemma (in group) diff_neutralizes:
  assumes "subgroup H G" "R ∈ rcosets H"
  shows "⋀r1 r2. ⟦ r1 ∈ R; r2 ∈ R ⟧ ⟹ r1 ⊗ (inv r2) ∈ H"
proof -
  fix r1 r2 assume r1: "r1 ∈ R" and r2: "r2 ∈ R"
  obtain g where g: "g ∈ carrier G" "R = H #> g"
    using assms unfolding RCOSETS_def by blast
  then obtain h1 h2 where h1: "h1 ∈ H" "r1 = h1 ⊗ g"
                      and h2: "h2 ∈ H" "r2 = h2 ⊗ g"
    using r1 r2 unfolding r_coset_def by blast
  hence "r1 ⊗ (inv r2) = (h1 ⊗ g) ⊗ ((inv g) ⊗ (inv h2))"
    using inv_mult_group is_group assms(1) g(1) subgroup.mem_carrier by
fastforce
  also have " ... =  (h1 ⊗ (g ⊗ inv g) ⊗ inv h2)"
    using h1 h2 assms(1) g(1) inv_closed m_closed monoid.m_assoc
          monoid_axioms subgroup.mem_carrier
  proof -
    have "h1 ∈ carrier G"
      by (meson subgroup.mem_carrier assms(1) h1(1))
    moreover have "h2 ∈ carrier G"
      by (meson subgroup.mem_carrier assms(1) h2(1))
    ultimately show ?thesis
      using g(1) inv_closed m_assoc m_closed by presburger
  qed
  finally have "r1 ⊗ inv r2 = h1 ⊗ inv h2"
    using assms(1) g(1) h1(1) subgroup.mem_carrier by fastforce
  thus "r1 ⊗ inv r2 ∈ H" by (metis assms(1) h1(1) h2(1) subgroup_def)
qed

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

  unfolding set_mult_def by (simp add: UN_mono)
```

## 7.1 Stable Operations for Subgroups

```
lemma set_mult_consistent [simp]:
  "N <#>_(G ⦇ carrier := H ⦈) K = N <#>_G K"
  unfolding set_mult_def by simp

lemma r_coset_consistent [simp]:
  "I #>_G ⦇ carrier := H ⦈ h = I #>_G h"
  unfolding r_coset_def by simp

lemma l_coset_consistent [simp]:
  "h <#_G ⦇ carrier := H ⦈ I = h <#_G I"
  unfolding l_coset_def by simp
```

## 7.2 Basic Properties of set multiplication

```
lemma (in group) setmult_subset_G:
  assumes "H ⊆ carrier G" "K ⊆ carrier G"
  shows "H <#> K ⊆ carrier G" using assms
  by (auto simp add: set_mult_def subsetD)

lemma (in monoid) set_mult_closed:
  assumes "H ⊆ carrier G" "K ⊆ carrier G"
  shows "H <#> K ⊆ carrier G"
  using assms by (auto simp add: set_mult_def subsetD)

lemma (in group) set_mult_assoc:
  assumes "M ⊆ carrier G" "H ⊆ carrier G" "K ⊆ carrier G"
  shows "(M <#> H) <#> K = M <#> (H <#> K)"
proof
  show "(M <#> H) <#> K ⊆ M <#> (H <#> K)"
  proof
    fix x assume "x ∈ (M <#> H) <#> K"
    then obtain m h k where x: "m ∈ M" "h ∈ H" "k ∈ K" "x = (m ⊗ h)
⊗ k"
      unfolding set_mult_def by blast
    hence "x = m ⊗ (h ⊗ k)"
      using assms m_assoc by blast
    thus "x ∈ M <#> (H <#> K)"
      unfolding set_mult_def using x by blast
  qed
next
  show "M <#> (H <#> K) ⊆ (M <#> H) <#> K"
  proof
    fix x assume "x ∈ M <#> (H <#> K)"
    then obtain m h k where x: "m ∈ M" "h ∈ H" "k ∈ K" "x = m ⊗ (h ⊗
k)"
      unfolding set_mult_def by blast
    hence "x = (m ⊗ h) ⊗ k"
      using assms m_assoc rev_subsetD by metis
    thus "x ∈ (M <#> H) <#> K"
      unfolding set_mult_def using x by blast
  qed
qed
```

## 7.3 Basic Properties of Cosets

```
lemma (in group) coset_mult_assoc:
  assumes "M ⊆ carrier G" "g ∈ carrier G" "h ∈ carrier G"
  shows "(M #> g) #> h = M #> (g ⊗ h)"
  using assms by (force simp add: r_coset_def m_assoc)

lemma (in group) coset_assoc:
  assumes "x ∈ carrier G" "y ∈ carrier G" "H ⊆ carrier G"
```

```
  shows "x <# (H #> y) = (x <# H) #> y"
  using set_mult_assoc[of "{x}" H "{y}"]
  by (simp add: l_coset_eq_set_mult r_coset_eq_set_mult assms)

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

lemma (in group) coset_mult_inv1:
  assumes "M #> (x ⊗ (inv y)) = M"
    and "x ∈ carrier G" "y ∈ carrier G" "M ⊆ carrier G"
  shows "M #> x = M #> y" using assms
  by (metis coset_mult_assoc group.inv_solve_right is_group subgroup_def
subgroup_self)

lemma (in group) coset_mult_inv2:
  assumes "M #> x = M #> y"
    and "x ∈ carrier G"  "y ∈ carrier G" "M ⊆ carrier G"
  shows "M #> (x ⊗ (inv y)) = M " using assms
  by (metis group.coset_mult_assoc group.coset_mult_one inv_closed is_group
r_inv)

lemma (in group) coset_join1:
  assumes "H #> x = H"
    and "x ∈ carrier G" "subgroup H G"
  shows "x ∈ H"
  using assms r_coset_def l_one subgroup.one_closed sym by fastforce

lemma (in group) solve_equation:
  assumes "subgroup H G" "x ∈ H" "y ∈ H"
  shows "∃h ∈ H. y = h ⊗ x"
proof -
  have "y = (y ⊗ (inv x)) ⊗ x" using assms
    by (simp add: m_assoc subgroup.mem_carrier)
  moreover have "y ⊗ (inv x) ∈ H" using assms
    by (simp add: subgroup_def)
  ultimately show ?thesis by blast
qed

lemma (in group_hom) inj_on_one_iff:
   "inj_on h (carrier G) ⟷ (∀x. x ∈ carrier G ⟶ h x = one H ⟶ x
= one G)"
using G.solve_equation G.subgroup_self by (force simp: inj_on_def)

lemma inj_on_one_iff':
   "⟦h ∈ hom G H; group G; group H⟧ ⟹ inj_on h (carrier G) ⟷ (∀x.
x ∈ carrier G ⟶ h x = one H ⟶ x = one G)"
  using group_hom.inj_on_one_iff group_hom.intro group_hom_axioms.intro
by blast
```

```
lemma mon_iff_hom_one:
   "⟦group G; group H⟧ ⟹ f ∈ mon G H ⟷ f ∈ hom G H ∧ (∀x. x ∈ carrier
G ∧ f x = 𝟭⇘H⇙ ⟶ x = 𝟭⇘G⇙)"
  by (auto simp: mon_def inj_on_one_iff')

lemma (in group_hom) iso_iff: "h ∈ iso G H ⟷ carrier H ⊆ h ` carrier
G ∧ (∀x∈carrier G. h x = 𝟭⇘H⇙ ⟶ x = 𝟭)"
  by (auto simp: iso_def bij_betw_def inj_on_one_iff)

lemma (in group) repr_independence:
  assumes "y ∈ H #> x" "x ∈ carrier G" "subgroup H G"
  shows "H #> x = H #> y" using assms
by (auto simp add: r_coset_def m_assoc [symmetric]
                   subgroup.subset [THEN subsetD]
                   subgroup.m_closed solve_equation)

lemma (in group) coset_join2:
  assumes "x ∈ carrier G" "subgroup H G" "x ∈ H"
  shows "H #> x = H" using assms
  — Alternative proof is to put x = 𝟭 in repr_independence.
by (force simp add: subgroup.m_closed r_coset_def solve_equation)

lemma (in group) coset_join3:
  assumes "x ∈ carrier G" "subgroup H G" "x ∈ H"
  shows "x <# H = H"
proof
  have "⋀h. h ∈ H ⟹ x ⊗ h ∈ H" using assms
    by (simp add: subgroup.m_closed)
  thus "x <# H ⊆ H" unfolding l_coset_def by blast
next
  have "⋀h. h ∈ H ⟹ x ⊗ ((inv x) ⊗ h) = h"
    by (metis (no_types, lifting) assms group.inv_closed group.inv_solve_left
is_group
              monoid.m_closed monoid_axioms subgroup.mem_carrier)
  moreover have "⋀h. h ∈ H ⟹ (inv x) ⊗ h ∈ H"
    by (simp add: assms subgroup.m_closed subgroup.m_inv_closed)
  ultimately show "H ⊆ x <# H" unfolding l_coset_def by blast
qed

lemma (in monoid) r_coset_subset_G:
  "⟦ H ⊆ carrier G; x ∈ carrier G ⟧ ⟹ H #> x ⊆ carrier G"
by (auto simp add: r_coset_def)

lemma (in group) rcosI:
  "⟦ h ∈ H; H ⊆ carrier G; x ∈ carrier G ⟧ ⟹ h ⊗ x ∈ H #> x"
by (auto simp add: r_coset_def)

lemma (in group) rcosetsI:
```

```
    "⟦H ⊆ carrier G; x ∈ carrier G⟧ ⟹ H #> x ∈ rcosets H"
by (auto simp add: RCOSETS_def)

lemma (in group) rcos_self:
  "⟦ x ∈ carrier G; subgroup H G ⟧ ⟹ x ∈ H #> x"
  by (metis l_one rcosI subgroup_def)
```

Opposite of "repr_independence"

```
lemma (in group) repr_independenceD:
  assumes "subgroup H G" "y ∈ carrier G"
    and "H #> x = H #> y"
  shows "y ∈ H #> x"
  using assms by (simp add: rcos_self)
```

Elements of a right coset are in the carrier

```
lemma (in subgroup) elemrcos_carrier:
  assumes "group G" "a ∈ carrier G"
    and "a' ∈ H #> a"
  shows "a' ∈ carrier G"
  by (meson assms group.is_monoid monoid.r_coset_subset_G subset subsetCE)

lemma (in subgroup) rcos_const:
  assumes "group G" "h ∈ H"
  shows "H #> h = H"
  using group.coset_join2[OF assms(1), of h H]
  by (simp add: assms(2) subgroup_axioms)

lemma (in subgroup) rcos_module_imp:
  assumes "group G" "x ∈ carrier G"
    and "x' ∈ H #> x"
  shows "(x' ⊗ inv x) ∈ H"
proof -
  obtain h where h: "h ∈ H" "x' = h ⊗ x"
    using assms(3) unfolding r_coset_def by blast
  hence "x' ⊗ inv x = h"
    by (metis assms elemrcos_carrier group.inv_solve_right mem_carrier)
  thus ?thesis using h by blast
qed

lemma (in subgroup) rcos_module_rev:
  assumes "group G" "x ∈ carrier G" "x' ∈ carrier G"
    and "(x' ⊗ inv x) ∈ H"
  shows "x' ∈ H #> x"
proof -
  obtain h where h: "h ∈ H" "x' ⊗ inv x = h"
    using assms(4) unfolding r_coset_def by blast
  hence "x' = h ⊗ x"
    by (metis assms group.inv_solve_right mem_carrier)
  thus ?thesis using h unfolding r_coset_def by blast
```

**qed**

Module property of right cosets

```
lemma (in subgroup) rcos_module:
  assumes "group G" "x ∈ carrier G" "x' ∈ carrier G"
  shows "(x' ∈ H #> x) = (x' ⊗ inv x ∈ H)"
  using rcos_module_rev rcos_module_imp assms by blast
```

Right cosets are subsets of the carrier.

```
lemma (in subgroup) rcosets_carrier:
  assumes "group G" "X ∈ rcosets H"
  shows "X ⊆ carrier G"
  using assms elemrcos_carrier singletonD
  subset_eq unfolding RCOSETS_def by force
```

Multiplication of general subsets

```
lemma (in comm_group) mult_subgroups:
  assumes HG: "subgroup H G" and KG: "subgroup K G"
  shows "subgroup (H <#> K) G"
proof (rule subgroup.intro)
  show "H <#> K ⊆ carrier G"
    by (simp add: setmult_subset_G assms subgroup.subset)
next
  have "1 ⊗ 1 ∈ H <#> K"
    unfolding set_mult_def using assms subgroup.one_closed by blast
  thus "1 ∈ H <#> K" by simp
next
  show "⋀x. x ∈ H <#> K ⟹ inv x ∈ H <#> K"
  proof -
    fix x assume "x ∈ H <#> K"
    then obtain h k where hk: "h ∈ H" "k ∈ K" "x = h ⊗ k"
      unfolding set_mult_def by blast
    hence "inv x = (inv k) ⊗ (inv h)"
      by (meson inv_mult_group assms subgroup.mem_carrier)
    hence "inv x = (inv h) ⊗ (inv k)"
      by (metis hk inv_mult assms subgroup.mem_carrier)
    thus "inv x ∈ H <#> K"
      unfolding set_mult_def using hk assms
      by (metis (no_types, lifting) UN_iff singletonI subgroup_def)
  qed
next
  show "⋀x y. x ∈ H <#> K ⟹ y ∈ H <#> K ⟹ x ⊗ y ∈ H <#> K"
  proof -
    fix x y assume "x ∈ H <#> K" "y ∈ H <#> K"
    then obtain h1 k1 h2 k2 where h1k1: "h1 ∈ H" "k1 ∈ K" "x = h1 ⊗
k1"
                               and h2k2: "h2 ∈ H" "k2 ∈ K" "y = h2 ⊗ k2"
      unfolding set_mult_def by blast
```

```
    with KG HG have carr: "k1 ∈ carrier G" "h1 ∈ carrier G" "k2 ∈ carrier
G" "h2 ∈ carrier G"
        by (meson subgroup.mem_carrier)+
    have "x ⊗ y = (h1 ⊗ k1) ⊗ (h2 ⊗ k2)"
      using h1k1 h2k2 by simp
    also have " ... = h1 ⊗ (k1 ⊗ h2) ⊗ k2"
        by (simp add: carr comm_groupE(3) comm_group_axioms)
    also have " ... = h1 ⊗ (h2 ⊗ k1) ⊗ k2"
      by (simp add: carr m_comm)
    finally have "x ⊗ y  = (h1 ⊗ h2) ⊗ (k1 ⊗ k2)"
      by (simp add: carr comm_groupE(3) comm_group_axioms)
    thus "x ⊗ y ∈ H <#> K" unfolding set_mult_def
      using subgroup.m_closed[OF assms(1) h1k1(1) h2k2(1)]
            subgroup.m_closed[OF assms(2) h1k1(2) h2k2(2)] by blast
  qed
qed

lemma (in subgroup) lcos_module_rev:
  assumes "group G" "x ∈ carrier G" "x' ∈ carrier G"
    and "(inv x ⊗ x') ∈ H"
  shows "x' ∈ x <# H"
proof -
  obtain h where h: "h ∈ H" "inv x ⊗ x' = h"
    using assms(4) unfolding l_coset_def by blast
  hence "x' = x ⊗ h"
    by (metis assms group.inv_solve_left mem_carrier)
  thus ?thesis using h unfolding l_coset_def by blast
qed
```

## 7.4 Normal subgroups

```
lemma normal_imp_subgroup: "H ◁ G ⟹ subgroup H G"
  by (rule normal.axioms(1))

lemma (in group) normalI:
  "subgroup H G ⟹ (∀x ∈ carrier G. H #> x = x <# H) ⟹ H ◁ G"
  by (simp add: normal_def normal_axioms_def is_group)

lemma (in normal) inv_op_closed1:
  assumes "x ∈ carrier G" and "h ∈ H"
  shows "(inv x) ⊗ h ⊗ x ∈ H"
proof -
  have "h ⊗ x ∈ x <# H"
    using assms coset_eq assms(1) unfolding r_coset_def by blast
  then obtain h' where "h' ∈ H" "h ⊗ x = x ⊗ h'"
    unfolding l_coset_def by blast
  thus ?thesis by (metis assms inv_closed l_inv l_one m_assoc mem_carrier)
qed
```

```
lemma (in normal) inv_op_closed2:
  assumes "x ∈ carrier G" and "h ∈ H"
  shows "x ⊗ h ⊗ (inv x) ∈ H"
  using assms inv_op_closed1 by (metis inv_closed inv_inv)

lemma (in comm_group) normal_iff_subgroup:
  "N ◁ G ⟷ subgroup N G"
proof
  assume "subgroup N G"
  then show "N ◁ G"
    by unfold_locales (auto simp: subgroupE subgroup.one_closed l_coset_def
r_coset_def m_comm subgroup.mem_carrier)
qed (simp add: normal_imp_subgroup)
```

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
  assume N: "N ◁ G"
  show ?rhs
    by (blast intro: N normal.inv_op_closed2 normal_imp_subgroup)
next
  assume ?rhs
  hence sg: "subgroup N G"
    and closed: "⋀x. x∈carrier G ⟹ ∀h∈N. x ⊗ h ⊗ inv x ∈ N" by auto
  hence sb: "N ⊆ carrier G" by (simp add: subgroup.subset)
  show "N ◁ G"
  proof (intro normalI [OF sg], simp add: l_coset_def r_coset_def, clarify)
    fix x
    assume x: "x ∈ carrier G"
    show "(⋃h∈N. {h ⊗ x}) = (⋃h∈N. {x ⊗ h})"
    proof
      show "(⋃h∈N. {h ⊗ x}) ⊆ (⋃h∈N. {x ⊗ h})"
      proof clarify
        fix n
        assume n: "n ∈ N"
        show "n ⊗ x ∈ (⋃h∈N. {x ⊗ h})"
        proof
          from closed [of "inv x"]
          show "inv x ⊗ n ⊗ x ∈ N" by (simp add: x n)
          show "n ⊗ x ∈ {x ⊗ (inv x ⊗ n ⊗ x)}"
            by (simp add: x n m_assoc [symmetric] sb [THEN subsetD])
        qed
      qed
    next
      show "(⋃h∈N. {x ⊗ h}) ⊆ (⋃h∈N. {h ⊗ x})"
      proof clarify
```

```
        fix n
        assume n: "n ∈ N"
        show "x ⊗ n ∈ (⋃h∈N. {h ⊗ x})"
        proof
          show "x ⊗ n ⊗ inv x ∈ N" by (simp add: x n closed)
          show "x ⊗ n ∈ {x ⊗ n ⊗ inv x ⊗ x}"
            by (simp add: x n m_assoc sb [THEN subsetD])
        qed
      qed
    qed
  qed
qed
```

```
corollary (in group) normal_invI:
  assumes "subgroup N G" and "⋀x h. ⟦ x ∈ carrier G; h ∈ N ⟧ ⟹ x ⊗
h ⊗ inv x ∈ N"
  shows "N ◁ G"
  using assms normal_inv_iff by blast
```

```
corollary (in group) normal_invE:
  assumes "N ◁ G"
  shows "subgroup N G" and "⋀x h. ⟦ x ∈ carrier G; h ∈ N ⟧ ⟹ x ⊗ h
⊗ inv x ∈ N"
  using assms normal_inv_iff apply blast
  by (simp add: assms normal.inv_op_closed2)
```

```
lemma (in group) one_is_normal: "{1} ◁ G"
  using normal_invI triv_subgroup by force
```

The intersection of two normal subgroups is, again, a normal subgroup.

```
lemma (in group) normal_subgroup_intersect:
  assumes "M ◁ G" and "N ◁ G" shows "M ∩ N ◁ G"
  using  assms normal_inv_iff subgroups_Inter_pair by force
```

Being a normal subgroup is preserved by surjective homomorphisms.

```
lemma (in normal) surj_hom_normal_subgroup:
  assumes φ: "group_hom G F φ"
  assumes φsurj: "φ ‘ (carrier G) = carrier F"
  shows "(φ ‘ H) ◁ F"
proof (rule group.normalI)
  show "group F"
    using φ group_hom.axioms(2) by blast
next
  show "subgroup (φ ‘ H) F"
    using φ group_hom.subgroup_img_is_subgroup subgroup_axioms by blast
next
  show "∀x∈carrier F. φ ‘ H #>F x = x <#F φ ‘ H"
  proof
    fix f
```

```
    assume f: "f ∈ carrier F"
    with φsurj obtain g where g: "g ∈ carrier G" "f = φ g" by auto
    hence "φ ` H #>F f = φ ` H #>F φ g" by simp
    also have "... = (λx. (φ x) ⊗F (φ g)) ` H"
      unfolding r_coset_def image_def by auto
    also have "... = (λx. φ (x ⊗ g)) ` H"
      using subset g φ group_hom.hom_mult unfolding image_def by fastforce
    also have "... = φ ` (H #> g)"
      using φ unfolding r_coset_def by auto
    also have "... = φ ` (g <# H)"
      by (metis coset_eq g(1))
    also have "... = (λx. φ (g ⊗ x)) ` H"
      using φ unfolding l_coset_def by auto
    also have "... = (λx. (φ g) ⊗F (φ x)) ` H"
      using subset g φ group_hom.hom_mult by fastforce
    also have "... = φ g <#F φ ` H"
      unfolding l_coset_def image_def by auto
    also have "... = f <#F φ ` H"
      using g by simp
    finally show "φ ` H #>F f = f <#F φ ` H".
  qed
qed
```

Being a normal subgroup is preserved by group isomorphisms.

```
lemma iso_normal_subgroup:
  assumes φ: "φ ∈ iso G F" "group G" "group F" "H ◁ G"
  shows "(φ ` H) ◁ F"
  by (meson assms Group.iso_iff group_hom_axioms_def group_hom_def normal.surj_hom_normal_s
```

The set product of two normal subgroups is a normal subgroup.

```
lemma (in group) setmult_lcos_assoc:
  "⟦H ⊆ carrier G; K ⊆ carrier G; x ∈ carrier G⟧
      ⟹ (x <# H) <#> K = x <# (H <#> K)"
  by (force simp add: l_coset_def set_mult_def m_assoc)
```

## 7.5 More Properties of Left Cosets

```
lemma (in group) l_repr_independence:
  assumes "y ∈ x <# H" "x ∈ carrier G" and HG: "subgroup H G"
  shows "x <# H = y <# H"
proof -
  obtain h' where h': "h' ∈ H" "y = x ⊗ h'"
    using assms(1) unfolding l_coset_def by blast
  hence "x ⊗ h = y ⊗ ((inv h') ⊗ h)" if "h ∈ H" for h
  proof -
    have "h' ∈ carrier G"
      by (meson HG h'(1) subgroup.mem_carrier)
    moreover have "h ∈ carrier G"
      by (meson HG subgroup.mem_carrier that)
```

```
    ultimately show ?thesis
      by (metis assms(2) h'(2) inv_closed inv_solve_right m_assoc m_closed)
  qed
  hence "⋀xh. xh ∈ x <# H ⟹ xh ∈ y <# H"
    unfolding l_coset_def by (metis (no_types, lifting) UN_iff HG h'(1)
subgroup_def)
  moreover have "⋀h. h ∈ H ⟹ y ⊗ h = x ⊗ (h' ⊗ h)"
    using h' by (meson assms(2) HG m_assoc subgroup.mem_carrier)
  hence "⋀yh. yh ∈ y <# H ⟹ yh ∈ x <# H"
    unfolding l_coset_def using subgroup.m_closed[OF HG h'(1)] by blast
  ultimately show ?thesis by blast
qed

lemma (in group) lcos_m_assoc:
  "⟦ M ⊆ carrier G; g ∈ carrier G; h ∈ carrier G ⟧ ⟹ g <# (h <# M) =
(g ⊗ h) <# M"
by (force simp add: l_coset_def m_assoc)

lemma (in group) lcos_mult_one: "M ⊆ carrier G ⟹ 𝟭 <# M = M"
by (force simp add: l_coset_def)

lemma (in group) l_coset_subset_G:
  "⟦ H ⊆ carrier G; x ∈ carrier G ⟧ ⟹ x <# H ⊆ carrier G"
by (auto simp add: l_coset_def subsetD)

lemma (in group) l_coset_carrier:
  "⟦ y ∈ x <# H; x ∈ carrier G; subgroup H G ⟧ ⟹ y ∈ carrier G"
  by (auto simp add: l_coset_def m_assoc  subgroup.subset [THEN subsetD]
subgroup.m_closed)

lemma (in group) l_coset_swap:
  assumes "y ∈ x <# H" "x ∈ carrier G" "subgroup H G"
  shows "x ∈ y <# H"
  using assms(2) l_repr_independence[OF assms] subgroup.one_closed[OF
assms(3)]
  unfolding l_coset_def by fastforce

lemma (in group) subgroup_mult_id:
  assumes "subgroup H G"
  shows "H <#> H = H"
proof
  show "H <#> H ⊆ H"
    unfolding set_mult_def using subgroup.m_closed[OF assms] by (simp
add: UN_subset_iff)
  show "H ⊆ H <#> H"
  proof
    fix x assume x: "x ∈ H" thus "x ∈ H <#> H" unfolding set_mult_def
      using subgroup.m_closed[OF assms subgroup.one_closed[OF assms] x]
subgroup.one_closed[OF assms]
```

```
    using assms subgroup.mem_carrier by force
  qed
qed
```

### 7.5.1 Set of Inverses of an r_coset.

```
lemma (in normal) rcos_inv:
  assumes x:     "x ∈ carrier G"
  shows "set_inv (H #> x) = H #> (inv x)"
proof (simp add: r_coset_def SET_INV_def x inv_mult_group, safe)
  fix h
  assume h: "h ∈ H"
  show "inv x ⊗ inv h ∈ (⋃j∈H. {j ⊗ inv x})"
  proof
    show "inv x ⊗ inv h ⊗ x ∈ H"
      by (simp add: inv_op_closed1 h x)
    show "inv x ⊗ inv h ∈ {inv x ⊗ inv h ⊗ x ⊗ inv x}"
      by (simp add: h x m_assoc)
  qed
  show "h ⊗ inv x ∈ (⋃j∈H. {inv x ⊗ inv j})"
  proof
    show "x ⊗ inv h ⊗ inv x ∈ H"
      by (simp add: inv_op_closed2 h x)
    show "h ⊗ inv x ∈ {inv x ⊗ inv (x ⊗ inv h ⊗ inv x)}"
      by (simp add: h x m_assoc [symmetric] inv_mult_group)
  qed
qed
```

### 7.5.2 Theorems for <#> with #> or <#.

```
lemma (in group) setmult_rcos_assoc:
  "⟦H ⊆ carrier G; K ⊆ carrier G; x ∈ carrier G⟧ ⟹
    H <#> (K #> x) = (H <#> K) #> x"
  using set_mult_assoc[of H K "{x}"] by (simp add: r_coset_eq_set_mult)

lemma (in group) rcos_assoc_lcos:
  "⟦H ⊆ carrier G; K ⊆ carrier G; x ∈ carrier G⟧ ⟹
   (H #> x) <#> K = H <#> (x <# K)"
  using set_mult_assoc[of H "{x}" K]
  by (simp add: l_coset_eq_set_mult r_coset_eq_set_mult)

lemma (in normal) rcos_mult_step1:
  "⟦x ∈ carrier G; y ∈ carrier G⟧ ⟹
   (H #> x) <#> (H #> y) = (H <#> (x <# H)) #> y"
  by (simp add: setmult_rcos_assoc r_coset_subset_G
               subset l_coset_subset_G rcos_assoc_lcos)

lemma (in normal) rcos_mult_step2:
     "⟦x ∈ carrier G; y ∈ carrier G⟧
      ⟹ (H <#> (x <# H)) #> y = (H <#> (H #> x)) #> y"
```

```
by (insert coset_eq, simp add: normal_def)

lemma (in normal) rcos_mult_step3:
     "⟦x ∈ carrier G; y ∈ carrier G⟧
      ⟹ (H <#> (H #> x)) #> y = H #> (x ⊗ y)"
by (simp add: setmult_rcos_assoc coset_mult_assoc
              subgroup_mult_id normal.axioms subset normal_axioms)

lemma (in normal) rcos_sum:
     "⟦x ∈ carrier G; y ∈ carrier G⟧
      ⟹ (H #> x) <#> (H #> y) = H #> (x ⊗ y)"
by (simp add: rcos_mult_step1 rcos_mult_step2 rcos_mult_step3)

lemma (in normal) rcosets_mult_eq: "M ∈ rcosets H ⟹ H <#> M = M"
  — generalizes subgroup_mult_id
  by (auto simp add: RCOSETS_def subset
        setmult_rcos_assoc subgroup_mult_id normal.axioms normal_axioms)
```

### 7.5.3 An Equivalence Relation

```
definition
  r_congruent :: "[('a,'b)monoid_scheme, 'a set] ⇒ ('a*'a)set"
    (‹(‹open_block notation=‹prefix rcong››rcongı _)›)
  where "rcong_G H = {(x,y). x ∈ carrier G ∧ y ∈ carrier G ∧ inv_G x ⊗_G
y ∈ H}"


lemma (in subgroup) equiv_rcong:
   assumes "group G"
   shows "equiv (carrier G) (rcong H)"
proof -
  interpret group G by fact
  show ?thesis
  proof (intro equivI)
    have "rcong H ⊆ carrier G × carrier G"
      by (auto simp add: r_congruent_def)
    thus "refl_on (carrier G) (rcong H)"
      by (auto simp add: r_congruent_def refl_on_def)
  next
    show "sym (rcong H)"
    proof (simp add: r_congruent_def sym_def, clarify)
      fix x y
      assume [simp]: "x ∈ carrier G" "y ∈ carrier G"
         and "inv x ⊗ y ∈ H"
      hence "inv (inv x ⊗ y) ∈ H" by simp
      thus "inv y ⊗ x ∈ H" by (simp add: inv_mult_group)
    qed
  next
    show "trans (rcong H)"
```

```
    proof (simp add: r_congruent_def trans_def, clarify)
      fix x y z
      assume [simp]: "x ∈ carrier G" "y ∈ carrier G" "z ∈ carrier G"
         and "inv x ⊗ y ∈ H" and "inv y ⊗ z ∈ H"
      hence "(inv x ⊗ y) ⊗ (inv y ⊗ z) ∈ H" by simp
      hence "inv x ⊗ (y ⊗ inv y) ⊗ z ∈ H"
        by (simp add: m_assoc del: r_inv Units_r_inv)
      thus "inv x ⊗ z ∈ H" by simp
    qed
  qed
qed
```

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 -
  interpret group G by fact
  show ?thesis by (force simp add: r_congruent_def l_coset_def m_assoc
[symmetric] a )
qed
```

### 7.5.4 Two Distinct Right Cosets are Disjoint

```
lemma (in group) rcos_equation:
  assumes "subgroup H G"
  assumes p: "ha ⊗ a = h ⊗ b" "a ∈ carrier G" "b ∈ carrier G" "h ∈ H"
"ha ∈ H" "hb ∈ H"
  shows "hb ⊗ a ∈ (⋃h∈H. {h ⊗ b})"
proof -
  interpret subgroup H G by fact
  from p show ?thesis
    by (rule_tac UN_I [of "hb ⊗ ((inv ha) ⊗ h)"]) (auto simp: inv_solve_left
m_assoc)
qed

lemma (in group) rcos_disjoint:
  assumes "subgroup H G"
  shows "pairwise disjnt (rcosets H)"
proof -
  interpret subgroup H G by fact
  show ?thesis
    unfolding RCOSETS_def r_coset_def pairwise_def disjnt_def
    by (blast intro: rcos_equation assms sym)
qed
```

## 7.6 Further lemmas for r_congruent

The relation is a congruence

```
lemma (in normal) congruent_rcong:
  shows "congruent2 (rcong H) (rcong H) (λa b. a ⊗ b <# H)"
proof (intro congruent2I[of "carrier G" _ "carrier G" _] equiv_rcong is_group)
  fix a b c
  assume abrcong: "(a, b) ∈ rcong H"
    and ccarr: "c ∈ carrier G"

  from abrcong
      have acarr: "a ∈ carrier G"
        and bcarr: "b ∈ carrier G"
        and abH: "inv a ⊗ b ∈ H"
      unfolding r_congruent_def
      by fast+

  note carr = acarr bcarr ccarr

  from ccarr and abH
      have "inv c ⊗ (inv a ⊗ b) ⊗ c ∈ H" by (rule inv_op_closed1)
  moreover
      from carr and inv_closed
      have "inv c ⊗ (inv a ⊗ b) ⊗ c = (inv c ⊗ inv a) ⊗ (b ⊗ c)"
      by (force cong: m_assoc)
  moreover
      from carr and inv_closed
      have "... = (inv (a ⊗ c)) ⊗ (b ⊗ c)"
      by (simp add: inv_mult_group)
  ultimately
      have "(inv (a ⊗ c)) ⊗ (b ⊗ c) ∈ H" by simp
  from carr and this
     have "(b ⊗ c) ∈ (a ⊗ c) <# H"
     by (simp add: lcos_module_rev[OF is_group])
  from carr and this and is_subgroup
     show "(a ⊗ c) <# H = (b ⊗ c) <# H" by (intro l_repr_independence,
simp+)
next
  fix a b c
  assume abrcong: "(a, b) ∈ rcong H"
    and ccarr: "c ∈ carrier G"

  from ccarr have "c ∈ Units G" by simp
  hence cinvc_one: "inv c ⊗ c = 1" by (rule Units_l_inv)

  from abrcong
      have acarr: "a ∈ carrier G"
       and bcarr: "b ∈ carrier G"
       and abH: "inv a ⊗ b ∈ H"
```

```
    by (unfold r_congruent_def, fast+)

  note carr = acarr bcarr ccarr

  from carr and inv_closed
     have "inv a ⊗ b = inv a ⊗ (𝟭 ⊗ b)" by simp
  also from carr and inv_closed
      have "... = inv a ⊗ (inv c ⊗ c) ⊗ b" by simp
  also from carr and inv_closed
      have "... = (inv a ⊗ inv c) ⊗ (c ⊗ b)" by (force cong: m_assoc)
  also from carr and inv_closed
      have "... = inv (c ⊗ a) ⊗ (c ⊗ b)" by (simp add: inv_mult_group)
  finally
      have "inv a ⊗ b = inv (c ⊗ a) ⊗ (c ⊗ b)" .
  from abH and this
      have "inv (c ⊗ a) ⊗ (c ⊗ b) ∈ H" by simp

  from carr and this
     have "(c ⊗ b) ∈ (c ⊗ a) <# H"
     by (simp add: lcos_module_rev[OF is_group])
  from carr and this and is_subgroup
     show "(c ⊗ a) <# H = (c ⊗ b) <# H" by (intro l_repr_independence,
simp+)
qed
```

## 7.7 Order of a Group and Lagrange's Theorem

```
definition
  order :: "('a, 'b) monoid_scheme ⇒ nat"
  where "order S = card (carrier S)"

lemma iso_same_order:
  assumes "φ ∈ iso G H"
  shows "order G = order H"
  by (metis assms is_isoI iso_same_card order_def order_def)

lemma (in monoid) order_gt_0_iff_finite: "0 < order G ⟷ finite (carrier
G)"
  by(auto simp add: order_def card_gt_0_iff)

lemma (in group) order_one_triv_iff:
  shows "(order G = 1) = (carrier G = {𝟭})"
  by (metis One_nat_def card.empty card_Suc_eq empty_iff one_closed order_def
singleton_iff)

lemma (in group) rcosets_part_G:
  assumes "subgroup H G"
  shows "⋃(rcosets H) = carrier G"
proof -
```

```
  interpret subgroup H G by fact
  show ?thesis
    unfolding RCOSETS_def r_coset_def by auto
qed

lemma (in group) cosets_finite:
     "⟦c ∈ rcosets H;  H ⊆ carrier G;  finite (carrier G)⟧ ⟹ finite
c"
  unfolding RCOSETS_def
  by (auto simp add: r_coset_subset_G [THEN finite_subset])
```

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

```
lemma (in group) inj_on_f:
  assumes "H ⊆ carrier G" and a: "a ∈ carrier G"
  shows "inj_on (λy. y ⊗ inv a) (H #> a)"
proof
  fix x y
  assume "x ∈ H #> a" "y ∈ H #> a" and xy: "x ⊗ inv a = y ⊗ inv a"
  then have "x ∈ carrier G" "y ∈ carrier G"
    using assms r_coset_subset_G by blast+
  with xy a show "x = y"
    by auto
qed

lemma (in group) inj_on_g:
    "⟦H ⊆ carrier G;  a ∈ carrier G⟧ ⟹ inj_on (λy. y ⊗ a) H"
by (force simp add: inj_on_def subsetD)



lemma (in group) card_cosets_equal:
  assumes "R ∈ rcosets H" "H ⊆ carrier G"
  shows "∃f. bij_betw f H R"
proof -
  obtain g where g: "g ∈ carrier G" "R = H #> g"
    using assms(1) unfolding RCOSETS_def by blast

  let ?f = "λh. h ⊗ g"
  have "⋀r. r ∈ R ⟹ ∃h ∈ H. ?f h = r"
  proof -
    fix r assume "r ∈ R"
    then obtain h where "h ∈ H" "r = h ⊗ g"
      using g unfolding r_coset_def by blast
    thus "∃h ∈ H. ?f h = r" by blast
  qed
  hence "R ⊆ ?f ‘ H" by blast
  moreover have "?f ‘ H ⊆ R"
    using g unfolding r_coset_def by blast
  ultimately show ?thesis using inj_on_g unfolding bij_betw_def
```

```
    using assms(2) g(1) by auto
qed

corollary (in group) card_rcosets_equal:
  assumes "R ∈ rcosets H" "H ⊆ carrier G"
  shows "card H = card R"
  using card_cosets_equal assms bij_betw_same_card by blast

corollary (in group) rcosets_finite:
  assumes "R ∈ rcosets H" "H ⊆ carrier G" "finite H"
  shows "finite R"
  using card_cosets_equal assms bij_betw_finite is_group by blast



lemma (in group) rcosets_subset_PowG:
     "subgroup H G  ⟹ rcosets H ⊆ Pow(carrier G)"
  using rcosets_part_G by auto

proposition (in group) lagrange_finite:
  assumes "finite(carrier G)" and HG: "subgroup H G"
  shows "card(rcosets H) * card(H) = order(G)"
proof -
  have "card H * card (rcosets H) = card (⋃(rcosets H))"
  proof (rule card_partition)
    show "⋀c1 c2. ⟦c1 ∈ rcosets H; c2 ∈ rcosets H; c1 ≠ c2⟧ ⟹ c1 ∩
c2 = {}"
      using HG rcos_disjoint by (auto simp: pairwise_def disjnt_def)
  qed (auto simp: assms finite_UnionD rcosets_part_G card_rcosets_equal
subgroup.subset)
  then show ?thesis
    by (simp add: HG mult.commute order_def rcosets_part_G)
qed

theorem (in group) lagrange:
  assumes "subgroup H G"
  shows "card (rcosets H) * card H = order G"
proof (cases "finite (carrier G)")
  case True thus ?thesis using lagrange_finite assms by simp
next
  case False
  thus ?thesis
  proof (cases "finite H")
    case False thus ?thesis using ‹infinite (carrier G)›  by (simp add:
order_def)
  next
    case True
    have "infinite (rcosets H)"
    proof
```

```
      assume "finite (rcosets H)"
      hence finite_rcos: "finite (rcosets H)" by simp
      hence "card (⋃(rcosets H)) = (∑R∈(rcosets H). card R)"
        using card_Union_disjoint[of "rcosets H"] ‹finite H› rcos_disjoint[OF
assms(1)]
              rcosets_finite[where ?H = H] by (simp add: assms subgroup.subset)
      hence "order G = (∑R∈(rcosets H). card R)"
        by (simp add: assms order_def rcosets_part_G)
      hence "order G = (∑R∈(rcosets H). card H)"
        using card_rcosets_equal by (simp add: assms subgroup.subset)
      hence "order G = (card H) * (card (rcosets H))" by simp
      hence "order G ≠ 0" using finite_rcos ‹finite H› assms ex_in_conv
                              rcosets_part_G subgroup.one_closed by
fastforce
      thus False using ‹infinite (carrier G)› order_gt_0_iff_finite by
blast
    qed
    thus ?thesis using ‹infinite (carrier G)› by (simp add: order_def)
  qed
qed
```

The cardinality of the right cosets of the trivial subgroup is the cardinality of the group itself:

```
corollary (in group) card_rcosets_triv:
  assumes "finite (carrier G)"
  shows "card (rcosets {𝟭}) = order G"
  using lagrange triv_subgroup by fastforce
```

## 7.8 Quotient Groups: Factorization of a Group

```
definition
  FactGroup :: "[('a,'b) monoid_scheme, 'a set] ⇒ ('a set) monoid" (infixl
‹Mod› 65)
     — Actually defined for groups rather than monoids
   where "FactGroup G H = ⦇carrier = rcosets_G H, mult = set_mult G, one
= H⦈"

lemma (in normal) setmult_closed:
     "⟦K1 ∈ rcosets H; K2 ∈ rcosets H⟧ ⟹ K1 <#> K2 ∈ rcosets H"
by (auto simp add: rcos_sum RCOSETS_def)

lemma (in normal) setinv_closed:
     "K ∈ rcosets H ⟹ set_inv K ∈ rcosets H"
by (auto simp add: rcos_inv RCOSETS_def)

lemma (in normal) rcosets_assoc:
     "⟦M1 ∈ rcosets H; M2 ∈ rcosets H; M3 ∈ rcosets H⟧
      ⟹ M1 <#> M2 <#> M3 = M1 <#> (M2 <#> M3)"
  by (simp add: group.set_mult_assoc is_group rcosets_carrier)
```

```
lemma (in subgroup) subgroup_in_rcosets:
  assumes "group G"
  shows "H ∈ rcosets H"
proof -
  interpret group G by fact
  from _ subgroup_axioms have "H #> 1 = H"
    by (rule coset_join2) auto
  then show ?thesis
    by (auto simp add: RCOSETS_def)
qed

lemma (in normal) rcosets_inv_mult_group_eq:
     "M ∈ rcosets H ⟹ set_inv M <#> M = H"
by (auto simp add: RCOSETS_def rcos_inv rcos_sum subgroup.subset normal.axioms
normal_axioms)

theorem (in normal) factorgroup_is_group: "group (G Mod H)"
proof -
  have "⋀x. x ∈ rcosets H ⟹ ∃y∈rcosets H. y <#> x = H"
    using rcosets_inv_mult_group_eq setinv_closed by blast
  then show ?thesis
    unfolding FactGroup_def
    by (intro groupI)
      (auto simp: setmult_closed subgroup_in_rcosets rcosets_assoc rcosets_mult_eq)
qed

lemma carrier_FactGroup: "carrier(G Mod N) = (λx. r_coset G N x) ` carrier
G"
  by (auto simp: FactGroup_def RCOSETS_def)

lemma one_FactGroup [simp]: "one(G Mod N) = N"
  by (auto simp: FactGroup_def)

lemma mult_FactGroup [simp]: "monoid.mult (G Mod N) = set_mult G"
  by (auto simp: FactGroup_def)

lemma (in normal) inv_FactGroup:
  assumes "X ∈ carrier (G Mod H)"
  shows "inv⇘G Mod H⇙ X = set_inv X"
proof -
  have X: "X ∈ rcosets H"
    using assms by (simp add: FactGroup_def)
  moreover have "set_inv X <#> X = H"
    using X by (simp add: normal.rcosets_inv_mult_group_eq normal_axioms)
  moreover have "Group.group (G Mod H)"
    using normal.factorgroup_is_group normal_axioms by blast
  ultimately show ?thesis
    by (simp add: FactGroup_def group.inv_equality normal.setinv_closed
```

```
normal_axioms)
qed
```

The coset map is a homomorphism from `G` to the quotient group `G Mod H`

```
lemma (in normal) r_coset_hom_Mod:
  "(λa. H #> a) ∈ hom G (G Mod H)"
  by (auto simp add: FactGroup_def RCOSETS_def Pi_def hom_def rcos_sum)


lemma (in comm_group) set_mult_commute:
  assumes "N ⊆ carrier G" "x ∈ rcosets N" "y ∈ rcosets N"
  shows "x <#> y = y <#> x"
  using assms unfolding set_mult_def RCOSETS_def
  by auto (metis m_comm r_coset_subset_G subsetCE)+

lemma (in comm_group) abelian_FactGroup:
  assumes "subgroup N G" shows "comm_group(G Mod N)"
proof (rule group.group_comm_groupI)
  have "N ◁ G"
    by (simp add: assms normal_iff_subgroup)
  then show "Group.group (G Mod N)"
    by (simp add: normal.factorgroup_is_group)
  fix x :: "'a set" and y :: "'a set"
  assume "x ∈ carrier (G Mod N)" "y ∈ carrier (G Mod N)"
  then show "x ⊗G Mod N y = y ⊗G Mod N x"
    by (metis FactGroup_def assms mult_FactGroup partial_object.simps(1)
set_mult_commute subgroup_def)
qed


lemma FactGroup_universal:
  assumes "h ∈ hom G H" "N ◁ G"
    and h: "⋀x y. ⟦x ∈ carrier G; y ∈ carrier G; r_coset G N x = r_coset
G N y⟧ ⟹ h x = h y"
  obtains g
  where "g ∈ hom (G Mod N) H" "⋀x. x ∈ carrier G ⟹ g(r_coset G N x)
= h x"
proof -
  obtain g where g: "⋀x. x ∈ carrier G ⟹ h x = g(r_coset G N x)"
    using h function_factors_left_gen [of "λx. x ∈ carrier G" "r_coset
G N" h] by blast
  show thesis
  proof
    show "g ∈ hom (G Mod N) H"
    proof (rule homI)
      show "g (u ⊗G Mod N v) = g u ⊗H g v"
        if "u ∈ carrier (G Mod N)" "v ∈ carrier (G Mod N)" for u v
      proof -
        from that
```

```
        obtain x y where xy: "x ∈ carrier G" "u = r_coset G N x" "y ∈
carrier G"  "v = r_coset G N y"
          by (auto simp: carrier_FactGroup)
        then have "h (x ⊗G y) = h x ⊗H h y"
           by (metis hom_mult [OF ‹h ∈ hom G H›])
        then show ?thesis
          by (metis Coset.mult_FactGroup xy ‹N ◁ G› g group.subgroup_self
normal.axioms(2) normal.rcos_sum subgroup_def)
      qed
    qed (use ‹h ∈ hom G H› in ‹auto simp: carrier_FactGroup Pi_iff hom_def
simp flip: g›)
  qed (auto simp flip: g)
qed


lemma (in normal) FactGroup_pow:
  fixes k::nat
  assumes "a ∈ carrier G"
  shows "pow (FactGroup G H) (r_coset G H a) k = r_coset G H (pow G a
k)"
proof (induction k)
  case 0
  then show ?case
    by (simp add: r_coset_def)
next
  case (Suc k)
  then show ?case
    by (simp add: assms rcos_sum)
qed

lemma (in normal) FactGroup_int_pow:
  fixes k::int
  assumes "a ∈ carrier G"
  shows "pow (FactGroup G H) (r_coset G H a) k = r_coset G H (pow G a
k)"
  by (metis Group.group.axioms(1) image_eqI is_group monoid.nat_pow_closed
int_pow_def2 assms
         FactGroup_pow carrier_FactGroup inv_FactGroup rcos_inv)
```

## 7.9 The First Isomorphism Theorem

The quotient by the kernel of a homomorphism is isomorphic to the range of that homomorphism.

```
definition
  kernel :: "('a, 'm) monoid_scheme ⇒ ('b, 'n) monoid_scheme ⇒  ('a
⇒ 'b) ⇒ 'a set"
    — the kernel of a homomorphism
  where "kernel G H h = {x. x ∈ carrier G ∧ h x = 1H}"
```

```
lemma (in group_hom) subgroup_kernel: "subgroup (kernel G H h) G"
  by (auto simp add: kernel_def group.intro intro: subgroup.intro)
```

The kernel of a homomorphism is a normal subgroup

```
lemma (in group_hom) normal_kernel: "(kernel G H h) ◁ G"
  apply (simp only: G.normal_inv_iff subgroup_kernel)
  apply (simp add: kernel_def)
  done

lemma iso_kernel_image:
  assumes "group G" "group H"
  shows "f ∈ iso G H ⟷ f ∈ hom G H ∧ kernel G H f = {𝟭_G} ∧ f ` carrier
G = carrier H"
    (is "?lhs = ?rhs")
proof (intro iffI conjI)
  assume f: ?lhs
  show "f ∈ hom G H"
    using Group.iso_iff f by blast
  show "kernel G H f = {𝟭_G}"
    using assms f Group.group_def hom_one
    by (fastforce simp add: kernel_def iso_iff_mon_epi mon_iff_hom_one
set_eq_iff)
  show "f ` carrier G = carrier H"
    by (meson Group.iso_iff f)
next
  assume ?rhs
  with assms show ?lhs
    by (auto simp: kernel_def iso_def bij_betw_def inj_on_one_iff')
qed


lemma (in group_hom) FactGroup_nonempty:
  assumes "X ∈ carrier (G Mod kernel G H h)"
  shows "X ≠ {}"
  using assms unfolding FactGroup_def
  by (metis group_hom.subgroup_kernel group_hom_axioms partial_object.simps(1)
subgroup.rcosets_non_empty)


lemma (in group_hom) FactGroup_universal_kernel:
  assumes "N ◁ G" and h: "N ⊆ kernel G H h"
  obtains g where "g ∈ hom (G Mod N) H" "⋀x. x ∈ carrier G ⟹ g(r_coset
G N x) = h x"
proof -
  have "h x = h y"
    if "x ∈ carrier G" "y ∈ carrier G" "r_coset G N x = r_coset G N y"
for x y
  proof -
    have "x ⊗_G inv_G y ∈ N"
```

```
      using ‹N ◁ G› group.rcos_self normal.axioms(2) normal_imp_subgroup
         subgroup.rcos_module_imp that by metis
    with h have xy: "x ⊗G invG y ∈ kernel G H h"
      by blast
    have "h x ⊗H invH(h y) = h (x ⊗G invG y)"
      by (simp add: that)
    also have "... = 1H"
      using xy by (simp add: kernel_def)
    finally have "h x ⊗H invH(h y) = 1H" .
    then show ?thesis
      using H.inv_equality that by fastforce
  qed
  with FactGroup_universal [OF homh ‹N ◁ G›] that show thesis
    by metis
qed

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 -
  from X
  obtain g where g: "g ∈ carrier G"
           and "X = kernel G H h #> g"
    by (auto simp add: FactGroup_def RCOSETS_def)
  hence "h ‘ X = {h g}" by (auto simp add: kernel_def r_coset_def g intro!:
imageI)
  thus ?thesis by (auto simp add: g)
qed

lemma (in group_hom) FactGroup_hom:
     "(λX. the_elem (h‘X)) ∈ hom (G Mod (kernel G H h)) H"
proof -
  have "the_elem (h ‘ (X <#> X’)) = the_elem (h ‘ X) ⊗H the_elem (h ‘
X’)"
    if X: "X  ∈ carrier (G Mod kernel G H h)" and X’: "X’ ∈ carrier (G
Mod kernel G H h)" for X X’
  proof -
    obtain g and g’
      where "g ∈ carrier G" and "g’ ∈ carrier G"
        and "X = kernel G H h #> g" and "X’ = kernel G H h #> g’"
      using X X’ by (auto simp add: FactGroup_def RCOSETS_def)
    hence all: "∀x∈X. h x = h g" "∀x∈X’. h x = h g’"
      and Xsub: "X ⊆ carrier G" and X’sub: "X’ ⊆ carrier G"
      by (force simp add: kernel_def r_coset_def image_def)+
    hence "h ‘ (X <#> X’) = {h g ⊗H h g’}" using X X’
      by (auto dest!: FactGroup_nonempty intro!: image_eqI
          simp add: set_mult_def
          subsetD [OF Xsub] subsetD [OF X’sub])
    then show "the_elem (h ‘ (X <#> X’)) = the_elem (h ‘ X) ⊗H the_elem
```

```
(h ‘ X’)"
      by (auto simp add: all FactGroup_nonempty X X’ the_elem_image_unique)
  qed
  then show ?thesis
    by (simp add: hom_def FactGroup_the_elem_mem normal.factorgroup_is_group
[OF normal_kernel] group.axioms monoid.m_closed)
qed
```

Lemma for the following injectivity result

```
lemma (in group_hom) FactGroup_subset:
  assumes "g ∈ carrier G" "g’ ∈ carrier G" "h g = h g’"
  shows "kernel G H h #> g ⊆ kernel G H h #> g’"
  unfolding kernel_def r_coset_def
proof clarsimp
  fix y
  assume "y ∈ carrier G" "h y = 1H"
  with assms show "∃x. x ∈ carrier G ∧ h x = 1H ∧ y ⊗ g = x ⊗ g’"
    by (rule_tac x="y ⊗ g ⊗ inv g’" in exI) (auto simp: G.m_assoc)
qed

lemma (in group_hom) FactGroup_inj_on:
     "inj_on (λX. the_elem (h ‘ X)) (carrier (G Mod kernel G H h))"
proof (simp add: inj_on_def, clarify)
  fix X and X’
  assume X:  "X  ∈ carrier (G Mod kernel G H h)"
     and X’: "X’ ∈ carrier (G Mod kernel G H h)"
  then
  obtain g and g’
           where gX: "g ∈ carrier G"  "g’ ∈ carrier G"
              "X = kernel G H h #> g" "X’ = kernel G H h #> g’"
    by (auto simp add: FactGroup_def RCOSETS_def)
  hence all: "∀x∈X. h x = h g" "∀x∈X’. h x = h g’"
    by (force simp add: kernel_def r_coset_def image_def)+
  assume "the_elem (h ‘ X) = the_elem (h ‘ X’)"
  hence h: "h g = h g’"
    by (simp add: all FactGroup_nonempty X X’ the_elem_image_unique)
  show "X=X’" by (rule equalityI) (simp_all add: FactGroup_subset h gX)
qed
```

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
  show "(λX. the_elem (h ‘ X)) ‘ carrier (G Mod kernel G H h) ⊆ carrier
H"
    by (auto simp add: FactGroup_the_elem_mem)
```

```
  show "carrier H ⊆ (λX. the_elem (h ` X)) ` carrier (G Mod kernel G
H h)"
  proof
    fix y
    assume y: "y ∈ carrier H"
    with h obtain g where g: "g ∈ carrier G" "h g = y"
      by (blast elim: equalityE)
    hence "(⋃x∈kernel G H h #> g. {h x}) = {y}"
      by (auto simp add: y kernel_def r_coset_def)
    with g show "y ∈ (λX. the_elem (h ` X)) ` carrier (G Mod kernel G
H h)"
      apply (auto intro!: bexI image_eqI simp add: FactGroup_def RCOSETS_def)
      apply (subst the_elem_image_unique)
      apply auto
      done
  qed
qed
```

If `h` is a homomorphism from `G` onto `H`, then the quotient group `G Mod kernel G H h` is isomorphic to `H`.

```
theorem (in group_hom) FactGroup_iso_set:
  "h ` carrier G = carrier H
   ⟹ (λX. the_elem (h`X)) ∈ iso (G Mod (kernel G H h)) H"
by (simp add: iso_def FactGroup_hom FactGroup_inj_on bij_betw_def
              FactGroup_onto)

corollary (in group_hom) FactGroup_iso :
  "h ` carrier G = carrier H
   ⟹ (G Mod (kernel G H h))≅ H"
  using FactGroup_iso_set unfolding is_iso_def by auto


lemma (in group_hom) trivial_hom_iff:
  "h ` (carrier G) = { 1_H } ⟷ kernel G H h = carrier G"
  unfolding kernel_def using one_closed by force

lemma (in group_hom) trivial_ker_imp_inj:
  assumes "kernel G H h = { 1 }"
  shows "inj_on h (carrier G)"
proof (rule inj_onI)
  fix g1 g2 assume A: "g1 ∈ carrier G" "g2 ∈ carrier G" "h g1 = h g2"
  hence "h (g1 ⊗ (inv g2)) = 1_H" by simp
  hence "g1 ⊗ (inv g2) = 1"
    using A assms unfolding kernel_def by blast
  thus "g1 = g2"
    using A G.inv_equality G.inv_inv by blast
qed

lemma (in group_hom) inj_iff_trivial_ker:
```

```
  shows "inj_on h (carrier G) ⟷ kernel G H h = { 1 }"
proof
  assume inj: "inj_on h (carrier G)" show "kernel G H h = { 1 }"
    unfolding kernel_def
  proof (auto)
    fix a assume "a ∈ carrier G" "h a = 1_H" thus "a = 1"
      using inj hom_one unfolding inj_on_def by force
  qed
next
  show "kernel G H h = { 1 } ⟹ inj_on h (carrier G)"
    using trivial_ker_imp_inj by simp
qed

lemma (in group_hom) induced_group_hom':
  assumes "subgroup I G" shows "group_hom (G ⦇ carrier := I ⦈) H h"
proof -
  have "h ∈ hom (G ⦇ carrier := I ⦈) H"
    using homh subgroup.subset[OF assms] unfolding hom_def by (auto, meson
hom_mult subsetCE)
  thus ?thesis
    using subgroup.subgroup_is_group[OF assms G.group_axioms] group_axioms
    unfolding group_hom_def group_hom_axioms_def by auto
qed

lemma (in group_hom) inj_on_subgroup_iff_trivial_ker:
  assumes "subgroup I G"
  shows "inj_on h I ⟷ kernel (G ⦇ carrier := I ⦈) H h = { 1 }"
  using group_hom.inj_iff_trivial_ker[OF induced_group_hom'[OF assms]]
by simp

lemma set_mult_hom:
  assumes "h ∈ hom G H" "I ⊆ carrier G" and "J ⊆ carrier G"
  shows "h ` (I <#>_G J) = (h ` I) <#>_H (h ` J)"
proof
  show "h ` (I <#>_G J) ⊆ (h ` I) <#>_H (h ` J)"
  proof
    fix a assume "a ∈ h ` (I <#>_G J)"
    then obtain i j where i: "i ∈ I" and j: "j ∈ J" and "a = h (i ⊗_G
j)"
      unfolding set_mult_def by auto
    hence "a = (h i) ⊗_H (h j)"
      using assms unfolding hom_def by blast
    thus "a ∈ (h ` I) <#>_H (h ` J)"
      using i and j unfolding set_mult_def by auto
  qed
next
  show "(h ` I) <#>_H (h ` J) ⊆ h ` (I <#>_G J)"
  proof
    fix a assume "a ∈ (h ` I) <#>_H (h ` J)"
```

```
    then obtain i j where i: "i ∈ I" and j: "j ∈ J" and "a = (h i)
⊗H (h j)"
      unfolding set_mult_def by auto
    hence "a = h (i ⊗G j)"
      using assms unfolding hom_def by fastforce
    thus "a ∈ h ‘ (I <#>G J)"
      using i and j unfolding set_mult_def by auto
  qed
qed

corollary coset_hom:
  assumes "h ∈ hom G H" "I ⊆ carrier G" "a ∈ carrier G"
  shows "h ‘ (a <#G I) = h a <#H (h ‘ I)" and "h ‘ (I #>G a) = (h ‘ I)
#>H h a"
  unfolding l_coset_eq_set_mult r_coset_eq_set_mult using assms set_mult_hom[OF
assms(1)] by auto

corollary (in group_hom) set_mult_ker_hom:
  assumes "I ⊆ carrier G"
  shows "h ‘ (I <#> (kernel G H h)) = h ‘ I" and "h ‘ ((kernel G H h)
<#> I) = h ‘ I"
proof -
  have ker_in_carrier: "kernel G H h ⊆ carrier G"
    unfolding kernel_def by auto

  have "h ‘ (kernel G H h) = { 1H }"
    unfolding kernel_def by force
  moreover have "h ‘ I ⊆ carrier H"
    using assms by auto
  hence "(h ‘ I) <#>H { 1H } = h ‘ I" and "{ 1H } <#>H (h ‘ I) = h ‘
I"
    unfolding set_mult_def by force+
  ultimately show "h ‘ (I <#> (kernel G H h)) = h ‘ I" and "h ‘ ((kernel
G H h) <#> I) = h ‘ I"
    using set_mult_hom[OF homh assms ker_in_carrier] set_mult_hom[OF homh
ker_in_carrier assms] by simp+
qed
```

### 7.9.1 Trivial homomorphisms

```
definition trivial_homomorphism where
 "trivial_homomorphism G H f ≡ f ∈ hom G H ∧ (∀x ∈ carrier G. f x =
one H)"

lemma trivial_homomorphism_kernel:
   "trivial_homomorphism G H f ⟷ f ∈ hom G H ∧ kernel G H f = carrier
G"
  by (auto simp: trivial_homomorphism_def kernel_def)
```

**lemma** (**in** group) trivial_homomorphism_image:
  "trivial_homomorphism G H f ⟷ f ∈ hom G H ∧ f ‘ carrier G = {one H}"
  **by** (auto simp: trivial_homomorphism_def) (metis one_closed rev_image_eqI)

## 7.10 Image kernel theorems

```
lemma group_Int_image_ker:
  assumes f: "f ∈ hom G H" and g: "g ∈ hom H K"
    and "inj_on (g ∘ f) (carrier G)" "group G" "group H" "group K"
  shows "(f ‘ carrier G) ∩ (kernel H K g) = {1_H}"
proof -
  have "(f ‘ carrier G) ∩ (kernel H K g) ⊆ {1_H}"
    using assms
    apply (clarsimp simp: kernel_def o_def)
    by (metis group.is_monoid hom_one inj_on_eq_iff monoid.one_closed)
  moreover have "one H ∈ f ‘ carrier G"
    by (metis f ‹group G› ‹group H› group.is_monoid hom_one image_iff
monoid.one_closed)
  moreover have "one H ∈ kernel H K g"
    unfolding kernel_def using Group.group_def ‹group H› ‹group K› g
hom_one by blast
  ultimately show ?thesis
    by blast
qed


lemma group_sum_image_ker:
  assumes f: "f ∈ hom G H" and g: "g ∈ hom H K" and eq: "(g ∘ f) ‘ (carrier
G) = carrier K"
     and "group G" "group H" "group K"
  shows "set_mult H (f ‘ carrier G) (kernel H K g) = carrier H" (is "?lhs
= ?rhs")
proof
  show "?lhs ⊆ ?rhs"
    apply (clarsimp simp: kernel_def set_mult_def)
    by (meson ‹group H› f group.is_monoid hom_in_carrier monoid.m_closed)
  have "∃x∈carrier G. ∃z. z ∈ carrier H ∧ g z = 1_K ∧ y = f x ⊗_H z"
    if y: "y ∈ carrier H" for y
  proof -
    have "g y ∈ carrier K"
      using g hom_carrier that by blast
    with assms obtain x where x: "x ∈ carrier G" "(g ∘ f) x = g y"
      by (metis image_iff)
    with assms have invf: "inv_H f x ⊗_H y ∈ carrier H"
      by (metis group.subgroup_self hom_carrier image_subset_iff subgroup_def
y)
    moreover
    have "g (inv_H f x ⊗_H y) = 1_K"
```

```
    proof -
      have "invH f x ∈ carrier H"
        by (meson ‹group H› f group.inv_closed hom_carrier image_subset_iff
x(1))
      then have "g (invH f x ⊗H y) = g (invH f x) ⊗K g y"
        by (simp add: hom_mult [OF g] y)
      also have "... = invK (g (f x)) ⊗K g y"
        using assms x(1)
        by (metis (mono_tags, lifting) group_hom.hom_inv group_hom.intro
group_hom_axioms.intro hom_carrier image_subset_iff)
      also have "... = 1K"
        using ‹g y ∈ carrier K› assms(6) group.l_inv x(2) by fastforce
      finally show ?thesis .
    qed
    moreover
    have "y = f x ⊗H (invH f x ⊗H y)"
      using x y
      by (meson ‹group H› invf f group.inv_solve_left' hom_in_carrier)
    ultimately
    show ?thesis
      using x y by force
  qed
  then show "?rhs ⊆ ?lhs"
    by (auto simp: kernel_def set_mult_def)
qed


lemma group_sum_ker_image:
  assumes f: "f ∈ hom G H" and g: "g ∈ hom H K" and eq: "(g ∘ f) ` (carrier
G) = carrier K"
     and "group G" "group H" "group K"
   shows "set_mult H (kernel H K g) (f ` carrier G) = carrier H" (is "?lhs
= ?rhs")
proof
  show "?lhs ⊆ ?rhs"
    apply (clarsimp simp: kernel_def set_mult_def)
    by (meson ‹group H› f group.is_monoid hom_in_carrier monoid.m_closed)
  have "∃w∈carrier H. ∃x ∈ carrier G. g w = 1K ∧ y = w ⊗H f x"
    if y: "y ∈ carrier H" for y
  proof -
    have "g y ∈ carrier K"
      using g hom_carrier that by blast
    with assms obtain x where x: "x ∈ carrier G" "(g ∘ f) x = g y"
      by (metis image_iff)
    with assms have carr: "(y ⊗H invH f x) ∈ carrier H"
      by (metis group.subgroup_self hom_carrier image_subset_iff subgroup_def
y)
    moreover
    have "g (y ⊗H invH f x) = 1K"
```

```
    proof -
      have "invH f x ∈ carrier H"
        by (meson ‹group H› f group.inv_closed hom_carrier image_subset_iff
x(1))
      then have "g (y ⊗H invH f x) = g y ⊗K g (invH f x)"
        by (simp add: hom_mult [OF g] y)
      also have "... = g y ⊗K invK (g (f x))"
        using assms x(1)
        by (metis group_hom.hom_inv group_hom_axioms.intro group_hom_def
hom_in_carrier)
      also have "... = 1K"
        using ‹g y ∈ carrier K› assms(6) group.l_inv x(2)
        by (simp add: group.r_inv)
      finally show ?thesis .
    qed
    moreover
    have "y = (y ⊗H invH f x) ⊗H f x"
      using x y by (meson ‹group H› carr f group.inv_solve_right hom_carrier
image_subset_iff)
    ultimately
    show ?thesis
      using x y by force
  qed
  then show "?rhs ⊆ ?lhs"
    by (force simp: kernel_def set_mult_def)
qed

lemma group_semidirect_sum_ker_image:
  assumes "(g ∘ f) ∈ iso G K" "f ∈ hom G H" "g ∈ hom H K" "group G" "group
H" "group K"
  shows "(kernel H K g) ∩ (f ‘ carrier G) = {1H}"
        "kernel H K g <#>H (f ‘ carrier G) = carrier H"
  using assms
  by (simp_all add: iso_iff_mon_epi group_Int_image_ker group_sum_ker_image
epi_def mon_def Int_commute [of "kernel H K g"])

lemma group_semidirect_sum_image_ker:
  assumes f: "f ∈ hom G H" and g: "g ∈ hom H K" and iso: "(g ∘ f) ∈
iso G K"
     and "group G" "group H" "group K"
   shows "(f ‘ carrier G) ∩ (kernel H K g) = {1H}"
         "f ‘ carrier G <#>H (kernel H K g) = carrier H"
  using group_Int_image_ker [OF f g] group_sum_image_ker [OF f g] assms
  by (simp_all add: iso_def bij_betw_def)
```

## 7.11 Factor Groups and Direct product

```
lemma (in group) DirProd_normal :
  assumes "group K"
```

```
    and "H ◁ G"
    and "N ◁ K"
  shows "H × N ◁ G ×× K"
proof (intro group.normal_invI[OF DirProd_group[OF group_axioms assms(1)]])
  show sub : "subgroup (H × N) (G ×× K)"
    using DirProd_subgroups[OF group_axioms normal_imp_subgroup[OF assms(2)]assms(1)
         normal_imp_subgroup[OF assms(3)]].
  show "⋀x h. x ∈ carrier (G××K) ⟹ h ∈ H×N ⟹ x ⊗G××K h ⊗G××K
invG××K x ∈ H×N"
  proof-
    fix x h assume xGK : "x ∈ carrier (G ×× K)" and hHN : " h ∈ H ×
N"
    hence hGK : "h ∈ carrier (G ×× K)" using subgroup.subset[OF sub]
by auto
    from xGK obtain x1 x2 where x1x2 :"x1 ∈ carrier G" "x2 ∈ carrier
K" "x = (x1,x2)"
      unfolding DirProd_def by fastforce
    from hHN obtain h1 h2 where h1h2 : "h1 ∈ H" "h2 ∈ N" "h = (h1,h2)"
      unfolding DirProd_def by fastforce
    hence h1h2GK : "h1 ∈ carrier G" "h2 ∈ carrier K"
      using normal_imp_subgroup subgroup.subset assms by blast+
    have "invG ×× K x = (invG x1,invK x2)"
      using inv_DirProd[OF group_axioms assms(1) x1x2(1)x1x2(2)] x1x2
by auto
    hence "x ⊗G ×× K h ⊗G ×× K invG ×× K x = (x1 ⊗ h1 ⊗ inv x1,x2
⊗K h2 ⊗K invK x2)"
      using h1h2 x1x2 h1h2GK by auto
    moreover have "x1 ⊗ h1 ⊗ inv x1 ∈ H" "x2 ⊗K h2 ⊗K invK x2 ∈ N"
      using assms x1x2 h1h2 assms by (simp_all add: normal.inv_op_closed2)
    hence "(x1 ⊗ h1 ⊗ inv x1, x2 ⊗K h2 ⊗K invK x2)∈ H × N" by auto
    ultimately show " x ⊗G ×× K h ⊗G ×× K invG ×× K x ∈ H × N" by
auto
  qed
qed

lemma (in group) FactGroup_DirProd_multiplication_iso_set :
  assumes "group K"
    and "H ◁ G"
    and "N ◁ K"
  shows "(λ (X, Y). X × Y) ∈ iso  ((G Mod H) ×× (K Mod N)) (G ×× K
Mod H × N)"
proof-
  have R :"(λ(X, Y). X × Y) ∈ carrier (G Mod H) × carrier (K Mod N)
→ carrier (G ×× K Mod H × N)"
    unfolding r_coset_def Sigma_def DirProd_def FactGroup_def RCOSETS_def
by force
  moreover have "(∀x∈carrier (G Mod H). ∀y∈carrier (K Mod N). ∀xa∈carrier
(G Mod H).
                ∀ya∈carrier (K Mod N). (x <#> xa) × (y <#>K ya) =  x
```

```
× y <#>G ×× K xa × ya)"
    unfolding set_mult_def by force
  moreover have "(∀x∈carrier (G Mod H). ∀y∈carrier (K Mod N). ∀xa∈carrier
(G Mod H).
                ∀ya∈carrier (K Mod N).  x × y = xa × ya ⟶ x = xa
∧ y = ya)"
    unfolding  FactGroup_def using times_eq_iff subgroup.rcosets_non_empty
    by (metis assms(2) assms(3) normal_def partial_object.select_convs(1))
  moreover have "(λ(X, Y). X × Y) ` (carrier (G Mod H) × carrier (K
Mod N)) =
                                     carrier (G ×× K Mod H × N)"
  proof -
    have 1: "⋀x a b. ⟦a ∈ carrier (G Mod H); b ∈ carrier (K Mod N)⟧ ⟹
a × b ∈ carrier (G ×× K Mod H × N)"
      using R by force
    have 2: "⋀z. z ∈ carrier (G ×× K Mod H × N) ⟹ ∃x∈carrier (G Mod
H). ∃y∈carrier (K Mod N). z = x × y"
      unfolding DirProd_def FactGroup_def RCOSETS_def r_coset_def by force
    show ?thesis
      unfolding image_def by (auto simp: intro: 1 2)
  qed
  ultimately show ?thesis
    unfolding iso_def hom_def bij_betw_def inj_on_def by simp
qed

corollary (in group) FactGroup_DirProd_multiplication_iso_1 :
  assumes "group K"
    and "H ⊲ G"
    and "N ⊲ K"
  shows "  ((G Mod H) ×× (K Mod N)) ≅ (G ×× K Mod H × N)"
  unfolding is_iso_def using FactGroup_DirProd_multiplication_iso_set
assms by auto

corollary (in group) FactGroup_DirProd_multiplication_iso_2 :
  assumes "group K"
    and "H ⊲ G"
    and "N ⊲ K"
  shows "(G ×× K Mod H × N) ≅ ((G Mod H) ×× (K Mod N))"
  using FactGroup_DirProd_multiplication_iso_1 group.iso_sym assms
        DirProd_group[OF normal.factorgroup_is_group normal.factorgroup_is_group]
  by blast
```

### 7.11.1 More Lemmas about set multiplication

A group multiplied by a subgroup stays the same

```
lemma (in group) set_mult_carrier_idem:
  assumes "subgroup H G"
  shows "(carrier G) <#> H = carrier G"
proof
```

```
  show "(carrier G)<#>H ⊆ carrier G"
    unfolding set_mult_def using subgroup.subset assms by blast
next
  have " (carrier G) #>  1 = carrier G" unfolding set_mult_def r_coset_def
group_axioms by simp
  moreover have "(carrier G) #>  1 ⊆ (carrier G) <#> H" unfolding set_mult_def
r_coset_def
    using assms subgroup.one_closed[OF assms] by blast
  ultimately show "carrier G ⊆ (carrier G) <#> H" by simp
qed
```

Same lemma as above, but everything is included in a subgroup

```
lemma (in group) set_mult_subgroup_idem:
  assumes HG: "subgroup H G" and NG: "subgroup N (G ⦇ carrier := H ⦈)"
  shows "H <#> N = H"
  using group.set_mult_carrier_idem[OF subgroup.subgroup_is_group[OF HG
group_axioms] NG] by simp
```

A normal subgroup is commutative with set multiplication

```
lemma (in group) commut_normal:
  assumes "subgroup H G" and "N⊲G"
  shows "H<#>N = N<#>H"
proof-
  have aux1: "{H <#> N} = {⋃h∈H. h <# N }" unfolding set_mult_def l_coset_def
by auto
  also have "... = {⋃h∈H. N #> h }" using assms normal.coset_eq subgroup.mem_carrier
by fastforce
  moreover have aux2: "{N <#> H} = {⋃h∈H. N #> h }"unfolding set_mult_def
r_coset_def by auto
  ultimately show "H<#>N = N<#>H" by simp
qed
```

Same lemma as above, but everything is included in a subgroup

```
lemma (in group) commut_normal_subgroup:
  assumes "subgroup H G" and "N ⊲ (G⦇ carrier := H ⦈)"
    and "subgroup K (G ⦇ carrier := H ⦈)"
  shows "K <#> N = N <#> K"
  by (metis assms(2) assms(3) group.commut_normal normal.axioms(2) set_mult_consistent)
```

### 7.11.2 Lemmas about intersection and normal subgroups

Mostly by Jakob von Raumer

```
lemma (in group) normal_inter:
  assumes "subgroup H G"
    and "subgroup K G"
    and "H1⊲G⦇carrier := H⦈"
  shows "(H1∩K)⊲(G⦇carrier:= (H∩K)⦈)"
proof-
```

```
  define HK and H1K and GH and GHK
    where "HK = H∩K" and "H1K=H1∩K" and "GH =G(|carrier := H|)" and "GHK
= (G(|carrier:= (H∩K)|))"
  show "H1K◁GHK"
  proof (intro group.normal_invI[of GHK H1K])
    show "Group.group GHK"
      using GHK_def subgroups_Inter_pair subgroup_imp_group assms by blast

  next
    have  H1K_incl:"subgroup H1K (G(|carrier:= (H∩K)|))"
    proof(intro subgroup_incl)
      show "subgroup H1K G"
        using assms normal_imp_subgroup subgroups_Inter_pair incl_subgroup
H1K_def by blast
    next
      show "subgroup (H∩K) G" using HK_def subgroups_Inter_pair assms
by auto
    next
      have "H1 ⊆ (carrier (G(|carrier:=H|)))"
        using  assms(3) normal_imp_subgroup subgroup.subset by blast
      also have "... ⊆ H" by simp
      thus "H1K ⊆H∩K"
        using H1K_def calculation by auto
    qed
    thus "subgroup H1K GHK" using GHK_def by simp
  next
    show "⋀ x h. x∈carrier GHK ⟹ h∈H1K ⟹ x ⊗GHK h ⊗GHK invGHK x∈
H1K"
    proof-
      have invHK: "⟦y∈HK⟧ ⟹ invGHK y = invGH y"
        using m_inv_consistent assms HK_def GH_def GHK_def subgroups_Inter_pair
by simp
      have multHK : "⟦x∈HK;y∈HK⟧ ⟹  x ⊗(G(|carrier:=HK|)) y =  x ⊗ y"
        using HK_def by simp
      fix x assume p: "x∈carrier GHK"
      fix h assume p2 : "h:H1K"
      have "carrier(GHK)⊆HK"
        using GHK_def HK_def by simp
      hence xHK:"x∈HK" using p by auto
      hence invx:"invGHK x = invGH x"
        using invHK assms GHK_def HK_def GH_def m_inv_consistent subgroups_Inter_pair
by simp
      have "H1⊆carrier(GH)"
        using assms GH_def normal_imp_subgroup subgroup.subset by blast
      hence hHK:"h∈HK"
        using p2 H1K_def HK_def GH_def by auto
      hence xhx_egal : "x ⊗GHK h ⊗GHK invGHKx =  x ⊗GH h ⊗GH invGH x"
        using invx invHK multHK GHK_def GH_def by auto
      have xH:"x∈carrier(GH)"
```

```
        using xHK HK_def GH_def by auto
      have hH:"h∈carrier(GH)"
        using hHK HK_def GH_def by auto
      have  "(∀x∈carrier (GH). ∀h∈H1.  x ⊗GH h ⊗GH invGH x ∈ H1)"
        using assms GH_def normal.inv_op_closed2 by fastforce
      hence INCL_1 : "x ⊗GH h ⊗GH invGH x ∈ H1"
        using  xH H1K_def p2 by blast
      have " x ⊗GH h ⊗GH invGH x ∈ HK"
        using assms HK_def subgroups_Inter_pair hHK xHK
        by (metis GH_def inf.cobounded1 subgroup_def subgroup_incl)
      hence " x ⊗GH h ⊗GH invGH x ∈ K" using HK_def by simp
      hence " x ⊗GH h ⊗GH invGH x ∈ H1K" using INCL_1 H1K_def by auto
      thus  "x ⊗GHK h ⊗GHK invGHK x ∈ H1K" using xhx_egal by simp
    qed
  qed
qed

lemma (in group) normal_Int_subgroup:
  assumes "subgroup H G"
    and "N ⊲ G"
  shows "(N∩H) ⊲ (G⦇carrier := H⦈)"
proof -
  define K where "K = carrier G"
  have "G⦇carrier := K⦈ =  G" using K_def by auto
  moreover have "subgroup K G" using K_def subgroup_self by blast
  moreover have "normal N (G ⦇carrier :=K⦈)" using assms K_def by simp
  ultimately have "N ∩ H ⊲ G⦇carrier := K ∩ H⦈"
    using normal_inter[of K H N] assms(1) by blast
  moreover have "K ∩ H = H" using K_def assms subgroup.subset by blast
  ultimately show "normal (N∩H) (G⦇carrier := H⦈)"
 by auto
qed

lemma (in group) normal_restrict_supergroup:
  assumes "subgroup S G" "N ⊲ G" "N ⊆ S"
  shows "N ⊲ (G⦇carrier := S⦈)"
  by (metis assms inf.absorb_iff1 normal_Int_subgroup)
```

A subgroup relation survives factoring by a normal subgroup.

```
lemma (in group) normal_subgroup_factorize:
  assumes "N ⊲ G" and "N ⊆ H" and "subgroup H G"
  shows "subgroup (rcosetsG⦇carrier := H⦈ N) (G Mod N)"
proof -
  interpret GModN: group "G Mod N"
    using assms(1) by (rule normal.factorgroup_is_group)
  have "N ⊲ G⦇carrier := H⦈"
    using assms by (metis normal_restrict_supergroup)
  hence grpHN: "group (G⦇carrier := H⦈ Mod N)"
    by (rule normal.factorgroup_is_group)
```

```
  have "(<#>G⦇carrier:=H⦈) = (λU K. (⋃h∈U. ⋃k∈K. {h ⊗G⦇carrier := H⦈ k}))"

    using set_mult_def by metis
  moreover have "... = (λU K. (⋃h∈U. ⋃k∈K. {h ⊗G k}))"
    by auto
  moreover have "(<#>) = (λU K. (⋃h∈U. ⋃k∈K. {h ⊗ k}))"
    using set_mult_def by metis
  ultimately have "(<#>G⦇carrier:=H⦈) = (<#>G)"
    by simp
  with grpHN have "group ((G Mod N)⦇carrier := (rcosetsG⦇carrier := H⦈
N)⦈)"
    unfolding FactGroup_def by auto
  moreover have "rcosetsG⦇carrier := H⦈ N ⊆ carrier (G Mod N)"
    unfolding FactGroup_def RCOSETS_def r_coset_def using assms(3) subgroup.subset

    by fastforce
  ultimately show ?thesis
    using GModN.group_incl_imp_subgroup by blast
qed
```

A normality relation survives factoring by a normal subgroup.

```
lemma (in group) normality_factorization:
  assumes NG: "N ◁ G" and NH: "N ⊆ H" and HG: "H ◁ G"
  shows "(rcosetsG⦇carrier := H⦈ N) ◁ (G Mod N)"
proof -
  from assms(1) interpret GModN: group "G Mod N" by (metis normal.factorgroup_is_group)
  show ?thesis
    unfolding GModN.normal_inv_iff
  proof (intro conjI strip)
    show "subgroup (rcosetsG⦇carrier := H⦈ N) (G Mod N)"
      using assms normal_imp_subgroup normal_subgroup_factorize by force
  next
    fix U V
    assume U: "U ∈ carrier (G Mod N)" and V: "V ∈ rcosetsG⦇carrier := H⦈
N"
    then obtain g where g: "g ∈ carrier G" "U = N #> g"
      unfolding FactGroup_def RCOSETS_def by auto
    from V obtain h where h: "h ∈ H" "V = N #> h"
      unfolding FactGroup_def RCOSETS_def r_coset_def by auto
    hence hG: "h ∈ carrier G"
      using HG normal_imp_subgroup subgroup.mem_carrier by force
    hence ghG: "g ⊗ h ∈ carrier G"
      using g m_closed by auto
    from g h have "g ⊗ h ⊗ inv g ∈ H"
      using HG normal_inv_iff by auto
    moreover have "U <#> V <#> invG Mod N U = N #> (g ⊗ h ⊗ inv g)"
    proof -
      from g U have "invG Mod N U = N #> inv g"
        using NG normal.inv_FactGroup normal.rcos_inv by fastforce
```

```
    hence "U <#> V <#> inv_G Mod N U = (N #> g) <#> (N #> h) <#> (N #>
inv g)"
      using g h by simp
    also have "... = N #> (g ⊗ h ⊗ inv g)"
      using g hG NG inv_closed ghG normal.rcos_sum by force
    finally show ?thesis .
  qed
  ultimately show "U ⊗_G Mod N V ⊗_G Mod N inv_G Mod N U ∈ rcosets_G(|carrier := H|)
N"
    unfolding RCOSETS_def r_coset_def by auto
 qed
qed
```

Factorizing by the trivial subgroup is an isomorphism.

```
lemma (in group) trivial_factor_iso:
 shows "the_elem ∈ iso (G Mod {1}) G"
proof -
 have "group_hom G G (λx. x)"
  unfolding group_hom_def group_hom_axioms_def hom_def using is_group
by simp
 moreover have "(λx. x) ‘ carrier G = carrier G"
  by simp
 moreover have "kernel G G (λx. x) = {1}"
  unfolding kernel_def by auto
 ultimately show ?thesis using group_hom.FactGroup_iso_set
  by force
qed
```

And the dual theorem to the previous one: Factorizing by the group itself gives the trivial group

```
lemma (in group) self_factor_iso:
 shows "(λX. the_elem ((λx. 1) ‘ X)) ∈ iso (G Mod (carrier G)) (G(| carrier
:= {1} |))"
proof -
 have "group (G(|carrier := {1}|))"
  by (metis subgroup_imp_group triv_subgroup)
 hence "group_hom G (G(|carrier := {1}|)) (λx. 1)"
  unfolding group_hom_def group_hom_axioms_def hom_def using is_group
by auto
 moreover have "(λx. 1) ‘ carrier G = carrier (G(|carrier := {1}|))"
  by auto
 moreover have "kernel G (G(|carrier := {1}|)) (λx. 1) = carrier G"
  unfolding kernel_def by auto
 ultimately show ?thesis using group_hom.FactGroup_iso_set
  by force
qed
```

Factoring by a normal subgroups yields the trivial group iff the subgroup is the whole group.

```
lemma (in normal) fact_group_trivial_iff:
  assumes "finite (carrier G)"
  shows "(carrier (G Mod H) = {1G Mod H}) ⟷ (H = carrier G)"
proof
  assume "carrier (G Mod H) = {1G Mod H}"
  moreover have "order (G Mod H) * card H = order G"
    by (simp add: FactGroup_def lagrange order_def subgroup_axioms)
  ultimately have "card H = order G" unfolding order_def by auto
  thus "H = carrier G"
    by (simp add: assms card_subset_eq order_def subset)
next
  assume "H = carrier G"
  with assms is_subgroup lagrange
  have "card (rcosets H) * order G = order G"
    by (simp add: order_def)
  then have "card (rcosets H) = 1"
    using assms order_gt_0_iff_finite by auto
  hence "order (G Mod H) = 1"
    unfolding order_def FactGroup_def by auto
  thus "carrier (G Mod H) = {1G Mod H}"
    using factorgroup_is_group by (metis group.order_one_triv_iff)
qed
```

The union of all the cosets contained in a subgroup of a quotient group acts as a represenation for that subgroup.

```
lemma (in normal) factgroup_subgroup_union_char:
  assumes "subgroup A (G Mod H)"
  shows "(⋃A) = {x ∈ carrier G. H #> x ∈ A}"
proof
  show "⋃A ⊆ {x ∈ carrier G. H #> x ∈ A}"
  proof
    fix x
    assume x: "x ∈ ⋃A"
    then obtain a where a: "a ∈ A" "x ∈ a" and xx: "x ∈ carrier G"
      using subgroup.subset assms by (force simp add: FactGroup_def RCOSETS_def
r_coset_def)
    from assms a obtain y where y: "y ∈ carrier G" "a = H #> y"
      using subgroup.subset unfolding FactGroup_def RCOSETS_def by force
    with a have "x ∈ H #> y" by simp
    hence "H #> y = H #> x" using y is_subgroup repr_independence by
auto
    with y(2) a(1) have "H #> x ∈ A"
      by auto
    with xx show "x ∈ {x ∈ carrier G. H #> x ∈ A}" by simp
  qed
next
  show "{x ∈ carrier G. H #> x ∈ A} ⊆ ⋃A"
    using rcos_self subgroup_axioms by auto
qed
```

```
lemma (in normal) factgroup_subgroup_union_subgroup:
  assumes "subgroup A (G Mod H)"
  shows "subgroup (⋃A) G"
proof -
  have "subgroup {x ∈ carrier G. H #> x ∈ A} G"
  proof
    show "{x ∈ carrier G. H #> x ∈ A} ⊆ carrier G" by auto
  next
    fix x y
    assume xy: "x ∈ {x ∈ carrier G. H #> x ∈ A}" "y ∈ {x ∈ carrier G.
H #> x ∈ A}"
    then have "(H #> x) <#> (H #> y) ∈ A"
      using subgroup.m_closed assms unfolding FactGroup_def  by fastforce
    hence "H #> (x ⊗ y) ∈ A"
      using xy rcos_sum by force
    with xy show "x ⊗ y ∈ {x ∈ carrier G. H #> x ∈ A}" by blast
  next
    have "H #> 1 ∈ A"
      using assms subgroup.one_closed subset by fastforce
    with assms one_closed show "1 ∈ {x ∈ carrier G. H #> x ∈ A}" by
simp
  next
    fix x
    assume x: "x ∈ {x ∈ carrier G. H #> x ∈ A}"
    hence invx: "inv x ∈ carrier G" using inv_closed by simp
    from assms x have "set_inv (H #> x) ∈ A" using subgroup.m_inv_closed
      using inv_FactGroup subgroup.mem_carrier by fastforce
    with invx show "inv x ∈ {x ∈ carrier G. H #> x ∈ A}"
      using rcos_inv x by force
  qed
  with assms factgroup_subgroup_union_char show ?thesis by auto
qed

lemma (in normal) factgroup_subgroup_union_normal:
  assumes "A ◁ (G Mod H)"
  shows "⋃A ◁ G"
proof -
  have "{x ∈ carrier G. H #> x ∈ A} ◁ G"
    unfolding normal_def normal_axioms_def
  proof (intro conjI strip)
    from assms show "subgroup {x ∈ carrier G. H #> x ∈ A} G"
      by (metis (full_types) factgroup_subgroup_union_char factgroup_subgroup_union_subgrou
normal_imp_subgroup)
  next
    interpret Anormal: normal A "(G Mod H)" using assms by simp
    show "{x ∈ carrier G. H #> x ∈ A} #> x = x <# {x ∈ carrier G. H #>
x ∈ A}" if x: "x ∈ carrier G" for x
    proof -
```

```
      have "y ∈ x <# {x ∈ carrier G. H #> x ∈ A}"
        if y: "y ∈ {x ∈ carrier G. H #> x ∈ A} #> x" for y
      proof -
        from that obtain x' where x': "x' ∈ carrier G" "H #> x' ∈ A"
"y = x' ⊗ x"
          unfolding r_coset_def by auto
        from x(1) have Hx: "H #> x ∈ carrier (G Mod H)"
          unfolding FactGroup_def RCOSETS_def by force
        with x' have "(invG Mod H (H #> x)) ⊗G Mod H (H #> x') ⊗G Mod H
(H #> x) ∈ A"
          using Anormal.inv_op_closed1 by auto
        hence "(set_inv (H #> x)) <#> (H #> x') <#> (H #> x) ∈ A"
          using inv_FactGroup Hx unfolding FactGroup_def by auto
        hence "(H #> (inv x)) <#> (H #> x') <#> (H #> x) ∈ A"
          using x(1) by (metis rcos_inv)
        hence "H #> (inv x ⊗ x' ⊗ x) ∈ A"
          by (metis inv_closed m_closed rcos_sum x'(1) x(1))
        moreover have "inv x ⊗ x' ⊗ x ∈ carrier G"
          using x x' by (metis inv_closed m_closed)
        ultimately have xcoset: "x ⊗ (inv x ⊗ x' ⊗ x) ∈ x <# {x ∈ carrier
G. H #> x ∈ A}"
          unfolding l_coset_def using x(1) by auto
        have "x ⊗ (inv x ⊗ x' ⊗ x) = (x ⊗ inv x) ⊗ x' ⊗ x"
          by (metis Units_eq Units_inv_Units m_assoc m_closed x'(1) x(1))
        also have "... = y"
          by (simp add: x x')
        finally have "x ⊗ (inv x ⊗ x' ⊗ x) = y" .
        with xcoset show ?thesis by auto
      qed
      moreover have "y ∈ {x ∈ carrier G. H #> x ∈ A} #> x"
        if y: "y ∈ x <# {x ∈ carrier G. H #> x ∈ A}" for y
      proof -
        from that obtain x' where x': "x' ∈ carrier G" "H #> x' ∈ A"
"y = x ⊗ x'" unfolding l_coset_def by auto
        from x(1) have invx: "inv x ∈ carrier G"
          by (rule inv_closed)
        hence Hinvx: "H #> (inv x) ∈ carrier (G Mod H)"
          unfolding FactGroup_def RCOSETS_def by force
        with x' have "(invG Mod H (H #> inv x)) ⊗G Mod H (H #> x') ⊗G Mod H
(H #> inv x) ∈ A"
          using invx Anormal.inv_op_closed1 by auto
        hence "(set_inv (H #> inv x)) <#> (H #> x') <#> (H #> inv x) ∈
A"
          using inv_FactGroup Hinvx unfolding FactGroup_def by auto
        hence "H #> (x ⊗ x' ⊗ inv x) ∈ A"
          by (simp add: rcos_inv rcos_sum x x'(1))
        moreover have "x ⊗ x' ⊗ inv x ∈ carrier G" using x x' by (metis
inv_closed m_closed)
        ultimately have xcoset: "(x ⊗ x' ⊗ inv x) ⊗ x ∈ {x ∈ carrier
```

```
G. H #> x ∈ A} #> x"
          unfolding r_coset_def using invx by auto
        have "(x ⊗ x' ⊗ inv x) ⊗ x = (x ⊗ x') ⊗ (inv x ⊗ x)"
          by (metis Units_eq Units_inv_Units m_assoc m_closed x'(1) x(1))
        also have "... = y"
          by (simp add: x x')
        finally have "x ⊗ x' ⊗ inv x ⊗ x = y".
        with xcoset show ?thesis by auto
      qed
      ultimately show ?thesis
        by auto
    qed
  qed auto
  with assms show ?thesis
    by (metis (full_types) factgroup_subgroup_union_char normal_imp_subgroup)
qed

lemma (in normal) factgroup_subgroup_union_factor:
  assumes "subgroup A (G Mod H)"
  shows "A = rcosets_G(|carrier := ⋃A|) H"
  using assms subgroup.mem_carrier factgroup_subgroup_union_char by (fastforce
simp: RCOSETS_def FactGroup_def)
```

# 8 Flattening the type of group carriers

Flattening here means to convert the type of group elements from 'a set to 'a. This is possible whenever the empty set is not an element of the group. By Jakob von Raumer

```
definition flatten where
  "flatten (G::('a set, 'b) monoid_scheme) rep = (|carrier=(rep ` (carrier
G)),
      monoid.mult=(λ x y. rep ((the_inv_into (carrier G) rep x) ⊗_G (the_inv_into
(carrier G) rep y))),
      one=rep 1_G |)"

lemma flatten_set_group_hom:
  assumes group: "group G"
  assumes inj: "inj_on rep (carrier G)"
  shows "rep ∈ hom G (flatten G rep)"
  by (force simp add: hom_def flatten_def inj the_inv_into_f_f)

lemma flatten_set_group:
  assumes "group G" "inj_on rep (carrier G)"
  shows "group (flatten G rep)"
proof (rule groupI)
  fix x y
  assume "x ∈ carrier (flatten G rep)" and "y ∈ carrier (flatten G rep)"
  then show "x ⊗_flatten G rep y ∈ carrier (flatten G rep)"
```

```
    using assms group.surj_const_mult the_inv_into_f_f by (fastforce simp:
flatten_def)
next
  show "1flatten G rep ∈ carrier (flatten G rep)"
    unfolding flatten_def by (simp add: assms group.is_monoid)
next
  fix x y z
  assume "x ∈ carrier (flatten G rep)" "y ∈ carrier (flatten G rep)"
"z ∈ carrier (flatten G rep)"
  then show "x ⊗flatten G rep y ⊗flatten G rep z = x ⊗flatten G rep (y
⊗flatten G rep z)"
    by (auto simp: assms flatten_def group.is_monoid monoid.m_assoc monoid.m_closed
the_inv_into_f_f)
next
  fix x
  assume x: "x ∈ carrier (flatten G rep)"
  then show "1flatten G rep ⊗flatten G rep x = x"
    by (auto simp: assms group.is_monoid the_inv_into_f_f flatten_def)
  then have "∃y∈carrier G. rep (y ⊗G z) = rep 1G" if "z ∈ carrier G"
for z
    by (metis ‹group G› group.l_inv_ex that)
  with assms x show "∃y∈carrier (flatten G rep). y ⊗flatten G rep x =
1flatten G rep"
    by (auto simp: flatten_def the_inv_into_f_f)
qed

lemma (in normal) flatten_set_group_mod_inj:
  shows "inj_on (λU. SOME g. g ∈ U) (carrier (G Mod H))"
proof (rule inj_onI)
  fix U V
  assume U: "U ∈ carrier (G Mod H)" and V: "V ∈ carrier (G Mod H)"
  then obtain g h where g: "U = H #> g" "g ∈ carrier G" and h: "V =
H #> h" "h ∈ carrier G"
    unfolding FactGroup_def RCOSETS_def by auto
  hence notempty: "U ≠ {}" "V ≠ {}"
    by (metis empty_iff is_subgroup rcos_self)+
  assume "(SOME g. g ∈ U) = (SOME g. g ∈ V)"
  with notempty have "(SOME g. g ∈ U) ∈ U ∩ V"
    by (metis IntI ex_in_conv someI)
  thus "U = V"
    by (metis Int_iff g h is_subgroup repr_independence)
qed

lemma (in normal) flatten_set_group_mod:
  shows "group (flatten (G Mod H) (λU. SOME g. g ∈ U))"
  by (simp add: factorgroup_is_group flatten_set_group flatten_set_group_mod_inj)

lemma (in normal) flatten_set_group_mod_iso:
  shows "(λU. SOME g. g ∈ U) ∈ iso (G Mod H) (flatten (G Mod H) (λU.
```

```
SOME g. g ∈ U))"
proof -
  have "(λU. SOME g. g ∈ U) ∈ hom (G Mod H) (flatten (G Mod H) (λU. SOME
g. g ∈ U))"
    using factorgroup_is_group flatten_set_group_hom flatten_set_group_mod_inj
by blast
  moreover
  have "inj_on (λU. SOME g. g ∈ U) (carrier (G Mod H))"
    using flatten_set_group_mod_inj by blast
  ultimately show ?thesis
    by (simp add: iso_def bij_betw_def flatten_def)
qed

end
```

```
theory Exponent
imports Main "HOL-Computational_Algebra.Primes"
begin
```

# 9 Sylow's Theorem

The Combinatorial Argument Underlying the First Sylow Theorem

needed in this form to prove Sylow's theorem

```
corollary (in algebraic_semidom) div_combine:
  "⟦prime_elem p; ¬ p ^ Suc r dvd n; p ^ (a + r) dvd n * k⟧ ⟹ p ^ a
dvd k"
  by (metis add_Suc_right mult.commute prime_elem_power_dvd_cases)

lemma exponent_p_a_m_k_equation:
  fixes p :: nat
  assumes "0 < m" "0 < k" "p ≠ 0" "k < p^a"
    shows "multiplicity p (p^a * m - k) = multiplicity p (p^a - k)"
proof (rule multiplicity_cong [OF iffI])
  fix r
  assume *: "p ^ r dvd p ^ a * m - k"
  show "p ^ r dvd p ^ a - k"
  proof -
    have "k ≤ p ^ a * m" using assms
      by (meson nat_dvd_not_less dvd_triv_left leI mult_pos_pos order.strict_trans)
    then have "r ≤ a"
      by (meson "*" ‹0 < k› ‹k < p^a› dvd_diffD1 dvd_triv_left leI less_imp_le_nat
nat_dvd_not_less power_le_dvd)
    then have "p^r dvd p^a * m" by (simp add: le_imp_power_dvd)
    thus ?thesis
      by (meson ‹k ≤ p ^ a * m› ‹r ≤ a› * dvd_diffD1 dvd_diff_nat le_imp_power_dvd)
  qed
```

```
next
  fix r
  assume *: "p ^ r dvd p ^ a - k"
  with assms have "r ≤ a"
    by (metis diff_diff_cancel less_imp_le_nat nat_dvd_not_less nat_le_linear
power_le_dvd zero_less_diff)
  show "p ^ r dvd p ^ a * m - k"
  proof -
    have "p^r dvd p^a*m"
      by (simp add: ‹r ≤ a› le_imp_power_dvd)
    then show ?thesis
      by (meson assms * dvd_diffD1 dvd_diff_nat le_imp_power_dvd less_imp_le_nat
‹r ≤ a›)
  qed
qed

lemma p_not_div_choose_lemma:
  fixes p :: nat
  assumes eeq: "⋀i. Suc i < K ⟹ multiplicity p (Suc i) = multiplicity
p (Suc (j + i))"
      and "k < K" and p: "prime p"
    shows "multiplicity p (j + k choose k) = 0"
  using ‹k < K›
proof (induction k)
  case 0 then show ?case by simp
next
  case (Suc k)
  then have *: "(Suc (j+k) choose Suc k) > 0" by simp
  then have "multiplicity p ((Suc (j+k) choose Suc k) * Suc k) = multiplicity
p (Suc k)"
    by (subst Suc_times_binomial_eq [symmetric], subst prime_elem_multiplicity_mult_distrib
       (insert p Suc.prems, simp_all add: eeq [symmetric] Suc.IH)
  with p * show ?case
    by (subst (asm) prime_elem_multiplicity_mult_distrib) simp_all
qed
```

The lemma above, with two changes of variables

```
lemma p_not_div_choose:
  assumes "k < K" and "k ≤ n"
      and eeq: "⋀j. ⟦0<j; j<K⟧ ⟹ multiplicity p (n - k + (K - j)) =
multiplicity p (K - j)" "prime p"
    shows "multiplicity p (n choose k) = 0"
apply (rule p_not_div_choose_lemma [of K p "n-k" k, simplified assms nat_minus_add_max
max_absorb1])
apply (metis add_Suc_right eeq diff_diff_cancel order_less_imp_le zero_less_Suc
zero_less_diff)
apply (rule TrueI)+
done
```

```
proposition const_p_fac:
  assumes "m>0" and prime: "prime p"
  shows "multiplicity p (p^a * m choose p^a) = multiplicity p m"
proof-
  from assms have p: "0 < p ^ a" "0 < p^a * m" "p^a ≤ p^a * m"
    by (auto simp: prime_gt_0_nat)
  have *: "multiplicity p ((p^a * m - 1) choose (p^a - 1)) = 0"
    apply (rule p_not_div_choose [where K = "p^a"])
    using p exponent_p_a_m_k_equation by (auto simp: diff_le_mono prime)
  have "multiplicity p ((p ^ a * m choose p ^ a) * p ^ a) = a + multiplicity
p m"
  proof -
    have "(p ^ a * m choose p ^ a) * p ^ a = p ^ a * m * (p ^ a * m -
1 choose (p ^ a - 1))"
      (is "_ = ?rhs") using prime
      by (subst times_binomial_minus1_eq [symmetric]) (auto simp: prime_gt_0_nat)
    also from p have "p ^ a - Suc 0 ≤ p ^ a * m - Suc 0" by linarith
    with prime * p have "multiplicity p ?rhs = multiplicity p (p ^ a
* m)"
      by (subst prime_elem_multiplicity_mult_distrib) auto
    also have "... = a + multiplicity p m"
      using prime p by (subst prime_elem_multiplicity_mult_distrib) simp_all
    finally show ?thesis .
  qed
  then show ?thesis
    using prime p by (subst (asm) prime_elem_multiplicity_mult_distrib)
simp_all
qed

end


theory Sylow
  imports Coset Exponent
begin
```

See also [4].

The combinatorial argument is in theory Exponent.

```
lemma le_extend_mult: "⟦0 < c; a ≤ b⟧ ⟹ a ≤ b * c" for c :: nat
  using gr0_conv_Suc by fastforce

locale sylow = group +
  fixes p and a and m and calM and RelM
  assumes prime_p: "prime p"
    and order_G: "order G = (p^a) * m"
    and finite_G[iff]: "finite (carrier G)"
  defines "calM ≡ {s. s ⊆ carrier G ∧ card s = p^a}"
    and "RelM ≡ {(N1, N2). N1 ∈ calM ∧ N2 ∈ calM ∧ (∃g ∈ carrier G.
N1 = N2 #> g)}"
```

```
begin

lemma RelM_subset: "RelM ⊆ calM × calM"
  by (auto simp only: RelM_def)

lemma RelM_refl_on: "refl_on calM RelM"
  by (auto simp: refl_on_def RelM_def calM_def) (blast intro!: coset_mult_one
[symmetric])

lemma RelM_sym: "sym RelM"
  unfolding sym_def RelM_def calM_def
  using coset_mult_assoc coset_mult_one r_inv_ex
  by (smt (verit, best) case_prod_conv mem_Collect_eq)

lemma RelM_trans: "trans RelM"
  by (auto simp add: trans_def RelM_def calM_def coset_mult_assoc)

lemma RelM_equiv: "equiv calM RelM"
  using RelM_subset RelM_refl_on RelM_sym RelM_trans by (intro equivI)

lemma M_subset_calM_prep: "M' ∈ calM // RelM  ⟹ M' ⊆ calM"
  unfolding RelM_def by (blast elim!: quotientE)

end
```

## 9.1 Main Part of the Proof

```
locale sylow_central = sylow +
  fixes H and M1 and M
  assumes M_in_quot: "M ∈ calM // RelM"
    and not_dvd_M: "¬ (p ^ Suc (multiplicity p m) dvd card M)"
    and M1_in_M: "M1 ∈ M"
  defines "H ≡ {g. g ∈ carrier G ∧ M1 #> g = M1}"
begin

lemma M_subset_calM: "M ⊆ calM"
  by (simp add: M_in_quot M_subset_calM_prep)

lemma card_M1: "card M1 = p^a"
  using M1_in_M M_subset_calM calM_def by blast

lemma exists_x_in_M1: "∃x. x ∈ M1"
  using prime_p [THEN prime_gt_Suc_0_nat] card_M1 one_in_subset by fastforce

lemma M1_subset_G [simp]: "M1 ⊆ carrier G"
  using M1_in_M M_subset_calM calM_def mem_Collect_eq subsetCE by blast

lemma M1_inj_H: "∃f ∈ H→M1. inj_on f H"
proof -
```

```
  from exists_x_in_M1 obtain m1 where m1M: "m1 ∈ M1"..
  show ?thesis
  proof
    have "m1 ∈ carrier G"
      by (simp add: m1M M1_subset_G [THEN subsetD])
    then show "inj_on (λz∈H. m1 ⊗ z) H"
      by (simp add: H_def inj_on_def)
    show "restrict ((⊗) m1) H ∈ H → M1"
      using H_def m1M rcosI by auto
  qed
qed

end
```

## 9.2 Discharging the Assumptions of sylow_central

```
context sylow
begin

lemma EmptyNotInEquivSet: "{} ∉ calM // RelM"
  using RelM_equiv in_quotient_imp_non_empty by blast

lemma existsM1inM: "M ∈ calM // RelM ⟹ ∃M1. M1 ∈ M"
  using RelM_equiv equiv_Eps_in by blast

lemma zero_less_o_G: "0 < order G"
  by (simp add: order_def card_gt_0_iff carrier_not_empty)

lemma zero_less_m: "m > 0"
  using zero_less_o_G by (simp add: order_G)

lemma card_calM: "card calM = (p^a) * m choose p^a"
  by (simp add: calM_def n_subsets order_G [symmetric] order_def)

lemma zero_less_card_calM: "card calM > 0"
  by (simp add: card_calM zero_less_binomial le_extend_mult zero_less_m)

lemma max_p_div_calM: "¬ (p ^ Suc (multiplicity p m) dvd card calM)"
proof
  assume "p ^ Suc (multiplicity p m) dvd card calM"
  with zero_less_card_calM prime_p
  have "Suc (multiplicity p m) ≤ multiplicity p (card calM)"
    by (intro multiplicity_geI) auto
  then show False
    by (simp add: card_calM const_p_fac prime_p zero_less_m)
qed

lemma finite_calM: "finite calM"
  unfolding calM_def by (rule finite_subset [where B = "Pow (carrier
```

```
G)"]) auto

lemma lemma_A1: "∃M ∈ calM // RelM. ¬ (p ^ Suc (multiplicity p m) dvd
card M)"
  using RelM_equiv equiv_imp_dvd_card finite_calM max_p_div_calM by blast

end
```

### 9.2.1 Introduction and Destruct Rules for H

```
context sylow_central
begin

lemma H_I: "⟦g ∈ carrier G; M1 #> g = M1⟧ ⟹ g ∈ H"
  by (simp add: H_def)

lemma H_into_carrier_G: "x ∈ H ⟹ x ∈ carrier G"
  by (simp add: H_def)

lemma in_H_imp_eq: "g ∈ H ⟹ M1 #> g = M1"
  by (simp add: H_def)

lemma H_m_closed: "⟦x ∈ H; y ∈ H⟧ ⟹ x ⊗ y ∈ H"
  by (simp add: H_def coset_mult_assoc [symmetric])

lemma H_not_empty: "H ≠ {}"
  by (force simp add: H_def intro: exI [of _ 1])

lemma H_is_subgroup: "subgroup H G"
proof (rule subgroupI)
  show "H ⊆ carrier G"
    using H_into_carrier_G by blast
  show "⋀a. a ∈ H ⟹ inv a ∈ H"
    by (metis H_I H_into_carrier_G M1_subset_G coset_mult_assoc coset_mult_one
in_H_imp_eq inv_closed r_inv)
  show "⋀a b. ⟦a ∈ H; b ∈ H⟧ ⟹ a ⊗ b ∈ H"
    by (blast intro: H_m_closed)
qed (use H_not_empty in auto)

lemma rcosetGM1g_subset_G: "⟦g ∈ carrier G; x ∈ M1 #> g⟧ ⟹ x ∈ carrier
G"
  by (blast intro: M1_subset_G [THEN r_coset_subset_G, THEN subsetD])

lemma finite_M1: "finite M1"
  by (rule finite_subset [OF M1_subset_G finite_G])

lemma finite_rcosetGM1g: "g ∈ carrier G ⟹ finite (M1 #> g)"
  using rcosetGM1g_subset_G finite_G M1_subset_G cosets_finite rcosetsI
by blast
```

```
lemma M1_cardeq_rcosetGM1g: "g ∈ carrier G ⟹ card (M1 #> g) = card
M1"
  by (metis M1_subset_G card_rcosets_equal rcosetsI)

lemma M1_RelM_rcosetGM1g:
  assumes "g ∈ carrier G"
  shows "(M1, M1 #> g) ∈ RelM"
proof -
  have "M1 #> g ⊆ carrier G"
    by (simp add: assms r_coset_subset_G)
  moreover have "card (M1 #> g) = p ^ a"
    using assms by (simp add: card_M1 M1_cardeq_rcosetGM1g)
  moreover have "∃h∈carrier G. M1 = M1 #> g #> h"
    by (metis assms M1_subset_G coset_mult_assoc coset_mult_one r_inv_ex)
  ultimately show ?thesis
    by (simp add: RelM_def calM_def card_M1)
qed

end
```

## 9.3 Equal Cardinalities of M and the Set of Cosets

Injections between M and $\mathtt{rcosets}_{\mathtt{G}}$ H show that their cardinalities are equal.

```
lemma ElemClassEquiv: "⟦equiv A r; C ∈ A // r⟧ ⟹ ∀x ∈ C. ∀y ∈ C. (x,
y) ∈ r"
  unfolding equiv_def quotient_def sym_def trans_def by blast

context sylow_central
begin

lemma M_elem_map: "M2 ∈ M ⟹ ∃g. g ∈ carrier G ∧ M1 #> g = M2"
  using M1_in_M M_in_quot [THEN RelM_equiv [THEN ElemClassEquiv]]
  by (simp add: RelM_def) (blast dest!: bspec)

lemmas M_elem_map_carrier = M_elem_map [THEN someI_ex, THEN conjunct1]

lemmas M_elem_map_eq = M_elem_map [THEN someI_ex, THEN conjunct2]

lemma M_funcset_rcosets_H:
  "(λx∈M. H #> (SOME g. g ∈ carrier G ∧ M1 #> g = x)) ∈ M → rcosets
H"
  by (metis (lifting) H_is_subgroup M_elem_map_carrier rcosetsI restrictI
subgroup.subset)

lemma inj_M_GmodH: "∃f ∈ M → rcosets H. inj_on f M"
proof
  let ?inv = "λx. SOME g. g ∈ carrier G ∧ M1 #> g = x"
  show "inj_on (λx∈M. H #> ?inv x) M"
```

```
  proof (rule inj_onI, simp)
    fix x y
    assume eq: "H #> ?inv x = H #> ?inv y" and xy: "x ∈ M" "y ∈ M"
    have "x = M1 #> ?inv x"
      by (simp add: M_elem_map_eq ‹x ∈ M›)
    also have "... = M1 #> ?inv y"
    proof (rule coset_mult_inv1 [OF in_H_imp_eq [OF coset_join1]])
      show "H #> ?inv x ⊗ inv (?inv y) = H"
        by (simp add: H_into_carrier_G M_elem_map_carrier xy coset_mult_inv2
eq subsetI)
    qed (simp_all add: H_is_subgroup M_elem_map_carrier xy)
    also have "... = y"
      using M_elem_map_eq ‹y ∈ M› by simp
    finally show "x=y" .
  qed
  show "(λx∈M. H #> ?inv x) ∈ M → rcosets H"
    by (rule M_funcset_rcosets_H)
qed

end
```

### 9.3.1 The Opposite Injection

```
context sylow_central
begin

lemma H_elem_map: "H1 ∈ rcosets H ⟹ ∃g. g ∈ carrier G ∧ H #> g =
H1"
  by (auto simp: RCOSETS_def)

lemmas H_elem_map_carrier = H_elem_map [THEN someI_ex, THEN conjunct1]

lemmas H_elem_map_eq = H_elem_map [THEN someI_ex, THEN conjunct2]

lemma rcosets_H_funcset_M:
  "(λC ∈ rcosets H. M1 #> (SOME g. g ∈ carrier G ∧ H #> g = C)) ∈ rcosets
H → M"
  using in_quotient_imp_closed [OF RelM_equiv M_in_quot _  M1_RelM_rcosetGM1g]
  by (simp add: M1_in_M H_elem_map_carrier RCOSETS_def)

lemma inj_GmodH_M: "∃g ∈ rcosets H→M. inj_on g (rcosets H)"
proof
  let ?inv = "λx. SOME g. g ∈ carrier G ∧ H #> g = x"
  show "inj_on (λC∈rcosets H. M1 #> ?inv C) (rcosets H)"
  proof (rule inj_onI, simp)
    fix x y
    assume eq: "M1 #> ?inv x = M1 #> ?inv y" and xy: "x ∈ rcosets H"
"y ∈ rcosets H"
    have "x = H #> ?inv x"
```

```
      by (simp add: H_elem_map_eq ‹x ∈ rcosets H›)
    also have "... = H #> ?inv y"
    proof (rule coset_mult_inv1 [OF coset_join2])
      show "?inv x ⊗ inv (?inv y) ∈ carrier G"
        by (simp add: H_elem_map_carrier ‹x ∈ rcosets H› ‹y ∈ rcosets
H›)
      then show "(?inv x) ⊗ inv (?inv y) ∈ H"
        by (simp add: H_I H_elem_map_carrier xy coset_mult_inv2 eq)
      show "H ⊆ carrier G"
        by (simp add: H_is_subgroup subgroup.subset)
    qed (simp_all add: H_is_subgroup H_elem_map_carrier xy)
    also have "... = y"
      by (simp add: H_elem_map_eq ‹y ∈ rcosets H›)
    finally show "x=y" .
  qed
  show "(λC∈rcosets H. M1 #> ?inv C) ∈ rcosets H → M"
    using rcosets_H_funcset_M by blast
qed

lemma calM_subset_PowG: "calM ⊆ Pow (carrier G)"
  by (auto simp: calM_def)


lemma finite_M: "finite M"
  by (metis M_subset_calM finite_calM rev_finite_subset)

lemma cardMeqIndexH: "card M = card (rcosets H)"
  using inj_M_GmodH inj_GmodH_M
  by (metis H_is_subgroup card_bij finite_G finite_M finite_UnionD rcosets_part_G)

lemma index_lem: "card M * card H = order G"
  by (simp add: cardMeqIndexH lagrange H_is_subgroup)

lemma card_H_eq: "card H = p^a"
proof (rule antisym)
  show "p^a ≤ card H"
  proof (rule dvd_imp_le)
    have "p ^ (a + multiplicity p m) dvd card M * card H"
      by (simp add: index_lem multiplicity_dvd order_G power_add)
    then show "p ^ a dvd card H"
      using div_combine not_dvd_M prime_p by blast
    show "0 < card H"
      by (blast intro: subgroup.finite_imp_card_positive H_is_subgroup)
  qed
next
  show "card H ≤ p^a"
    using M1_inj_H card_M1 card_inj finite_M1 by fastforce
qed
```

**end**

```
lemma (in sylow) sylow_thm: "∃H. subgroup H G ∧ card H = p^a"
proof -
  obtain M where M: "M ∈ calM // RelM" "¬ (p ^ Suc (multiplicity p m)
dvd card M)"
    using lemma_A1 by blast
  then obtain M1 where "M1 ∈ M"
    by (metis existsM1inM)
  define H where "H ≡ {g. g ∈ carrier G ∧ M1 #> g = M1}"
  with M ‹M1 ∈ M›
  interpret sylow_central G p a m calM RelM H M1 M
    by unfold_locales (auto simp add: H_def calM_def RelM_def)
  show ?thesis
    using H_is_subgroup card_H_eq by blast
qed
```

Needed because the locale's automatic definition refers to `semigroup G` and `Group.group_axioms G` rather than simply to `Group.group G`.

```
lemma sylow_eq: "sylow G p a m ⟷ group G ∧ sylow_axioms G p a m"
  by (simp add: sylow_def group_def)
```

## 9.4 Sylow's Theorem

```
theorem sylow_thm:
  "⟦prime p; group G; order G = (p^a) * m; finite (carrier G)⟧
    ⟹ ∃H. subgroup H G ∧ card H = p^a"
  by (rule sylow.sylow_thm [of G p a m]) (simp add: sylow_eq sylow_axioms_def)

end
```

```
theory Bij
imports Group
begin
```

# 10 Bijections of a Set, Permutation and Automorphism Groups

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

definition
  BijGroup :: "'a set ⇒ ('a ⇒ 'a) monoid"
  where "BijGroup S =
    ⦇carrier = Bij S,
```

```
    mult = λg ∈ Bij S. λf ∈ Bij S. compose S g f,
    one = λx ∈ S. x⦈"
```

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

lemma Bij_imp_extensional: "f ∈ Bij S ⟹ f ∈ extensional S"
  by (simp add: Bij_def)

lemma Bij_imp_funcset: "f ∈ Bij S ⟹ f ∈ S → S"
  by (auto simp add: Bij_def bij_betw_imp_funcset)
```

## 10.1 Bijections Form a Group

```
lemma restrict_inv_into_Bij: "f ∈ Bij S ⟹ (λx ∈ S. (inv_into S f)
x) ∈ Bij S"
  by (simp add: Bij_def bij_betw_inv_into)

lemma id_Bij: "(λx∈S. x) ∈ Bij S "
  by (auto simp add: Bij_def bij_betw_def inj_on_def)

lemma compose_Bij: "⟦x ∈ Bij S; y ∈ Bij S⟧ ⟹ compose S x y ∈ Bij S"
  by (auto simp add: Bij_def bij_betw_compose)

lemma Bij_compose_restrict_eq:
    "f ∈ Bij S ⟹ compose S (restrict (inv_into S f) S) f = (λx∈S.
x)"
  by (simp add: Bij_def compose_inv_into_id)

theorem group_BijGroup: "group (BijGroup S)"
  apply (simp add: BijGroup_def)
  apply (rule groupI)
      apply (auto simp: compose_Bij id_Bij Bij_imp_funcset Bij_imp_extensional
compose_assoc [symmetric])
  apply (blast intro: Bij_compose_restrict_eq restrict_inv_into_Bij)
  done
```

## 10.2 Automorphisms Form a Group

```
lemma Bij_inv_into_mem: "⟦ f ∈ Bij S;  x ∈ S⟧ ⟹ inv_into S f x ∈ S"
by (simp add: Bij_def bij_betw_def inv_into_into)

lemma Bij_inv_into_lemma:
  assumes eq: "⋀x y. ⟦x ∈ S; y ∈ S⟧ ⟹ h(g x y) = g (h x) (h y)"
      and hg: "h ∈ Bij S" "g ∈ S → S → S" and "x ∈ S" "y ∈ S"
  shows "inv_into S h (g x y) = g (inv_into S h x) (inv_into S h y)"
proof -
  have "h ‘ S = S"
    by (metis (no_types) Bij_def Int_iff assms(2) bij_betw_def mem_Collect_eq)
  with ‹x ∈ S› ‹y ∈ S› have "∃x'∈S. ∃y'∈S. x = h x' ∧ y = h y'"
```

```
    by auto
  then show ?thesis
    using assms
    by (auto simp add: Bij_def bij_betw_def eq [symmetric] inv_f_f funcset_mem
[THEN funcset_mem])
qed


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"
  by (simp add: auto_def hom_def restrictI group.axioms id_Bij)

lemma (in group) mult_funcset: "mult G ∈ carrier G → carrier G → carrier
G"
  by (simp add:  Pi_I group.axioms)

lemma (in group) restrict_inv_into_hom:
      "[|h ∈ hom G G; h ∈ Bij (carrier G)|]
       ⟹ restrict (inv_into (carrier G) h) (carrier G) ∈ hom G G"
  by (simp add: hom_def Bij_inv_into_mem restrictI mult_funcset
                group.axioms Bij_inv_into_lemma)

lemma inv_BijGroup:
     "f ∈ Bij S ⟹ m_inv (BijGroup S) f = (λx ∈ S. (inv_into S f) x)"
apply (rule group.inv_equality [OF group_BijGroup])
apply (simp_all add:BijGroup_def restrict_inv_into_Bij Bij_compose_restrict_eq)
done

lemma (in group) subgroup_auto:
      "subgroup (auto G) (BijGroup (carrier G))"
proof (rule subgroup.intro)
  show "auto G ⊆ carrier (BijGroup (carrier G))"
    by (force simp add: auto_def BijGroup_def)
next
  fix x y
  assume "x ∈ auto G" "y ∈ auto G"
  thus "x ⊗BijGroup (carrier G) y ∈ auto G"
    by (force simp add: BijGroup_def is_group auto_def Bij_imp_funcset

                            group.hom_compose compose_Bij)
next
  show "1BijGroup (carrier G) ∈ auto G" by (simp add:  BijGroup_def id_in_auto)
```

```
next
  fix x
  assume "x ∈ auto G"
  thus "inv_{BijGroup (carrier G)} x ∈ auto G"
    by (simp del: restrict_apply
        add: inv_BijGroup auto_def restrict_inv_into_Bij restrict_inv_into_hom)
qed

theorem (in group) AutoGroup: "group (AutoGroup G)"
by (simp add: AutoGroup_def subgroup.subgroup_is_group subgroup_auto
              group_BijGroup)

end


theory Ring
imports FiniteProduct
begin
```

# 11 The Algebraic Hierarchy of Rings

## 11.1 Abelian Groups

```
record 'a ring = "'a monoid" +
  zero :: 'a (‹𝟬ı›)
  add :: "['a, 'a] ⇒ 'a" (infixl ‹⊕ı› 65)

abbreviation
  add_monoid :: "('a, 'm) ring_scheme ⇒ ('a, 'm) monoid_scheme"
  where "add_monoid R ≡ ⦇ carrier = carrier R, mult = add R, one = zero
R, ... = (undefined :: 'm) ⦈"
```

Derived operations.

```
definition
  a_inv :: "[('a, 'm) ring_scheme, 'a ] ⇒ 'a" (‹(‹open_block notation=‹prefix
⊖››⊖ı _)› [81] 80)
  where "a_inv R = m_inv (add_monoid R)"

definition
  a_minus :: "[('a, 'm) ring_scheme, 'a, 'a] => 'a" (‹(‹notation=‹infix
⊖››_ ⊖ı _)› [65,66] 65)
  where "x ⊖_R y = x ⊕_R (⊖_R y)"

definition
  add_pow :: "[_, ('b :: semiring_1), 'a] ⇒ 'a"
    (‹(‹open_block notation=‹mixfix ·››[_] ·ı _)› [81, 81] 80)
  where "[k] ·_R a = pow (add_monoid R) a k"

locale abelian_monoid =
```

```
  fixes G (structure)
  assumes a_comm_monoid:
     "comm_monoid (add_monoid G)"

definition
  finsum :: "[('b, 'm) ring_scheme, 'a ⇒ 'b, 'a set] ⇒ 'b" where
  "finsum G = finprod (add_monoid G)"

syntax
  "_finsum" :: "index ⇒ idt ⇒ 'a set ⇒ 'b ⇒ 'b"
    (‹(‹indent=3 notation=‹binder ⨁››⨁__∈_. _)› [1000, 0, 51, 10]
10)
syntax_consts
  "_finsum" ⇌ finsum
translations
  "⨁Gi∈A. b" ⇌ "CONST finsum G (λi. b) A"
  — Beware of argument permutation!


locale abelian_group = abelian_monoid +
  assumes a_comm_group:
     "comm_group (add_monoid G)"
```

## 11.2 Basic Properties

```
lemma abelian_monoidI:
  fixes R (structure)
  assumes "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y ∈ carrier
R"
      and "0 ∈ carrier R"
      and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)"
      and "⋀x. x ∈ carrier R ⟹ 0 ⊕ x = x"
      and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y = y ⊕ x"
  shows "abelian_monoid R"
  by (auto intro!: abelian_monoid.intro comm_monoidI intro: assms)

lemma abelian_monoidE:
  fixes R (structure)
  assumes "abelian_monoid R"
  shows "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y ∈ carrier
R"
    and "0 ∈ carrier R"
    and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)"
    and "⋀x. x ∈ carrier R ⟹ 0 ⊕ x = x"
    and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y = y ⊕ x"
  using assms unfolding abelian_monoid_def comm_monoid_def comm_monoid_axioms_def
monoid_def by auto
```

```
lemma abelian_groupI:
  fixes R (structure)
  assumes "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y ∈ carrier
R"
      and "𝟬 ∈ carrier R"
      and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)"
      and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y = y ⊕ x"
      and "⋀x. x ∈ carrier R ⟹ 𝟬 ⊕ x = x"
      and "⋀x. x ∈ carrier R ⟹ ∃y ∈ carrier R. y ⊕ x = 𝟬"
  shows "abelian_group R"
  by (auto intro!: abelian_group.intro abelian_monoidI
      abelian_group_axioms.intro comm_monoidI comm_groupI
    intro: assms)

lemma abelian_groupE:
  fixes R (structure)
  assumes "abelian_group R"
  shows "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y ∈ carrier
R"
    and "𝟬 ∈ carrier R"
    and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)"
    and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊕ y = y ⊕ x"
    and "⋀x. x ∈ carrier R ⟹ 𝟬 ⊕ x = x"
    and "⋀x. x ∈ carrier R ⟹ ∃y ∈ carrier R. y ⊕ x = 𝟬"
  using abelian_group.a_comm_group assms comm_groupE by fastforce+

lemma (in abelian_monoid) a_monoid:
  "monoid (add_monoid G)"
by (rule comm_monoid.axioms, rule a_comm_monoid)

lemma (in abelian_group) a_group:
  "group (add_monoid G)"
  by (simp add: group_def a_monoid)
    (simp add: comm_group.axioms group.axioms a_comm_group)

lemmas monoid_record_simps = partial_object.simps monoid.simps
```

Transfer facts from multiplicative structures via interpretation.

```
sublocale abelian_monoid <
       add: monoid "(add_monoid G)"
  rewrites "carrier (add_monoid G) = carrier G"
       and "mult    (add_monoid G) = add G"
       and "one     (add_monoid G) = zero G"
       and "(λa k. pow (add_monoid G) a k) = (λa k. add_pow G k a)"
  by (rule a_monoid) (auto simp add: add_pow_def)
```

```
context abelian_monoid
begin

lemmas a_closed = add.m_closed
lemmas zero_closed = add.one_closed
lemmas a_assoc = add.m_assoc
lemmas l_zero = add.l_one
lemmas r_zero = add.r_one
lemmas minus_unique = add.inv_unique

end

sublocale abelian_monoid <
  add: comm_monoid "(add_monoid G)"
  rewrites "carrier (add_monoid G) = carrier G"
       and "mult    (add_monoid G) = add G"
       and "one     (add_monoid G) = zero G"
       and "finprod (add_monoid G) = finsum G"
       and "pow     (add_monoid G) = (λa k. add_pow G k a)"
  by (rule a_comm_monoid) (auto simp: finsum_def add_pow_def)

context abelian_monoid begin

lemmas a_comm = add.m_comm
lemmas a_lcomm = add.m_lcomm
lemmas a_ac = a_assoc a_comm a_lcomm

lemmas finsum_empty = add.finprod_empty
lemmas finsum_insert = add.finprod_insert
lemmas finsum_zero = add.finprod_one
lemmas finsum_closed = add.finprod_closed
lemmas finsum_Un_Int = add.finprod_Un_Int
lemmas finsum_Un_disjoint = add.finprod_Un_disjoint
lemmas finsum_addf = add.finprod_multf
lemmas finsum_cong' = add.finprod_cong'
lemmas finsum_0 = add.finprod_0
lemmas finsum_Suc = add.finprod_Suc
lemmas finsum_Suc2 = add.finprod_Suc2
lemmas finsum_infinite = add.finprod_infinite

lemmas finsum_cong = add.finprod_cong
```

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

```
lemmas finsum_reindex = add.finprod_reindex
```

```
lemmas finsum_singleton = add.finprod_singleton

end

sublocale abelian_group <
        add: group "(add_monoid G)"
  rewrites "carrier (add_monoid G) = carrier G"
       and "mult    (add_monoid G) = add G"
       and "one     (add_monoid G) = zero G"
       and "m_inv   (add_monoid G) = a_inv G"
       and "pow     (add_monoid G) = (λa k. add_pow G k a)"
  by (rule a_group) (auto simp: m_inv_def a_inv_def add_pow_def)

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"
  by (simp add: a_minus_def)

lemmas l_neg = add.l_inv [simp del]
lemmas r_neg = add.r_inv [simp del]
lemmas minus_minus = add.inv_inv
lemmas a_inv_inj = add.inv_inj
lemmas minus_equality = add.inv_equality

end

sublocale abelian_group <
   add: comm_group "(add_monoid G)"
  rewrites "carrier (add_monoid G) = carrier G"
       and "mult    (add_monoid G) = add G"
       and "one     (add_monoid G) = zero G"
       and "m_inv   (add_monoid G) = a_inv G"
       and "finprod (add_monoid G) = finsum G"
       and "pow     (add_monoid G) = (λa k. add_pow G k a)"
  by (rule a_comm_group) (auto simp: m_inv_def a_inv_def finsum_def add_pow_def)

lemmas (in abelian_group) minus_add = add.inv_mult
```

Derive an abelian_group from a comm_group

```
lemma comm_group_abelian_groupI:
  fixes G (structure)
  assumes cg: "comm_group (add_monoid G)"
  shows "abelian_group G"
proof -
  interpret comm_group "(add_monoid G)"
```

```
    by (rule cg)
  show "abelian_group G" ..
qed
```

## 11.3 Rings: Basic Definitions

```
locale semiring = abelian_monoid R + monoid R for R (structure) +
  assumes l_distr: "⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
      and r_distr: "⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
z ⊗ (x ⊕ y) = z ⊗ x ⊕ z ⊗ y"
      and l_null[simp]: "x ∈ carrier R ⟹ 0 ⊗ x = 0"
      and r_null[simp]: "x ∈ carrier R ⟹ x ⊗ 0 = 0"

locale ring = abelian_group R + monoid R for R (structure) +
  assumes "⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹ (x ⊕ y)
⊗ z = x ⊗ z ⊕ y ⊗ z"
      and "⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹ z ⊗ (x
⊕ y) = z ⊗ x ⊕ z ⊗ y"

locale cring = ring + comm_monoid R

locale "domain" = cring +
  assumes one_not_zero [simp]: "1 ≠ 0"
      and integral: "⟦ a ⊗ b = 0; a ∈ carrier R; b ∈ carrier R ⟧ ⟹
a = 0 ∨ b = 0"

locale field = "domain" +
  assumes field_Units: "Units R = carrier R - {0}"
```

## 11.4 Rings

```
lemma ringI:
  fixes R (structure)
  assumes "abelian_group R"
      and "monoid R"
      and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
      and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
z ⊗ (x ⊕ y) = z ⊗ x ⊕ z ⊗ y"
  shows "ring R"
  by (auto intro: ring.intro
    abelian_group.axioms ring_axioms.intro assms)

lemma ringE:
  fixes R (structure)
  assumes "ring R"
  shows "abelian_group R"
    and "monoid R"
```

```
    and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
    and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
z ⊗ (x ⊕ y) = z ⊗ x ⊕ z ⊗ y"
  using assms unfolding ring_def ring_axioms_def by auto

context ring begin

lemma is_abelian_group: "abelian_group R" ..

lemma is_monoid: "monoid R"
  by (auto intro!: monoidI m_assoc)

end

thm monoid_record_simps
lemmas ring_record_simps = monoid_record_simps ring.simps

lemma cringI:
  fixes R (structure)
  assumes abelian_group: "abelian_group R"
    and comm_monoid: "comm_monoid R"
    and l_distr: "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier
R ⟧ ⟹
                            (x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
  shows "cring R"
proof (intro cring.intro ring.intro)
  show "ring_axioms R"
    — Right-distributivity follows from left-distributivity and commutativity.
  proof (rule ring_axioms.intro)
    fix x y z
    assume R: "x ∈ carrier R" "y ∈ carrier R" "z ∈ carrier R"
    note [simp] = comm_monoid.axioms [OF comm_monoid]
      abelian_group.axioms [OF abelian_group]
      abelian_monoid.a_closed

    from R have "z ⊗ (x ⊕ y) = (x ⊕ y) ⊗ z"
      by (simp add: comm_monoid.m_comm [OF comm_monoid.intro])
    also from R have "... = x ⊗ z ⊕ y ⊗ z" by (simp add: l_distr)
    also from R have "... = z ⊗ x ⊕ z ⊗ y"
      by (simp add: comm_monoid.m_comm [OF comm_monoid.intro])
    finally show "z ⊗ (x ⊕ y) = z ⊗ x ⊕ z ⊗ y" .
  qed (rule l_distr)
qed (auto intro: cring.intro
  abelian_group.axioms comm_monoid.axioms ring_axioms.intro assms)

lemma cringE:
  fixes R (structure)
  assumes "cring R"
```

```
  shows "comm_monoid R"
    and "⋀x y z. ⟦ x ∈ carrier R; y ∈ carrier R; z ∈ carrier R ⟧ ⟹
(x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
  using assms cring_def by auto (simp add: assms cring.axioms(1) ringE(3))

lemma (in cring) is_cring:
  "cring R" by (rule cring_axioms)

lemma (in ring) minus_zero [simp]: "⊖ 0 = 0"
  by (simp add: a_inv_def)
```

### 11.4.1 Normaliser for Rings

```
lemma (in abelian_group) r_neg1:
  "⟦ x ∈ carrier G; y ∈ carrier G ⟧ ⟹ (⊖ x) ⊕ (x ⊕ y) = y"
proof -
  assume G: "x ∈ carrier G" "y ∈ carrier G"
  then have "(⊖ x ⊕ x) ⊕ y = y"
    by (simp only: l_neg l_zero)
  with G show ?thesis by (simp add: a_ac)
qed

lemma (in abelian_group) r_neg2:
  "⟦ x ∈ carrier G; y ∈ carrier G ⟧ ⟹ x ⊕ ((⊖ x) ⊕ y) = y"
proof -
  assume G: "x ∈ carrier G" "y ∈ carrier G"
  then have "(x ⊕ ⊖ x) ⊕ y = y"
    by (simp only: r_neg l_zero)
  with G show ?thesis
    by (simp add: a_ac)
qed

context ring begin
```

The following proofs are from Jacobson, Basic Algebra I, pp. 88–89.

```
sublocale semiring
proof -
  note [simp] = ring_axioms[unfolded ring_def ring_axioms_def]
  show "semiring R"
  proof (unfold_locales)
    fix x
    assume R: "x ∈ carrier R"
    then have "0 ⊗ x ⊕ 0 ⊗ x = (0 ⊕ 0) ⊗ x"
      by (simp del: l_zero r_zero)
    also from R have "... = 0 ⊗ x ⊕ 0" by simp
    finally have "0 ⊗ x ⊕ 0 ⊗ x = 0 ⊗ x ⊕ 0" .
    with R show "0 ⊗ x = 0" by (simp del: r_zero)
    from R have "x ⊗ 0 ⊕ x ⊗ 0 = x ⊗ (0 ⊕ 0)"
      by (simp del: l_zero r_zero)
```

```
    also from R have "... = x ⊗ 0 ⊕ 0" by simp
    finally have "x ⊗ 0 ⊕ x ⊗ 0 = x ⊗ 0 ⊕ 0" .
    with R show "x ⊗ 0 = 0" by (simp del: r_zero)
  qed auto
qed

lemma l_minus:
  "⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ (⊖ x) ⊗ y = ⊖ (x ⊗ y)"
proof -
  assume R: "x ∈ carrier R" "y ∈ carrier R"
  then have "(⊖ x) ⊗ y ⊕ x ⊗ y = (⊖ x ⊕ x) ⊗ y" by (simp add: l_distr)
  also from R have "... = 0" by (simp add: l_neg)
  finally have "(⊖ x) ⊗ y ⊕ x ⊗ y = 0" .
  with R have "(⊖ x) ⊗ y ⊕ x ⊗ y ⊕ ⊖ (x ⊗ y) = 0 ⊕ ⊖ (x ⊗ y)" by
simp
  with R show ?thesis by (simp add: a_assoc r_neg)
qed

lemma r_minus:
  "⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ x ⊗ (⊖ y) = ⊖ (x ⊗ y)"
proof -
  assume R: "x ∈ carrier R" "y ∈ carrier R"
  then have "x ⊗ (⊖ y) ⊕ x ⊗ y = x ⊗ (⊖ y ⊕ y)" by (simp add: r_distr)
  also from R have "... = 0" by (simp add: l_neg)
  finally have "x ⊗ (⊖ y) ⊕ x ⊗ y = 0" .
  with R have "x ⊗ (⊖ y) ⊕ x ⊗ y ⊕ ⊖ (x ⊗ y) = 0 ⊕ ⊖ (x ⊗ y)" by
simp
  with R show ?thesis by (simp add: a_assoc r_neg )
qed

end

lemma (in abelian_group) minus_eq: "x ⊖ y = x ⊕ (⊖ y)"
  by (rule a_minus_def)
```

Setup algebra method: compute distributive normal form in locale contexts

```
ML_file ‹ringsimp.ML›

attribute_setup algebra = ‹
  Scan.lift ((Args.add >> K true || Args.del >> K false) --| Args.colon
|| Scan.succeed true)
    -- Scan.lift Args.name -- Scan.repeat Args.term
    >> (fn ((b, n), ts) => if b then Ringsimp.add_struct (n, ts) else
Ringsimp.del_struct (n, ts))
› "theorems controlling algebra method"

method_setup algebra = ‹
  Scan.succeed (SIMPLE_METHOD' o Ringsimp.algebra_tac)
› "normalisation of algebraic structure"
```

```
lemmas (in semiring) semiring_simprules
  [algebra ring "zero R" "add R" "a_inv R" "a_minus R" "one R" "mult R"]
=
  a_closed zero_closed  m_closed one_closed
  a_assoc l_zero  a_comm m_assoc l_one l_distr r_zero
  a_lcomm r_distr l_null r_null

lemmas (in ring) ring_simprules
  [algebra ring "zero R" "add R" "a_inv R" "a_minus R" "one R" "mult R"]
=
  a_closed zero_closed a_inv_closed minus_closed m_closed one_closed
  a_assoc l_zero l_neg a_comm m_assoc l_one l_distr minus_eq
  r_zero r_neg r_neg2 r_neg1 minus_add minus_minus minus_zero
  a_lcomm r_distr l_null r_null l_minus r_minus

lemmas (in cring)
  [algebra del: ring "zero R" "add R" "a_inv R" "a_minus R" "one R" "mult
R"] =

  _

lemmas (in cring) cring_simprules
  [algebra add: cring "zero R" "add R" "a_inv R" "a_minus R" "one R" "mult
R"] =
  a_closed zero_closed a_inv_closed minus_closed m_closed one_closed
  a_assoc l_zero l_neg a_comm m_assoc l_one l_distr m_comm minus_eq
  r_zero r_neg r_neg2 r_neg1 minus_add minus_minus minus_zero
  a_lcomm m_lcomm r_distr l_null r_null l_minus r_minus

lemma (in semiring) nat_pow_zero:
  "(n::nat) ≠ 0 ⟹ 0 [^] n = 0"
  by (induct n) simp_all

context semiring begin

lemma one_zeroD:
  assumes onezero: "1 = 0"
  shows "carrier R = {0}"
proof (rule, rule)
  fix x
  assume xcarr: "x ∈ carrier R"
  from xcarr have "x = x ⊗ 1" by simp
  with onezero have "x = x ⊗ 0" by simp
  with xcarr have "x = 0" by simp
  then show "x ∈ {0}" by fast
qed fast

lemma one_zeroI:
  assumes carrzero: "carrier R = {0}"
```

```
  shows "1 = 0"
proof -
  from one_closed and carrzero
      show "1 = 0" by simp
qed

lemma carrier_one_zero: "(carrier R = {0}) = (1 = 0)"
  using one_zeroD by blast

lemma carrier_one_not_zero: "(carrier R ≠ {0}) = (1 ≠ 0)"
  by (simp add: carrier_one_zero)

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 -
  interpret ring R by fact
  interpret cring S by fact
  from RS show ?thesis by algebra
qed

lemma
  fixes R (structure)
  assumes "ring R"
  assumes R: "a ∈ carrier R" "b ∈ carrier R"
  shows "a ⊖ (a ⊖ b) = b"
proof -
  interpret ring R by fact
  from R show ?thesis by algebra
qed
```

### 11.4.2 Sums over Finite Sets

```
lemma (in semiring) finsum_ldistr:
  "⟦ finite A; a ∈ carrier R; f: A → carrier R ⟧ ⟹
    (⨁ i ∈ A. (f i)) ⊗ a = (⨁ i ∈ A. ((f i) ⊗ a))"
proof (induct set: finite)
  case empty then show ?case by simp
next
  case (insert x F) then show ?case by (simp add: Pi_def l_distr)
qed

lemma (in semiring) finsum_rdistr:
```

```
  "⟦ finite A; a ∈ carrier R; f: A → carrier R ⟧ ⟹
   a ⊗ (⨁ i ∈ A. (f i)) = (⨁ i ∈ A. (a ⊗ (f i)))"
proof (induct set: finite)
  case empty then show ?case by simp
next
  case (insert x F) then show ?case by (simp add: Pi_def r_distr)
qed
```

A quick detour

```
lemma add_pow_int_ge: "(k :: int) ≥ 0 ⟹ [ k ] ·R a = [ nat k ] ·R a"

  by (simp add: add_pow_def int_pow_def nat_pow_def)

lemma add_pow_int_lt: "(k :: int) < 0 ⟹ [ k ] ·R a = ⊖R ([ nat (- k)
] ·R a)"
  by (simp add: int_pow_def nat_pow_def a_inv_def add_pow_def)

corollary (in semiring) add_pow_ldistr:
  assumes "a ∈ carrier R" "b ∈ carrier R"
  shows "([(k :: nat)] · a) ⊗ b = [k] · (a ⊗ b)"
proof -
  have "([k] · a) ⊗ b = (⨁ i ∈ {..< k}. a) ⊗ b"
    using add.finprod_const[OF assms(1), of "{..<k}"] by simp
  also have " ... = (⨁ i ∈ {..< k}. (a ⊗ b))"
    using finsum_ldistr[of "{..<k}" b "λx. a"] assms by simp
  also have " ... = [k] · (a ⊗ b)"
    using add.finprod_const[of "a ⊗ b" "{..<k}"] assms by simp
  finally show ?thesis .
qed

corollary (in semiring) add_pow_rdistr:
  assumes "a ∈ carrier R" "b ∈ carrier R"
  shows "a ⊗ ([(k :: nat)] · b) = [k] · (a ⊗ b)"
proof -
  have "a ⊗ ([k] · b) = a ⊗ (⨁ i ∈ {..< k}. b)"
    using add.finprod_const[OF assms(2), of "{..<k}"] by simp
  also have " ... = (⨁ i ∈ {..< k}. (a ⊗ b))"
    using finsum_rdistr[of "{..<k}" a "λx. b"] assms by simp
  also have " ... = [k] · (a ⊗ b)"
    using add.finprod_const[of "a ⊗ b" "{..<k}"] assms by simp
  finally show ?thesis .
qed


lemma (in ring) add_pow_ldistr_int:
  assumes "a ∈ carrier R" "b ∈ carrier R"
  shows "([(k :: int)] · a) ⊗ b = [k] · (a ⊗ b)"
proof (cases "k ≥ 0")
  case True thus ?thesis
```

```
    using add_pow_int_ge[of k R] add_pow_ldistr[OF assms] by auto
next
  case False thus ?thesis
    using add_pow_int_lt[of k R a] add_pow_int_lt[of k R "a ⊗ b"]
          add_pow_ldistr[OF assms, of "nat (- k)"] assms l_minus by auto
qed

lemma (in ring) add_pow_rdistr_int:
  assumes "a ∈ carrier R" "b ∈ carrier R"
  shows "a ⊗ ([(k :: int)] · b) = [k] · (a ⊗ b)"
proof (cases "k ≥ 0")
  case True thus ?thesis
    using add_pow_int_ge[of k R] add_pow_rdistr[OF assms] by auto
next
  case False thus ?thesis
    using add_pow_int_lt[of k R b] add_pow_int_lt[of k R "a ⊗ b"]
          add_pow_rdistr[OF assms, of "nat (- k)"] assms r_minus by auto
qed
```

## 11.5 Integral Domains

```
context "domain" begin

lemma zero_not_one [simp]: "0 ≠ 1"
  by (rule not_sym) simp

lemma integral_iff:
  "⟦ a ∈ carrier R; b ∈ carrier R ⟧ ⟹ (a ⊗ b = 0) = (a = 0 ∨ b = 0)"
proof
  assume "a ∈ carrier R" "b ∈ carrier R" "a ⊗ b = 0"
  then show "a = 0 ∨ b = 0" by (simp add: integral)
next
  assume "a ∈ carrier R" "b ∈ carrier R" "a = 0 ∨ b = 0"
  then show "a ⊗ b = 0" by auto
qed

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
  assume eq: "a ⊗ b = a ⊗ c"
  with R have "a ⊗ (b ⊖ c) = 0" by algebra
  with R have "a = 0 ∨ (b ⊖ c) = 0" by (simp add: integral_iff)
  with prem and R have "b ⊖ c = 0" by auto
  with R have "b = b ⊖ (b ⊖ c)" by algebra
  also from R have "b ⊖ (b ⊖ c) = c" by algebra
  finally show "b = c" .
next
```

```
  assume "b = c" then show "a ⊗ b = a ⊗ c" by simp
qed

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 -
  from prem and R have "(a ⊗ b = a ⊗ c) = (b = c)" by (rule m_lcancel)
  with R show ?thesis by algebra
qed

end
```

## 11.6 Fields

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

```
lemma (in field) is_ring: "ring R"
  using ring_axioms .

lemma fieldE :
  fixes R (structure)
  assumes "field R"
  shows "cring R"
    and one_not_zero : "1 ≠ 0"
    and integral: "⋀a b. ⟦ a ⊗ b = 0; a ∈ carrier R; b ∈ carrier R ⟧
⟹ a = 0 ∨ b = 0"
  and field_Units: "Units R = carrier R - {0}"
  using assms unfolding field_def field_axioms_def domain_def domain_axioms_def
by simp_all

lemma (in cring) cring_fieldI:
  assumes field_Units: "Units R = carrier R - {0}"
  shows "field R"
proof
  from field_Units have "0 ∉ Units R" by fast
  moreover have "1 ∈ Units R" by fast
  ultimately show "1 ≠ 0" by force
next
  fix a b
  assume acarr: "a ∈ carrier R"
    and bcarr: "b ∈ carrier R"
    and ab: "a ⊗ b = 0"
  show "a = 0 ∨ b = 0"
  proof (cases "a = 0", simp)
    assume "a ≠ 0"
    with field_Units and acarr have aUnit: "a ∈ Units R" by fast
    from bcarr have "b = 1 ⊗ b" by algebra
```

```
    also from aUnit acarr have "... = (inv a ⊗ a) ⊗ b" by simp
    also from acarr bcarr aUnit[THEN Units_inv_closed]
    have "... = (inv a) ⊗ (a ⊗ b)" by algebra
    also from ab and acarr bcarr aUnit have "... = (inv a) ⊗ 0" by simp
    also from aUnit[THEN Units_inv_closed] have "... = 0" by algebra
    finally have "b = 0" .
    then show "a = 0 ∨ b = 0" by simp
  qed
qed (rule field_Units)
```

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 -
  have *: "carrier R - {0} ⊆ {y ∈ carrier R. ∃x∈carrier R. x ⊗ y = 1
∧ y ⊗ x = 1}"
  proof (clarsimp)
    fix x
    assume xcarr: "x ∈ carrier R" and "x ≠ 0"
    obtain y where ycarr: "y ∈ carrier R" and xy: "x ⊗ y = 1"
      using ‹x ≠ 0› invex xcarr by blast
    with ycarr and xy show "∃y∈carrier R. y ⊗ x = 1 ∧ x ⊗ y = 1"
      using m_comm xcarr by fastforce
  qed
  show ?thesis
    apply (rule cring_fieldI, simp add: Units_def)
    using *
    using group_l_invI notzero set_diff_eq by auto
qed
```

## 11.7 Morphisms

```
definition
  ring_hom :: "[('a, 'm) ring_scheme, ('b, 'n) ring_scheme] => ('a =>
'b) set"
  where "ring_hom R S =
    {h. h ∈ carrier R → carrier S ∧
      (∀x y. x ∈ carrier R ∧ y ∈ carrier R ⟶
        h (x ⊗R y) = h x ⊗S h y ∧ h (x ⊕R y) = h x ⊕S h y) ∧
      h 1R = 1S}"

lemma ring_hom_memI:
  fixes R (structure) and S (structure)
  assumes "⋀x. x ∈ carrier R ⟹ h x ∈ carrier S"
      and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ h (x ⊗ y) = h
x ⊗S h y"
```

```
      and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ h (x ⊕ y) = h
x ⊕S h y"
      and "h 1 = 1S"
  shows "h ∈ ring_hom R S"
  by (auto simp add: ring_hom_def assms Pi_def)

lemma ring_hom_memE:
  fixes R (structure) and S (structure)
  assumes "h ∈ ring_hom R S"
  shows "⋀x. x ∈ carrier R ⟹ h x ∈ carrier S"
    and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ h (x ⊗ y) = h x
⊗S h y"
    and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ h (x ⊕ y) = h x
⊕S h y"
    and "h 1 = 1S"
  using assms unfolding ring_hom_def by auto

lemma ring_hom_closed:
  "⟦ h ∈ ring_hom R S; x ∈ carrier R ⟧ ⟹ h x ∈ carrier S"
  by (auto simp add: ring_hom_def funcset_mem)

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"
    by (simp add: ring_hom_def)

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"
    by (simp add: ring_hom_def)

lemma ring_hom_one:
  fixes R (structure) and S (structure)
  shows "h ∈ ring_hom R S ⟹ h 1 = 1S"
  by (simp add: ring_hom_def)

lemma ring_hom_zero:
  fixes R (structure) and S (structure)
  assumes "h ∈ ring_hom R S" "ring R" "ring S"
  shows "h 0 = 0S"
proof -
  have "h 0 = h 0 ⊕S h 0"
    using ring_hom_add[OF assms(1), of 0 0] assms(2)
    by (simp add: ring.ring_simprules(2) ring.ring_simprules(15))
  thus ?thesis
    by (metis abelian_group.l_neg assms ring.is_abelian_group ring.ring_simprules(18)
ring.ring_simprules(2) ring_hom_closed)
```

```
qed

locale ring_hom_cring =
  R?: cring R + S?: cring S for R (structure) and S (structure) + fixes
h
  assumes homh [simp, intro]: "h ∈ ring_hom R S"
  notes hom_closed [simp, intro] = ring_hom_closed [OF homh]
    and hom_mult [simp] = ring_hom_mult [OF homh]
    and hom_add [simp] = ring_hom_add [OF homh]
    and hom_one [simp] = ring_hom_one [OF homh]

lemma (in ring_hom_cring) hom_zero [simp]: "h 0 = 0⇘S⇙"
proof -
  have "h 0 ⊕⇘S⇙ h 0 = h 0 ⊕⇘S⇙ 0⇘S⇙"
    by (simp add: hom_add [symmetric] del: hom_add)
  then show ?thesis by (simp del: S.r_zero)
qed

lemma (in ring_hom_cring) hom_a_inv [simp]:
  "x ∈ carrier R ⟹ h (⊖ x) = ⊖⇘S⇙ h x"
proof -
  assume R: "x ∈ carrier R"
  then have "h x ⊕⇘S⇙ h (⊖ x) = h x ⊕⇘S⇙ (⊖⇘S⇙ h x)"
    by (simp add: hom_add [symmetric] R.r_neg S.r_neg del: hom_add)
  with R show ?thesis by simp
qed

lemma (in ring_hom_cring) hom_finsum [simp]:
  assumes "f: A → carrier R"
  shows "h (⨁ i ∈ A. f i) = (⨁⇘S⇙ i ∈ A. (h o f) i)"
  using assms by (induct A rule: infinite_finite_induct, auto simp: Pi_def)

lemma (in ring_hom_cring) hom_finprod:
  assumes "f: A → carrier R"
  shows "h (⨂ i ∈ A. f i) = (⨂⇘S⇙ i ∈ A. (h o f) i)"
  using assms by (induct A rule: infinite_finite_induct, auto simp: Pi_def)

declare ring_hom_cring.hom_finprod [simp]

lemma id_ring_hom [simp]: "id ∈ ring_hom R R"
  by (auto intro!: ring_hom_memI)

lemma ring_hom_trans:
  "⟦ f ∈ ring_hom R S; g ∈ ring_hom S T ⟧ ⟹ g ∘ f ∈ ring_hom R T"
  by (rule ring_hom_memI) (auto simp add: ring_hom_closed ring_hom_mult
ring_hom_add ring_hom_one)
```

## 11.8 Jeremy Avigad's More_Finite_Product material

```
lemma (in cring) sum_zero_eq_neg: "x ∈ carrier R ⟹ y ∈ carrier R ⟹
x ⊕ y = 0 ⟹ x = ⊖ y"
  by (metis minus_equality)

lemma (in domain) square_eq_one:
  fixes x
  assumes [simp]: "x ∈ carrier R"
    and "x ⊗ x = 1"
  shows "x = 1 ∨ x = ⊖1"
proof -
  have "(x ⊕ 1) ⊗ (x ⊕ ⊖ 1) = x ⊗ x ⊕ ⊖ 1"
    by (simp add: ring_simprules)
  also from ‹x ⊗ x = 1› have "... = 0"
    by (simp add: ring_simprules)
  finally have "(x ⊕ 1) ⊗ (x ⊕ ⊖ 1) = 0" .
  then have "(x ⊕ 1) = 0 ∨ (x ⊕ ⊖ 1) = 0"
    by (intro integral) auto
  then show ?thesis
    by (metis add.inv_closed add.inv_solve_right assms(1) l_zero one_closed
zero_closed)
qed

lemma (in domain) inv_eq_self: "x ∈ Units R ⟹ x = inv x ⟹ x = 1
∨ x = ⊖1"
  by (metis Units_closed Units_l_inv square_eq_one)
```

The following translates theorems about groups to the facts about the units of a ring. (The list should be expanded as more things are needed.)

```
lemma (in ring) finite_ring_finite_units [intro]: "finite (carrier R)
⟹ finite (Units R)"
  by (rule finite_subset) auto

lemma (in monoid) units_of_pow:
  fixes n :: nat
  shows "x ∈ Units G ⟹ x [^]units_of G n = x [^]G n"
  apply (induct n)
  apply (auto simp add: units_group group.is_monoid
    monoid.nat_pow_0 monoid.nat_pow_Suc units_of_one units_of_mult)
  done

lemma (in cring) units_power_order_eq_one:
  "finite (Units R) ⟹ a ∈ Units R ⟹ a [^] card(Units R) = 1"
  by (metis comm_group.power_order_eq_one units_comm_group units_of_carrier
units_of_one units_of_pow)
```

## 11.9 Jeremy Avigad's More_Ring material

```
lemma (in cring) field_intro2:
```

```
  assumes "0R ≠ 1R" and un: "⋀x. x ∈ carrier R - {0R} ⟹ x ∈ Units
R"
  shows "field R"
proof unfold_locales
  show "1 ≠ 0" using assms by auto
  show "⟦a ⊗ b = 0; a ∈ carrier R;
            b ∈ carrier R⟧
           ⟹ a = 0 ∨ b = 0" for a b
    by (metis un Units_l_cancel insert_Diff_single insert_iff r_null zero_closed)
qed (use assms in ‹auto simp: Units_def›)

lemma (in monoid) inv_char:
  assumes "x ∈ carrier G" "y ∈ carrier G" "x ⊗ y = 1" "y ⊗ x = 1"
  shows "inv x = y"
  using assms inv_unique' by auto

lemma (in comm_monoid) comm_inv_char: "x ∈ carrier G ⟹ y ∈ carrier
G ⟹ x ⊗ y = 1 ⟹ inv x = y"
  by (simp add: inv_char m_comm)

lemma (in ring) inv_neg_one [simp]: "inv (⊖ 1) = ⊖ 1"
  by (simp add: inv_char local.ring_axioms ring.r_minus)

lemma (in monoid) inv_eq_imp_eq [dest!]: "inv x = inv y ⟹ x ∈ Units
G ⟹ y ∈ Units G ⟹ x = y"
  by (metis Units_inv_inv)

lemma (in ring) Units_minus_one_closed [intro]: "⊖ 1 ∈ Units R"
  by (simp add: Units_def) (metis add.l_inv_ex local.minus_minus minus_equality
one_closed r_minus r_one)

lemma (in ring) inv_eq_neg_one_eq: "x ∈ Units R ⟹ inv x = ⊖ 1 ⟷
x = ⊖ 1"
  by (metis Units_inv_inv inv_neg_one)

lemma (in monoid) inv_eq_one_eq: "x ∈ Units G ⟹ inv x = 1 ⟷ x =
1"
  by (metis Units_inv_inv inv_one)

end


theory Module
imports Ring
begin
```

# 12 Modules over an Abelian Group

## 12.1 Definitions

```
record ('a, 'b) module = "'b ring" +
  smult :: "['a, 'b] => 'b" (infixl ‹⊙ı› 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"
  by (auto intro: module.intro cring.axioms abelian_group.axioms
    module_axioms.intro assms)

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"
  apply intro_locales
             apply (rule cring.axioms ring.axioms abelian_group.axioms
comm_monoid.axioms assms)+
      apply (rule module_axioms.intro)
          apply (simp add: smult_closed)
         apply (simp add: smult_l_distr)
        apply (simp add: smult_r_distr)
       apply (simp add: smult_assoc1)
      apply (simp add: smult_one)
     apply (rule cring.axioms ring.axioms abelian_group.axioms comm_monoid.axioms
assms)+
  apply (rule algebra_axioms.intro)
  apply (simp add: smult_assoc2)
  done

lemma (in algebra) R_cring: "cring R" ..

lemma (in algebra) M_cring: "cring M" ..

lemma (in algebra) module: "module R M"
  by (auto intro: moduleI R_cring is_abelian_group smult_l_distr smult_r_distr
smult_assoc1)
```

## 12.2 Basic Properties of Modules

```
lemma (in module) smult_l_null [simp]:
 "x ∈ carrier M ==> 0 ⊙M x = 0M"
proof-
```

```
  assume M : "x ∈ carrier M"
  note facts = M smult_closed [OF R.zero_closed]
  from facts have "0 ⊙M x = (0 ⊕ 0) ⊙M x "
    using smult_l_distr by auto
  also have "... = 0 ⊙M x ⊕M 0 ⊙M x"
    using smult_l_distr[of 0 0 x] facts by auto
  finally show "0 ⊙M x = 0M" using facts
    by (metis M.add.r_cancel_one')
qed

lemma (in module) smult_r_null [simp]:
  "a ∈ carrier R ==> a ⊙M 0M = 0M"
proof -
  assume R: "a ∈ carrier R"
  note facts = R smult_closed
  from facts have "a ⊙M 0M = (a ⊙M 0M ⊕M a ⊙M 0M) ⊕M ⊖M (a ⊙M 0M)"
    by (simp add: M.add.inv_solve_right)
  also from R have "... = a ⊙M (0M ⊕M 0M) ⊕M ⊖M (a ⊙M 0M)"
    by (simp add: smult_r_distr del: M.l_zero M.r_zero)
  also from facts have "... = 0M"
    by (simp add: M.r_neg)
  finally show ?thesis .
qed

lemma (in module) smult_l_minus:
"⟦ a ∈ carrier R; x ∈ carrier M ⟧ ⟹ (⊖a) ⊙M x = ⊖M (a ⊙M x)"
proof-
  assume RM: "a ∈ carrier R" "x ∈ carrier M"
  from RM have a_smult: "a ⊙M x ∈ carrier M" by simp
  from RM have ma_smult: "⊖a ⊙M x ∈ carrier M" by simp
  note facts = RM a_smult ma_smult
  from facts have "(⊖a) ⊙M x = (⊖a ⊙M x ⊕M a ⊙M x) ⊕M ⊖M(a ⊙M x)"
    by (simp add: M.add.inv_solve_right)
  also from RM have "... = (⊖a ⊕ a) ⊙M x ⊕M ⊖M(a ⊙M x)"
    by (simp add: smult_l_distr)
  also from facts smult_l_null have "... = ⊖M(a ⊙M x)"
    by (simp add: R.l_neg)
  finally show ?thesis .
qed

lemma (in module) smult_r_minus:
  "[| a ∈ carrier R; x ∈ carrier M |] ==> a ⊙M (⊖Mx) = ⊖M (a ⊙M x)"
proof -
  assume RM: "a ∈ carrier R" "x ∈ carrier M"
  note facts = RM smult_closed
  from facts have "a ⊙M (⊖Mx) = (a ⊙M ⊖Mx ⊕M a ⊙M x) ⊕M ⊖M(a ⊙M x)"
    by (simp add: M.add.inv_solve_right)
  also from RM have "... = a ⊙M (⊖Mx ⊕M x) ⊕M ⊖M(a ⊙M x)"
    by (simp add: smult_r_distr)
```

```
  also from facts smult_l_null have "... = ⊖M(a ⊙M x)"
    by (metis M.add.inv_closed M.add.inv_solve_right M.l_neg R.zero_closed
r_null smult_assoc1)
  finally show ?thesis .
qed

lemma (in module) finsum_smult_ldistr:
  "⟦ finite A; a ∈ carrier R; f: A → carrier M ⟧ ⟹
     a ⊙M (⨁M i ∈ A. (f i)) = (⨁M i ∈ A. ( a ⊙M (f i)))"
proof (induct set: finite)
  case empty then show ?case
    by (metis M.finsum_empty M.zero_closed R.zero_closed r_null smult_assoc1
smult_l_null)
next
  case (insert x F) then show ?case
    by (simp add: Pi_def smult_r_distr)
qed
```

## 12.3 Submodules

```
locale submodule = subgroup H "add_monoid M" for H and R :: "('a, 'b)
ring_scheme" and M (structure)
+ assumes smult_closed [simp, intro]:
      "⟦a ∈ carrier R; x ∈ H⟧ ⟹ a ⊙M x ∈ H"


lemma (in module) submoduleI :
  assumes subset: "H ⊆ carrier M"
    and zero: "𝟬M ∈ H"
    and a_inv: "!!a. a ∈ H ⟹ ⊖M a ∈ H"
    and add : "⋀ a b. ⟦a ∈ H ; b ∈ H⟧ ⟹ a ⊕M b ∈ H"
    and smult_closed : "⋀ a x. ⟦a ∈ carrier R; x ∈ H⟧ ⟹ a ⊙M x ∈ H"
  shows "submodule H R M"
  apply (intro submodule.intro subgroup.intro)
  using assms unfolding submodule_axioms_def
  by (simp_all add : a_inv_def)


lemma (in module) submoduleE :
  assumes "submodule H R M"
  shows "H ⊆ carrier M"
    and "H ≠ {}"
    and "⋀a. a ∈ H ⟹ ⊖M a ∈ H"
    and "⋀a b. ⟦a ∈ carrier R; b ∈ H⟧ ⟹ a ⊙M b ∈ H"
    and "⋀ a b. ⟦a ∈ H ; b ∈ H⟧ ⟹ a ⊕M b ∈ H"
    and "⋀ x. x ∈ H ⟹ (a_inv M x) ∈ H"
  using group.subgroupE[of "add_monoid M" H, OF _ submodule.axioms(1)[OF
assms]] a_comm_group
        submodule.smult_closed[OF assms]
```

```
  unfolding comm_group_def a_inv_def
  by auto


lemma (in module) carrier_is_submodule :
"submodule (carrier M) R M"
  apply (intro  submoduleI)
  using a_comm_group group.inv_closed unfolding comm_group_def a_inv_def
group_def monoid_def
  by auto

lemma (in submodule) submodule_is_module :
  assumes "module R M"
  shows "module R (M(|carrier := H|))"
proof (auto intro! : moduleI abelian_group.intro)
  show "cring (R)" using assms unfolding module_def by auto
  show "abelian_monoid (M(|carrier := H|))"
    using comm_monoid.submonoid_is_comm_monoid[OF _ subgroup_is_submonoid]
assms
    unfolding abelian_monoid_def module_def abelian_group_def
    by auto
  thus "abelian_group_axioms (M(|carrier := H|))"
    using subgroup_is_group assms
    unfolding abelian_group_axioms_def comm_group_def abelian_monoid_def
module_def abelian_group_def
    by auto
  show "⋀z. z ∈ H ⟹ 1_R ⊙ z = z"
    using subgroup.subset[OF subgroup_axioms] module.smult_one[OF assms]
    by auto
  fix a b x y assume a : "a ∈ carrier R" and b : "b ∈ carrier R" and
x : "x ∈ H" and y : "y ∈ H"
  show "(a ⊕_R b) ⊙ x = a ⊙ x ⊕ b ⊙ x"
    using a b x module.smult_l_distr[OF assms] subgroup.subset[OF subgroup_axioms]
    by auto
  show "a ⊙ (x ⊕ y) = a ⊙ x ⊕ a ⊙ y"
    using a x y module.smult_r_distr[OF assms] subgroup.subset[OF subgroup_axioms]
    by auto
  show "a ⊗_R b ⊙ x = a ⊙ (b ⊙ x)"
    using a b x module.smult_assoc1[OF assms] subgroup.subset[OF subgroup_axioms]
    by auto
qed


lemma (in module) module_incl_imp_submodule :
  assumes "H ⊆ carrier M"
    and "module R (M(|carrier := H|))"
  shows "submodule H R M"
  apply (intro submodule.intro)
  using add.group_incl_imp_subgroup[OF assms(1)] assms module.axioms(2)[OF
```

```
assms(2)]
       module.smult_closed[OF assms(2)]
  unfolding abelian_group_def abelian_group_axioms_def comm_group_def
submodule_axioms_def
  by simp_all


end


theory AbelCoset
imports Coset Ring
begin
```

## 12.4 More Lifting from Groups to Abelian Groups

### 12.4.1 Definitions

Hiding `<+>` from `HOL.Sum_Type` until I come up with better syntax here

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

definition
  a_r_coset    :: "[_, 'a set, 'a] ⇒ 'a set"    (infixl ‹+>ı› 60)
  where "a_r_coset G = r_coset (add_monoid G)"

definition
  a_l_coset    :: "[_, 'a, 'a set] ⇒ 'a set"    (infixl ‹<+ı› 60)
  where "a_l_coset G = l_coset (add_monoid G)"

definition
  A_RCOSETS  :: "[_, 'a set] ⇒ ('a set)set"
    (‹(‹open_block notation=‹prefix a_rcosets››a'_rcosetsı _)› [81]
80)
  where "A_RCOSETS G H = RCOSETS (add_monoid G) H"

definition
  set_add  :: "[_, 'a set ,'a set] ⇒ 'a set" (infixl ‹<+>ı› 60)
  where "set_add G = set_mult (add_monoid G)"

definition
  A_SET_INV :: "[_,'a set] ⇒ 'a set"
    (‹(‹open_block notation=‹prefix a_set_inv››a'_set'_invı _)› [81]
80)
  where "A_SET_INV G H = SET_INV (add_monoid G) H"

definition
  a_r_congruent :: "[('a,'b)ring_scheme, 'a set] ⇒ ('a*'a)set"  (‹racongı›)
  where "a_r_congruent G = r_congruent (add_monoid G)"
```

```
definition
  A_FactGroup :: "[('a,'b) ring_scheme, 'a set] ⇒ ('a set) monoid" (infixl
‹A'_Mod› 65)
    — Actually defined for groups rather than monoids
  where "A_FactGroup G H = FactGroup (add_monoid G) H"

definition
  a_kernel :: "('a, 'm) ring_scheme ⇒ ('b, 'n) ring_scheme ⇒  ('a ⇒
'b) ⇒ 'a set"
    — the kernel of a homomorphism (additive)
  where "a_kernel G H h = kernel (add_monoid G) (add_monoid H) h"

locale abelian_group_hom = G?: abelian_group G + H?: abelian_group H
    for G (structure) and H (structure) +
  fixes h
  assumes a_group_hom: "group_hom (add_monoid G) (add_monoid H) h"

lemmas a_r_coset_defs =
  a_r_coset_def r_coset_def

lemma a_r_coset_def':
  fixes G (structure)
  shows "H +> a ≡ ⋃h∈H. {h ⊕ a}"
  unfolding a_r_coset_defs by simp

lemmas a_l_coset_defs =
  a_l_coset_def l_coset_def

lemma a_l_coset_def':
  fixes G (structure)
  shows "a <+ H ≡ ⋃h∈H. {a ⊕ h}"
  unfolding a_l_coset_defs by simp

lemmas A_RCOSETS_defs =
  A_RCOSETS_def RCOSETS_def

lemma A_RCOSETS_def':
  fixes G (structure)
  shows "a_rcosets H ≡ ⋃a∈carrier G. {H +> a}"
  unfolding A_RCOSETS_defs by (fold a_r_coset_def, simp)

lemmas set_add_defs =
  set_add_def set_mult_def

lemma set_add_def':
  fixes G (structure)
  shows "H <+> K ≡ ⋃h∈H. ⋃k∈K. {h ⊕ k}"
  unfolding set_add_defs by simp
```

```
lemmas A_SET_INV_defs =
  A_SET_INV_def SET_INV_def

lemma A_SET_INV_def':
  fixes G (structure)
  shows "a_set_inv H ≡ ⋃h∈H. {⊖ h}"
  unfolding A_SET_INV_defs by (fold a_inv_def)
```

### 12.4.2 Cosets

```
sublocale abelian_group <
        add: group "(add_monoid G)"
  rewrites "carrier (add_monoid G) =   carrier G"
       and "   mult (add_monoid G) =       add G"
       and "    one (add_monoid G) =      zero G"
       and "  m_inv (add_monoid G) =     a_inv G"
       and "finprod (add_monoid G) =    finsum G"
       and "r_coset (add_monoid G) = a_r_coset G"
       and "l_coset (add_monoid G) = a_l_coset G"
       and "(λa k. pow (add_monoid G) a k) = (λa k. add_pow G k a)"
  by (rule a_group)
     (auto simp: m_inv_def a_inv_def finsum_def a_r_coset_def a_l_coset_def
add_pow_def)

context abelian_group
begin

thm add.coset_mult_assoc
lemmas a_repr_independence' = add.repr_independence



end

lemma (in abelian_group) a_coset_add_assoc:
     "[| M ⊆ carrier G; g ∈ carrier G; h ∈ carrier G |]
      ==> (M +> g) +> h = M +> (g ⊕ h)"
by (rule group.coset_mult_assoc [OF a_group,
    folded a_r_coset_def, simplified monoid_record_simps])

thm abelian_group.a_coset_add_assoc

lemma (in abelian_group) a_coset_add_zero [simp]:
  "M ⊆ carrier G ==> M +> 0 = M"
by (rule group.coset_mult_one [OF a_group,
    folded a_r_coset_def, simplified monoid_record_simps])

lemma (in abelian_group) a_coset_add_inv1:
     "[| M +> (x ⊕ (⊖ y)) = M;  x ∈ carrier G ; y ∈ carrier G;
```

```
        M ⊆ carrier G |] ==> M +> x = M +> y"
by (rule group.coset_mult_inv1 [OF a_group,
    folded a_r_coset_def a_inv_def, simplified monoid_record_simps])

lemma (in abelian_group) a_coset_add_inv2:
     "[| M +> x = M +> y;  x ∈ carrier G;  y ∈ carrier G;  M ⊆ carrier
G |]
      ==> M +> (x ⊕ (⊖ y)) = M"
by (rule group.coset_mult_inv2 [OF a_group,
    folded a_r_coset_def a_inv_def, simplified monoid_record_simps])

lemma (in abelian_group) a_coset_join1:
     "[| H +> x = H;  x ∈ carrier G;  subgroup H (add_monoid G) |] ==>
x ∈ H"
by (rule group.coset_join1 [OF a_group,
    folded a_r_coset_def, simplified monoid_record_simps])

lemma (in abelian_group) a_solve_equation:
    "⟦subgroup H (add_monoid G); x ∈ H; y ∈ H⟧ ⟹ ∃h∈H. y = h ⊕ x"
by (rule group.solve_equation [OF a_group,
    folded a_r_coset_def, simplified monoid_record_simps])

lemma (in abelian_group) a_repr_independence:
  "⟦ y ∈ H +> x; x ∈ carrier G; subgroup H (add_monoid G) ⟧ ⟹
     H +> x = H +> y"
  using a_repr_independence' by (simp add: a_r_coset_def)

lemma (in abelian_group) a_coset_join2:
     "⟦x ∈ carrier G;  subgroup H (add_monoid G); x∈H⟧ ⟹ H +> x = H"
by (rule group.coset_join2 [OF a_group,
    folded a_r_coset_def, simplified monoid_record_simps])

lemma (in abelian_monoid) a_r_coset_subset_G:
     "[| H ⊆ carrier G; x ∈ carrier G |] ==> H +> x ⊆ carrier G"
by (rule monoid.r_coset_subset_G [OF a_monoid,
    folded a_r_coset_def, simplified monoid_record_simps])

lemma (in abelian_group) a_rcosI:
     "[| h ∈ H; H ⊆ carrier G; x ∈ carrier G|] ==> h ⊕ x ∈ H +> x"
by (rule group.rcosI [OF a_group,
    folded a_r_coset_def, simplified monoid_record_simps])

lemma (in abelian_group) a_rcosetsI:
     "⟦H ⊆ carrier G; x ∈ carrier G⟧ ⟹ H +> x ∈ a_rcosets H"
by (rule group.rcosetsI [OF a_group,
    folded a_r_coset_def A_RCOSETS_def, simplified monoid_record_simps])
```

Really needed?

```
lemma (in abelian_group) a_transpose_inv:
```

```
    "[| x ⊕ y = z;  x ∈ carrier G;  y ∈ carrier G;  z ∈ carrier G |]
     ==> (⊖ x) ⊕ z = y"
  using r_neg1 by blast
```

### 12.4.3 Subgroups

```
locale additive_subgroup =
  fixes H and G (structure)
  assumes a_subgroup: "subgroup H (add_monoid G)"

lemma (in additive_subgroup) is_additive_subgroup:
  shows "additive_subgroup H G"
by (rule additive_subgroup_axioms)

lemma additive_subgroupI:
  fixes G (structure)
  assumes a_subgroup: "subgroup H (add_monoid G)"
  shows "additive_subgroup H G"
by (rule additive_subgroup.intro) (rule a_subgroup)

lemma (in additive_subgroup) a_subset:
     "H ⊆ carrier G"
by (rule subgroup.subset[OF a_subgroup,
    simplified monoid_record_simps])

lemma (in additive_subgroup) a_closed [intro, simp]:
     "⟦x ∈ H; y ∈ H⟧ ⟹ x ⊕ y ∈ H"
by (rule subgroup.m_closed[OF a_subgroup,
    simplified monoid_record_simps])

lemma (in additive_subgroup) zero_closed [simp]:
     "𝟬 ∈ H"
by (rule subgroup.one_closed[OF a_subgroup,
    simplified monoid_record_simps])

lemma (in additive_subgroup) a_inv_closed [intro,simp]:
     "x ∈ H ⟹ ⊖ x ∈ H"
by (rule subgroup.m_inv_closed[OF a_subgroup,
    folded a_inv_def, simplified monoid_record_simps])
```

### 12.4.4 Additive subgroups are normal

Every subgroup of an abelian_group is normal

```
locale abelian_subgroup = additive_subgroup + abelian_group G +
  assumes a_normal: "normal H (add_monoid G)"

lemma (in abelian_subgroup) is_abelian_subgroup:
  shows "abelian_subgroup H G"
by (rule abelian_subgroup_axioms)
```

```
lemma abelian_subgroupI:
  assumes a_normal: "normal H (add_monoid G)"
      and a_comm: "!!x y. [| x ∈ carrier G; y ∈ carrier G |] ==> x ⊕G
y = y ⊕G x"
  shows "abelian_subgroup H G"
proof -
  interpret normal "H" "(add_monoid G)"
    by (rule a_normal)

  show "abelian_subgroup H G"
    by standard (simp add: a_comm)
qed

lemma abelian_subgroupI2:
  fixes G (structure)
  assumes a_comm_group: "comm_group (add_monoid G)"
      and a_subgroup: "subgroup H (add_monoid G)"
  shows "abelian_subgroup H G"
proof -
  interpret comm_group "(add_monoid G)"
    by (rule a_comm_group)
  interpret subgroup "H" "(add_monoid G)"
    by (rule a_subgroup)
  have "(⋃xa∈H. {xa ⊕ x}) = (⋃xa∈H. {x ⊕ xa})" if "x ∈ carrier G" for
x
  proof -
    have "H ⊆ carrier G"
      using a_subgroup that unfolding subgroup_def by simp
    with that show "(⋃h∈H. {h ⊕G x}) = (⋃h∈H. {x ⊕G h})"
      using m_comm [simplified] by fastforce
  qed
  then show "abelian_subgroup H G"
    by unfold_locales (auto simp: r_coset_def l_coset_def)
qed

lemma abelian_subgroupI3:
  fixes G (structure)
  assumes "additive_subgroup H G"
    and "abelian_group G"
  shows "abelian_subgroup H G"
  using assms abelian_subgroupI2 abelian_group.a_comm_group additive_subgroup_def
by blast

lemma (in abelian_subgroup) a_coset_eq:
     "(∀x ∈ carrier G. H +> x = x <+ H)"
by (rule normal.coset_eq[OF a_normal,
    folded a_r_coset_def a_l_coset_def, simplified monoid_record_simps])
```

```
lemma (in abelian_subgroup) a_inv_op_closed1:
  shows "⟦x ∈ carrier G; h ∈ H⟧ ⟹ (⊖ x) ⊕ h ⊕ x ∈ H"
by (rule normal.inv_op_closed1 [OF a_normal,
    folded a_inv_def, simplified monoid_record_simps])

lemma (in abelian_subgroup) a_inv_op_closed2:
  shows "⟦x ∈ carrier G; h ∈ H⟧ ⟹ x ⊕ h ⊕ (⊖ x) ∈ H"
by (rule normal.inv_op_closed2 [OF a_normal,
    folded a_inv_def, simplified monoid_record_simps])

lemma (in abelian_group) a_lcos_m_assoc:
  "⟦ M ⊆ carrier G; g ∈ carrier G; h ∈ carrier G ⟧ ⟹ g <+ (h <+ M) =
(g ⊕ h) <+ M"
by (rule group.lcos_m_assoc [OF a_group,
    folded a_l_coset_def, simplified monoid_record_simps])

lemma (in abelian_group) a_lcos_mult_one:
     "M ⊆ carrier G ==> 0 <+ M = M"
by (rule group.lcos_mult_one [OF a_group,
    folded a_l_coset_def, simplified monoid_record_simps])

lemma (in abelian_group) a_l_coset_subset_G:
  "⟦ H ⊆ carrier G; x ∈ carrier G ⟧ ⟹ x <+ H ⊆ carrier G"
by (rule group.l_coset_subset_G [OF a_group,
    folded a_l_coset_def, simplified monoid_record_simps])

lemma (in abelian_group) a_l_coset_swap:
     "⟦y ∈ x <+ H;  x ∈ carrier G;  subgroup H (add_monoid G)⟧ ⟹ x ∈
y <+ H"
by (rule group.l_coset_swap [OF a_group,
    folded a_l_coset_def, simplified monoid_record_simps])

lemma (in abelian_group) a_l_coset_carrier:
     "[| y ∈ x <+ H;  x ∈ carrier G;  subgroup H (add_monoid G) |] ==>
y ∈ carrier G"
by (rule group.l_coset_carrier [OF a_group,
    folded a_l_coset_def, simplified monoid_record_simps])

lemma (in abelian_group) a_l_repr_imp_subset:
  assumes "y ∈ x <+ H" "x ∈ carrier G" "subgroup H (add_monoid G)"
  shows "y <+ H ⊆ x <+ H"
  by (metis (full_types) a_l_coset_defs(1) add.l_repr_independence assms
set_eq_subset)

lemma (in abelian_group) a_l_repr_independence:
  assumes y: "y ∈ x <+ H" and x: "x ∈ carrier G" and sb: "subgroup H
(add_monoid G)"
  shows "x <+ H = y <+ H"
apply (rule group.l_repr_independence [OF a_group,
```

```
    folded a_l_coset_def, simplified monoid_record_simps])
apply (rule y)
apply (rule x)
apply (rule sb)
done

lemma (in abelian_group) setadd_subset_G:
     "⟦H ⊆ carrier G; K ⊆ carrier G⟧ ⟹ H <+> K ⊆ carrier G"
by (rule group.setmult_subset_G [OF a_group,
    folded set_add_def, simplified monoid_record_simps])

lemma (in abelian_group) subgroup_add_id: "subgroup H (add_monoid G)
⟹ H <+> H = H"
by (rule group.subgroup_mult_id [OF a_group,
    folded set_add_def, simplified monoid_record_simps])

lemma (in abelian_subgroup) a_rcos_inv:
  assumes x:     "x ∈ carrier G"
  shows "a_set_inv (H +> x) = H +> (⊖ x)"
by (rule normal.rcos_inv [OF a_normal,
  folded a_r_coset_def a_inv_def A_SET_INV_def, simplified monoid_record_simps])
(rule x)

lemma (in abelian_group) a_setmult_rcos_assoc:
     "⟦H ⊆ carrier G; K ⊆ carrier G; x ∈ carrier G⟧
      ⟹ H <+> (K +> x) = (H <+> K) +> x"
by (rule group.setmult_rcos_assoc [OF a_group,
    folded set_add_def a_r_coset_def, simplified monoid_record_simps])

lemma (in abelian_group) a_rcos_assoc_lcos:
     "⟦H ⊆ carrier G; K ⊆ carrier G; x ∈ carrier G⟧
      ⟹ (H +> x) <+> K = H <+> (x <+ K)"
by (rule group.rcos_assoc_lcos [OF a_group,
     folded set_add_def a_r_coset_def a_l_coset_def, simplified monoid_record_simps])

lemma (in abelian_subgroup) a_rcos_sum:
     "⟦x ∈ carrier G; y ∈ carrier G⟧
      ⟹ (H +> x) <+> (H +> y) = H +> (x ⊕ y)"
by (rule normal.rcos_sum [OF a_normal,
    folded set_add_def a_r_coset_def, simplified monoid_record_simps])

lemma (in abelian_subgroup) rcosets_add_eq:
  "M ∈ a_rcosets H ⟹ H <+> M = M"
  — generalizes subgroup_mult_id
by (rule normal.rcosets_mult_eq [OF a_normal,
    folded set_add_def A_RCOSETS_def, simplified monoid_record_simps])
```

### 12.4.5 Congruence Relation

```
lemma (in abelian_subgroup) a_equiv_rcong:
   shows "equiv (carrier G) (racong H)"
by (rule subgroup.equiv_rcong [OF a_subgroup a_group,
    folded a_r_congruent_def, simplified monoid_record_simps])

lemma (in abelian_subgroup) a_l_coset_eq_rcong:
  assumes a: "a ∈ carrier G"
  shows "a <+ H = racong H `` {a}"
by (rule subgroup.l_coset_eq_rcong [OF a_subgroup a_group,
    folded a_r_congruent_def a_l_coset_def, simplified monoid_record_simps])
(rule a)

lemma (in abelian_subgroup) a_rcos_equation:
  shows
     "⟦ha ⊕ a = h ⊕ b; a ∈ carrier G;  b ∈ carrier G;
        h ∈ H;  ha ∈ H;  hb ∈ H⟧
      ⟹ hb ⊕ a ∈ (⋃h∈H. {h ⊕ b})"
by (rule group.rcos_equation [OF a_group a_subgroup,
    folded a_r_congruent_def a_l_coset_def, simplified monoid_record_simps])

lemma (in abelian_subgroup) a_rcos_disjoint: "pairwise disjnt (a_rcosets
H)"
by (rule group.rcos_disjoint [OF a_group a_subgroup,
    folded A_RCOSETS_def, simplified monoid_record_simps])

lemma (in abelian_subgroup) a_rcos_self:
  shows "x ∈ carrier G ⟹ x ∈ H +> x"
by (rule group.rcos_self [OF a_group _ a_subgroup,
    folded a_r_coset_def, simplified monoid_record_simps])

lemma (in abelian_subgroup) a_rcosets_part_G:
  shows "⋃(a_rcosets H) = carrier G"
by (rule group.rcosets_part_G [OF a_group a_subgroup,
    folded A_RCOSETS_def, simplified monoid_record_simps])

lemma (in abelian_subgroup) a_cosets_finite:
     "⟦c ∈ a_rcosets H;  H ⊆ carrier G;  finite (carrier G)⟧ ⟹ finite
c"
by (rule group.cosets_finite [OF a_group,
    folded A_RCOSETS_def, simplified monoid_record_simps])

lemma (in abelian_group) a_card_cosets_equal:
     "⟦c ∈ a_rcosets H;  H ⊆ carrier G; finite(carrier G)⟧
      ⟹ card c = card H"
  by (simp add: A_RCOSETS_defs(1) add.card_rcosets_equal)

lemma (in abelian_group) rcosets_subset_PowG:
     "additive_subgroup H G  ⟹ a_rcosets H ⊆ Pow(carrier G)"
```

```
by (rule group.rcosets_subset_PowG [OF a_group,
    folded A_RCOSETS_def, simplified monoid_record_simps],
    rule additive_subgroup.a_subgroup)
```

**theorem** (**in** abelian_group) a_lagrange:

```
     "⟦finite(carrier G); additive_subgroup H G⟧
      ⟹ card(a_rcosets H) * card(H) = order(G)"
by (rule group.lagrange [OF a_group,
    folded A_RCOSETS_def, simplified monoid_record_simps order_def, folded
order_def])
    (fast intro!: additive_subgroup.a_subgroup)+
```

### 12.4.6 Factorization

**lemmas** A_FactGroup_defs = A_FactGroup_def FactGroup_def

**lemma** A_FactGroup_def':
  **fixes** G (**structure**)
  **shows** "G A_Mod H ≡ ⦇carrier = a_rcosets$_G$ H, mult = set_add G, one =
H⦈"
**unfolding** A_FactGroup_defs
**by** (fold A_RCOSETS_def set_add_def)

**lemma** (**in** abelian_subgroup) a_setmult_closed:

```
     "⟦K1 ∈ a_rcosets H; K2 ∈ a_rcosets H⟧ ⟹ K1 <+> K2 ∈ a_rcosets H"
by (rule normal.setmult_closed [OF a_normal,
    folded A_RCOSETS_def set_add_def, simplified monoid_record_simps])
```

**lemma** (**in** abelian_subgroup) a_setinv_closed:

```
     "K ∈ a_rcosets H ⟹ a_set_inv K ∈ a_rcosets H"
by (rule normal.setinv_closed [OF a_normal,
    folded A_RCOSETS_def A_SET_INV_def, simplified monoid_record_simps])
```

**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)"
by (rule normal.rcosets_assoc [OF a_normal,
    folded A_RCOSETS_def set_add_def, simplified monoid_record_simps])
```

**lemma** (**in** abelian_subgroup) a_subgroup_in_rcosets:

```
     "H ∈ a_rcosets H"
by (rule subgroup.subgroup_in_rcosets [OF a_subgroup a_group,
    folded A_RCOSETS_def, simplified monoid_record_simps])
```

**lemma** (**in** abelian_subgroup) a_rcosets_inv_mult_group_eq:

```
     "M ∈ a_rcosets H ⟹ a_set_inv M <+> M = H"
by (rule normal.rcosets_inv_mult_group_eq [OF a_normal,
    folded A_RCOSETS_def A_SET_INV_def set_add_def, simplified monoid_record_simps])
```

```
theorem (in abelian_subgroup) a_factorgroup_is_group:
  "group (G A_Mod H)"
by (rule normal.factorgroup_is_group [OF a_normal,
    folded A_FactGroup_def, simplified monoid_record_simps])
```

Since the Factorization is based on an *abelian* subgroup, is results in a commutative group

```
theorem (in abelian_subgroup) a_factorgroup_is_comm_group: "comm_group
(G A_Mod H)"
proof -
  have "Group.comm_monoid_axioms (G A_Mod H)"
    apply (rule comm_monoid_axioms.intro)
    apply (auto simp: A_FactGroup_def FactGroup_def RCOSETS_def a_normal
add.m_comm normal.rcos_sum)
    done
  then show ?thesis
    by (intro comm_group.intro comm_monoid.intro) (simp_all add: a_factorgroup_is_group
group.is_monoid)
qed

lemma add_A_FactGroup [simp]: "X ⊗(G A_Mod H) X' = X <+>G X'"
by (simp add: A_FactGroup_def set_add_def)

lemma (in abelian_subgroup) a_inv_FactGroup:
     "X ∈ carrier (G A_Mod H) ⟹ invG A_Mod H X = a_set_inv X"
by (rule normal.inv_FactGroup [OF a_normal,
    folded A_FactGroup_def A_SET_INV_def, simplified monoid_record_simps])
```

The coset map is a homomorphism from `G` to the quotient group `G Mod H`

```
lemma (in abelian_subgroup) a_r_coset_hom_A_Mod:
  "(λa. H +> a) ∈ hom (add_monoid G) (G A_Mod H)"
by (rule normal.r_coset_hom_Mod [OF a_normal,
    folded A_FactGroup_def a_r_coset_def, simplified monoid_record_simps])
```

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

### 12.4.7 The First Isomorphism Theorem

The quotient by the kernel of a homomorphism is isomorphic to the range of that homomorphism.

```
lemmas a_kernel_defs =
  a_kernel_def kernel_def

lemma a_kernel_def':
  "a_kernel R S h = {x ∈ carrier R. h x = 0S}"
by (rule a_kernel_def[unfolded kernel_def, simplified ring_record_simps])
```

### 12.4.8 Homomorphisms

```
lemma abelian_group_homI:
  assumes "abelian_group G"
  assumes "abelian_group H"
  assumes a_group_hom: "group_hom (add_monoid G)
                                  (add_monoid H) h"
  shows "abelian_group_hom G H h"
proof -
  interpret G: abelian_group G by fact
  interpret H: abelian_group H by fact
  show ?thesis
    by (intro abelian_group_hom.intro abelian_group_hom_axioms.intro
        G.abelian_group_axioms H.abelian_group_axioms a_group_hom)
qed

lemma (in abelian_group_hom) is_abelian_group_hom:
  "abelian_group_hom G H h"
  ..

lemma (in abelian_group_hom) hom_add [simp]:
  "[| x ∈ carrier G; y ∈ carrier G |]
        ==> h (x ⊕G y) = h x ⊕H h y"
by (rule group_hom.hom_mult[OF a_group_hom,
    simplified ring_record_simps])

lemma (in abelian_group_hom) hom_closed [simp]:
  "x ∈ carrier G ⟹ h x ∈ carrier H"
by (rule group_hom.hom_closed[OF a_group_hom,
    simplified ring_record_simps])

lemma (in abelian_group_hom) zero_closed [simp]:
  "h 0 ∈ carrier H"
  by simp

lemma (in abelian_group_hom) hom_zero [simp]:
  "h 0 = 0H"
by (rule group_hom.hom_one[OF a_group_hom,
    simplified ring_record_simps])

lemma (in abelian_group_hom) a_inv_closed [simp]:
  "x ∈ carrier G ==> h (⊖x) ∈ carrier H"
  by simp

lemma (in abelian_group_hom) hom_a_inv [simp]:
  "x ∈ carrier G ==> h (⊖x) = ⊖H (h x)"
by (rule group_hom.hom_inv[OF a_group_hom,
    folded a_inv_def, simplified ring_record_simps])

lemma (in abelian_group_hom) additive_subgroup_a_kernel:
```

```
  "additive_subgroup (a_kernel G H h) G"
  by (simp add: additive_subgroup.intro a_group_hom a_kernel_def group_hom.subgroup_kernel)
```

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"
  apply (rule abelian_subgroupI)
   apply (simp add: G.abelian_group_axioms abelian_subgroup.a_normal abelian_subgroupI3
additive_subgroup_a_kernel)
  apply (simp add: G.a_comm)
  done

lemma (in abelian_group_hom) A_FactGroup_nonempty:
  assumes X: "X ∈ carrier (G A_Mod a_kernel G H h)"
  shows "X ≠ {}"
by (rule group_hom.FactGroup_nonempty[OF a_group_hom,
    folded a_kernel_def A_FactGroup_def, simplified ring_record_simps])
(rule X)

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"
by (rule group_hom.FactGroup_the_elem_mem[OF a_group_hom,
    folded a_kernel_def A_FactGroup_def, simplified ring_record_simps])
(rule X)

lemma (in abelian_group_hom) A_FactGroup_hom:
     "(λX. the_elem (h`X)) ∈ hom (G A_Mod (a_kernel G H h))
          (add_monoid H)"
by (rule group_hom.FactGroup_hom[OF a_group_hom,
    folded a_kernel_def A_FactGroup_def, simplified ring_record_simps])

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))"
by (rule group_hom.FactGroup_inj_on[OF a_group_hom,
    folded a_kernel_def A_FactGroup_def, simplified ring_record_simps])
```

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"
by (rule group_hom.FactGroup_onto[OF a_group_hom,
    folded a_kernel_def A_FactGroup_def, simplified ring_record_simps])
(rule h)
```

If `h` is a homomorphism from `G` onto `H`, then the quotient group `G Mod kernel G H h` is isomorphic to `H`.

```
theorem (in abelian_group_hom) A_FactGroup_iso_set:
  "h ‘ carrier G = carrier H
   ⟹ (λX. the_elem (h‘X)) ∈ iso (G A_Mod (a_kernel G H h)) (add_monoid
H)"
by (rule group_hom.FactGroup_iso_set[OF a_group_hom,
    folded a_kernel_def A_FactGroup_def, simplified ring_record_simps])

corollary (in abelian_group_hom) A_FactGroup_iso :
  "h ‘ carrier G = carrier H
   ⟹  (G A_Mod (a_kernel G H h)) ≅  (add_monoid H)"
  using A_FactGroup_iso_set unfolding is_iso_def by auto
```

### 12.4.9 Cosets

Not eveything from CosetExt.thy is lifted here.

```
lemma (in additive_subgroup) a_Hcarr [simp]:
  assumes hH: "h ∈ H"
  shows "h ∈ carrier G"
by (rule subgroup.mem_carrier [OF a_subgroup,
    simplified monoid_record_simps]) (rule hH)


lemma (in abelian_subgroup) a_elemrcos_carrier:
  assumes acarr: "a ∈ carrier G"
      and a’: "a’ ∈ H +> a"
  shows "a’ ∈ carrier G"
by (rule subgroup.elemrcos_carrier [OF a_subgroup a_group,
    folded a_r_coset_def, simplified monoid_record_simps]) (rule acarr,
rule a’)

lemma (in abelian_subgroup) a_rcos_const:
  assumes hH: "h ∈ H"
  shows "H +> h = H"
by (rule subgroup.rcos_const [OF a_subgroup a_group,
    folded a_r_coset_def, simplified monoid_record_simps]) (rule hH)

lemma (in abelian_subgroup) a_rcos_module_imp:
  assumes xcarr: "x ∈ carrier G"
      and x’cos: "x’ ∈ H +> x"
  shows "(x’ ⊕ ⊖x) ∈ H"
by (rule subgroup.rcos_module_imp [OF a_subgroup a_group,
    folded a_r_coset_def a_inv_def, simplified monoid_record_simps]) (rule
xcarr, rule x’cos)

lemma (in abelian_subgroup) a_rcos_module_rev:
  assumes "x ∈ carrier G" "x’ ∈ carrier G"
      and "(x’ ⊕ ⊖x) ∈ H"
  shows "x’ ∈ H +> x"
using assms
```

```
by (rule subgroup.rcos_module_rev [OF a_subgroup a_group,
    folded a_r_coset_def a_inv_def, simplified monoid_record_simps])

lemma (in abelian_subgroup) a_rcos_module:
  assumes "x ∈ carrier G" "x' ∈ carrier G"
  shows "(x' ∈ H +> x) = (x' ⊕ ⊖x ∈ H)"
using assms
by (rule subgroup.rcos_module [OF a_subgroup a_group,
    folded a_r_coset_def a_inv_def, simplified monoid_record_simps])
```

— variant

```
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 -
  interpret G: ring G by fact
  from carr
  have "(x' ∈ H +> x) = (x' ⊕ ⊖x ∈ H)" by (rule a_rcos_module)
  with carr
  show "(x' ∈ H +> x) = (x' ⊖ x ∈ H)"
    by (simp add: minus_eq)
qed

lemma (in abelian_subgroup) a_repr_independence':
  assumes "y ∈ H +> x" "x ∈ carrier G"
  shows "H +> x = H +> y"
  using a_repr_independence a_subgroup assms by blast

lemma (in abelian_subgroup) a_repr_independenceD:
  assumes "y ∈ carrier G" "H +> x = H +> y"
  shows "y ∈ H +> x"
  by (simp add: a_rcos_self assms)


lemma (in abelian_subgroup) a_rcosets_carrier:
  "X ∈ a_rcosets H ⟹ X ⊆ carrier G"
  using a_rcosets_part_G by auto
```

### 12.4.10 Addition of Subgroups

```
lemma (in abelian_monoid) set_add_closed:
  assumes "A ⊆ carrier G" "B ⊆ carrier G"
  shows "A <+> B ⊆ carrier G"
  by (simp add: assms add.set_mult_closed set_add_defs(1))

lemma (in abelian_group) add_additive_subgroups:
  assumes subH: "additive_subgroup H G"
    and subK: "additive_subgroup K G"
```

```
  shows "additive_subgroup (H <+> K) G"
  unfolding set_add_def
  using add.mult_subgroups additive_subgroup_def subH subK
  by (blast intro: additive_subgroup.intro)

end
```

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

# 13 Ideals

## 13.1 Definitions

### 13.1.1 General definition

```
locale ideal = additive_subgroup I R + ring R for I and R (structure) +
  assumes I_l_closed: "⟦a ∈ I; x ∈ carrier R⟧ ⟹ x ⊗ a ∈ I"
      and I_r_closed: "⟦a ∈ I; x ∈ carrier R⟧ ⟹ a ⊗ x ∈ I"

sublocale ideal ⊆ abelian_subgroup I R
proof (intro abelian_subgroupI3 abelian_group.intro)
  show "additive_subgroup I R"
    by (simp add: is_additive_subgroup)
  show "abelian_monoid R"
    by (simp add: abelian_monoid_axioms)
  show "abelian_group_axioms R"
    using abelian_group_def is_abelian_group by blast
qed

lemma (in ideal) is_ideal: "ideal I R"
  by (rule ideal_axioms)

lemma idealI:
  fixes R (structure)
  assumes "ring R"
  assumes a_subgroup: "subgroup I (add_monoid R)"
    and I_l_closed: "⋀a x. ⟦a ∈ I; x ∈ carrier R⟧ ⟹ x ⊗ a ∈ I"
    and I_r_closed: "⋀a x. ⟦a ∈ I; x ∈ carrier R⟧ ⟹ a ⊗ x ∈ I"
  shows "ideal I R"
proof -
  interpret ring R by fact
  show ?thesis
    by (auto simp: ideal.intro ideal_axioms.intro additive_subgroupI a_subgroup
ring_axioms I_l_closed I_r_closed)
qed
```

### 13.1.2 Ideals Generated by a Subset of carrier R

```
definition genideal :: "_ ⇒ 'a set ⇒ 'a set"
    (‹(‹open_block notation=‹prefix Idl››Idl_ı _)› [80] 79)
  where "Idl_R S = ⋂{I. ideal I R ∧ S ⊆ I}"
```

### 13.1.3 Principal Ideals

```
locale principalideal = ideal +
  assumes generate: "∃i ∈ carrier R. I = Idl {i}"

lemma (in principalideal) is_principalideal: "principalideal I R"
  by (rule principalideal_axioms)

lemma principalidealI:
  fixes R (structure)
  assumes "ideal I R"
    and generate: "∃i ∈ carrier R. I = Idl {i}"
  shows "principalideal I R"
proof -
  interpret ideal I R by fact
  show ?thesis
    by (intro principalideal.intro principalideal_axioms.intro)
      (rule is_ideal, rule generate)
qed


lemma (in ideal) rcos_const_imp_mem:
  assumes "i ∈ carrier R" and "I +> i = I" shows "i ∈ I"
  using additive_subgroup.zero_closed[OF ideal.axioms(1)[OF ideal_axioms]]
assms
  by (force simp add: a_r_coset_def')


lemma (in ring) a_rcos_zero:
  assumes "ideal I R" "i ∈ I" shows "I +> i = I"
  using abelian_subgroupI3[OF ideal.axioms(1) is_abelian_group]
  by (simp add: abelian_subgroup.a_rcos_const assms)


lemma (in ring) ideal_is_normal:
  assumes "ideal I R" shows "I ◁ (add_monoid R)"
  using abelian_subgroup.a_normal[OF abelian_subgroupI3[OF ideal.axioms(1)]]
        abelian_group_axioms assms
  by auto
```

```
lemma (in ideal) a_rcos_sum:
  assumes "a ∈ carrier R" and "b ∈ carrier R" shows "(I +> a) <+> (I
+> b) = I +> (a ⊕ b)"
  using normal.rcos_sum[OF ideal_is_normal[OF ideal_axioms]] assms
  unfolding set_add_def a_r_coset_def by simp
```

```
lemma (in ring) set_add_comm:
  assumes "I ⊆ carrier R" "J ⊆ carrier R" shows "I <+> J = J <+> I"
proof -
  have "I <+> J ⊆ J <+> I" if "I ⊆ carrier R" "J ⊆ carrier R" for I J
    using that a_comm unfolding set_add_def' by (auto, blast)
  thus ?thesis
    using assms by auto
qed
```

### 13.1.4 Maximal Ideals

```
locale maximalideal = ideal +
  assumes I_notcarr: "carrier R ≠ I"
    and I_maximal: "⟦ideal J R; I ⊆ J; J ⊆ carrier R⟧ ⟹ (J = I) ∨ (J
= carrier R)"

lemma (in maximalideal) is_maximalideal: "maximalideal I R"
  by (rule maximalideal_axioms)

lemma maximalidealI:
  fixes R
  assumes "ideal I R"
    and I_notcarr: "carrier R ≠ I"
    and I_maximal: "⋀J. ⟦ideal J R; I ⊆ J; J ⊆ carrier R⟧ ⟹ (J = I)
∨ (J = carrier R)"
  shows "maximalideal I R"
proof -
  interpret ideal I R by fact
  show ?thesis
    by (intro maximalideal.intro maximalideal_axioms.intro)
      (rule is_ideal, rule I_notcarr, rule I_maximal)
qed
```

### 13.1.5 Prime Ideals

```
locale primeideal = ideal + cring +
  assumes I_notcarr: "carrier R ≠ I"
    and I_prime: "⟦a ∈ carrier R; b ∈ carrier R; a ⊗ b ∈ I⟧ ⟹ a ∈
I ∨ b ∈ I"

lemma (in primeideal) primeideal: "primeideal I R"
  by (rule primeideal_axioms)
```

```
lemma primeidealI:
  fixes R (structure)
  assumes "ideal I R"
    and "cring R"
    and I_notcarr: "carrier R ≠ I"
    and I_prime: "⋀a b. ⟦a ∈ carrier R; b ∈ carrier R; a ⊗ b ∈ I⟧ ⟹
a ∈ I ∨ b ∈ I"
  shows "primeideal I R"
proof -
  interpret ideal I R by fact
  interpret cring R by fact
  show ?thesis
    by (intro primeideal.intro primeideal_axioms.intro)
      (rule is_ideal, rule is_cring, rule I_notcarr, rule I_prime)
qed

lemma primeidealI2:
  fixes R (structure)
  assumes "additive_subgroup I R"
    and "cring R"
    and I_l_closed: "⋀a x. ⟦a ∈ I; x ∈ carrier R⟧ ⟹ x ⊗ a ∈ I"
    and I_r_closed: "⋀a x. ⟦a ∈ I; x ∈ carrier R⟧ ⟹ a ⊗ x ∈ I"
    and I_notcarr: "carrier R ≠ I"
    and I_prime: "⋀a b. ⟦a ∈ carrier R; b ∈ carrier R; a ⊗ b ∈ I⟧ ⟹
a ∈ I ∨ b ∈ I"
  shows "primeideal I R"
proof -
  interpret additive_subgroup I R by fact
  interpret cring R by fact
  show ?thesis apply intro_locales
    apply (intro ideal_axioms.intro)
    apply (erule (1) I_l_closed)
    apply (erule (1) I_r_closed)
    by (simp add: I_notcarr I_prime primeideal_axioms.intro)
qed
```

## 13.2 Special Ideals

```
lemma (in ring) zeroideal: "ideal {0} R"
  by (intro idealI subgroup.intro) (simp_all add: ring_axioms)

lemma (in ring) oneideal: "ideal (carrier R) R"
  by (rule idealI) (auto intro: ring_axioms add.subgroupI)

lemma (in "domain") zeroprimeideal: "primeideal {0} R"
proof -
  have "carrier R ≠ {0}"
    by (simp add: carrier_one_not_zero)
```

```
  then show ?thesis
    by (metis (no_types, lifting) domain_axioms domain_def integral primeidealI
singleton_iff zeroideal)
qed
```

## 13.3 General Ideal Properties

```
lemma (in ideal) one_imp_carrier:
  assumes I_one_closed: "1 ∈ I"
  shows "I = carrier R"
proof
  show "carrier R ⊆ I"
    using I_r_closed assms by fastforce
  show "I ⊆ carrier R"
    by (rule a_subset)
qed

lemma (in ideal) Icarr:
  assumes iI: "i ∈ I"
  shows "i ∈ carrier R"
  using iI by (rule a_Hcarr)

lemma (in ring) quotient_eq_iff_same_a_r_cos:
  assumes "ideal I R" and "a ∈ carrier R" and "b ∈ carrier R"
  shows "a ⊖ b ∈ I ⟷ I +> a = I +> b"
proof
  assume "I +> a = I +> b"
  then obtain i where "i ∈ I" and "0 ⊕ a = i ⊕ b"
    using additive_subgroup.zero_closed[OF ideal.axioms(1)[OF assms(1)]]
assms(2)
    unfolding a_r_coset_def' by blast
  hence "a ⊖ b = i"
    using assms(2-3) by (metis a_minus_def add.inv_solve_right assms(1)
ideal.Icarr l_zero)
  with ‹i ∈ I› show "a ⊖ b ∈ I"
    by simp
next
  assume "a ⊖ b ∈ I"
  then obtain i where "i ∈ I" and "a = i ⊕ b"
    using ideal.Icarr[OF assms(1)] assms(2-3)
    by (metis a_minus_def add.inv_solve_right)
  hence "I +> a = (I +> i) +> b"
    using ideal.Icarr[OF assms(1)] assms(3)
    by (simp add: a_coset_add_assoc subsetI)
  with ‹i ∈ I› show "I +> a = I +> b"
    using a_rcos_zero[OF assms(1)] by simp
qed
```

## 13.4 Intersection of Ideals

**Intersection of two ideals** The intersection of any two ideals is again an ideal in R

```
lemma (in ring) i_intersect:
  assumes "ideal I R"
  assumes "ideal J R"
  shows "ideal (I ∩ J) R"
proof -
  interpret ideal I R by fact
  interpret ideal J R by fact
  have IJ: "I ∩ J ⊆ carrier R"
    by (force simp: a_subset)
  show ?thesis
    apply (intro idealI subgroup.intro)
    apply (simp_all add: IJ ring_axioms I_l_closed assms ideal.I_l_closed
ideal.I_r_closed flip: a_inv_def)
    done
qed
```

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

```
lemma (in ring) i_Intersect:
  assumes Sideals: "⋀I. I ∈ S ⟹ ideal I R" and notempty: "S ≠ {}"
  shows "ideal (⋂S) R"
proof -
  have "x ⊕ y ∈ J" if "∀I∈S. x ∈ I" "∀I∈S. y ∈ I" and JS: "J ∈ S" for
x y J
  proof -
    interpret ideal J R by (rule Sideals[OF JS])
    show ?thesis by (simp add: JS ‹∀I∈S. x ∈ I› ‹∀I∈S. y ∈ I›)
  qed
  moreover have "𝟬 ∈ J" if "J ∈ S" for J
    by (simp add: that Sideals additive_subgroup.zero_closed ideal.axioms(1))

  moreover have "⊖ x ∈ J" if "∀I∈S. x ∈ I" and JS: "J ∈ S" for x J
  proof -
    interpret ideal J R by (rule Sideals[OF JS])
    show ?thesis by (simp add: JS ‹∀I∈S. x ∈ I›)
  qed
  moreover have "y ⊗ x ∈ J" "x ⊗ y ∈ J"
    if "∀I∈S. x ∈ I" and ycarr: "y ∈ carrier R" and JS: "J ∈ S" for
x y J
  proof -
    interpret ideal J R by (rule Sideals[OF JS])
    show "y ⊗ x ∈ J" "x ⊗ y ∈ J" using I_l_closed I_r_closed JS ‹∀I∈S.
x ∈ I› ycarr by blast+
  qed
  moreover have "x ∈ carrier R" if "∀I∈S. x ∈ I" for x
    proof -
```

```
    obtain I0 where I0S: "I0 ∈ S"
      using notempty by blast
    interpret ideal I0 R by (rule Sideals[OF I0S])
    have "x ∈ I0"
      by (simp add: I0S ‹∀I∈S. x ∈ I›)
    with a_subset show ?thesis by fast
  qed
  ultimately show ?thesis
    by unfold_locales (auto simp: Inter_eq simp flip: a_inv_def)
qed
```

## 13.5 Addition of Ideals

```
lemma (in ring) add_ideals:
  assumes idealI: "ideal I R" and idealJ: "ideal J R"
  shows "ideal (I <+> J) R"
proof (rule ideal.intro)
  show "additive_subgroup (I <+> J) R"
    by (intro ideal.axioms[OF idealI] ideal.axioms[OF idealJ] add_additive_subgroups)
  show "ring R"
    by (rule ring_axioms)
  show "ideal_axioms (I <+> J) R"
  proof -
    have "∃h∈I. ∃k∈J. (i ⊕ j) ⊗ x = h ⊕ k"
      if xcarr: "x ∈ carrier R" and iI: "i ∈ I" and jJ: "j ∈ J" for x
i j
      using xcarr ideal.Icarr[OF idealI iI] ideal.Icarr[OF idealJ jJ]
      by (meson iI ideal.I_r_closed idealJ jJ l_distr local.idealI)
    moreover have "∃h∈I. ∃k∈J. x ⊗ (i ⊕ j) = h ⊕ k"
      if xcarr: "x ∈ carrier R" and iI: "i ∈ I" and jJ: "j ∈ J" for x
i j
      using xcarr ideal.Icarr[OF idealI iI] ideal.Icarr[OF idealJ jJ]
        by (meson iI ideal.I_l_closed idealJ jJ local.idealI r_distr)
    ultimately show "ideal_axioms (I <+> J) R"
      by (intro ideal_axioms.intro) (auto simp: set_add_defs)
  qed
qed
```

## 13.6 Ideals generated by a subset of `carrier R`

`genideal` generates an ideal

```
lemma (in ring) genideal_ideal:
  assumes Scarr: "S ⊆ carrier R"
  shows "ideal (Idl S) R"
unfolding genideal_def
proof (rule i_Intersect, fast, simp)
  from oneideal and Scarr
  show "∃I. ideal I R ∧ S ≤ I" by fast
qed
```

```
lemma (in ring) genideal_self:
  assumes "S ⊆ carrier R"
  shows "S ⊆ Idl S"
  unfolding genideal_def by fast

lemma (in ring) genideal_self':
  assumes carr: "i ∈ carrier R"
  shows "i ∈ Idl {i}"
  by (simp add: genideal_def)
```

genideal generates the minimal ideal

```
lemma (in ring) genideal_minimal:
  assumes "ideal I R" "S ⊆ I"
  shows "Idl S ⊆ I"
  unfolding genideal_def by rule (elim InterD, simp add: assms)
```

Generated ideals and subsets

```
lemma (in ring) Idl_subset_ideal:
  assumes Iideal: "ideal I R"
    and Hcarr: "H ⊆ carrier R"
  shows "(Idl H ⊆ I) = (H ⊆ I)"
proof
  assume a: "Idl H ⊆ I"
  from Hcarr have "H ⊆ Idl H" by (rule genideal_self)
  with a show "H ⊆ I" by simp
next
  fix x
  assume "H ⊆ I"
  with Iideal have "I ∈ {I. ideal I R ∧ H ⊆ I}" by fast
  then show "Idl H ⊆ I" unfolding genideal_def by fast
qed

lemma (in ring) subset_Idl_subset:
  assumes Icarr: "I ⊆ carrier R"
    and HI: "H ⊆ I"
  shows "Idl H ⊆ Idl I"
proof -
  from Icarr have Iideal: "ideal (Idl I) R"
    by (rule genideal_ideal)
  from HI and Icarr have "H ⊆ carrier R"
    by fast
  with Iideal have "(H ⊆ Idl I) = (Idl H ⊆ Idl I)"
    by (rule Idl_subset_ideal[symmetric])
  then show "Idl H ⊆ Idl I"
    by (meson HI Icarr genideal_self order_trans)
qed

lemma (in ring) Idl_subset_ideal':
```

```
  assumes acarr: "a ∈ carrier R" and bcarr: "b ∈ carrier R"
  shows "Idl {a} ⊆ Idl {b} ⟷ a ∈ Idl {b}"
proof -
  have "Idl {a} ⊆ Idl {b} ⟷ {a} ⊆ Idl {b}"
    by (simp add: Idl_subset_ideal acarr bcarr genideal_ideal)
  also have "... ⟷ a ∈ Idl {b}"
    by blast
  finally show ?thesis .
qed

lemma (in ring) genideal_zero: "Idl {𝟬} = {𝟬}"
proof
  show "Idl {𝟬} ⊆ {𝟬}"
    by (simp add: genideal_minimal zeroideal)
  show "{𝟬} ⊆ Idl {𝟬}"
    by (simp add: genideal_self')
qed

lemma (in ring) genideal_one: "Idl {𝟭} = carrier R"
proof -
  interpret ideal "Idl {𝟭}" "R" by (rule genideal_ideal) fast
  show "Idl {𝟭} = carrier R"
    using genideal_self' one_imp_carrier by blast
qed
```

Generation of Principal Ideals in Commutative Rings

```
definition cgenideal :: "_ ⇒ 'a ⇒ 'a set"
    (‹(‹open_block notation=‹prefix PIdl››PIdlı _)› [80] 79)
  where "PIdl_R a = {x ⊗_R a | x. x ∈ carrier R}"
```

genhideal (?) really generates an ideal

```
lemma (in cring) cgenideal_ideal:
  assumes acarr: "a ∈ carrier R"
  shows "ideal (PIdl a) R"
  unfolding cgenideal_def
proof (intro subgroup.intro idealI[OF ring_axioms], simp_all)
  show "{x ⊗ a |x. x ∈ carrier R} ⊆ carrier R"
    by (blast intro: acarr)
  show "⋀x y. ⟦∃u. x = u ⊗ a ∧ u ∈ carrier R; ∃x. y = x ⊗ a ∧ x ∈
carrier R⟧
              ⟹ ∃v. x ⊕ y = v ⊗ a ∧ v ∈ carrier R"
    by (metis assms cring.cring_simprules(1) is_cring l_distr)
  show "∃x. 𝟬 = x ⊗ a ∧ x ∈ carrier R"
    by (metis assms l_null zero_closed)
  show "⋀x. ∃u. x = u ⊗ a ∧ u ∈ carrier R
            ⟹ ∃v. inv_add_monoid R x = v ⊗ a ∧ v ∈ carrier R"
    by (metis a_inv_def add.inv_closed assms l_minus)
  show "⋀b x. ⟦∃x. b = x ⊗ a ∧ x ∈ carrier R; x ∈ carrier R⟧
        ⟹ ∃z. x ⊗ b = z ⊗ a ∧ z ∈ carrier R"
```

```
    by (metis assms m_assoc m_closed)
  show "⋀b x. ⟦∃x. b = x ⊗ a ∧ x ∈ carrier R; x ∈ carrier R⟧
       ⟹ ∃z. b ⊗ x = z ⊗ a ∧ z ∈ carrier R"
    by (metis assms m_assoc m_comm m_closed)
qed

lemma (in ring) cgenideal_self:
  assumes icarr: "i ∈ carrier R"
  shows "i ∈ PIdl i"
  unfolding cgenideal_def
proof simp
  from icarr have "i = 𝟭 ⊗ i"
    by simp
  with icarr show "∃x. i = x ⊗ i ∧ x ∈ carrier R"
    by fast
qed
```

cgenideal is minimal

```
lemma (in ring) cgenideal_minimal:
  assumes "ideal J R"
  assumes aJ: "a ∈ J"
  shows "PIdl a ⊆ J"
proof -
  interpret ideal J R by fact
  show ?thesis
    unfolding cgenideal_def
    using I_l_closed aJ by blast
qed

lemma (in cring) cgenideal_eq_genideal:
  assumes icarr: "i ∈ carrier R"
  shows "PIdl i = Idl {i}"
proof
  show "PIdl i ⊆ Idl {i}"
    by (simp add: cgenideal_minimal genideal_ideal genideal_self' icarr)
  show "Idl {i} ⊆ PIdl i"
    by (simp add: cgenideal_ideal cgenideal_self genideal_minimal icarr)
qed

lemma (in cring) cgenideal_eq_rcos: "PIdl i = carrier R #> i"
  unfolding cgenideal_def r_coset_def by fast

lemma (in cring) cgenideal_is_principalideal:
  assumes "i ∈ carrier R"
  shows "principalideal (PIdl i) R"
proof -
  have "∃i'∈carrier R. PIdl i = Idl {i'}"
    using cgenideal_eq_genideal assms by auto
  then show ?thesis
```

```
    by (simp add: cgenideal_ideal assms principalidealI)
qed
```

## 13.7 Union of Ideals

```
lemma (in ring) union_genideal:
  assumes idealI: "ideal I R" and idealJ: "ideal J R"
  shows "Idl (I ∪ J) = I <+> J"
proof
  show "Idl (I ∪ J) ⊆ I <+> J"
  proof (rule ring.genideal_minimal [OF ring_axioms])
    show "ideal (I <+> J) R"
      by (rule add_ideals[OF idealI idealJ])
    have "⋀x. x ∈ I ⟹ ∃xa∈I. ∃xb∈J. x = xa ⊕ xb"
      by (metis additive_subgroup.zero_closed ideal.Icarr idealJ ideal_def
local.idealI r_zero)
    moreover have "⋀x. x ∈ J ⟹ ∃xa∈I. ∃xb∈J. x = xa ⊕ xb"
      by (metis additive_subgroup.zero_closed ideal.Icarr idealJ ideal_def
l_zero local.idealI)
    ultimately show "I ∪ J ⊆ I <+> J"
      by (auto simp: set_add_defs)
  qed
next
  show "I <+> J ⊆ Idl (I ∪ J)"
    by (auto simp: set_add_defs genideal_def additive_subgroup.a_closed
ideal_def subsetD)
qed
```

## 13.8 Properties of Principal Ideals

The zero ideal is a principal ideal

```
corollary (in ring) zeropideal: "principalideal {0} R"
  using genideal_zero principalidealI zeroideal by blast
```

The unit ideal is a principal ideal

```
corollary (in ring) onepideal: "principalideal (carrier R) R"
  using genideal_one oneideal principalidealI by blast
```

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 -
  interpret cring R by fact
  from generate obtain i where icarr: "i ∈ carrier R" and I1: "I = Idl
{i}"
    by fast+
  then have "I = PIdl i"
```

```
    by (simp add: cgenideal_eq_genideal)
  moreover have "i ∈ I"
    by (simp add: I1 genideal_self' icarr)
  moreover have "PIdl i = carrier R #> i"
    unfolding cgenideal_def r_coset_def by fast
  ultimately show "∃x∈I. I = carrier R #> x"
    by fast
qed
```

This next lemma would be trivial if placed in a theory that imports QuotRing, but it makes more sense to have it here (easier to find and coherent with the previous developments).

```
lemma (in cring) cgenideal_prod:
  assumes "a ∈ carrier R" "b ∈ carrier R"
  shows "(PIdl a) <#> (PIdl b) = PIdl (a ⊗ b)"
proof -
  have "(carrier R #> a) <#> (carrier R #> b) = carrier R #> (a ⊗ b)"
  proof
    show "(carrier R #> a) <#> (carrier R #> b) ⊆ carrier R #> a ⊗ b"
    proof
      fix x assume "x ∈ (carrier R #> a) <#> (carrier R #> b)"
      then obtain r1 r2 where r1: "r1 ∈ carrier R" and r2: "r2 ∈ carrier
R"
                             and "x = (r1 ⊗ a) ⊗ (r2 ⊗ b)"
        unfolding set_mult_def r_coset_def by blast
      hence "x = (r1 ⊗ r2) ⊗ (a ⊗ b)"
        by (simp add: assms local.ring_axioms m_lcomm ring.ring_simprules(11))
      thus "x ∈ carrier R #> a ⊗ b"
        unfolding r_coset_def using r1 r2 assms by blast
    qed
  next
    show "carrier R #> a ⊗ b ⊆ (carrier R #> a) <#> (carrier R #> b)"
    proof
      fix x assume "x ∈ carrier R #> a ⊗ b"
      then obtain r where r: "r ∈ carrier R" "x = r ⊗ (a ⊗ b)"
        unfolding r_coset_def by blast
      hence "x = (r ⊗ a) ⊗ (1 ⊗ b)"
        using assms by (simp add: m_assoc)
      thus "x ∈ (carrier R #> a) <#> (carrier R #> b)"
        unfolding set_mult_def r_coset_def using assms r by blast
    qed
  qed
  thus ?thesis
    using cgenideal_eq_rcos[of a] cgenideal_eq_rcos[of b] cgenideal_eq_rcos[of
"a ⊗ b"] by simp
qed
```

## 13.9 Prime Ideals

```
lemma (in ideal) primeidealCD:
  assumes "cring R"
  assumes notprime: "¬ primeideal I R"
  shows "carrier R = I ∨ (∃a b. a ∈ carrier R ∧ b ∈ carrier R ∧ a ⊗
b ∈ I ∧ a ∉ I ∧ b ∉ I)"
proof (rule ccontr, clarsimp)
  interpret cring R by fact
  assume InR: "carrier R ≠ I"
    and "∀a. a ∈ carrier R ⟶ (∀b. a ⊗ b ∈ I ⟶ b ∈ carrier R ⟶
a ∈ I ∨ b ∈ I)"
  then have I_prime: "⋀ a b. ⟦a ∈ carrier R; b ∈ carrier R; a ⊗ b ∈
I⟧ ⟹ a ∈ I ∨ b ∈ I"
    by simp
  have "primeideal I R"
    by (simp add: I_prime InR is_cring is_ideal primeidealI)
  with notprime show False by simp
qed

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 -
  interpret R: cring R by fact
  assume "carrier R = I ==> thesis"
    and "∃a b. a ∈ carrier R ∧ b ∈ carrier R ∧ a ⊗ b ∈ I ∧ a ∉ I ∧
b ∉ I ⟹ thesis"
  then show thesis using primeidealCD [OF R.is_cring notprime] by blast
qed
```

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 (intro domain.intro is_cring domain_axioms.intro)
  show "𝟭 ≠ 𝟬"
    using genideal_one genideal_zero pi primeideal.I_notcarr by force
  show "a = 𝟬 ∨ b = 𝟬" if ab: "a ⊗ b = 𝟬" and carr: "a ∈ carrier R"
"b ∈ carrier R" for a b
  proof -
    interpret primeideal "{𝟬}" "R" by (rule pi)
    show "a = 𝟬 ∨ b = 𝟬"
      using I_prime ab carr by blast
  qed
qed
```

```
corollary (in cring) domain_eq_zeroprimeideal: "domain R = primeideal {0}
R"
  using domain.zeroprimeideal zeroprimeideal_domainI by blast
```

## 13.10 Maximal Ideals

```
lemma (in ideal) helper_I_closed:
  assumes carr: "a ∈ carrier R" "x ∈ carrier R" "y ∈ carrier R"
    and axI: "a ⊗ x ∈ I"
  shows "a ⊗ (x ⊗ y) ∈ I"
proof -
  from axI and carr have "(a ⊗ x) ⊗ y ∈ I"
    by (simp add: I_r_closed)
  also from carr have "(a ⊗ x) ⊗ y = a ⊗ (x ⊗ y)"
    by (simp add: m_assoc)
  finally show "a ⊗ (x ⊗ y) ∈ I" .
qed

lemma (in ideal) helper_max_prime:
  assumes "cring R"
  assumes acarr: "a ∈ carrier R"
  shows "ideal {x∈carrier R. a ⊗ x ∈ I} R"
proof -
  interpret cring R by fact
  show ?thesis
  proof (rule idealI, simp_all)
    show "ring R"
      by (simp add: local.ring_axioms)
    show "subgroup {x ∈ carrier R. a ⊗ x ∈ I} (add_monoid R)"
      by (rule subgroup.intro) (auto simp: r_distr acarr r_minus simp
flip: a_inv_def)
    show "⋀b x. ⟦b ∈ carrier R ∧ a ⊗ b ∈ I; x ∈ carrier R⟧
                 ⟹ a ⊗ (x ⊗ b) ∈ I"
      using acarr helper_I_closed m_comm by auto
    show "⋀b x. ⟦b ∈ carrier R ∧ a ⊗ b ∈ I; x ∈ carrier R⟧
                ⟹ a ⊗ (b ⊗ x) ∈ I"
      by (simp add: acarr helper_I_closed)
  qed
qed
```

In a cring every maximal ideal is prime

```
lemma (in cring) maximalideal_prime:
  assumes "maximalideal I R"
  shows "primeideal I R"
proof -
  interpret maximalideal I R by fact
  show ?thesis
  proof (rule ccontr)
    assume neg: "¬ primeideal I R"
```

```
    then obtain a b where acarr: "a ∈ carrier R" and bcarr: "b ∈ carrier
R"
      and abI: "a ⊗ b ∈ I" and anI: "a ∉ I" and bnI: "b ∉ I"
      using primeidealCE [OF is_cring]
      by (metis I_notcarr)
    define J where "J = {x∈carrier R. a ⊗ x ∈ I}"
    from is_cring and acarr have idealJ: "ideal J R"
      unfolding J_def by (rule helper_max_prime)
    have IsubJ: "I ⊆ J"
      using I_l_closed J_def a_Hcarr acarr by blast
    from abI and acarr bcarr have "b ∈ J"
      unfolding J_def by fast
    with bnI have JnI: "J ≠ I" by fast
    have "1 ∉ J"
      unfolding J_def by (simp add: acarr anI)
    then have Jncarr: "J ≠ carrier R" by fast
    interpret ideal J R by (rule idealJ)
    have "J = I ∨ J = carrier R"
      by (simp add: I_maximal IsubJ a_subset is_ideal)
    with JnI and Jncarr show False by simp
  qed
qed
```

## 13.11 Derived Theorems

A non-zero cring that has only the two trivial ideals is a field

```
lemma (in cring) trivialideals_fieldI:
  assumes carrnzero: "carrier R ≠ {0}"
    and haveideals: "{I. ideal I R} = {{0}, carrier R}"
  shows "field R"
proof (intro cring_fieldI equalityI)
  show "Units R ⊆ carrier R - {0}"
    by (metis Diff_empty Units_closed Units_r_inv_ex carrnzero l_null
one_zeroD subsetI subset_Diff_insert)
  show "carrier R - {0} ⊆ Units R"
  proof
    fix x
    assume xcarr': "x ∈ carrier R - {0}"
    then have xcarr: "x ∈ carrier R" and xnZ: "x ≠ 0" by auto
    from xcarr have xIdl: "ideal (PIdl x) R"
      by (intro cgenideal_ideal) fast
    have "PIdl x ≠ {0}"
      using xcarr xnZ cgenideal_self by blast
    with haveideals have "PIdl x = carrier R"
      by (blast intro!: xIdl)
    then have "1 ∈ PIdl x" by simp
    then have "∃y. 1 = y ⊗ x ∧ y ∈ carrier R"
      unfolding cgenideal_def by blast
    then obtain y where ycarr: " y ∈ carrier R" and ylinv: "1 = y ⊗
```

```
x"
      by fast
    have "∃y ∈ carrier R. y ⊗ x = 1 ∧ x ⊗ y = 1"
      using m_comm xcarr ycarr ylinv by auto
    with xcarr show "x ∈ Units R"
      unfolding Units_def by fast
  qed
qed

lemma (in field) all_ideals: "{I. ideal I R} = {{0}, carrier R}"
proof (intro equalityI subsetI)
  fix I
  assume a: "I ∈ {I. ideal I R}"
  then interpret ideal I R by simp

  show "I ∈ {{0}, carrier R}"
  proof (cases "∃a. a ∈ I - {0}")
    case True
    then obtain a where aI: "a ∈ I" and anZ: "a ≠ 0"
      by fast+
    have aUnit: "a ∈ Units R"
      by (simp add: aI anZ field_Units)
    then have a: "a ⊗ inv a = 1" by (rule Units_r_inv)
    from aI and aUnit have "a ⊗ inv a ∈ I"
      by (simp add: I_r_closed del: Units_r_inv)
    then have oneI: "1 ∈ I" by (simp add: a[symmetric])
    have "carrier R ⊆ I"
      using oneI one_imp_carrier by auto
    with a_subset have "I = carrier R" by fast
    then show "I ∈ {{0}, carrier R}" by fast
  next
    case False
    then have IZ: "⋀a. a ∈ I ⟹ a = 0" by simp
    have a: "I ⊆ {0}"
      using False by auto
    have "0 ∈ I" by simp
    with a have "I = {0}" by fast
    then show "I ∈ {{0}, carrier R}" by fast
  qed
qed (auto simp: zeroideal oneideal)
```

— "Jacobson Theorem 2.2"

```
lemma (in cring) trivialideals_eq_field:
  assumes carrnzero: "carrier R ≠ {0}"
  shows "({I. ideal I R} = {{0}, carrier R}) = field R"
  by (fast intro!: trivialideals_fieldI[OF carrnzero] field.all_ideals)
```

Like zeroprimeideal for domains

```
lemma (in field) zeromaximalideal: "maximalideal {0} R"
```

```
proof (intro maximalidealI zeroideal)
  from one_not_zero have "1 ∉ {0}" by simp
  with one_closed show "carrier R ≠ {0}" by fast
next
  fix J
  assume Jideal: "ideal J R"
  then have "J ∈ {I. ideal I R}" by fast
  with all_ideals show "J = {0} ∨ J = carrier R"
    by simp
qed

lemma (in cring) zeromaximalideal_fieldI:
  assumes zeromax: "maximalideal {0} R"
  shows "field R"
proof (intro trivialideals_fieldI maximalideal.I_notcarr[OF zeromax])
  have "J = carrier R" if Jn0: "J ≠ {0}" and idealJ: "ideal J R" for J
  proof -
    interpret ideal J R by (rule idealJ)
    have "{0} ⊆ J"
      by force
    from zeromax idealJ this a_subset
    have "J = {0} ∨ J = carrier R"
      by (rule maximalideal.I_maximal)
    with Jn0 show "J = carrier R"
      by simp
  qed
  then show "{I. ideal I R} = {{0}, carrier R}"
    by (auto simp: zeroideal oneideal)
qed

lemma (in cring) zeromaximalideal_eq_field: "maximalideal {0} R = field
R"
  using field.zeromaximalideal zeromaximalideal_fieldI by blast

end


theory RingHom
imports Ideal
begin
```

# 14 Homomorphisms of Non-Commutative Rings

Lifting existing lemmas in a `ring_hom_ring` locale

```
locale ring_hom_ring = R?: ring R + S?: ring S
    for R (structure) and S (structure) +
  fixes h
  assumes homh: "h ∈ ring_hom R S"
```

```
  notes hom_mult [simp] = ring_hom_mult [OF homh]
    and hom_one [simp] = ring_hom_one [OF homh]

sublocale ring_hom_cring ⊆ ring: ring_hom_ring
  by standard (rule homh)

sublocale ring_hom_ring ⊆ abelian_group?: abelian_group_hom R S
proof
  show "h ∈ hom (add_monoid R) (add_monoid S)"
    using homh by (simp add: hom_def ring_hom_def)
qed

lemma (in ring_hom_ring) is_ring_hom_ring:
  "ring_hom_ring R S h"
  by (rule ring_hom_ring_axioms)

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 -
  interpret ring R by fact
  interpret ring S by fact
  show ?thesis
  proof
    show "h ∈ ring_hom R S"
      unfolding ring_hom_def
      by (auto simp: compatible_mult compatible_add compatible_one hom_closed)
  qed
qed

lemma ring_hom_ringI2:
  assumes "ring R" "ring S"
  assumes h: "h ∈ ring_hom R S"
  shows "ring_hom_ring R S h"
proof -
  interpret R: ring R by fact
  interpret S: ring S by fact
  show ?thesis
  proof
    show "h ∈ ring_hom R S"
      using h .
  qed
```

```
qed

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 -
  interpret abelian_group_hom R S h by fact
  interpret R: ring R by fact
  interpret S: ring S by fact
  show ?thesis
  proof
    show "h ∈ ring_hom R S"
      unfolding ring_hom_def by (auto simp: compatible_one compatible_mult)
  qed
qed

lemma ring_hom_cringI:
  assumes "ring_hom_ring R S h" "cring R" "cring S"
  shows "ring_hom_cring R S h"
proof -
  interpret ring_hom_ring R S h by fact
  interpret R: cring R by fact
  interpret S: cring S by fact
  show ?thesis
  proof
    show "h ∈ ring_hom R S"
      by (simp add: homh)
  qed
qed
```

## 14.1 The Kernel of a Ring Homomorphism

```
lemma (in ring_hom_ring) kernel_is_ideal: "ideal (a_kernel R S h) R"
  apply (rule idealI [OF R.ring_axioms])
    apply (rule additive_subgroup.a_subgroup[OF additive_subgroup_a_kernel])
   apply (auto simp: a_kernel_def')
  done
```

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"
by (simp add: a_kernel_defs)
```

## 14.2 Cosets

Cosets of the kernel correspond to the elements of the image of the homomorphism

```
lemma (in ring_hom_ring) rcos_imp_homeq:
  assumes acarr: "a ∈ carrier R"
      and xrcos: "x ∈ a_kernel R S h +> a"
  shows "h x = h a"
proof -
  interpret ideal "a_kernel R S h" "R" by (rule kernel_is_ideal)

  from xrcos
      have "∃i ∈ a_kernel R S h. x = i ⊕ a" by (simp add: a_r_coset_defs)
  from this obtain i
      where iker: "i ∈ a_kernel R S h"
        and x: "x = i ⊕ a"
      by fast+
  note carr = acarr iker[THEN a_Hcarr]

  from x
      have "h x = h (i ⊕ a)" by simp
  also from carr
      have "... = h i ⊕S h a" by simp
  also from iker
      have "... = 0S ⊕S h a" by simp
  also from carr
      have "... = h a" by simp
  finally
      show "h x = h a" .
qed

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 -
  interpret ideal "a_kernel R S h" "R" by (rule kernel_is_ideal)

  note carr = acarr xcarr
  note hcarr = acarr[THEN hom_closed] xcarr[THEN hom_closed]

  from hx and hcarr
      have a: "h x ⊕S ⊖Sh a = 0S" by algebra
  from carr
      have "h x ⊕S ⊖Sh a = h (x ⊕ ⊖a)" by simp
  from a and this
      have b: "h (x ⊕ ⊖a) = 0S" by simp
```

```
  from carr have "x ⊕ ⊖a ∈ carrier R" by simp
  from this and b
      have "x ⊕ ⊖a ∈ a_kernel R S h"
      unfolding a_kernel_def'
      by fast

  from this and carr
      show "x ∈ a_kernel R S h +> a" by (simp add: a_rcos_module_rev)
qed

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 -
  interpret ideal "a_kernel R S h" "R" by (rule kernel_is_ideal)
  show ?thesis
    using assms by (auto simp: intro: homeq_imp_rcos rcos_imp_homeq a_elemrcos_carrier)
qed

lemma (in ring_hom_ring) hom_nat_pow:
  "x ∈ carrier R ⟹ h (x [^] (n :: nat)) = (h x) [^]_S n"
  by (induct n) (auto)


lemma (in ring_hom_ring) inj_on_domain:
  assumes "inj_on h (carrier R)"
  shows "domain S ⟹ domain R"
proof -
  assume A: "domain S" show "domain R"
  proof
    have "h 1 = 1_S ∧ h 0 = 0_S" by simp
    hence "h 1 ≠ h 0"
      using domain.one_not_zero[OF A] by simp
    thus "1 ≠ 0"
      using assms unfolding inj_on_def by fastforce
  next
    fix a b
    assume a: "a ∈ carrier R"
       and b: "b ∈ carrier R"
    have "h (a ⊗ b) = (h a) ⊗_S (h b)" by (simp add: a b)
    also have " ... = (h b) ⊗_S (h a)" using a b A cringE(1)[of S]
      by (simp add: cring.cring_simprules(14) domain_def)
    also have " ... = h (b ⊗ a)" by (simp add: a b)
    finally have "h (a ⊗ b) = h (b ⊗ a)" .
    thus "a ⊗ b = b ⊗ a"
      using assms a b unfolding inj_on_def by simp

    assume  ab: "a ⊗ b = 0"
    hence "h (a ⊗ b) = 0_S" by simp
```

```
    hence "(h a) ⊗S (h b) = 0S" using a b by simp
    hence "h a =  0S ∨ h b =  0S" using a b domain.integral[OF A] by
simp
    thus "a = 0 ∨ b = 0"
      using a b assms unfolding inj_on_def by force
  qed
qed

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 (rule classical)
  assume "¬ ?thesis"
  then have "m < n" by arith
  with bound have "f n = z" ..
  with nonzero show ?thesis by contradiction
qed

record ('a, 'p) up_ring = "('a, 'p) module" +
```

```
  monom :: "['a, nat] => 'p"
  coeff :: "['p, nat] => 'a"
```

**definition**

```
  up :: "('a, 'm) ring_scheme => (nat => 'a) set"
  where "up R = {f. f ∈ UNIV → carrier R ∧ (∃n. bound 0R n f)}"
```

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

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

Properties of the set of polynomials `up`.

**lemma** `mem_upI [intro]:`

```
  "[| ⋀n. f n ∈ carrier R; ∃n. bound (zero R) n f |] ==> f ∈ up R"
  by (simp add: up_def Pi_def)
```

**lemma** `mem_upD [dest]:`

```
  "f ∈ up R ==> f n ∈ carrier R"
  by (simp add: up_def Pi_def)
```

**context** `ring`
**begin**

**lemma** `bound_upD [dest]: "f ∈ up R ⟹ ∃n. bound 0 n f"` **by** `(simp add: up_def)`

**lemma** `up_one_closed: "(λn. if n = 0 then 1 else 0) ∈ up R"` **using** `up_def` **by** `force`

**lemma** `up_smult_closed: "[| a ∈ carrier R; p ∈ up R |] ==> (λi. a ⊗ p i) ∈ up R"` **by** `force`

**lemma** `up_add_closed:`

```
  "[| p ∈ up R; q ∈ up R |] ==> (λi. p i ⊕ q i) ∈ up R"
proof
  fix n
  assume "p ∈ up R" and "q ∈ up R"
  then show "p n ⊕ q n ∈ carrier R"
    by auto
next
  assume UP: "p ∈ up R" "q ∈ up R"
  show "∃n. bound 0 n (λi. p i ⊕ q i)"
```

```
  proof -
    from UP obtain n where boundn: "bound 0 n p" by fast
    from UP obtain m where boundm: "bound 0 m q" by fast
    have "bound 0 (max n m) (λi. p i ⊕ q i)"
    proof
      fix i
      assume "max n m < i"
      with boundn and boundm and UP show "p i ⊕ q i = 0" by fastforce
    qed
    then show ?thesis ..
  qed
qed

lemma up_a_inv_closed:
  "p ∈ up R ==> (λi. ⊖ (p i)) ∈ up R"
proof
  assume R: "p ∈ up R"
  then obtain n where "bound 0 n p" by auto
  then have "bound 0 n (λi. ⊖ p i)"
    by (simp add: bound_def minus_equality)
  then show "∃n. bound 0 n (λi. ⊖ p i)" by auto
qed auto

lemma up_minus_closed:
  "[| p ∈ up R; q ∈ up R |] ==> (λi. p i ⊖ q i) ∈ up R"
  unfolding a_minus_def
  using mem_upD [of p R] mem_upD [of q R] up_add_closed up_a_inv_closed
by auto

lemma up_mult_closed:
  "[| p ∈ up R; q ∈ up R |] ==>
  (λn. ⨁i ∈ {..n}. p i ⊗ q (n-i)) ∈ up R"
proof
  fix n
  assume "p ∈ up R" "q ∈ up R"
  then show "(⨁i ∈ {..n}. p i ⊗ q (n-i)) ∈ carrier R"
    by (simp add: mem_upD  funcsetI)
next
  assume UP: "p ∈ up R" "q ∈ up R"
  show "∃n. bound 0 n (λn. ⨁i ∈ {..n}. p i ⊗ q (n-i))"
  proof -
    from UP obtain n where boundn: "bound 0 n p" by fast
    from UP obtain m where boundm: "bound 0 m q" by fast
    have "bound 0 (n + m) (λn. ⨁i ∈ {..n}. p i ⊗ q (n - i))"
    proof
      fix k assume bound: "n + m < k"
      {
        fix i
        have "p i ⊗ q (k-i) = 0"
```

```
        proof (cases "n < i")
          case True
          with boundn have "p i = 0" by auto
          moreover from UP have "q (k-i) ∈ carrier R" by auto
          ultimately show ?thesis by simp
        next
          case False
          with bound have "m < k-i" by arith
          with boundm have "q (k-i) = 0" by auto
          moreover from UP have "p i ∈ carrier R" by auto
          ultimately show ?thesis by simp
        qed
      }
      then show "(⨁i ∈ {..k}. p i ⊗ q (k-i)) = 0"
        by (simp add: Pi_def)
    qed
    then show ?thesis by fast
  qed
qed

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
  by intro_locales [1] (rule P_def)

locale UP_domain = UP + R?: "domain" R

sublocale UP_domain < UP_cring
  by intro_locales [1] (rule P_def)

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
  fix x
  from prem and R show "p x = q x" by (simp add: UP_def)
qed

lemma coeff_closed [simp]:
  "p ∈ carrier P ==> coeff P p n ∈ carrier R" by (auto simp add: UP_def)

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 -
  assume R: "a ∈ carrier R"
  then have "(λn. if n = m then a else 0) ∈ up R"
    using up_def by force
  with R show ?thesis by (simp add: UP_def)
qed

lemma coeff_zero [simp]: "coeff P 0_P n = 0" by (auto simp add: UP_def)

lemma coeff_one [simp]: "coeff P 1_P n = (if n=0 then 1 else 0)"
  using up_one_closed by (simp add: UP_def)

lemma coeff_smult [simp]:
  "[| a ∈ carrier R; p ∈ carrier P |] ==> coeff P (a ⊙_P p) n = a ⊗ coeff
P p n"
  by (simp add: UP_def up_smult_closed)

lemma coeff_add [simp]:
  "[| p ∈ carrier P; q ∈ carrier P |] ==> coeff P (p ⊕_P q) n = coeff
P p n ⊕ coeff P q n"
  by (simp add: UP_def up_add_closed)

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))"
  by (simp add: UP_def up_mult_closed)

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" by (simp
add: UP_def up_mult_closed)

lemma UP_one_closed [simp]:
  "1P ∈ carrier P" by (simp add: UP_def up_one_closed)

lemma UP_zero_closed [intro, simp]:
  "0P ∈ carrier P" by (auto simp add: UP_def)

lemma UP_a_closed [intro, simp]:
  "[| p ∈ carrier P; q ∈ carrier P |] ==> p ⊕P q ∈ carrier P" by (simp
add: UP_def up_add_closed)

lemma monom_closed [simp]:
  "a ∈ carrier R ==> monom P a n ∈ carrier P" by (auto simp add: UP_def
up_def Pi_def)

lemma UP_smult_closed [simp]:
  "[| a ∈ carrier R; p ∈ carrier P |] ==> a ⊙P p ∈ carrier P" by (simp
add: UP_def up_smult_closed)

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)" by (rule up_eqI, simp add:
a_assoc R, simp_all add: R)

lemma UP_l_zero [simp]:
  assumes R: "p ∈ carrier P"
  shows "0P ⊕P p = p" by (rule up_eqI, simp_all add: R)

lemma UP_l_neg_ex:
  assumes R: "p ∈ carrier P"
  shows "∃q ∈ carrier P. q ⊕P p = 0P"
proof -
  let ?q = "λi. ⊖ (p i)"
```

```
  from R have closed: "?q ∈ carrier P"
    by (simp add: UP_def P_def up_a_inv_closed)
  from R have coeff: "!!n. coeff P ?q n = ⊖ (coeff P p n)"
    by (simp add: UP_def P_def up_a_inv_closed)
  show ?thesis
  proof
    show "?q ⊕P p = 0P"
      by (auto intro!: up_eqI simp add: R closed coeff R.l_neg)
  qed (rule closed)
qed

lemma UP_a_comm:
  assumes R: "p ∈ carrier P" "q ∈ carrier P"
  shows "p ⊕P q = q ⊕P p" by (rule up_eqI, simp add: a_comm R, simp_all
add: R)

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 (rule up_eqI)
  fix n
  {
    fix k and a b c :: "nat=>'a"
    assume R: "a ∈ UNIV → carrier R" "b ∈ UNIV → carrier R"
      "c ∈ UNIV → carrier R"
    then have "k <= n ==>
      (⨁j ∈ {..k}. (⨁i ∈ {..j}. a i ⊗ b (j-i)) ⊗ c (n-j)) =
      (⨁j ∈ {..k}. a j ⊗ (⨁i ∈ {..k-j}. b i ⊗ c (n-j-i)))"
      (is "_ ⟹ ?eq k")
    proof (induct k)
      case 0 then show ?case by (simp add: Pi_def m_assoc)
    next
      case (Suc k)
      then have "k <= n" by arith
      from this R have "?eq k" by (rule Suc)
      with R show ?case
        by (simp cong: finsum_cong
             add: Suc_diff_le Pi_def l_distr r_distr m_assoc)
           (simp cong: finsum_cong add: Pi_def a_ac finsum_ldistr m_assoc)
    qed
  }
  with R show "coeff P ((p ⊗P q) ⊗P r) n = coeff P (p ⊗P (q ⊗P r))
n"
    by (simp add: Pi_def)
qed (simp_all add: R)

lemma UP_r_one [simp]:
  assumes R: "p ∈ carrier P" shows "p ⊗P 1P = p"
proof (rule up_eqI)
```

```
  fix n
  show "coeff P (p ⊗P 1P) n = coeff P p n"
  proof (cases n)
    case 0
    {
      with R show ?thesis by simp
    }
  next
    case Suc
    {

      fix nn assume Succ: "n = Suc nn"
      have "coeff P (p ⊗P 1P) (Suc nn) = coeff P p (Suc nn)"
      proof -
        have "coeff P (p ⊗P 1P) (Suc nn) = (⨁i∈{..Suc nn}. coeff P
p i ⊗ (if Suc nn ≤ i then 1 else 0))" using R by simp
        also have "... = coeff P p (Suc nn) ⊗ (if Suc nn ≤ Suc nn then
1 else 0) ⊕ (⨁i∈{..nn}. coeff P p i ⊗ (if Suc nn ≤ i then 1 else 0))"
          using finsum_Suc [of "(λi::nat. coeff P p i ⊗ (if Suc nn ≤
i then 1 else 0))" "nn"] unfolding Pi_def using R by simp
        also have "... = coeff P p (Suc nn) ⊗ (if Suc nn ≤ Suc nn then
1 else 0)"
        proof -
          have "(⨁i∈{..nn}. coeff P p i ⊗ (if Suc nn ≤ i then 1 else
0)) = (⨁i∈{..nn}. 0)"
            using finsum_cong [of "{..nn}" "{..nn}" "(λi::nat. coeff P
p i ⊗ (if Suc nn ≤ i then 1 else 0))" "(λi::nat. 0)"] using R
            unfolding Pi_def by simp
          also have "... = 0" by simp
          finally show ?thesis using r_zero R by simp
        qed
        also have "... = coeff P p (Suc nn)" using R by simp
        finally show ?thesis by simp
      qed
      then show ?thesis using Succ by simp
    }
  qed
qed (simp_all add: R)

lemma UP_l_one [simp]:
  assumes R: "p ∈ carrier P"
  shows "1P ⊗P p = p"
proof (rule up_eqI)
  fix n
  show "coeff P (1P ⊗P p) n = coeff P p n"
  proof (cases n)
    case 0 with R show ?thesis by simp
  next
    case Suc with R show ?thesis
```

```
      by (simp del: finsum_Suc add: finsum_Suc2 Pi_def)
  qed
qed (simp_all add: R)

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)"
  by (rule up_eqI) (simp add: l_distr R Pi_def, simp_all add: R)

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)"
  by (rule up_eqI) (simp add: r_distr R Pi_def, simp_all add: R)

theorem UP_ring: "ring P"
  by (auto intro!: ringI abelian_groupI monoidI UP_a_assoc)
    (auto intro: UP_a_comm UP_l_neg_ex UP_m_assoc UP_l_distr UP_r_distr)

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 (rule up_eqI)
  fix n
  have l: "(⨁i ∈ {..k}. a i ⊗ b (n-i)) = (⨁i ∈ {..k}. a (k-i) ⊗ b
(i+n-k))" (is "?eq k")
    if "a ∈ UNIV → carrier R" "b ∈ UNIV → carrier R" and "k <= n"
    for k and a b :: "nat=>'a"
  using that
  proof (induct k)
    case 0 then show ?case by (simp add: Pi_def)
  next
    case (Suc k) then show ?case
      by (subst (2) finsum_Suc2) (simp add: Pi_def a_comm)+
  qed
  from R1 R2 show "coeff P (p ⊗P q) n =  coeff P (q ⊗P p) n"
    unfolding coeff_mult [OF R1 R2, of n]
    unfolding coeff_mult [OF R2 R1, of n]
    using l [of "(λi. coeff P p i)" "(λi. coeff P q i)" "n"] by (simp
add: Pi_def m_comm)
qed (simp_all add: R1 R2)
```

## 15.5 Polynomials over a commutative ring for a commutative ring

```
theorem UP_cring:
  "cring P" using UP_ring unfolding cring_def by (auto intro!: comm_monoidI
UP_m_assoc UP_m_comm)

end

context UP_ring
begin

lemma UP_a_inv_closed [intro, simp]:
  "p ∈ carrier P ==> ⊖P p ∈ carrier P"
  by (rule abelian_group.a_inv_closed [OF ring.is_abelian_group [OF UP_ring]])

lemma coeff_a_inv [simp]:
  assumes R: "p ∈ carrier P"
  shows "coeff P (⊖P p) n = ⊖ (coeff P p n)"
proof -
  from R coeff_closed UP_a_inv_closed have
    "coeff P (⊖P p) n = ⊖ coeff P p n ⊕ (coeff P p n ⊕ coeff P (⊖P p)
n)"
    by algebra
  also from R have "... =  ⊖ (coeff P p n)"
    by (simp del: coeff_add add: coeff_add [THEN sym]
      abelian_group.r_neg [OF ring.is_abelian_group [OF UP_ring]])
  finally show ?thesis .
qed

end

sublocale UP_ring < P?: ring P using UP_ring .
sublocale UP_cring < P?: cring P using UP_cring .
```

## 15.6 Polynomials Form an Algebra

```
context UP_ring
begin

lemma UP_smult_l_distr:
  "[| a ∈ carrier R; b ∈ carrier R; p ∈ carrier P |] ==>
  (a ⊕ b) ⊙P p = a ⊙P p ⊕P b ⊙P p"
  by (rule up_eqI) (simp_all add: R.l_distr)

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"
  by (rule up_eqI) (simp_all add: R.r_distr)
```

```
lemma UP_smult_assoc1:
      "[| a ∈ carrier R; b ∈ carrier R; p ∈ carrier P |] ==>
      (a ⊗ b) ⊙P p = a ⊙P (b ⊙P p)"
  by (rule up_eqI) (simp_all add: R.m_assoc)

lemma UP_smult_zero [simp]:
      "p ∈ carrier P ==> 0 ⊙P p = 0P"
  by (rule up_eqI) simp_all

lemma UP_smult_one [simp]:
      "p ∈ carrier P ==> 1 ⊙P p = p"
  by (rule up_eqI) simp_all

lemma UP_smult_assoc2:
  "[| a ∈ carrier R; p ∈ carrier P; q ∈ carrier P |] ==>
  (a ⊙P p) ⊗P q = a ⊙P (p ⊗P q)"
  by (rule up_eqI) (simp_all add: R.finsum_rdistr R.m_assoc Pi_def)

end
```

Interpretation of lemmas from `algebra`.

```
lemma (in UP_cring) UP_algebra:
  "algebra R P" by (auto intro!: algebraI R.cring_axioms UP_cring UP_smult_l_distr
UP_smult_r_distr
    UP_smult_assoc1 UP_smult_assoc2)

sublocale UP_cring < algebra R P using UP_algebra .
```

## 15.7 Further Lemmas Involving Monomials

```
context UP_ring
begin

lemma monom_zero [simp]:
  "monom P 0 n = 0P" by (simp add: UP_def P_def)

lemma monom_mult_is_smult:
  assumes R: "a ∈ carrier R" "p ∈ carrier P"
  shows "monom P a 0 ⊗P p = a ⊙P p"
proof (rule up_eqI)
  fix n
  show "coeff P (monom P a 0 ⊗P p) n = coeff P (a ⊙P p) n"
  proof (cases n)
    case 0 with R show ?thesis by simp
  next
    case Suc with R show ?thesis
      using R.finsum_Suc2 by (simp del: R.finsum_Suc add: Pi_def)
  qed
qed (simp_all add: R)
```

```
lemma monom_one [simp]:
  "monom P 1 0 = 1P"
  by (rule up_eqI) simp_all

lemma monom_add [simp]:
  "[| a ∈ carrier R; b ∈ carrier R |] ==>
  monom P (a ⊕ b) n = monom P a n ⊕P monom P b n"
  by (rule up_eqI) simp_all

lemma monom_one_Suc:
  "monom P 1 (Suc n) = monom P 1 n ⊗P monom P 1 1"
proof (rule up_eqI)
  fix k
  show "coeff P (monom P 1 (Suc n)) k = coeff P (monom P 1 n ⊗P monom
P 1 1) k"
  proof (cases "k = Suc n")
    case True show ?thesis
    proof -
      fix m
      from True have less_add_diff:
        "!!i. [| n < i; i <= n + m |] ==> n + m - i < m" by arith
      from True have "coeff P (monom P 1 (Suc n)) k = 1" by simp
      also from True
      have "... = (⨁i ∈ {..<n} ∪ {n}. coeff P (monom P 1 n) i ⊗
        coeff P (monom P 1 1) (k - i))"
        by (simp cong: R.finsum_cong add: Pi_def)
      also have "... = (⨁i ∈  {..n}. coeff P (monom P 1 n) i ⊗
        coeff P (monom P 1 1) (k - i))"
        by (simp only: ivl_disj_un_singleton)
      also from True
      have "... = (⨁i ∈ {..n} ∪ {n<..k}. coeff P (monom P 1 n) i ⊗
        coeff P (monom P 1 1) (k - i))"
        by (simp cong: R.finsum_cong add: R.finsum_Un_disjoint ivl_disj_int_one
          order_less_imp_not_eq Pi_def)
      also from True have "... = coeff P (monom P 1 n ⊗P monom P 1 1)
k"
        by (simp add: ivl_disj_un_one)
      finally show ?thesis .
    qed
  next
    case False
    note neq = False
    let ?s =
      "λi. (if n = i then 1 else 0) ⊗ (if Suc 0 = k - i then 1 else 0)"
    from neq have "coeff P (monom P 1 (Suc n)) k = 0" by simp
    also have "... = (⨁i ∈ {..k}. ?s i)"
    proof -
      have f1: "(⨁i ∈ {..<n}. ?s i) = 0"
```

```
        by (simp cong: R.finsum_cong add: Pi_def)
      from neq have f2: "(⨁i ∈ {n}. ?s i) = 0"
        by (simp cong: R.finsum_cong add: Pi_def) arith
      have f3: "n < k ==> (⨁i ∈ {n<..k}. ?s i) = 0"
        by (simp cong: R.finsum_cong add: order_less_imp_not_eq Pi_def)
      show ?thesis
      proof (cases "k < n")
        case True then show ?thesis by (simp cong: R.finsum_cong add:
Pi_def)
      next
        case False then have n_le_k: "n <= k" by arith
        show ?thesis
        proof (cases "n = k")
          case True
          then have "0 = (⨁i ∈ {..<n} ∪ {n}. ?s i)"
            by (simp cong: R.finsum_cong add: Pi_def)
          also from True have "... = (⨁i ∈ {..k}. ?s i)"
            by (simp only: ivl_disj_un_singleton)
          finally show ?thesis .
        next
          case False with n_le_k have n_less_k: "n < k" by arith
          with neq have "0 = (⨁i ∈ {..<n} ∪ {n}. ?s i)"
            by (simp add: R.finsum_Un_disjoint f1 f2 Pi_def del: Un_insert_right)
          also have "... = (⨁i ∈ {..n}. ?s i)"
            by (simp only: ivl_disj_un_singleton)
          also from n_less_k neq have "... = (⨁i ∈ {..n} ∪ {n<..k}.
?s i)"
            by (simp add: R.finsum_Un_disjoint f3 ivl_disj_int_one Pi_def)
          also from n_less_k have "... = (⨁i ∈ {..k}. ?s i)"
            by (simp only: ivl_disj_un_one)
          finally show ?thesis .
        qed
      qed
    qed
    also have "... = coeff P (monom P 1 n ⊗P monom P 1 1) k" by simp
    finally show ?thesis .
  qed
qed (simp_all)

lemma monom_one_Suc2:
  "monom P 1 (Suc n) = monom P 1 1 ⊗P monom P 1 n"
proof (induct n)
  case 0 show ?case by simp
next
  case Suc
  {
    fix k:: nat
    assume hypo: "monom P 1 (Suc k) = monom P 1 1 ⊗P monom P 1 k"
    then show "monom P 1 (Suc (Suc k)) = monom P 1 1 ⊗P monom P 1 (Suc
```

```
k)"
    proof -
      have lhs: "monom P 1 (Suc (Suc k)) = monom P 1 1 ⊗P monom P 1 k
⊗P monom P 1 1"
        unfolding monom_one_Suc [of "Suc k"] unfolding hypo ..
      note cl = monom_closed [OF R.one_closed, of 1]
      note clk = monom_closed [OF R.one_closed, of k]
      have rhs: "monom P 1 1 ⊗P monom P 1 (Suc k) = monom P 1 1 ⊗P monom
P 1 k ⊗P monom P 1 1"
        unfolding monom_one_Suc [of k] unfolding sym [OF m_assoc  [OF
cl clk cl]] ..
      from lhs rhs show ?thesis by simp
    qed
  }
qed
```

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"
  unfolding monom_one_Suc [symmetric] monom_one_Suc2 [symmetric] ..

lemma monom_mult_smult:
  "[| a ∈ carrier R; b ∈ carrier R |] ==> monom P (a ⊗ b) n = a ⊙P monom
P b n"
  by (rule up_eqI) simp_all

lemma monom_one_mult:
  "monom P 1 (n + m) = monom P 1 n ⊗P monom P 1 m"
proof (induct n)
  case 0 show ?case by simp
next
  case Suc then show ?case
    unfolding add_Suc unfolding monom_one_Suc unfolding Suc.hyps
    using m_assoc monom_one_comm [of m] by simp
qed

lemma monom_one_mult_comm: "monom P 1 n ⊗P monom P 1 m = monom P 1 m
⊗P monom P 1 n"
  unfolding monom_one_mult [symmetric] by (rule up_eqI) simp_all

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 (rule up_eqI)
  fix k
  show "coeff P (monom P (a ⊗ b) (n + m)) k = coeff P (monom P a n ⊗P
monom P b m) k"
```

```
  proof (cases "n + m = k")
    case True
    {
      show ?thesis
        unfolding True [symmetric]
          coeff_mult [OF monom_closed [OF a_in_R, of n] monom_closed [OF
b_in_R, of m], of "n + m"]
          coeff_monom [OF a_in_R, of n] coeff_monom [OF b_in_R, of m]
        using R.finsum_cong [of "{.. n + m}" "{.. n + m}" "(λi. (if n
= i then a else 0) ⊗ (if m = n + m - i then b else 0))"
          "(λi. if n = i then a ⊗ b else 0)"]
          a_in_R b_in_R
        unfolding simp_implies_def
        using R.finsum_singleton [of n "{.. n + m}" "(λi. a ⊗ b)"]
        unfolding Pi_def by auto
    }
  next
    case False
    {
      show ?thesis
        unfolding coeff_monom [OF R.m_closed [OF a_in_R b_in_R], of "n
+ m" k] apply (simp add: False)
        unfolding coeff_mult [OF monom_closed [OF a_in_R, of n] monom_closed
[OF b_in_R, of m], of k]
        unfolding coeff_monom [OF a_in_R, of n] unfolding coeff_monom
[OF b_in_R, of m] using False
        using R.finsum_cong [of "{..k}" "{..k}" "(λi. (if n = i then a
else 0) ⊗ (if m = k - i then b else 0))" "(λi. 0)"]
        unfolding Pi_def simp_implies_def using a_in_R b_in_R by force
    }
  qed
qed (simp_all add: a_in_R b_in_R)

lemma monom_a_inv [simp]:
  "a ∈ carrier R ==> monom P (⊖ a) n = ⊖P monom P a n"
  by (rule up_eqI) auto

lemma monom_inj:
  "inj_on (λa. monom P a n) (carrier R)"
proof (rule inj_onI)
  fix x y
  assume R: "x ∈ carrier R" "y ∈ carrier R" and eq: "monom P x n = monom
P y n"
  then have "coeff P (monom P x n) n = coeff P (monom P y n) n" by simp
  with R show "x = y" by simp
qed

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"
  by (unfold deg_def P_def) (fast intro: Least_le)



lemma deg_aboveD:
  assumes "deg R p < m" and "p ∈ carrier P"
  shows "coeff P p m = 0"
proof -
  from ‹p ∈ carrier P› obtain n where "bound 0 n (coeff P p)"
    by (auto simp add: UP_def P_def)
  then have "bound 0 (deg R p) (coeff P p)"
    by (auto simp: deg_def P_def dest: LeastI)
  from this and ‹deg R p < m› show ?thesis ..
qed

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 (cases "n=0")
  case True then show ?thesis by simp
next
  case False then have "coeff P p n ≠ 0" by (rule non_zero)
  then have "¬ deg R p < n" by (fast dest: deg_aboveD intro: R)
  then show ?thesis by arith
qed

lemma lcoeff_nonzero_deg:
  assumes deg: "deg R p ≠ 0" and R: "p ∈ carrier P"
  shows "coeff P p (deg R p) ≠ 0"
proof -
  from R obtain m where "deg R p ≤ m" and m_coeff: "coeff P p m ≠ 0"
  proof -
    have minus: "⋀(n::nat) m. n ≠ 0 ⟹ (n - Suc 0 < m) = (n ≤ m)"
      by arith
    from deg have "deg R p - 1 < (LEAST n. bound 0 n (coeff P p))"
      by (unfold deg_def P_def) simp
```

```
    then have "¬ bound 0 (deg R p - 1) (coeff P p)" by (rule not_less_Least)
    then have "∃m. deg R p - 1 < m ∧ coeff P p m ≠ 0"
      by (unfold bound_def) fast
    then have "∃m. deg R p ≤ m ∧ coeff P p m ≠ 0" by (simp add: deg
minus)
    then show ?thesis by (auto intro: that)
  qed
  with deg_belowI R have "deg R p = m" by fastforce
  with m_coeff show ?thesis by simp
qed

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 -
  have "∃m. coeff P p m ≠ 0"
  proof (rule classical)
    assume "¬ ?thesis"
    with R have "p = 0P" by (auto intro: up_eqI)
    with nonzero show ?thesis by contradiction
  qed
  then obtain m where coeff: "coeff P p m ≠ 0" ..
  from this and R have "m ≤ deg R p" by (rule deg_belowI)
  then have "m = 0" by (simp add: deg)
  with coeff show ?thesis by simp
qed

lemma lcoeff_nonzero:
  assumes neq: "p ≠ 0P" and R: "p ∈ carrier P"
  shows "coeff P p (deg R p) ≠ 0"
proof (cases "deg R p = 0")
  case True with neq R show ?thesis by (simp add: lcoeff_nonzero_nonzero)
next
  case False with neq R show ?thesis by (simp add: lcoeff_nonzero_deg)
qed

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"
by (fast intro: le_antisym deg_aboveI deg_belowI)
```

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)"
by(rule deg_aboveI)(simp_all add: deg_aboveD)
```

```
lemma deg_monom_le:
  "a ∈ carrier R ⟹ deg R (monom P a n) ≤ n"
  by (intro deg_aboveI) simp_all

lemma deg_monom [simp]:
  "[| a ≠ 0; a ∈ carrier R |] ==> deg R (monom P a n) = n"
  by (fastforce intro: le_antisym deg_aboveI deg_belowI)

lemma deg_const [simp]:
  assumes R: "a ∈ carrier R" shows "deg R (monom P a 0) = 0"
proof (rule le_antisym)
  show "deg R (monom P a 0) ≤ 0" by (rule deg_aboveI) (simp_all add:
R)
next
  show "0 ≤ deg R (monom P a 0)" by (rule deg_belowI) (simp_all add:
R)
qed

lemma deg_zero [simp]:
  "deg R 0P = 0"
proof (rule le_antisym)
  show "deg R 0P ≤ 0" by (rule deg_aboveI) simp_all
next
  show "0 ≤ deg R 0P" by (rule deg_belowI) simp_all
qed

lemma deg_one [simp]:
  "deg R 1P = 0"
proof (rule le_antisym)
  show "deg R 1P ≤ 0" by (rule deg_aboveI) simp_all
next
  show "0 ≤ deg R 1P" by (rule deg_belowI) simp_all
qed

lemma deg_uminus [simp]:
  assumes R: "p ∈ carrier P" shows "deg R (⊖P p) = deg R p"
proof (rule le_antisym)
  show "deg R (⊖P p) ≤ deg R p" by (simp add: deg_aboveI deg_aboveD
R)
next
  show "deg R p ≤ deg R (⊖P p)"
    by (simp add: deg_belowI lcoeff_nonzero_deg
      inj_on_eq_iff [OF R.a_inv_inj, of _ "0", simplified] R)
qed
```

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)"
  by (cases "a = 0") (simp add: deg_aboveI deg_aboveD)+

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 (rule le_antisym)
  show "deg R (a ⊙P p) ≤ (if a = 0 then 0 else deg R p)"
    using R by (rule deg_smult_ring)
next
  show "(if a = 0 then 0 else deg R p) ≤ deg R (a ⊙P p)"
  proof (cases "a = 0")
  qed (simp, simp add: deg_belowI lcoeff_nonzero_deg integral_iff R)
qed

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 (rule deg_aboveI)
  fix m
  assume boundm: "deg R p + deg R q < m"
  {
    fix k i
    assume boundk: "deg R p + deg R q < k"
    then have "coeff P p i ⊗ coeff P q (k - i) = 0"
    proof (cases "deg R p < i")
      case True then show ?thesis by (simp add: deg_aboveD R)
    next
      case False with boundk have "deg R q < k - i" by arith
      then show ?thesis by (simp add: deg_aboveD R)
    qed
  }
  with boundm R show "coeff P (p ⊗P q) m = 0" by simp
qed (simp add: R)

end

context UP_domain
begin
```

```
lemma deg_mult [simp]:
  "[| p ≠ 0P; q ≠ 0P; p ∈ carrier P; q ∈ carrier P |] ==>
  deg R (p ⊗P q) = deg R p + deg R q"
proof (rule le_antisym)
  assume "p ∈ carrier P" " q ∈ carrier P"
  then show "deg R (p ⊗P q) ≤ deg R p + deg R q" by (rule deg_mult_ring)
next
  let ?s = "(λi. coeff P p i ⊗ coeff P q (deg R p + deg R q - i))"
  assume R: "p ∈ carrier P" "q ∈ carrier P" and nz: "p ≠ 0P" "q ≠ 0P"
  have less_add_diff: "!!(k::nat) n m. k < n ==> m < n + m - k" by arith
  show "deg R p + deg R q ≤ deg R (p ⊗P q)"
  proof (rule deg_belowI, simp add: R)
    have "(⨁i ∈ {.. deg R p + deg R q}. ?s i)
      = (⨁i ∈ {..< deg R p} ∪ {deg R p .. deg R p + deg R q}. ?s i)"
      by (simp only: ivl_disj_un_one)
    also have "... = (⨁i ∈ {deg R p .. deg R p + deg R q}. ?s i)"
      by (simp cong: R.finsum_cong add: R.finsum_Un_disjoint ivl_disj_int_one
        deg_aboveD less_add_diff R Pi_def)
    also have "...= (⨁i ∈ {deg R p} ∪ {deg R p <.. deg R p + deg R q}.
?s i)"
      by (simp only: ivl_disj_un_singleton)
    also have "... = coeff P p (deg R p) ⊗ coeff P q (deg R q)"
      by (simp cong: R.finsum_cong add: deg_aboveD R Pi_def)
    finally have "(⨁i ∈ {.. deg R p + deg R q}. ?s i)
      = coeff P p (deg R p) ⊗ coeff P q (deg R q)" .
    with nz show "(⨁i ∈ {.. deg R p + deg R q}. ?s i) ≠ 0"
      by (simp add: integral_iff lcoeff_nonzero R)
  qed (simp add: R)
qed

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)"
  using fin by induct (auto simp: Pi_def)

lemma up_repr:
  assumes R: "p ∈ carrier P"
  shows "(⨁P i ∈ {..deg R p}. monom P (coeff P p i) i) = p"
proof (rule up_eqI)
  let ?s = "(λi. monom P (coeff P p i) i)"
  fix k
```

```
  from R have RR: "!!i. (if i = k then coeff P p i else 0) ∈ carrier
R"
    by simp
  show "coeff P (⨁P i ∈ {..deg R p}. ?s i) k = coeff P p k"
  proof (cases "k ≤ deg R p")
    case True
    hence "coeff P (⨁P i ∈ {..deg R p}. ?s i) k =
          coeff P (⨁P i ∈ {..k} ∪ {k<..deg R p}. ?s i) k"
      by (simp only: ivl_disj_un_one)
    also from True
    have "... = coeff P (⨁P i ∈ {..k}. ?s i) k"
      by (simp cong: R.finsum_cong add: R.finsum_Un_disjoint
        ivl_disj_int_one order_less_imp_not_eq2 coeff_finsum R RR Pi_def)
    also
    have "... = coeff P (⨁P i ∈ {..<k} ∪ {k}. ?s i) k"
      by (simp only: ivl_disj_un_singleton)
    also have "... = coeff P p k"
      by (simp cong: R.finsum_cong add: coeff_finsum deg_aboveD R RR Pi_def)
    finally show ?thesis .
  next
    case False
    hence "coeff P (⨁P i ∈ {..deg R p}. ?s i) k =
          coeff P (⨁P i ∈ {..<deg R p} ∪ {deg R p}. ?s i) k"
      by (simp only: ivl_disj_un_singleton)
    also from False have "... = coeff P p k"
      by (simp cong: R.finsum_cong add: coeff_finsum deg_aboveD R Pi_def)
    finally show ?thesis .
  qed
qed (simp_all add: R Pi_def)

lemma up_repr_le:
  "[| deg R p <= n; p ∈ carrier P |] ==>
  (⨁P i ∈ {..n}. monom P (coeff P p i) i) = p"
proof -
  let ?s = "(λi. monom P (coeff P p i) i)"
  assume R: "p ∈ carrier P" and "deg R p <= n"
  then have "finsum P ?s {..n} = finsum P ?s ({..deg R p} ∪ {deg R p<..n})"
    by (simp only: ivl_disj_un_one)
  also have "... = finsum P ?s {..deg R p}"
    by (simp cong: P.finsum_cong add: P.finsum_Un_disjoint ivl_disj_int_one
      deg_aboveD R Pi_def)
  also have "... = p" using R by (rule up_repr)
  finally show ?thesis .
qed

end
```

## 15.9 Polynomials over Integral Domains

```
lemma domainI:
  assumes cring: "cring R"
    and one_not_zero: "one R ≠ zero R"
    and integral: "⋀a b. [| mult R a b = zero R; a ∈ carrier R;
      b ∈ carrier R |] ==> a = zero R ∨ b = zero R"
  shows "domain R"
  by (auto intro!: domain.intro domain_axioms.intro cring.axioms assms
    del: disjCI)

context UP_domain
begin

lemma UP_one_not_zero:
  "𝟭P ≠ 𝟬P"
proof
  assume "𝟭P = 𝟬P"
  hence "coeff P 𝟭P 0 = (coeff P 𝟬P 0)" by simp
  hence "𝟭 = 𝟬" by simp
  with R.one_not_zero show "False" by contradiction
qed

lemma UP_integral:
  "[| p ⊗P q = 𝟬P; p ∈ carrier P; q ∈ carrier P |] ==> p = 𝟬P ∨ q = 𝟬P"
proof -
  fix p q
  assume pq: "p ⊗P q = 𝟬P" and R: "p ∈ carrier P" "q ∈ carrier P"
  show "p = 𝟬P ∨ q = 𝟬P"
  proof (rule classical)
    assume c: "¬ (p = 𝟬P ∨ q = 𝟬P)"
    with R have "deg R p + deg R q = deg R (p ⊗P q)" by simp
    also from pq have "... = 0" by simp
    finally have "deg R p + deg R q = 0" .
    then have f1: "deg R p = 0 ∧ deg R q = 0" by simp
    from f1 R have "p = (⨁P i ∈ {..0}. monom P (coeff P p i) i)"
      by (simp only: up_repr_le)
    also from R have "... = monom P (coeff P p 0) 0" by simp
    finally have p: "p = monom P (coeff P p 0) 0" .
    from f1 R have "q = (⨁P i ∈ {..0}. monom P (coeff P q i) i)"
      by (simp only: up_repr_le)
    also from R have "... = monom P (coeff P q 0) 0" by simp
    finally have q: "q = monom P (coeff P q 0) 0" .
    from R have "coeff P p 0 ⊗ coeff P q 0 = coeff P (p ⊗P q) 0" by
simp
    also from pq have "... = 𝟬" by simp
    finally have "coeff P p 0 ⊗ coeff P q 0 = 𝟬" .
    with R have "coeff P p 0 = 𝟬 ∨ coeff P q 0 = 𝟬"
      by (simp add: R.integral_iff)
    with p q show "p = 𝟬P ∨ q = 𝟬P" by fastforce
```

```
  qed
qed

theorem UP_domain:
  "domain P"
  by (auto intro!: domainI UP_cring UP_one_not_zero UP_integral del: disjCI)

end
```

Interpretation of theorems from `domain`.

```
sublocale UP_domain < "domain" P
  by intro_locales (rule domain.axioms UP_domain)+
```

## 15.10 The Evaluation Homomorphism and Universal Property

```
lemma (in abelian_monoid) boundD_carrier:
  "[| bound 0 n f; n < m |] ==> f m ∈ carrier G"
  by auto

context ring
begin

theorem diagonal_sum:
  "[| f ∈ {..n + m::nat} → carrier R; g ∈ {..n + m} → carrier R |] ==>
  (⨁k ∈ {..n + m}. ⨁i ∈ {..k}. f i ⊗ g (k - i)) =
  (⨁k ∈ {..n + m}. ⨁i ∈ {..n + m - k}. f k ⊗ g i)"
proof -
  assume Rf: "f ∈ {..n + m} → carrier R" and Rg: "g ∈ {..n + m} →
carrier R"
  {
    fix j
    have "j <= n + m ==>
      (⨁k ∈ {..j}. ⨁i ∈ {..k}. f i ⊗ g (k - i)) =
      (⨁k ∈ {..j}. ⨁i ∈ {..j - k}. f k ⊗ g i)"
    proof (induct j)
      case 0 from Rf Rg show ?case by (simp add: Pi_def)
    next
      case (Suc j)
      have R6: "!!i k. [| k <= j; i <= Suc j - k |] ==> g i ∈ carrier
R"
        using Suc by (auto intro!: funcset_mem [OF Rg])
      have R8: "!!i k. [| k <= Suc j; i <= k |] ==> g (k - i) ∈ carrier
R"
        using Suc by (auto intro!: funcset_mem [OF Rg])
      have R9: "!!i k. [| k <= Suc j |] ==> f k ∈ carrier R"
        using Suc by (auto intro!: funcset_mem [OF Rf])
      have R10: "!!i k. [| k <= Suc j; i <= Suc j - k |] ==> g i ∈ carrier
R"
```

```
      using Suc by (auto intro!: funcset_mem [OF Rg])
    have R11: "g 0 ∈ carrier R"
      using Suc by (auto intro!: funcset_mem [OF Rg])
    from Suc show ?case
      by (simp cong: finsum_cong add: Suc_diff_le a_ac
        Pi_def R6 R8 R9 R10 R11)
  qed
 }
 then show ?thesis by fast
qed

theorem cauchy_product:
  assumes bf: "bound 0 n f" and bg: "bound 0 m g"
    and Rf: "f ∈ {..n} → carrier R" and Rg: "g ∈ {..m} → carrier R"
  shows "(⨁k ∈ {..n + m}. ⨁i ∈ {..k}. f i ⊗ g (k - i)) =
    (⨁i ∈ {..n}. f i) ⊗ (⨁i ∈ {..m}. g i)"
proof -
  have f: "!!x. f x ∈ carrier R"
  proof -
    fix x
    show "f x ∈ carrier R"
      using Rf bf boundD_carrier by (cases "x <= n") (auto simp: Pi_def)
  qed
  have g: "!!x. g x ∈ carrier R"
  proof -
    fix x
    show "g x ∈ carrier R"
      using Rg bg boundD_carrier by (cases "x <= m") (auto simp: Pi_def)
  qed
  from f g have "(⨁k ∈ {..n + m}. ⨁i ∈ {..k}. f i ⊗ g (k - i)) =
      (⨁k ∈ {..n + m}. ⨁i ∈ {..n + m - k}. f k ⊗ g i)"
    by (simp add: diagonal_sum Pi_def)
  also have "... = (⨁k ∈ {..n} ∪ {n<..n + m}. ⨁i ∈ {..n + m - k}.
f k ⊗ g i)"
    by (simp only: ivl_disj_un_one)
  also from f g have "... = (⨁k ∈ {..n}. ⨁i ∈ {..n + m - k}. f k ⊗
g i)"
    by (simp cong: finsum_cong
      add: bound.bound [OF bf] finsum_Un_disjoint ivl_disj_int_one Pi_def)
  also from f g
  have "... = (⨁k ∈ {..n}. ⨁i ∈ {..m} ∪ {m<..n + m - k}. f k ⊗ g i)"
    by (simp cong: finsum_cong add: ivl_disj_un_one le_add_diff Pi_def)
  also from f g have "... = (⨁k ∈ {..n}. ⨁i ∈ {..m}. f k ⊗ g i)"
    by (simp cong: finsum_cong
      add: bound.bound [OF bg] finsum_Un_disjoint ivl_disj_int_one Pi_def)
  also from f g have "... = (⨁i ∈ {..n}. f i) ⊗ (⨁i ∈ {..m}. g i)"
    by (simp add: finsum_ldistr diagonal_sum Pi_def,
      simp cong: finsum_cong add: finsum_rdistr Pi_def)
  finally show ?thesis .
```

**qed**

**end**

```
lemma (in UP_ring) const_ring_hom:
  "(λa. monom P a 0) ∈ ring_hom R P"
  by (auto intro!: ring_hom_memI intro: up_eqI simp: monom_mult_is_smult)

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).
    ⨁Si ∈ {..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)"
  by (unfold eval_def, fold P_def) simp

lemma eval_extensional:
  "eval R S phi p ∈ extensional (carrier P)"
  by (unfold eval_def, fold P_def) simp

end
```

The universal property of the polynomial ring

```
locale UP_pre_univ_prop = ring_hom_cring + UP_cring

locale UP_univ_prop = UP_pre_univ_prop +
  fixes s and Eval
  assumes indet_img_carrier [simp, intro]: "s ∈ carrier S"
  defines Eval_def: "Eval == eval R S h s"
```

JE: I have moved the following lemma from Ring.thy and lifted then to the locale `ring_hom_ring` from `ring_hom_cring`.

JE: I was considering using it in `eval_ring_hom`, but that property does not hold for non commutative rings, so maybe it is not that necessary.

```
lemma (in ring_hom_ring) hom_finsum [simp]:
  "f ∈ A → carrier R ⟹
  h (finsum R f A) = finsum S (h ∘ f) A"
  by (induct A rule: infinite_finite_induct, auto simp: Pi_def)

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 (rule ring_hom_memI)
  fix p
  assume R: "p ∈ carrier P"
  then show "eval R S h s p ∈ carrier S"
    by (simp only: eval_on_carrier) (simp add: S Pi_def)
next
  fix p q
  assume R: "p ∈ carrier P" "q ∈ carrier P"
  then show "eval R S h s (p ⊕P q) = eval R S h s p ⊕S eval R S h s
q"
  proof (simp only: eval_on_carrier P.a_closed)
    from S R have
      "(⨁S i∈{..deg R (p ⊕P q)}. h (coeff P (p ⊕P q) i) ⊗S s [^]S i)
=
      (⨁S i∈{..deg R (p ⊕P q)} ∪ {deg R (p ⊕P q)<..max (deg R p) (deg
R q)}.
        h (coeff P (p ⊕P q) i) ⊗S s [^]S i)"
      by (simp cong: S.finsum_cong
        add: deg_aboveD S.finsum_Un_disjoint ivl_disj_int_one Pi_def del:
coeff_add)
    also from R have "... =
        (⨁S i ∈ {..max (deg R p) (deg R q)}.
          h (coeff P (p ⊕P q) i) ⊗S s [^]S i)"
      by (simp add: ivl_disj_un_one)
    also from R S have "... =
      (⨁Si∈{..max (deg R p) (deg R q)}. h (coeff P p i) ⊗S s [^]S i)
⊕S
      (⨁Si∈{..max (deg R p) (deg R q)}. h (coeff P q i) ⊗S s [^]S i)"
      by (simp cong: S.finsum_cong
        add: S.l_distr deg_aboveD ivl_disj_int_one Pi_def)
    also have "... =
        (⨁S i ∈ {..deg R p} ∪ {deg R p<..max (deg R p) (deg R q)}.
          h (coeff P p i) ⊗S s [^]S i) ⊕S
        (⨁S i ∈ {..deg R q} ∪ {deg R q<..max (deg R p) (deg R q)}.
          h (coeff P q i) ⊗S s [^]S i)"
      by (simp only: ivl_disj_un_one max.cobounded1 max.cobounded2)
    also from R S have "... =
      (⨁S i ∈ {..deg R p}. h (coeff P p i) ⊗S s [^]S i) ⊕S
      (⨁S i ∈ {..deg R q}. h (coeff P q i) ⊗S s [^]S i)"
      by (simp cong: S.finsum_cong
        add: deg_aboveD S.finsum_Un_disjoint ivl_disj_int_one Pi_def)
    finally show
      "(⨁Si ∈ {..deg R (p ⊕P q)}. h (coeff P (p ⊕P q) i) ⊗S s [^]S
i) =
```

```
      (⨁Si ∈ {..deg R p}. h (coeff P p i) ⊗S s [^]S i) ⊕S
      (⨁Si ∈ {..deg R q}. h (coeff P q i) ⊗S s [^]S i)" .
  qed
next
  show "eval R S h s 1P = 1S"
    by (simp only: eval_on_carrier UP_one_closed) simp
next
  fix p q
  assume R: "p ∈ carrier P" "q ∈ carrier P"
  then show "eval R S h s (p ⊗P q) = eval R S h s p ⊗S eval R S h s
q"
  proof (simp only: eval_on_carrier UP_mult_closed)
    from R S have
      "(⨁S i ∈ {..deg R (p ⊗P q)}. h (coeff P (p ⊗P q) i) ⊗S s [^]S
i) =
      (⨁S i ∈ {..deg R (p ⊗P q)} ∪ {deg R (p ⊗P q)<..deg R p + deg
R q}.
        h (coeff P (p ⊗P q) i) ⊗S s [^]S i)"
      by (simp cong: S.finsum_cong
        add: deg_aboveD S.finsum_Un_disjoint ivl_disj_int_one Pi_def
        del: coeff_mult)
    also from R have "... =
      (⨁S i ∈ {..deg R p + deg R q}. h (coeff P (p ⊗P q) i) ⊗S s [^]S
i)"
      by (simp only: ivl_disj_un_one deg_mult_ring)
    also from R S have "... =
      (⨁S i ∈ {..deg R p + deg R q}.
        ⨁S k ∈ {..i}.
          h (coeff P p k) ⊗S h (coeff P q (i - k)) ⊗S
          (s [^]S k ⊗S s [^]S (i - k)))"
      by (simp cong: S.finsum_cong add: S.nat_pow_mult Pi_def
        S.m_ac S.finsum_rdistr)
    also from R S have "... =
      (⨁S i∈{..deg R p}. h (coeff P p i) ⊗S s [^]S i) ⊗S
      (⨁S i∈{..deg R q}. h (coeff P q i) ⊗S s [^]S i)"
      by (simp add: S.cauchy_product [THEN sym] bound.intro deg_aboveD
S.m_ac
        Pi_def)
    finally show
      "(⨁S i ∈ {..deg R (p ⊗P q)}. h (coeff P (p ⊗P q) i) ⊗S s [^]S
i) =
      (⨁S i ∈ {..deg R p}. h (coeff P p i) ⊗S s [^]S i) ⊗S
      (⨁S i ∈ {..deg R q}. h (coeff P q i) ⊗S s [^]S i)" .
  qed
qed
```

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"
  by (simp only: eval_on_carrier monom_closed) simp
```

Further properties of the evaluation homomorphism.

The following proof is complicated by the fact that in arbitrary rings one might have **1** = **0**.

```
lemma (in UP_pre_univ_prop) eval_monom1:
  assumes S: "s ∈ carrier S"
  shows "eval R S h s (monom P 1 1) = s"
proof (simp only: eval_on_carrier monom_closed R.one_closed)
   from S have
    "(⨁S i∈{..deg R (monom P 1 1)}. h (coeff P (monom P 1 1) i) ⊗S s
[^]S i) =
    (⨁S i∈{..deg R (monom P 1 1)} ∪ {deg R (monom P 1 1)<..1}.
      h (coeff P (monom P 1 1) i) ⊗S s [^]S i)"
    by (simp cong: S.finsum_cong del: coeff_monom
      add: deg_aboveD S.finsum_Un_disjoint ivl_disj_int_one Pi_def)
  also have "... =
    (⨁S i ∈ {..1}. h (coeff P (monom P 1 1) i) ⊗S s [^]S i)"
    by (simp only: ivl_disj_un_one deg_monom_le R.one_closed)
  also have "... = s"
  proof (cases "s = 0S")
    case True then show ?thesis by (simp add: Pi_def)
  next
    case False then show ?thesis by (simp add: S Pi_def)
  qed
  finally show "(⨁S i ∈ {..deg R (monom P 1 1)}.
    h (coeff P (monom P 1 1) i) ⊗S s [^]S i) = s" .
qed

end
```

Interpretation of ring homomorphism lemmas.

```
sublocale UP_univ_prop < ring_hom_cring P S Eval
  unfolding Eval_def
  by unfold_locales (fast intro: eval_ring_hom)

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 (induct m)
  case 0 from R show ?case by simp
next
  case Suc with R show ?case
    by (simp del: monom_mult add: monom_mult [THEN sym] add.commute)
qed
```

```
lemma (in ring_hom_cring) hom_pow [simp]:
  "x ∈ carrier R ==> h (x [^] n) = h x [^]S (n::nat)"
  by (induct n) simp_all

lemma (in UP_univ_prop) Eval_monom:
  "r ∈ carrier R ==> Eval (monom P r n) = h r ⊗S s [^]S n"
proof -
  assume R: "r ∈ carrier R"
  from R have "Eval (monom P r n) = Eval (monom P r 0 ⊗P (monom P 1 1)
[^]P n)"
    by (simp del: monom_mult add: monom_mult [THEN sym] monom_pow)
  also
  from R eval_monom1 [where s = s, folded Eval_def]
  have "... = h r ⊗S s [^]S n"
    by (simp add: eval_const [where s = s, folded Eval_def])
  finally show ?thesis .
qed

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 -
  interpret UP_univ_prop R S h P s "eval R S h s"
    using UP_pre_univ_prop_axioms P_def R S
    by (auto intro: UP_univ_prop.intro UP_univ_prop_axioms.intro)
  from R
  show ?thesis by (rule Eval_monom)
qed

lemma (in UP_univ_prop) Eval_smult:
  "[| r ∈ carrier R; p ∈ carrier P |] ==> Eval (r ⊙P p) = h r ⊗S Eval
p"
proof -
  assume R: "r ∈ carrier R" and P: "p ∈ carrier P"
  then show ?thesis
    by (simp add: monom_mult_is_smult [THEN sym]
      eval_const [where s = s, folded Eval_def])
qed

lemma ring_hom_cringI:
  assumes "cring R"
    and "cring S"
    and "h ∈ ring_hom R S"
  shows "ring_hom_cring R S h"
  by (fast intro: ring_hom_cring.intro ring_hom_cring_axioms.intro
    cring.axioms assms)

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 -
  interpret ring_hom_cring P S Phi by fact
  interpret ring_hom_cring P S Psi by fact
  have "Phi p =
      Phi (⨁P i ∈ {..deg R p}. monom P (coeff P p i) 0 ⊗P monom P 1
1 [^]P i)"
    by (simp add: up_repr P monom_mult [THEN sym] monom_pow del: monom_mult)
  also
  have "... =
      Psi (⨁P i∈{..deg R p}. monom P (coeff P p i) 0 ⊗P monom P 1 1
[^]P i)"
    by (simp add: Phi Psi P Pi_def comp_def)
  also have "... = Psi p"
    by (simp add: up_repr P monom_mult [THEN sym] monom_pow del: monom_mult)
  finally show ?thesis .
qed

lemma ring_homD:
  assumes Phi: "Phi ∈ ring_hom P S"
  shows "ring_hom_cring P S Phi"
  by unfold_locales (rule Phi)

theorem UP_universal_property:
  assumes S: "s ∈ carrier S"
  shows "∃!Phi. Phi ∈ ring_hom P S ∩ extensional (carrier P) ∧
    Phi (monom P 1 1) = s ∧
    (∀r ∈ carrier R. Phi (monom P r 0) = h r)"
  using S eval_monom1
  apply (auto intro: eval_ring_hom eval_const eval_extensional)
  apply (rule extensionalityI)
  apply (auto intro: UP_hom_unique ring_homD)
  done

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"
  **using** nat_pow_Suc [of x 0] **unfolding** nat_pow_0 [of x] **unfolding** l_one [OF x_in_G] **by** simp

**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 ≠ $\mathbf{0}_P$" **shows** "lcoeff p ≠ $\mathbf{0}$"
  **using** lcoeff_nonzero [OF p_not_zero p_in_R] .

## 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 $\ominus_P$ q) n = coeff P p n ⊖ coeff P q n"
  **by** (simp add: a_minus_def p q)

**lemma** lcoeff_closed [simp]: **assumes** p: "p ∈ carrier P" **shows** "lcoeff p ∈ carrier R"
  **using** coeff_closed [OF p, of "deg R p"] **by** simp

**lemma** deg_smult_decr: **assumes** a_in_R: "a ∈ carrier R" **and** f_in_P: "f ∈ carrier P" **shows** "deg R (a $\odot_P$ f) ≤ deg R f"
  **using** deg_smult_ring [OF a_in_R f_in_P] **by** (cases "a = $\mathbf{0}$", auto)

**lemma** coeff_monom_mult: **assumes** R: "c ∈ carrier R" **and** P: "p ∈ carrier P"
  **shows** "coeff P (monom P c n $\otimes_P$ p) (m + n) = c ⊗ (coeff P p m)"
**proof** -
  **have** "coeff P (monom P c n $\otimes_P$ p) (m + n) = (⨁i∈{..m + n}. (if n = i then c else $\mathbf{0}$) ⊗ coeff P p (m + n - i))"
    **unfolding** coeff_mult [OF monom_closed [OF R, of n] P, of "m + n"] **unfolding** coeff_monom [OF R, of n] **by** simp
  **also have** "(⨁i∈{..m + n}. (if n = i then c else $\mathbf{0}$) ⊗ coeff P p (m + n - i)) =
    (⨁i∈{..m + n}. (if n = i then c ⊗ coeff P p (m + n - i) else $\mathbf{0}$))"
    **using** R.finsum_cong [of "{..m + n}" "{..m + n}" "(λi::nat. (if n = i then c else $\mathbf{0}$) ⊗ coeff P p (m + n - i))"
      "(λi::nat. (if n = i then c ⊗ coeff P p (m + n - i) else $\mathbf{0}$))"]
    **using** coeff_closed [OF P] **unfolding** Pi_def simp_implies_def **using** R **by** auto

```
  also have "... = c ⊗ coeff P p m" using R.finsum_singleton [of n "{..m
+ n}" "(λi. c ⊗ coeff P p (m + n - i))"]
    unfolding Pi_def using coeff_closed [OF P] using P R by auto
  finally show ?thesis by simp
qed

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 -
  have deg_le: "deg R (p ⊕P q) ≤ deg R r"
  proof (rule deg_aboveI)
    fix m
    assume deg_r_le: "deg R r < m"
    show "coeff P (p ⊕P q) m = 0"
    proof -
      have slp: "deg R p < m" and "deg R q < m" using deg_R_p deg_R_q
using deg_r_le by auto
      then have max_sl: "max (deg R p) (deg R q) < m" by simp
      then have "deg R (p ⊕P q) < m" using deg_add [OF p_in_P q_in_P]
by arith
      with deg_R_p deg_R_q show ?thesis using coeff_add [OF p_in_P q_in_P,
of m]
        using deg_aboveD [of "p ⊕P q" m] using p_in_P q_in_P by simp
    qed
  qed (simp add: p_in_P q_in_P)
  moreover have deg_ne: "deg R (p ⊕P q) ≠ deg R r"
  proof (rule ccontr)
    assume nz: "¬ deg R (p ⊕P q) ≠ deg R r" then have deg_eq: "deg
R (p ⊕P q) = deg R r" by simp
    from deg_r_nonzero have r_nonzero: "r ≠ 0P" by (cases "r = 0P",
simp_all)
    have "coeff P (p ⊕P q) (deg R r) = 0R" using coeff_add [OF p_in_P
q_in_P, of "deg R r"] using coeff_R_p_eq_q
      using coeff_closed [OF p_in_P, of "deg R r"] coeff_closed [OF q_in_P,
of "deg R r"] by algebra
    with lcoeff_nonzero [OF r_nonzero r_in_P]  and deg_eq show False
using lcoeff_nonzero [of "p ⊕P q"] using p_in_P q_in_P
      using deg_r_nonzero by (cases "p ⊕P q ≠ 0P", auto)
  qed
  ultimately show ?thesis by simp
qed

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"
  using deg_mult_ring [OF g_in_P monom_closed [OF a_in_R, of "deg R f
- deg R g"]]
  apply (cases "a = 0") using g_in_P apply simp
  using deg_monom [OF _ a_in_R, of "deg R f - deg R g"] using deg_le by
simp

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"
  apply (rule up_eqI) using coeff_monom [OF coeff_closed [OF f_in_P],
of 0 0]
  using f_in_P deg_f using deg_aboveD [of f _] by auto

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"
  using exist by blast
```

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)"
  using f_in_P
proof (induct "deg R f" arbitrary: "f" rule: nat_less_induct)
  case (1 f)
  note f_in_P [simp] = "1.prems"
  let ?pred = "(λ 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))"
  let ?lg = "lcoeff g" and ?lf = "lcoeff f"
  show ?case
  proof (cases "deg R f < deg R g")
    case True
    have "?pred 0P f 0" using True by force
    then show ?thesis by blast
  next
```

```
    case False then have deg_g_le_deg_f: "deg R g ≤ deg R f" by simp
    {
      let ?k = "1::nat"
      let ?f1 = "(g ⊗P (monom P (?lf) (deg R f - deg R g))) ⊕P ⊖P (?lg
⊙P f)"
      let ?q = "monom P (?lf) (deg R f - deg R g)"
      have f1_in_carrier: "?f1 ∈ carrier P" and q_in_carrier: "?q ∈ carrier
P" by simp_all
      show ?thesis
      proof (cases "deg R f = 0")
        case True
        {
          have deg_g: "deg R g = 0" using True using deg_g_le_deg_f by
simp
          have "?pred f 0P 1"
            using deg_zero_impl_monom [OF g_in_P deg_g]
            using sym [OF monom_mult_is_smult [OF coeff_closed [OF g_in_P,
of 0] f_in_P]]
            using deg_g by simp
          then show ?thesis by blast
        }
      next
        case False note deg_f_nzero = False
        {
          have exist: "lcoeff g [^] ?k ⊙P f = g ⊗P ?q ⊕P ⊖P ?f1"
            by (simp add: minus_add r_neg sym [
              OF a_assoc [of "g ⊗P ?q" "⊖P (g ⊗P ?q)" "lcoeff g ⊙P f"]])
          have deg_remainder_l_f: "deg R (⊖P ?f1) < deg R f"
          proof (unfold deg_uminus [OF f1_in_carrier])
            show "deg R ?f1 < deg R f"
            proof (rule deg_lcoeff_cancel)
              show "deg R (⊖P (?lg ⊙P f)) ≤ deg R f"
                using deg_smult_ring [of ?lg f]
                using lcoeff_nonzero2 [OF g_in_P g_not_zero] by simp
              show "deg R (g ⊗P ?q) ≤ deg R f"
                by (simp add: monom_deg_mult [OF f_in_P g_in_P deg_g_le_deg_f,
of ?lf])
              show "coeff P (g ⊗P ?q) (deg R f) = ⊖ coeff P (⊖P (?lg
⊙P f)) (deg R f)"
                unfolding coeff_mult [OF g_in_P monom_closed
                  [OF lcoeff_closed [OF f_in_P],
                    of "deg R f - deg R g"], of "deg R f"]
                unfolding coeff_monom [OF lcoeff_closed
                  [OF f_in_P], of "(deg R f - deg R g)"]
                using R.finsum_cong' [of "{..deg R f}" "{..deg R f}"
                  "(λi. coeff P g i ⊗ (if deg R f - deg R g = deg R f
- i then ?lf else 0))"
                  "(λi. if deg R g = i then coeff P g i ⊗ ?lf else 0)"]
                using R.finsum_singleton [of "deg R g" "{.. deg R f}"
```

```
"(λi. coeff P g i ⊗ ?lf)"]
              unfolding Pi_def using deg_g_le_deg_f by force
          qed (simp_all add: deg_f_nzero)
        qed
        then obtain q' r' k'
          where rem_desc: "?lg [^] (k'::nat) ⊙P (⊖P ?f1) = g ⊗P q'
⊕P r'"
          and rem_deg: "(r' = 0P ∨ deg R r' < deg R g)"
          and q'_in_carrier: "q' ∈ carrier P" and r'_in_carrier: "r'
∈ carrier P"
          using "1.hyps" using f1_in_carrier by blast
        show ?thesis
        proof (rule exI3 [of _ "((?lg [^] k') ⊙P ?q ⊕P q')" r' "Suc
k'"], intro conjI)
          show "(?lg [^] (Suc k')) ⊙P f = g ⊗P ((?lg [^] k') ⊙P ?q
⊕P q') ⊕P r'"
          proof -
            have "(?lg [^] (Suc k')) ⊙P f = (?lg [^] k') ⊙P (g ⊗P
?q ⊕P ⊖P ?f1)"
              using smult_assoc1 [OF _ _ f_in_P] using exist by simp
            also have "... = (?lg [^] k') ⊙P (g ⊗P ?q) ⊕P ((?lg [^]
k') ⊙P ( ⊖P ?f1))"
              using UP_smult_r_distr by simp
            also have "... = (?lg [^] k') ⊙P (g ⊗P ?q) ⊕P (g ⊗P q'
⊕P r')"
              unfolding rem_desc ..
            also have "... = (?lg [^] k') ⊙P (g ⊗P ?q) ⊕P g ⊗P q' ⊕P
r'"
              using sym [OF a_assoc [of "?lg [^] k' ⊙P (g ⊗P ?q)" "g
⊗P q'" "r'"]]
              using r'_in_carrier q'_in_carrier by simp
            also have "... = (?lg [^] k') ⊙P (?q ⊗P g) ⊕P q' ⊗P g ⊕P
r'"
              using q'_in_carrier by (auto simp add: m_comm)
            also have "... = (((?lg [^] k') ⊙P ?q) ⊗P g) ⊕P q' ⊗P g
⊕P r'"
              using smult_assoc2 q'_in_carrier "1.prems" by auto
            also have "... = ((?lg [^] k') ⊙P ?q ⊕P q') ⊗P g ⊕P r'"
              using sym [OF l_distr] and q'_in_carrier by auto
            finally show ?thesis using m_comm q'_in_carrier by auto
          qed
        qed (simp_all add: rem_deg q'_in_carrier r'_in_carrier)
      }
    qed
  }
 qed
qed

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 1_R 1 ⊖_P monom P a 0) = 1"
  (is "deg R ?g = 1")
proof -
  have "deg R ?g ≤ 1"
  proof (rule deg_aboveI)
    fix m
    assume "(1::nat) < m"
    then show "coeff P ?g m = 0"
      using coeff_minus using a by auto algebra
  qed (simp add: a)
  moreover have "deg R ?g ≥ 1"
  proof (rule deg_belowI)
    show "coeff P ?g 1 ≠ 0"
      using a using R.carrier_one_not_zero R_not_trivial by simp algebra
  qed (simp add: a)
  ultimately show ?thesis by simp
qed

lemma lcoeff_monom:
  assumes a: "a ∈ carrier R" and R_not_trivial: "(carrier R ≠ {0})"
  shows "lcoeff (monom P 1_R 1 ⊖_P monom P a 0) = 1"
  using deg_minus_monom [OF a R_not_trivial]
  using coeff_minus a by auto algebra

lemma deg_nzero_nzero:
  assumes deg_p_nzero: "deg R p ≠ 0"
  shows "p ≠ 0_P"
  using deg_zero deg_p_nzero by auto

lemma deg_monom_minus:
  assumes a: "a ∈ carrier R"
  and R_not_trivial: "carrier R ≠ {0}"
  shows "deg R (monom P 1_R 1 ⊖_P monom P a 0) = 1"
  (is "deg R ?g = 1")
proof -
  have "deg R ?g ≤ 1"
  proof (rule deg_aboveI)
    fix m::nat assume "1 < m" then show "coeff P ?g m = 0"
      using coeff_minus [OF monom_closed [OF R.one_closed, of 1] monom_closed
[OF a, of 0], of m]
      using coeff_monom [OF R.one_closed, of 1 m] using coeff_monom [OF
a, of 0 m] by auto algebra
```

```
  qed (simp add: a)
  moreover have "1 ≤ deg R ?g"
  proof (rule deg_belowI)
    show "coeff P ?g 1 ≠ 0"
      using coeff_minus [OF monom_closed [OF R.one_closed, of 1] monom_closed
[OF a, of 0], of 1]
      using coeff_monom [OF R.one_closed, of 1 1] using coeff_monom [OF
a, of 0 1]
      using R_not_trivial using R.carrier_one_not_zero
      by auto algebra
  qed (simp add: a)
  ultimately show ?thesis by simp
qed

lemma eval_monom_expr:
  assumes a: "a ∈ carrier R"
  shows "eval R R id a (monom P 1_R 1 ⊖_P monom P a 0) = 0"
  (is "eval R R id a ?g = _")
proof -
  interpret UP_pre_univ_prop R R id by unfold_locales simp
  have eval_ring_hom: "eval R R id a ∈ ring_hom P R" using eval_ring_hom
[OF a] by simp
  interpret ring_hom_cring P R "eval R R id a" by unfold_locales (rule
eval_ring_hom)
  have mon1_closed: "monom P 1_R 1 ∈ carrier P"
    and mon0_closed: "monom P a 0 ∈ carrier P"
    and min_mon0_closed: "⊖_P monom P a 0 ∈ carrier P"
    using a R.a_inv_closed by auto
  have "eval R R id a ?g = eval R R id a (monom P 1 1) ⊖ eval R R id
a (monom P a 0)"
    by (simp add: a_minus_def mon0_closed)
  also have "... = a ⊖ a"
    using eval_monom [OF R.one_closed a, of 1] using eval_monom [OF a
a, of 0] using a by simp
  also have "... = 0"
    using a by algebra
  finally show ?thesis by simp
qed

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 1_R
1 ⊖_P monom P a 0) ⊗_P q ⊕_P r ∧ (deg R r = 0)"
  (is "∃ q r. (q ∈ carrier P) ∧ (r ∈ carrier P) ∧ f = ?g ⊗_P q ⊕_P r ∧
(deg R r = 0)")
proof -
  let ?g = "monom P 1_R 1 ⊖_P monom P a 0"
  from deg_minus_monom [OF a R_not_trivial]
```

```
  have deg_g_nzero: "deg R ?g ≠ 0" by simp
  have "∃q r (k::nat). q ∈ carrier P ∧ r ∈ carrier P ∧
    lcoeff ?g [^] k ⊙P f = ?g ⊗P q ⊕P r ∧ (r = 0P ∨ deg R r < deg R
?g)"
    using long_div_theorem [OF _ f deg_nzero_nzero [OF deg_g_nzero]] a
    by auto
  then show ?thesis
    unfolding lcoeff_monom [OF a R_not_trivial]
    unfolding deg_monom_minus [OF a R_not_trivial]
    using smult_one [OF f] using deg_zero by force
qed

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 -
  interpret UP_pre_univ_prop R R id P by standard simp
  have eval_ring_hom: "eval R R id a ∈ ring_hom P R"
    using eval_ring_hom [OF a] by simp
  have "eval R R id a f = eval R R id a ?gq ⊕R eval R R id a r"
    unfolding f_expr using ring_hom_add [OF eval_ring_hom] by auto
  also have "... = ((eval R R id a ?g) ⊗ (eval R R id a q)) ⊕R eval R
R id a r"
    using ring_hom_mult [OF eval_ring_hom] by auto
  also have "... = 0 ⊕ eval R R id a r"
    unfolding eval_monom_expr [OF a] using eval_ring_hom
    unfolding ring_hom_def using q unfolding Pi_def by simp
  also have "... = eval R R id a r"
    using eval_ring_hom unfolding ring_hom_def using r unfolding Pi_def
by simp
  finally have eval_eq: "eval R R id a f = eval R R id a r" by simp
  from deg_zero_impl_monom [OF r deg_r_0]
  have "r = monom P (coeff P r 0) 0" by simp
  with eval_const [OF a, of "coeff P r 0"] eval_eq
  show ?thesis by auto
qed

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 -
  from remainder_theorem_exist [OF f a R_not_trivial]
  obtain q r
    where q_r: "q ∈ carrier P ∧ r ∈ carrier P ∧ f = ?g ⊗P q ⊕P r"
    and deg_r: "deg R r = 0" by force
  with remainder_theorem_expression [OF f a _ _ R_not_trivial, of q r]
  show ?thesis by auto
qed

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"
  using assms
  by (auto intro!: UP_pre_univ_prop.intro ring_hom_cring.intro
    ring_hom_cring_axioms.intro UP_cring.intro)

definition
  INTEG :: "int ring"
  where "INTEG = (|carrier = UNIV, mult = (*), one = 1, zero = 0, add
= (+)|)"

lemma INTEG_cring: "cring INTEG"
  by (unfold INTEG_def) (auto intro!: cringI abelian_groupI comm_monoidI
    left_minus distrib_right)

lemma INTEG_id_eval:
  "UP_pre_univ_prop INTEG INTEG id"
  by (fast intro: UP_pre_univ_propI INTEG_cring id_ring_hom)
```

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"
  using INTEG_id_eval by simp_all

lemma INTEG_closed [intro, simp]:
  "z ∈ carrier INTEG"
  by (unfold INTEG_def) simp

lemma INTEG_mult [simp]:
  "mult INTEG z w = z * w"
  by (unfold INTEG_def) simp
```

```
lemma INTEG_pow [simp]:
  "pow INTEG z n = z ^ n"
  by (induct n) (simp_all add: INTEG_def nat_pow_def)

lemma "eval INTEG INTEG id 10 (monom (UP INTEG) 5 2) = 500"
  by (simp add: INTEG.eval_monom)

end
```

# 16 Generated Groups

```
theory Generated_Groups
  imports Group Coset

begin
```

## 16.1 Generated Groups

```
inductive_set generate :: "('a, 'b) monoid_scheme ⇒ 'a set ⇒ 'a set"
  for G and H where
    one:  "1G ∈ generate G H"
  | incl: "h ∈ H ⟹ h ∈ generate G H"
  | inv:  "h ∈ H ⟹ invG h ∈ generate G H"
  | eng:  "h1 ∈ generate G H ⟹ h2 ∈ generate G H ⟹ h1 ⊗G h2 ∈ generate
G H"
```

### 16.1.1 Basic Properties

```
lemma (in group) generate_consistent:
  assumes "K ⊆ H" "subgroup H G" shows "generate (G ⦇ carrier := H ⦈)
K = generate G K"
proof
  show "generate (G ⦇ carrier := H ⦈) K ⊆ generate G K"
  proof
    fix h assume "h ∈ generate (G ⦇ carrier := H ⦈) K" thus "h ∈ generate
G K"
    proof (induction, simp add: one, simp_all add: incl[of _ K G] eng)
      case inv thus ?case
        using m_inv_consistent assms generate.inv[of _ K G] by auto
    qed
  qed
next
  show "generate G K ⊆ generate (G ⦇ carrier := H ⦈) K"
  proof
    note gen_simps = one incl eng
    fix h assume "h ∈ generate G K" thus "h ∈ generate (G ⦇ carrier :=
H ⦈) K"
      using gen_simps[where ?G = "G ⦇ carrier := H ⦈"]
    proof (induction, auto)
```

```
      fix h assume "h ∈ K" thus "inv h ∈ generate (G ⦇ carrier := H ⦈)
K"
        using m_inv_consistent assms generate.inv[of h K "G ⦇ carrier
:= H ⦈"] by auto
    qed
  qed
qed

lemma (in group) generate_in_carrier:
  assumes "H ⊆ carrier G" and "h ∈ generate G H" shows "h ∈ carrier
G"
  using assms(2,1) by (induct h rule: generate.induct) (auto)

lemma (in group) generate_incl:
  assumes "H ⊆ carrier G" shows "generate G H ⊆ carrier G"
  using generate_in_carrier[OF assms(1)] by auto

lemma (in group) generate_m_inv_closed:
  assumes "H ⊆ carrier G" and "h ∈ generate G H" shows "(inv h) ∈ generate
G H"
  using assms(2,1)
proof (induction rule: generate.induct, auto simp add: one inv incl)
  fix h1 h2
  assume h1: "h1 ∈ generate G H" "inv h1 ∈ generate G H"
     and h2: "h2 ∈ generate G H" "inv h2 ∈ generate G H"
  hence "inv (h1 ⊗ h2) = (inv h2) ⊗ (inv h1)"
    by (meson assms generate_in_carrier group.inv_mult_group is_group)
  thus "inv (h1 ⊗ h2) ∈ generate G H"
    using generate.eng[OF h2(2) h1(2)] by simp
qed

lemma (in group) generate_is_subgroup:
  assumes "H ⊆ carrier G" shows "subgroup (generate G H) G"
  using subgroup.intro[OF generate_incl eng one generate_m_inv_closed]
assms by auto

lemma (in group) mono_generate:
  assumes "K ⊆ H" shows "generate G K ⊆ generate G H"
proof
  fix h assume "h ∈ generate G K" thus "h ∈ generate G H"
    using assms by (induction) (auto simp add: one incl inv eng)
qed

lemma (in group) generate_subgroup_incl:
  assumes "K ⊆ H" "subgroup H G" shows "generate G K ⊆ H"
  using group.generate_incl[OF subgroup_imp_group[OF assms(2)], of K]
assms(1)
  by (simp add: generate_consistent[OF assms])
```

```
lemma (in group) generate_minimal:
  assumes "H ⊆ carrier G" shows "generate G H = ⋂ { H'. subgroup H'
G ∧ H ⊆ H' }"
  using generate_subgroup_incl generate_is_subgroup[OF assms] incl[of
_ H] by blast

lemma (in group) generateI:
  assumes "subgroup E G" "H ⊆ E" and "⋀K. ⟦ subgroup K G; H ⊆ K ⟧ ⟹
E ⊆ K"
  shows "E = generate G H"
proof -
  have subset: "H ⊆ carrier G"
    using subgroup.subset assms by auto
  show ?thesis
    using assms unfolding generate_minimal[OF subset] by blast
qed

lemma (in group) normal_generateI:
  assumes "H ⊆ carrier G" and "⋀h g. ⟦ h ∈ H; g ∈ carrier G ⟧ ⟹ g
⊗ h ⊗ (inv g) ∈ H"
  shows "generate G H ◁ G"
proof (rule normal_invI[OF generate_is_subgroup[OF assms(1)]])
  fix g h assume g: "g ∈ carrier G" show "h ∈ generate G H ⟹ g ⊗ h
⊗ (inv g) ∈ generate G H"
  proof (induct h rule: generate.induct)
    case one thus ?case
      using g generate.one by auto
  next
    case incl show ?case
      using generate.incl[OF assms(2)[OF incl g]] .
  next
    case (inv h)
    hence h: "h ∈ carrier G"
      using assms(1) by auto
    hence "inv (g ⊗ h ⊗ (inv g)) = g ⊗ (inv h) ⊗ (inv g)"
      using g by (simp add: inv_mult_group m_assoc)
    thus ?case
      using generate_m_inv_closed[OF assms(1) generate.incl[OF assms(2)[OF
inv g]]] by simp
  next
    case (eng h1 h2)
    note in_carrier = eng(1,3)[THEN generate_in_carrier[OF assms(1)]]
    have "g ⊗ (h1 ⊗ h2) ⊗ inv g = (g ⊗ h1 ⊗ inv g) ⊗ (g ⊗ h2 ⊗ inv
g)"
      using in_carrier g by (simp add: inv_solve_left m_assoc)
    thus ?case
      using generate.eng[OF eng(2,4)] by simp
  qed
qed
```

```
lemma (in group) subgroup_int_pow_closed:
  assumes "subgroup H G" "h ∈ H" shows "h [^] (k :: int) ∈ H"
  using group.int_pow_closed[OF subgroup_imp_group[OF assms(1)]] assms(2)
  unfolding int_pow_consistent[OF assms] by simp

lemma (in group) generate_pow:
  assumes "a ∈ carrier G" shows "generate G { a } = { a [^] (k :: int)
| k. k ∈ UNIV }"
proof
  show "{ a [^] (k :: int) | k. k ∈ UNIV } ⊆ generate G { a }"
    using subgroup_int_pow_closed[OF generate_is_subgroup[of "{ a }"]
incl[of a]] assms by auto
next
  show "generate G { a } ⊆ { a [^] (k :: int) | k. k ∈ UNIV }"
  proof
    fix h assume "h ∈ generate G { a }" hence "∃k :: int. h = a [^] k"
    proof (induction)
      case one
      then show ?case
        using int_pow_0 [of G] by metis
    next
      case (incl h)
      then show ?case
        by (metis assms int_pow_1 singletonD)
    next
      case (inv h)
      then show ?case
        by (metis assms int_pow_1 int_pow_neg singletonD)
    next
      case (eng h1 h2)
      then show ?case
        using assms by (metis int_pow_mult)
    qed
    then show "h ∈ { a [^] (k :: int) | k. k ∈ UNIV }"
      by blast
  qed
qed

corollary (in group) generate_one: "generate G { 𝟭 } = { 𝟭 }"
  using generate_pow[of "𝟭", OF one_closed] by simp

corollary (in group) generate_empty: "generate G {} = { 𝟭 }"
  using mono_generate[of "{}" "{ 𝟭 }"] generate.one unfolding generate_one
by auto

lemma (in group_hom)
  "subgroup K G ⟹ subgroup (h ` K) H"
  using subgroup_img_is_subgroup by auto
```

```
lemma (in group_hom) generate_img:
  assumes "K ⊆ carrier G" shows "generate H (h ` K) = h ` (generate
G K)"
proof
  have "h ` K ⊆ h ` (generate G K)"
    using incl[of _ K G] by auto
  thus "generate H (h ` K) ⊆ h ` (generate G K)"
    using generate_subgroup_incl subgroup_img_is_subgroup[OF G.generate_is_subgroup[OF
assms]] by auto
next
  show "h ` (generate G K) ⊆ generate H (h ` K)"
  proof
    fix a assume "a ∈ h ` (generate G K)"
    then obtain k where "k ∈ generate G K" "a = h k"
      by blast
    show "a ∈ generate H (h ` K)"
      using ‹k ∈ generate G K› unfolding ‹a = h k›
    proof (induct k, auto simp add: generate.one[of H] generate.incl[of
_ "h ` K" H])
      case (inv k) show ?case
        using assms generate.inv[of "h k" "h ` K" H] inv by auto
    next
      case eng show ?case
        using generate.eng[OF eng(2,4)] eng(1,3)[THEN G.generate_in_carrier[OF
assms]] by auto
    qed
  qed
qed
```

## 16.2 Derived Subgroup

### 16.2.1 Definitions

```
abbreviation derived_set :: "('a, 'b) monoid_scheme ⇒ 'a set ⇒ 'a set"
  where "derived_set G H ≡
          ⋃h1 ∈ H. (⋃h2 ∈ H. { h1 ⊗G h2 ⊗G (invG h1) ⊗G (invG h2)
})"

definition derived :: "('a, 'b) monoid_scheme ⇒ 'a set ⇒ 'a set" where
  "derived G H = generate G (derived_set G H)"
```

### 16.2.2 Basic Properties

```
lemma (in group) derived_set_incl:
  assumes "K ⊆ H" "subgroup H G" shows "derived_set G K ⊆ H"
  using assms(1) subgroupE(3-4)[OF assms(2)] by (auto simp add: subset_iff)

lemma (in group) derived_incl:
  assumes "K ⊆ H" "subgroup H G" shows "derived G K ⊆ H"
```

```
  using generate_subgroup_incl[OF derived_set_incl] assms unfolding derived_def
by auto

lemma (in group) derived_set_in_carrier:
  assumes "H ⊆ carrier G" shows "derived_set G H ⊆ carrier G"
  using derived_set_incl[OF assms subgroup_self] .

lemma (in group) derived_in_carrier:
  assumes "H ⊆ carrier G" shows "derived G H ⊆ carrier G"
  using derived_incl[OF assms subgroup_self] .

lemma (in group) exp_of_derived_in_carrier:
  assumes "H ⊆ carrier G" shows "(derived G ^^ n) H ⊆ carrier G"
  using assms derived_in_carrier by (induct n) (auto)

lemma (in group) derived_is_subgroup:
  assumes "H ⊆ carrier G" shows "subgroup (derived G H) G"
  unfolding derived_def using generate_is_subgroup[OF derived_set_in_carrier[OF
assms]] .

lemma (in group) exp_of_derived_is_subgroup:
  assumes "subgroup H G" shows "subgroup ((derived G ^^ n) H) G"
  using assms derived_is_subgroup subgroup.subset by (induct n) (auto)

lemma (in group) exp_of_derived_is_subgroup':
  assumes "H ⊆ carrier G" shows "subgroup ((derived G ^^ (Suc n)) H)
G"
  using assms derived_is_subgroup[OF subgroup.subset] derived_is_subgroup
by (induct n) (auto)

lemma (in group) mono_derived_set:
  assumes "K ⊆ H" shows "derived_set G K ⊆ derived_set G H"
  using assms by auto

lemma (in group) mono_derived:
  assumes "K ⊆ H" shows "derived G K ⊆ derived G H"
  unfolding derived_def using mono_generate[OF mono_derived_set[OF assms]]
.

lemma (in group) mono_exp_of_derived:
  assumes "K ⊆ H" shows "(derived G ^^ n) K ⊆ (derived G ^^ n) H"
  using assms mono_derived by (induct n) (auto)

lemma (in group) derived_set_consistent:
  assumes "K ⊆ H" "subgroup H G" shows "derived_set (G ⦇ carrier :=
H ⦈) K = derived_set G K"
  using m_inv_consistent[OF assms(2)] assms(1) by (auto simp add: subset_iff)

lemma (in group) derived_consistent:
```

```
  assumes "K ⊆ H" "subgroup H G" shows "derived (G ⦇ carrier := H ⦈)
K = derived G K"
  using generate_consistent[OF derived_set_incl] derived_set_consistent
assms by (simp add: derived_def)

lemma (in comm_group) derived_eq_singleton:
  assumes "H ⊆ carrier G" shows "derived G H = { 𝟭 }"
proof (cases "derived_set G H = {}")
  case True show ?thesis
    using generate_empty unfolding derived_def True by simp
next
  case False
  have aux_lemma: "h ∈ derived_set G H ⟹ h = 𝟭" for h
    using assms by (auto simp add: subset_iff)
       (metis (no_types, lifting) m_comm m_closed inv_closed inv_solve_right
l_inv l_inv_ex)
  have "derived_set G H = { 𝟭 }"
  proof
    show "derived_set G H ⊆ { 𝟭 }"
      using aux_lemma by auto
  next
    obtain h where h: "h ∈ derived_set G H"
      using False by blast
    thus "{ 𝟭 } ⊆ derived_set G H"
      using aux_lemma[OF h] by auto
  qed
  thus ?thesis
    using generate_one unfolding derived_def by auto
qed

lemma (in group) derived_is_normal:
  assumes "H ⊲ G" shows "derived G H ⊲ G"
proof -
  interpret H: normal H G
    using assms .

  show ?thesis
    unfolding derived_def
  proof (rule normal_generateI[OF derived_set_in_carrier[OF H.subset]])
    fix h g assume "h ∈ derived_set G H" and g: "g ∈ carrier G"
    then obtain h1 h2 where h: "h1 ∈ H" "h2 ∈ H" "h = h1 ⊗ h2 ⊗ inv
h1 ⊗ inv h2"
      by auto
    hence in_carrier: "h1 ∈ carrier G" "h2 ∈ carrier G" "g ∈ carrier
G"
      using H.subset g by auto
    have "g ⊗ h ⊗ inv g =
           g ⊗ h1 ⊗ (inv g ⊗ g) ⊗ h2 ⊗ (inv g ⊗ g) ⊗ inv h1 ⊗ (inv
g ⊗ g) ⊗ inv h2 ⊗ inv g"
```

```
      unfolding h(3) by (simp add: in_carrier m_assoc)
    also have " ... =
          (g ⊗ h1 ⊗ inv g) ⊗ (g ⊗ h2 ⊗ inv g) ⊗ (g ⊗ inv h1 ⊗ inv
g) ⊗ (g ⊗ inv h2 ⊗ inv g)"
      using in_carrier m_assoc inv_closed m_closed by presburger
    finally have "g ⊗ h ⊗ inv g =
          (g ⊗ h1 ⊗ inv g) ⊗ (g ⊗ h2 ⊗ inv g) ⊗ inv (g ⊗ h1 ⊗ inv
g) ⊗ inv (g ⊗ h2 ⊗ inv g)"
      by (simp add: in_carrier inv_mult_group m_assoc)
    thus "g ⊗ h ⊗ inv g ∈ derived_set G H"
      using h(1-2)[THEN H.inv_op_closed2[OF g]] by auto
  qed
qed

lemma (in group) normal_self: "carrier G ◁ G"
  by (rule normal_invI[OF subgroup_self], simp)

corollary (in group) derived_self_is_normal: "derived G (carrier G) ◁
G"
  using derived_is_normal[OF normal_self] .

corollary (in group) derived_subgroup_is_normal:
  assumes "subgroup H G" shows "derived G H ◁ G ⦇ carrier := H ⦈"
  using group.derived_self_is_normal[OF subgroup_imp_group[OF assms]]
        derived_consistent[OF _ assms]
  by simp

corollary (in group) derived_quot_is_group: "group (G Mod (derived G (carrier
G)))"
  using normal.factorgroup_is_group[OF derived_self_is_normal] by auto

lemma (in group) derived_quot_is_comm_group: "comm_group (G Mod (derived
G (carrier G)))"
proof (rule group.group_comm_groupI[OF derived_quot_is_group], simp add:
FactGroup_def)
  interpret DG: normal "derived G (carrier G)" G
    using derived_self_is_normal .

  fix H K assume "H ∈ rcosets derived G (carrier G)" and "K ∈ rcosets
derived G (carrier G)"
  then obtain g1 g2
    where g1: "g1 ∈ carrier G" "H = derived G (carrier G) #> g1"
      and g2: "g2 ∈ carrier G" "K = derived G (carrier G) #> g2"
    unfolding RCOSETS_def by auto
  hence "H <#> K = derived G (carrier G) #> (g1 ⊗ g2)"
    by (simp add: DG.rcos_sum)
  also have " ... = derived G (carrier G) #> (g2 ⊗ g1)"
  proof -
    have "derived G (carrier G) #> (g1 ⊗ g2) ⊆ derived G (carrier G)
```

```
#> (g2 ⊗ g1)"
      if g1: "g1 ∈ carrier G" and g2: "g2 ∈ carrier G" for g1 g2
    proof
      fix h assume "h ∈ derived G (carrier G) #> (g1 ⊗ g2)"
      then obtain g' where h: "g' ∈ carrier G" "g' ∈ derived G (carrier
G)" "h = g' ⊗ (g1 ⊗ g2)"
        using DG.subset unfolding r_coset_def by auto
      hence "h = g' ⊗ (g1 ⊗ g2) ⊗ (inv g1 ⊗ inv g2 ⊗ g2 ⊗ g1)"
        using g1 g2 by (simp add: m_assoc)
      hence "h = (g' ⊗ (g1 ⊗ g2 ⊗ inv g1 ⊗ inv g2)) ⊗ (g2 ⊗ g1)"
        using h(1) g1 g2 inv_closed m_assoc m_closed by presburger
      moreover have "g1 ⊗ g2 ⊗ inv g1 ⊗ inv g2 ∈ derived G (carrier
G)"
        using incl[of _ "derived_set G (carrier G)"] g1 g2 unfolding derived_def
by blast
      hence "g' ⊗ (g1 ⊗ g2 ⊗ inv g1 ⊗ inv g2) ∈ derived G (carrier
G)"
        using DG.m_closed[OF h(2)] by simp
      ultimately show "h ∈ derived G (carrier G) #> (g2 ⊗ g1)"
        unfolding r_coset_def by blast
    qed
    thus ?thesis
      using g1(1) g2(1) by auto
  qed
  also have " ... = K <#> H"
    by (simp add: g1 g2 DG.rcos_sum)
  finally show "H <#> K = K <#> H" .
qed

corollary (in group) derived_quot_of_subgroup_is_comm_group:
  assumes "subgroup H G" shows "comm_group ((G ⦇ carrier := H ⦈) Mod
(derived G H))"
  using group.derived_quot_is_comm_group[OF subgroup_imp_group[OF assms]]
        derived_consistent[OF _ assms]
  by simp

proposition (in group) derived_minimal:
  assumes "H ◁ G" and "comm_group (G Mod H)" shows "derived G (carrier
G) ⊆ H"
proof -
  interpret H: normal H G
    using assms(1) .

  show ?thesis
    unfolding derived_def
  proof (rule generate_subgroup_incl[OF _ H.subgroup_axioms])
    show "derived_set G (carrier G) ⊆ H"
    proof
      fix h assume "h ∈ derived_set G (carrier G)"
```

```
      then obtain g1 g2 where h: "g1 ∈ carrier G" "g2 ∈ carrier G" "h
= g1 ⊗ g2 ⊗ inv g1 ⊗ inv g2"
        by auto
      have "H #> (g1 ⊗ g2) = (H #> g1) <#> (H #> g2)"
        by (simp add: h(1-2) H.rcos_sum)
      also have " ... = (H #> g2) <#> (H #> g1)"
        using comm_groupE(4)[OF assms(2)] h(1-2) unfolding FactGroup_def
RCOSETS_def by auto
      also have " ... = H #> (g2 ⊗ g1)"
        by (simp add: h(1-2) H.rcos_sum)
      finally have "H #> (g1 ⊗ g2) = H #> (g2 ⊗ g1)" .
      then obtain h' where "h' ∈ H" "1 ⊗ (g1 ⊗ g2) = h' ⊗ (g2 ⊗ g1)"
        using H.one_closed unfolding r_coset_def by blast
      thus "h ∈ H"
        using h m_assoc by auto
    qed
  qed
qed

proposition (in group) derived_of_subgroup_minimal:
  assumes "K ◁ G ⦇ carrier := H ⦈" "subgroup H G" and "comm_group ((G
⦇ carrier := H ⦈) Mod K)"
  shows "derived G H ⊆ K"
  using group.derived_minimal[OF subgroup_imp_group[OF assms(2)] assms(1,3)]
        derived_consistent[OF _ assms(2)]
  by simp

lemma (in group_hom) derived_img:
  assumes "K ⊆ carrier G" shows "derived H (h ` K) = h ` (derived G
K)"
proof -
  have "derived_set H (h ` K) = h ` (derived_set G K)"
  proof
    show "derived_set H (h ` K) ⊆ h ` derived_set G K"
    proof
      fix a assume "a ∈ derived_set H (h ` K)"
      then obtain k1 k2
        where "k1 ∈ K" "k2 ∈ K" "a = (h k1) ⊗H (h k2) ⊗H invH (h k1)
⊗H invH (h k2)"
        by auto
      hence "a = h (k1 ⊗ k2 ⊗ inv k1 ⊗ inv k2)"
        using assms by (simp add: subset_iff)
      from this ‹k1 ∈ K› and ‹k2 ∈ K› show "a ∈ h ` derived_set G
K" by auto
    qed
  next
    show "h ` (derived_set G K) ⊆ derived_set H (h ` K)"
    proof
      fix a assume "a ∈ h ` (derived_set G K)"
```

```
      then obtain k1 k2 where "k1 ∈ K" "k2 ∈ K" "a = h (k1 ⊗ k2 ⊗ inv
k1 ⊗ inv k2)"
        by auto
      hence "a = (h k1) ⊗H (h k2) ⊗H invH (h k1) ⊗H invH (h k2)"
        using assms by (simp add: subset_iff)
      from this ‹k1 ∈ K› and ‹k2 ∈ K› show "a ∈ derived_set H (h ‘
K)" by auto
    qed
  qed
  thus ?thesis
    unfolding derived_def using generate_img[OF G.derived_set_in_carrier[OF
assms]] by simp
qed

lemma (in group_hom) exp_of_derived_img:
  assumes "K ⊆ carrier G" shows "(derived H ^^ n) (h ‘ K) = h ‘ ((derived
G ^^ n) K)"
  using derived_img[OF G.exp_of_derived_in_carrier[OF assms]] by (induct
n) (auto)
```

### 16.2.3 Generated subgroup of a group

```
definition subgroup_generated :: "('a, 'b) monoid_scheme ⇒ 'a set ⇒ ('a,
'b) monoid_scheme"
  where "subgroup_generated G S = G(|carrier := generate G (carrier G
∩ S)|)"

lemma carrier_subgroup_generated: "carrier (subgroup_generated G S) =
generate G (carrier G ∩ S)"
  by (auto simp: subgroup_generated_def)

lemma (in group) subgroup_generated_subset_carrier_subset:
   "S ⊆ carrier G ⟹ S ⊆ carrier(subgroup_generated G S)"
  by (simp add: Int_absorb1 carrier_subgroup_generated generate.incl subsetI)

lemma (in group) subgroup_generated_minimal:
   "⟦subgroup H G; S ⊆ H⟧ ⟹ carrier(subgroup_generated G S) ⊆ H"
  by (simp add: carrier_subgroup_generated generate_subgroup_incl le_infI2)

lemma (in group) carrier_subgroup_generated_subset:
   "carrier (subgroup_generated G A) ⊆ carrier G"
  apply (clarsimp simp: carrier_subgroup_generated)
  by (meson Int_lower1 generate_in_carrier)

lemma (in group) group_subgroup_generated [simp]: "group (subgroup_generated
G S)"
    unfolding subgroup_generated_def
    by (simp add: generate_is_subgroup subgroup_imp_group)
```

```
lemma (in group) abelian_subgroup_generated:
  assumes "comm_group G"
  shows "comm_group (subgroup_generated G S)" (is "comm_group ?GS")
proof (rule group.group_comm_groupI)
  show "Group.group ?GS"
    by simp
next
  fix x y
  assume "x ∈ carrier ?GS" "y ∈ carrier ?GS"
  with assms show "x ⊗?GS y = y ⊗?GS x"
    apply (simp add: subgroup_generated_def)
    by (meson Int_lower1 comm_groupE(4) generate_in_carrier)
qed

lemma (in group) subgroup_of_subgroup_generated:
  assumes "H ⊆ B" "subgroup H G"
    shows "subgroup H (subgroup_generated G B)"
proof unfold_locales
  fix x
  assume "x ∈ H"
  with assms show "inv_subgroup_generated G B x ∈ H"
    unfolding subgroup_generated_def
    by (metis IntI Int_commute Int_lower2 generate.incl generate_is_subgroup
m_inv_consistent subgroup_def subsetCE)
next
  show "H ⊆ carrier (subgroup_generated G B)"
    using assms apply (auto simp: carrier_subgroup_generated)
    by (metis Int_iff generate.incl inf.orderE subgroup.mem_carrier)
qed (use assms in ‹auto simp: subgroup_generated_def subgroup_def›)

lemma carrier_subgroup_generated_alt:
  assumes "Group.group G" "S ⊆ carrier G"
  shows "carrier (subgroup_generated G S) = ⋂{H. subgroup H G ∧ carrier
G ∩ S ⊆ H}"
  using assms by (auto simp: group.generate_minimal subgroup_generated_def)

lemma one_subgroup_generated [simp]: "1_subgroup_generated G S = 1_G"
  by (auto simp: subgroup_generated_def)

lemma mult_subgroup_generated [simp]: "mult (subgroup_generated G S)
= mult G"
  by (auto simp: subgroup_generated_def)

lemma (in group) inv_subgroup_generated [simp]:
  assumes "f ∈ carrier (subgroup_generated G S)"
  shows "inv_subgroup_generated G S f = inv f"
proof (rule group.inv_equality)
  show "Group.group (subgroup_generated G S)"
    by simp
```

```
  have [simp]: "f ∈ carrier G"
    by (metis Int_lower1 assms carrier_subgroup_generated generate_in_carrier)
  show "inv f ⊗subgroup_generated G S f = 1subgroup_generated G S"
    by (simp add: assms)
  show "f ∈ carrier (subgroup_generated G S)"
    using assms by (simp add: generate.incl subgroup_generated_def)
  show "inv f ∈ carrier (subgroup_generated G S)"
    using assms by (simp add: subgroup_generated_def generate_m_inv_closed)
qed

lemma subgroup_generated_restrict [simp]:
   "subgroup_generated G (carrier G ∩ S) = subgroup_generated G S"
  by (simp add: subgroup_generated_def)

lemma (in subgroup) carrier_subgroup_generated_subgroup [simp]:
  "carrier (subgroup_generated G H) = H"
  by (auto simp: generate.incl carrier_subgroup_generated elim: generate.induct)

lemma (in group) subgroup_subgroup_generated_iff:
   "subgroup H (subgroup_generated G B) ⟷ subgroup H G ∧ H ⊆ carrier(subgroup_generated
G B)"
 (is "?lhs = ?rhs")
proof
  assume L: ?lhs
  then have Hsub: "H ⊆ generate G (carrier G ∩ B)"
    by (simp add: subgroup_def subgroup_generated_def)
  then have H: "H ⊆ carrier G" "H ⊆ carrier(subgroup_generated G B)"
    unfolding carrier_subgroup_generated
    using generate_incl by blast+
  with Hsub have "subgroup H G"
    by (metis Int_commute Int_lower2 L carrier_subgroup_generated generate_consistent
        generate_is_subgroup inf.orderE subgroup.carrier_subgroup_generated_subgroup
subgroup_generated_def)
  with H show ?rhs
    by blast
next
  assume ?rhs
  then show ?lhs
    by (simp add: generate_is_subgroup subgroup_generated_def subgroup_incl)
qed

lemma (in group) subgroup_subgroup_generated:
   "subgroup (carrier(subgroup_generated G S)) G"
  using group.subgroup_self group_subgroup_generated subgroup_subgroup_generated_iff
by blast

lemma pow_subgroup_generated:
   "pow (subgroup_generated G S) = (pow G :: 'a ⇒ nat ⇒ 'a)"
proof -
```

```
  have "x [^]subgroup_generated G S n = x [^]G n" for x and n::nat
    by (induction n) auto
  then show ?thesis
    by force
qed

lemma (in group) subgroup_generated2 [simp]: "subgroup_generated (subgroup_generated
G S) S = subgroup_generated G S"
proof -
  have *: "⋀A. carrier G ∩ A ⊆ carrier (subgroup_generated (subgroup_generated
G A) A)"
    by (metis (no_types, opaque_lifting) Int_assoc carrier_subgroup_generated
generate.incl inf.order_iff subset_iff)
  show ?thesis
  apply (auto intro!: monoid.equality)
    using group.carrier_subgroup_generated_subset group_subgroup_generated
apply blast
     apply (metis (no_types, opaque_lifting) "*" group.subgroup_subgroup_generated
group_subgroup_generated subgroup_generated_minimal
        subgroup_generated_restrict subgroup_subgroup_generated_iff subset_eq)
    apply (simp add: subgroup_generated_def)
    done
qed

lemma (in group) int_pow_subgroup_generated:
  fixes n::int
  assumes "x ∈ carrier (subgroup_generated G S)"
  shows "x [^]subgroup_generated G S n = x [^]G n"
proof -
  have "x [^] nat (- n) ∈ carrier (subgroup_generated G S)"
    by (metis assms group.is_monoid group_subgroup_generated monoid.nat_pow_closed
pow_subgroup_generated)
  then show ?thesis
    by (metis group.inv_subgroup_generated int_pow_def2 is_group pow_subgroup_generated)
qed

lemma kernel_from_subgroup_generated [simp]:
  "subgroup S G ⟹ kernel (subgroup_generated G S) H f = kernel G H f
∩ S"
  using subgroup.carrier_subgroup_generated_subgroup subgroup.subset
  by (fastforce simp add: kernel_def set_eq_iff)

lemma kernel_to_subgroup_generated [simp]:
  "kernel G (subgroup_generated H S) f = kernel G H f"
  by (simp add: kernel_def)
```

## 16.3 And homomorphisms

```
lemma (in group) hom_from_subgroup_generated:
```

```
   "h ∈ hom G H ⟹ h ∈ hom(subgroup_generated G A) H"
  apply (simp add: hom_def carrier_subgroup_generated Pi_iff)
  apply (metis group.generate_in_carrier inf_le1 is_group)
  done

lemma hom_into_subgroup:
   "⟦h ∈ hom G G'; h ` (carrier G) ⊆ H⟧ ⟹ h ∈ hom G (subgroup_generated
G' H)"
  by (auto simp: hom_def carrier_subgroup_generated Pi_iff generate.incl
image_subset_iff)

lemma hom_into_subgroup_eq_gen:
    "group G ⟹
     h ∈ hom K (subgroup_generated G H)
 ⟷ h ∈ hom K G ∧ h ` (carrier K) ⊆ carrier(subgroup_generated G H)"
  using group.carrier_subgroup_generated_subset [of G H] by (auto simp:
hom_def)

lemma hom_into_subgroup_eq:
   "⟦subgroup H G; group G⟧
    ⟹ (h ∈ hom K (subgroup_generated G H) ⟷ h ∈ hom K G ∧ h ` (carrier
K) ⊆ H)"
  by (simp add: hom_into_subgroup_eq_gen image_subset_iff subgroup.carrier_subgroup_generat

lemma (in group_hom) hom_between_subgroups:
  assumes "h ` A ⊆ B"
  shows "h ∈ hom (subgroup_generated G A) (subgroup_generated H B)"
proof -
  have [simp]: "group G" "group H"
    by (simp_all add: G.is_group H.is_group)
  have "x ∈ generate G (carrier G ∩ A) ⟹ h x ∈ generate H (carrier
H ∩ B)" for x
  proof (induction x rule: generate.induct)
    case (incl h)
    then show ?case
      by (meson IntE IntI assms generate.incl hom_closed image_subset_iff)
  next
    case (inv h)
    then show ?case
      by (metis G.inv_closed G.inv_inv IntE IntI assms generate.simps
hom_inv image_subset_iff local.inv_closed)
  next
    case (eng h1 h2)
    then show ?case
      by (metis G.generate_in_carrier generate.simps inf.cobounded1 local.hom_mult)
  qed (auto simp: generate.intros)
  then have "h ` carrier (subgroup_generated G A) ⊆ carrier (subgroup_generated
H B)"
    using group.carrier_subgroup_generated_subset [of G A]
```

```
    by (auto simp: carrier_subgroup_generated)
  then show ?thesis
    by (simp add: hom_into_subgroup_eq_gen group.hom_from_subgroup_generated
homh)
qed

lemma (in group_hom) subgroup_generated_by_image:
  assumes "S ⊆ carrier G"
  shows "carrier (subgroup_generated H (h ` S)) = h ` (carrier(subgroup_generated
G S))"
proof
  have "x ∈ generate H (carrier H ∩ h ` S) ⟹ x ∈ h ` generate G (carrier
G ∩ S)" for x
  proof (erule generate.induct)
    show "1_H ∈ h ` generate G (carrier G ∩ S)"
      using generate.one by force
  next
    fix f
    assume "f ∈ carrier H ∩ h ` S"
    with assms show "f ∈ h ` generate G (carrier G ∩ S)" "inv_H f ∈ h
` generate G (carrier G ∩ S)"
       apply (auto simp: Int_absorb1 generate.incl)
      apply (metis generate.simps hom_inv imageI subsetCE)
      done
  next
    fix h1 h2
    assume
      "h1 ∈ generate H (carrier H ∩ h ` S)" "h1 ∈ h ` generate G (carrier
G ∩ S)"
      "h2 ∈ generate H (carrier H ∩ h ` S)" "h2 ∈ h ` generate G (carrier
G ∩ S)"
    then show "h1 ⊗_H h2 ∈ h ` generate G (carrier G ∩ S)"
      using H.subgroupE(4) group.generate_is_subgroup subgroup_img_is_subgroup
      by (simp add: G.generate_is_subgroup)
  qed
  then
  show "carrier (subgroup_generated H (h ` S)) ⊆ h ` carrier (subgroup_generated
G S)"
    by (auto simp: carrier_subgroup_generated)
next
  have "h ` S ⊆ carrier H"
    by (metis (no_types) assms hom_closed image_subset_iff subsetCE)
  then show "h ` carrier (subgroup_generated G S) ⊆ carrier (subgroup_generated
H (h ` S))"
    apply (clarsimp simp: carrier_subgroup_generated)
    by (metis Int_absorb1 assms generate_img imageI)
qed

lemma (in group_hom) iso_between_subgroups:
```

```
  assumes "h ∈ iso G H" "S ⊆ carrier G" "h ` S = T"
  shows "h ∈ iso (subgroup_generated G S) (subgroup_generated H T)"
  using assms
  by (metis G.carrier_subgroup_generated_subset Group.iso_iff hom_between_subgroups
inj_on_subset subgroup_generated_by_image subsetI)

lemma (in group) subgroup_generated_group_carrier:
  "subgroup_generated G (carrier G) = G"
proof (rule monoid.equality)
  show "carrier (subgroup_generated G (carrier G)) = carrier G"
    by (simp add: subgroup.carrier_subgroup_generated_subgroup subgroup_self)
qed (auto simp: subgroup_generated_def)

lemma iso_onto_image:
  assumes "group G" "group H"
  shows
   "f ∈ iso G (subgroup_generated H (f ` carrier G)) ⟷ f ∈ hom G H
∧ inj_on f (carrier G)"
  using assms
  apply (auto simp: iso_def bij_betw_def hom_into_subgroup_eq_gen carrier_subgroup_generate
hom_carrier generate.incl Int_absorb1 Int_absorb2)
  by (metis group.generateI group.subgroupE(1) group.subgroup_self group_hom.generate_img
group_hom.intro group_hom_axioms.intro)

lemma (in group) iso_onto_image:
   "group H ⟹ f ∈ iso G (subgroup_generated H (f ` carrier G)) ⟷
f ∈ mon G H"
  by (simp add: mon_def epi_def hom_into_subgroup_eq_gen iso_onto_image)

end
```

# 17 Elementary Group Constructions

```
theory Elementary_Groups
imports Generated_Groups "HOL-Library.Infinite_Set"
begin
```

## 17.1 Direct sum/product lemmas

```
locale group_disjoint_sum = group G + AG: subgroup A G + BG: subgroup B
G for G (structure) and A B
begin

lemma subset_one: "A ∩ B ⊆ {1} ⟷ A ∩ B = {1}"
  by  auto

lemma sub_id_iff: "A ∩ B ⊆ {1} ⟷ (∀x∈A. ∀y∈B. x ⊗ y = 1 ⟶ x =
1 ∧ y = 1)"
        (is "?lhs = ?rhs")
```

```
proof -
  have "?lhs = (∀x∈A. ∀y∈B. x ⊗ inv y = 1 ⟶ x = 1 ∧ inv y = 1)"
  proof (intro ballI iffI impI)
    fix x y
    assume "A ∩ B ⊆ {1}" "x ∈ A" "y ∈ B" "x ⊗ inv y = 1"
    then have "y = x"
      using group.inv_equality group_l_invI by fastforce
    then show "x = 1 ∧ inv y = 1"
      using ‹A ∩ B ⊆ {1}› ‹x ∈ A› ‹y ∈ B› by fastforce
  next
    assume "∀x∈A. ∀y∈B. x ⊗ inv y = 1 ⟶ x = 1 ∧ inv y = 1"
    then show "A ∩ B ⊆ {1}"
      by auto
  qed
  also have "... = ?rhs"
    by (metis BG.mem_carrier BG.subgroup_axioms inv_inv subgroup_def)
  finally show ?thesis .
qed

lemma cancel: "A ∩ B ⊆ {1} ⟷ (∀x∈A. ∀y∈B. ∀x'∈A. ∀y'∈B. x ⊗ y
= x' ⊗ y' ⟶ x = x' ∧ y = y')"
        (is "?lhs = ?rhs")
proof -
  have "(∀x∈A. ∀y∈B. x ⊗ y = 1 ⟶ x = 1 ∧ y = 1) = ?rhs"
    (is "?med = _")
  proof (intro ballI iffI impI)
    fix x y x' y'
    assume * [rule_format]: "∀x∈A. ∀y∈B. x ⊗ y = 1 ⟶ x = 1 ∧ y =
1"
      and AB: "x ∈ A" "y ∈ B" "x' ∈ A" "y' ∈ B" and eq: "x ⊗ y = x'
⊗ y'"
    then have carr: "x ∈ carrier G" "x' ∈ carrier G" "y ∈ carrier G"
"y' ∈ carrier G"
      using AG.subset BG.subset by auto
    then have "inv x' ⊗ x ⊗ (y ⊗ inv y') = inv x' ⊗ (x ⊗ y) ⊗ inv y'"
      by (simp add: m_assoc)
    also have "... = 1"
      using carr  by (simp add: eq) (simp add: m_assoc)
    finally have 1: "inv x' ⊗ x ⊗ (y ⊗ inv y') = 1" .
    show "x = x' ∧ y = y'"
      using * [OF _ _ 1] AB by simp (metis carr inv_closed inv_inv local.inv_equality)
  next
    fix x y
    assume * [rule_format]: "∀x∈A. ∀y∈B. ∀x'∈A. ∀y'∈B. x ⊗ y = x'
⊗ y' ⟶ x = x' ∧ y = y'"
      and xy: "x ∈ A" "y ∈ B" "x ⊗ y = 1"
    show "x = 1 ∧ y = 1"
      by (rule *) (use xy in auto)
  qed
```

```
  then show ?thesis
    by (simp add: sub_id_iff)
qed

lemma commuting_imp_normal1:
  assumes sub: "carrier G ⊆ A <#> B"
     and mult: "⋀x y. ⟦x ∈ A; y ∈ B⟧ ⟹ x ⊗ y = y ⊗ x"
   shows "A ◁ G"
proof -
  have AB: "A ⊆ carrier G ∧ B ⊆ carrier G"
    by (simp add: AG.subset BG.subset)
  have "A #> x = x <# A"
    if x: "x ∈ carrier G" for x
  proof -
    obtain a b where xeq: "x = a ⊗ b" and "a ∈ A" "b ∈ B" and carr:
"a ∈ carrier G" "b ∈ carrier G"
      using x sub AB by (force simp: set_mult_def)
    have Ab: "A <#> {b} = {b} <#> A"
      using AB ‹a ∈ A› ‹b ∈ B› mult
      by (force simp: set_mult_def m_assoc subset_iff)
    have "A #> x = A <#> {a ⊗ b}"
      by (auto simp: l_coset_eq_set_mult r_coset_eq_set_mult xeq)
    also have "... = A <#> {a} <#> {b}"
      using AB ‹a ∈ A› ‹b ∈ B›
      by (auto simp: set_mult_def m_assoc subset_iff)
    also have "... = {a} <#> A <#> {b}"
      by (metis AG.rcos_const AG.subgroup_axioms ‹a ∈ A› coset_join3
is_group l_coset_eq_set_mult r_coset_eq_set_mult subgroup.mem_carrier)
    also have "... = {a} <#> {b} <#> A"
      by (simp add: is_group carr group.set_mult_assoc AB Ab)
    also have "... = {x} <#> A"
      by (auto simp: set_mult_def xeq)
    finally show "A #> x = x <# A"
      by (simp add: l_coset_eq_set_mult)
  qed
  then show ?thesis
    by (auto simp: normal_def normal_axioms_def AG.subgroup_axioms is_group)
qed

lemma commuting_imp_normal2:
  assumes"carrier G ⊆ A <#> B"  "⋀x y. ⟦x ∈ A; y ∈ B⟧ ⟹ x ⊗ y = y
⊗ x"
  shows "B ◁ G"
proof (rule group_disjoint_sum.commuting_imp_normal1)
  show "group_disjoint_sum G B A"
  proof qed
next
  show "carrier G ⊆ B <#> A"
    using BG.subgroup_axioms assms commut_normal commuting_imp_normal1
```

```
by blast
qed (use assms in auto)


lemma (in group) normal_imp_commuting:
  assumes "A ◁ G" "B ◁ G" "A ∩ B ⊆ {1}" "x ∈ A" "y ∈ B"
  shows "x ⊗ y = y ⊗ x"
proof -
  interpret AG: normal A G
    using assms by auto
  interpret BG: normal B G
    using assms by auto
  interpret group_disjoint_sum G A B
  proof qed
  have * [rule_format]: "(∀x∈A. ∀y∈B. ∀x'∈A. ∀y'∈B. x ⊗ y = x' ⊗ y'
⟶ x = x' ∧ y = y')"
    using cancel assms by (auto simp: normal_def)
  have carr: "x ∈ carrier G" "y ∈ carrier G"
    using assms AG.subset BG.subset by auto
  then show ?thesis
    using * [of x _ _ y] AG.coset_eq [rule_format, of y] BG.coset_eq [rule_format,
of x]
    by (clarsimp simp: l_coset_def r_coset_def set_eq_iff) (metis ‹x
∈ A› ‹y ∈ B›)
qed

lemma normal_eq_commuting:
  assumes "carrier G ⊆ A <#> B" "A ∩ B ⊆ {1}"
  shows "A ◁ G ∧ B ◁ G ⟷ (∀x∈A. ∀y∈B. x ⊗ y = y ⊗ x)"
  by (metis assms commuting_imp_normal1 commuting_imp_normal2 normal_imp_commuting)

lemma (in group) hom_group_mul_rev:
  assumes "(λ(x,y). x ⊗ y) ∈ hom (subgroup_generated G A ×× subgroup_generated
G B) G"
          (is "?h ∈ hom ?P G")
   and "x ∈ carrier G" "y ∈ carrier G" "x ∈ A" "y ∈ B"
 shows "x ⊗ y = y ⊗ x"
proof -
  interpret P: group_hom ?P G ?h
    by (simp add: assms DirProd_group group_hom.intro group_hom_axioms.intro
is_group)
  have xy: "(x,y) ∈ carrier ?P"
    by (auto simp: assms carrier_subgroup_generated generate.incl)
  have "x ⊗ (x ⊗ (y ⊗ y)) = x ⊗ (y ⊗ (x ⊗ y))"
    using P.hom_mult [OF xy xy] by (simp add: m_assoc assms)
  then have "x ⊗ (y ⊗ y) = y ⊗ (x ⊗ y)"
    using assms by simp
  then show ?thesis
    by (simp add: assms flip: m_assoc)
```

```
qed

lemma hom_group_mul_eq:
   "(λ(x,y). x ⊗ y) ∈ hom (subgroup_generated G A ×× subgroup_generated
G B) G
     ⟷ (∀x∈A. ∀y∈B. x ⊗ y = y ⊗ x)"
         (is "?lhs = ?rhs")
proof
  assume ?lhs then show ?rhs
    using hom_group_mul_rev AG.subset BG.subset by blast
next
  assume R: ?rhs
  have subG: "generate G (carrier G ∩ A) ⊆ carrier G" for A
    by (simp add: generate_incl)
  have *: "x ⊗ u ⊗ (y ⊗ v) = x ⊗ y ⊗ (u ⊗ v)"
    if eq [rule_format]: "∀x∈A. ∀y∈B. x ⊗ y = y ⊗ x"
      and gen: "x ∈ generate G (carrier G ∩ A)" "y ∈ generate G (carrier
G ∩ B)"
      "u ∈ generate G (carrier G ∩ A)" "v ∈ generate G (carrier G ∩ B)"
    for x y u v
  proof -
    have "u ⊗ y = y ⊗ u"
      by (metis AG.carrier_subgroup_generated_subgroup BG.carrier_subgroup_generated_subgro
carrier_subgroup_generated eq that(3) that(4))
    then have "x ⊗ u ⊗ y = x ⊗ y ⊗ u"
      using gen by (simp add: m_assoc subsetD [OF subG])
    then show ?thesis
      using gen by (simp add: subsetD [OF subG] flip: m_assoc)
  qed
  show ?lhs
    using R by (auto simp: hom_def carrier_subgroup_generated subsetD
[OF subG] *)
qed


lemma epi_group_mul_eq:
   "(λ(x,y). x ⊗ y) ∈ epi (subgroup_generated G A ×× subgroup_generated
G B) G
     ⟷ A <#> B = carrier G ∧ (∀x∈A. ∀y∈B. x ⊗ y = y ⊗ x)"
proof -
  have subGA: "generate G (carrier G ∩ A) ⊆ A"
    by (simp add: AG.subgroup_axioms generate_subgroup_incl)
  have subGB: "generate G (carrier G ∩ B) ⊆ B"
    by (simp add: BG.subgroup_axioms generate_subgroup_incl)
  have "((((λ(x, y). x ⊗ y) ` (generate G (carrier G ∩ A) × generate
G (carrier G ∩ B)))) = ((A <#> B))"
    by (auto simp: set_mult_def generate.incl pair_imageI dest: subsetD
[OF subGA] subsetD [OF subGB])
  then show ?thesis
```

```
    by (auto simp: epi_def hom_group_mul_eq carrier_subgroup_generated)
qed

lemma mon_group_mul_eq:
    "(λ(x,y). x ⊗ y) ∈ mon (subgroup_generated G A ×× subgroup_generated
G B) G
     ⟷ A ∩ B = {1} ∧ (∀x∈A. ∀y∈B. x ⊗ y = y ⊗ x)"
proof -
  have subGA: "generate G (carrier G ∩ A) ⊆ A"
    by (simp add: AG.subgroup_axioms generate_subgroup_incl)
  have subGB: "generate G (carrier G ∩ B) ⊆ B"
    by (simp add: BG.subgroup_axioms generate_subgroup_incl)
  show ?thesis
    apply (auto simp: mon_def hom_group_mul_eq simp flip: subset_one)
     apply (simp_all (no_asm_use) add: inj_on_def AG.carrier_subgroup_generated_subgroup
BG.carrier_subgroup_generated_subgroup)
    using cancel apply blast+
    done
qed

lemma iso_group_mul_alt:
    "(λ(x,y). x ⊗ y) ∈ iso (subgroup_generated G A ×× subgroup_generated
G B) G
     ⟷ A ∩ B = {1} ∧ A <#> B = carrier G ∧ (∀x∈A. ∀y∈B. x ⊗ y = y
⊗ x)"
  by (auto simp: iso_iff_mon_epi mon_group_mul_eq epi_group_mul_eq)

lemma iso_group_mul_eq:
    "(λ(x,y). x ⊗ y) ∈ iso (subgroup_generated G A ×× subgroup_generated
G B) G
     ⟷ A ∩ B = {1} ∧ A <#> B = carrier G ∧ A ◁ G ∧ B ◁ G"
  by (simp add: iso_group_mul_alt normal_eq_commuting cong: conj_cong)


lemma (in group) iso_group_mul_gen:
  assumes "A ◁ G" "B ◁ G"
  shows "(λ(x,y). x ⊗ y) ∈ iso (subgroup_generated G A ×× subgroup_generated
G B) G
     ⟷ A ∩ B ⊆ {1} ∧ A <#> B = carrier G"
proof -
  interpret group_disjoint_sum G A B
    using assms by (auto simp: group_disjoint_sum_def normal_def)
  show ?thesis
    by (simp add: subset_one iso_group_mul_eq assms)
qed


lemma iso_group_mul:
  assumes "comm_group G"
```

```
  shows "((λ(x,y). x ⊗ y) ∈ iso (DirProd (subgroup_generated G A) (subgroup_generated
G B)) G
     ⟷ A ∩ B ⊆ {1} ∧ A <#> B = carrier G)"
proof (rule iso_group_mul_gen)
  interpret comm_group
    by (rule assms)
  show "A ◁ G"
    by (simp add: AG.subgroup_axioms subgroup_imp_normal)
  show "B ◁ G"
    by (simp add: BG.subgroup_axioms subgroup_imp_normal)
qed

end
```

## 17.2 The one-element group on a given object

```
definition singleton_group :: "'a ⇒ 'a monoid"
  where "singleton_group a = (|carrier = {a}, monoid.mult = (λx y. a),
one = a|)"

lemma singleton_group [simp]: "group (singleton_group a)"
  unfolding singleton_group_def by (auto intro: groupI)

lemma singleton_abelian_group [simp]: "comm_group (singleton_group a)"
  by (metis group.group_comm_groupI monoid.simps(1) singleton_group singleton_group_def)

lemma carrier_singleton_group [simp]: "carrier (singleton_group a) =
{a}"
  by (auto simp: singleton_group_def)

lemma (in group) hom_into_singleton_iff [simp]:
  "h ∈ hom G (singleton_group a) ⟷ h ∈ carrier G → {a}"
  by (auto simp: hom_def singleton_group_def)

declare group.hom_into_singleton_iff [simp]

lemma (in group) id_hom_singleton: "id ∈ hom (singleton_group 1) G"
  by (simp add: hom_def singleton_group_def)
```

## 17.3 Similarly, trivial groups

```
definition trivial_group :: "('a, 'b) monoid_scheme ⇒ bool"
  where "trivial_group G ≡ group G ∧ carrier G = {one G}"

lemma trivial_imp_finite_group:
   "trivial_group G ⟹ finite(carrier G)"
  by (simp add: trivial_group_def)

lemma trivial_singleton_group [simp]: "trivial_group(singleton_group
a)"
```

```
  by (metis monoid.simps(2) partial_object.simps(1) singleton_group singleton_group_def
trivial_group_def)

lemma (in group) trivial_group_subset:
   "trivial_group G ⟷ carrier G ⊆ {one G}"
  using is_group trivial_group_def by fastforce

lemma (in group) trivial_group: "trivial_group G ⟷ (∃a. carrier G
= {a})"
  unfolding trivial_group_def using one_closed is_group by fastforce

lemma (in group) trivial_group_alt:
   "trivial_group G ⟷ (∃a. carrier G ⊆ {a})"
  by (auto simp: trivial_group)

lemma (in group) trivial_group_subgroup_generated:
  assumes "S ⊆ {one G}"
  shows "trivial_group(subgroup_generated G S)"
proof -
  have "carrier (subgroup_generated G S) ⊆ {𝟭}"
    using generate_empty generate_one subset_singletonD assms
    by (fastforce simp add: carrier_subgroup_generated)
  then show ?thesis
    by (simp add: group.trivial_group_subset)
qed

lemma (in group) trivial_group_subgroup_generated_eq:
  "trivial_group(subgroup_generated G s) ⟷ carrier G ∩ s ⊆ {one G}"
  apply (rule iffI)
   apply (force simp: trivial_group_def carrier_subgroup_generated generate.incl)
  by (metis subgroup_generated_restrict trivial_group_subgroup_generated)

lemma isomorphic_group_triviality1:
  assumes "G ≅ H" "group H" "trivial_group G"
  shows "trivial_group H"
  using assms
  by (auto simp: trivial_group_def is_iso_def iso_def group.is_monoid
Group.group_def bij_betw_def hom_one)

lemma isomorphic_group_triviality:
  assumes "G ≅ H" "group G" "group H"
  shows "trivial_group G ⟷ trivial_group H"
  by (meson assms group.iso_sym isomorphic_group_triviality1)

lemma (in group_hom) kernel_from_trivial_group:
   "trivial_group G ⟹ kernel G H h = carrier G"
  by (auto simp: trivial_group_def kernel_def)

lemma (in group_hom) image_from_trivial_group:
```

```
  "trivial_group G ⟹ h ` carrier G = {one H}"
 by (auto simp: trivial_group_def)
```

**lemma** (**in** group_hom) kernel_to_trivial_group:
```
  "trivial_group H ⟹ kernel G H h = carrier G"
 unfolding kernel_def trivial_group_def
 using hom_closed by blast
```

## 17.4 The additive group of integers

**definition** integer_group
```
 where "integer_group = ⦇carrier = UNIV, monoid.mult = (+), one = (0::int)⦈"
```

**lemma** group_integer_group [simp]: "group integer_group"
```
 unfolding integer_group_def
proof (rule groupI; simp)
 show "⋀x::int. ∃y. y + x = 0"
 by presburger
qed
```

**lemma** carrier_integer_group [simp]: "carrier integer_group = UNIV"
```
 by (auto simp: integer_group_def)
```

**lemma** one_integer_group [simp]: "$\mathbf{1}_{\texttt{integer\_group}}$ = 0"
```
 by (auto simp: integer_group_def)
```

**lemma** mult_integer_group [simp]: "x $\otimes_{\texttt{integer\_group}}$ y = x + y"
```
 by (auto simp: integer_group_def)
```

**lemma** inv_integer_group [simp]: "$\texttt{inv}_{\texttt{integer\_group}}$ x = -x"
```
 by (rule group.inv_equality [OF group_integer_group]) (auto simp: integer_group_def)
```

**lemma** abelian_integer_group: "comm_group integer_group"
```
 by (rule group.group_comm_groupI [OF group_integer_group]) (auto simp:
integer_group_def)
```

**lemma** group_nat_pow_integer_group [simp]:
```
 fixes n::nat and x::int
 shows "pow integer_group x n = int n * x"
 by (induction n) (auto simp: integer_group_def algebra_simps)
```

**lemma** group_int_pow_integer_group [simp]:
```
 fixes n::int and x::int
 shows "pow integer_group x n = n * x"
 by (simp add: int_pow_def2)
```

**lemma** (**in** group) hom_integer_group_pow:
```
  "x ∈ carrier G ⟹ pow G x ∈ hom integer_group G"
 by (rule homI) (auto simp: int_pow_mult)
```

## 17.5 Additive group of integers modulo n (n = 0 gives just the integers)

```
definition integer_mod_group :: "nat ⇒ int monoid"
  where
  "integer_mod_group n ≡
     if n = 0 then integer_group
     else (|carrier = {0..<int n}, monoid.mult = (λx y. (x+y) mod int n),
one = 0|)"

lemma carrier_integer_mod_group:
  "carrier(integer_mod_group n) = (if n=0 then UNIV else {0..<int n})"
  by (simp add: integer_mod_group_def)

lemma one_integer_mod_group[simp]: "one(integer_mod_group n) = 0"
  by (simp add: integer_mod_group_def)

lemma mult_integer_mod_group[simp]: "monoid.mult(integer_mod_group n)
= (λx y. (x + y) mod int n)"
  by (simp add: integer_mod_group_def integer_group_def)

lemma group_integer_mod_group [simp]: "group (integer_mod_group n)"
proof -
  have *: "∃y≥0. y < int n ∧ (y + x) mod int n = 0" if "x < int n" "0
≤ x" for x
  proof (cases "x=0")
    case False
    with that show ?thesis
      by (rule_tac x="int n - x" in exI) auto
  qed (use that in auto)
  show ?thesis
    apply (rule groupI)
        apply (auto simp: integer_mod_group_def Bex_def *, presburger+)
    done
qed

lemma inv_integer_mod_group[simp]:
  "x ∈ carrier (integer_mod_group n) ⟹ m_inv(integer_mod_group n) x
= (-x) mod int n"
  by (rule group.inv_equality [OF group_integer_mod_group]) (auto simp:
integer_mod_group_def add.commute mod_add_right_eq)


lemma pow_integer_mod_group [simp]:
  fixes m::nat
  shows "pow (integer_mod_group n) x m = (int m * x) mod int n"
proof (cases "n=0")
  case False
  show ?thesis
    by (induction m) (auto simp: add.commute mod_add_right_eq distrib_left
```

mult.commute)
**qed** (simp add: integer_mod_group_def)

**lemma** int_pow_integer_mod_group:
  "pow (integer_mod_group n) x m = (m * x) mod int n"
**proof** -
  **have** "$\mathrm{inv}_{\mathrm{integer\_mod\_group}\ n}$ (- (m * x) mod int n) = m * x mod int n"
    **by** (simp add: carrier_integer_mod_group mod_minus_eq)
  **then show** ?thesis
    **by** (simp add: int_pow_def2)
**qed**

**lemma** abelian_integer_mod_group [simp]: "comm_group(integer_mod_group n)"
  **by** (simp add: add.commute group.group_comm_groupI)

**lemma** integer_mod_group_0 [simp]: "0 ∈ carrier(integer_mod_group n)"
  **by** (simp add: integer_mod_group_def)

**lemma** integer_mod_group_1 [simp]: "1 ∈ carrier(integer_mod_group n) ⟷ (n ≠ 1)"
  **by** (auto simp: integer_mod_group_def)

**lemma** trivial_integer_mod_group: "trivial_group(integer_mod_group n) ⟷ n = 1"
  (**is** "?lhs = ?rhs")
**proof**
  **assume** ?lhs
  **then show** ?rhs
  **by** (simp add: trivial_group_def carrier_integer_mod_group set_eq_iff split: if_split_asm) (presburger+)
**next**
  **assume** ?rhs
  **then show** ?lhs
    **by** (force simp: trivial_group_def carrier_integer_mod_group)
**qed**

## 17.6 Cyclic groups

**lemma** (**in** group) subgroup_of_powers:
  "x ∈ carrier G ⟹ subgroup (range (λn::int. x [^] n)) G"
  **apply** (auto simp: subgroup_def image_iff simp flip: int_pow_mult int_pow_neg)
  **apply** (metis group.int_pow_diff int_pow_closed is_group r_inv)
  **done**

**lemma** (**in** group) carrier_subgroup_generated_by_singleton:
  **assumes** "x ∈ carrier G"
  **shows** "carrier(subgroup_generated G {x}) = (range (λn::int. x [^] n))"
**proof**

```
  show "carrier (subgroup_generated G {x}) ⊆ range (λn::int. x [^] n)"
  proof (rule subgroup_generated_minimal)
    show "subgroup (range (λn::int. x [^] n)) G"
      using assms subgroup_of_powers by blast
    show "{x} ⊆ range (λn::int. x [^] n)"
      by clarify (metis assms int_pow_1 range_eqI)
  qed
  have x: "x ∈ carrier (subgroup_generated G {x})"
    using assms subgroup_generated_subset_carrier_subset by auto
  show "range (λn::int. x [^] n) ⊆ carrier (subgroup_generated G {x})"
  proof clarify
    fix n :: "int"
    show "x [^] n ∈ carrier (subgroup_generated G {x})"
        by (simp add: x subgroup_int_pow_closed subgroup_subgroup_generated)
  qed
qed


definition cyclic_group
  where "cyclic_group G ≡ ∃x ∈ carrier G. subgroup_generated G {x} =
G"

lemma (in group) cyclic_group:
  "cyclic_group G ⟷ (∃x ∈ carrier G. carrier G = range (λn::int. x
[^] n))"
proof -
  have "⋀x. ⟦x ∈ carrier G; carrier G = range (λn::int. x [^] n)⟧
        ⟹ ∃x∈carrier G. subgroup_generated G {x} = G"
    by (rule_tac x=x in bexI) (auto simp: generate_pow subgroup_generated_def
intro!: monoid.equality)
  then show ?thesis
    unfolding cyclic_group_def
    using carrier_subgroup_generated_by_singleton by fastforce
qed

lemma cyclic_integer_group [simp]: "cyclic_group integer_group"
proof -
  have *: "int n ∈ generate integer_group {1}" for n
  proof (induction n)
    case 0
    then show ?case
    using generate.simps by force
  next
    case (Suc n)
    then show ?case
    by simp (metis generate.simps insert_subset integer_group_def monoid.simps(1)
subsetI)
  qed
  have **: "i ∈ generate integer_group {1}" for i
```

```
  proof (cases i rule: int_cases)
    case (nonneg n)
    then show ?thesis
      by (simp add: *)
  next
    case (neg n)
    then have "-i ∈ generate integer_group {1}"
      by (metis "*" add.inverse_inverse)
    then have "- (-i) ∈ generate integer_group {1}"
      by (metis UNIV_I group.generate_m_inv_closed group_integer_group
integer_group_def inv_integer_group partial_object.select_convs(1) subsetI)
    then show ?thesis
      by simp
  qed
  show ?thesis
    unfolding cyclic_group_def
    by (rule_tac x=1 in bexI)
       (auto simp: carrier_subgroup_generated ** intro: monoid.equality)
qed

lemma nontrivial_integer_group [simp]: "¬ trivial_group integer_group"
  using integer_mod_group_def trivial_integer_mod_group by presburger


lemma (in group) cyclic_imp_abelian_group:
  "cyclic_group G ⟹ comm_group G"
  apply (auto simp: cyclic_group comm_group_def is_group intro!: monoid_comm_monoidI)
  apply (metis add.commute int_pow_mult rangeI)
  done

lemma trivial_imp_cyclic_group:
   "trivial_group G ⟹ cyclic_group G"
  by (metis cyclic_group_def group.subgroup_generated_group_carrier insertI1
trivial_group_def)

lemma (in group) cyclic_group_alt:
   "cyclic_group G ⟷ (∃x. subgroup_generated G {x} = G)"
proof safe
  fix x
  assume *: "subgroup_generated G {x} = G"
  show "cyclic_group G"
  proof (cases "x ∈ carrier G")
    case True
    then show ?thesis
      using ‹subgroup_generated G {x} = G› cyclic_group_def by blast
  next
    case False
    then show ?thesis
      by (metis "*" Int_empty_right Int_insert_right_if0 carrier_subgroup_generated
```

```
generate_empty trivial_group trivial_imp_cyclic_group)
  qed
qed (auto simp: cyclic_group_def)

lemma (in group) cyclic_group_generated:
  "cyclic_group (subgroup_generated G {x})"
  using group.cyclic_group_alt group_subgroup_generated subgroup_generated2
by blast

lemma (in group) cyclic_group_epimorphic_image:
  assumes "h ∈ epi G H" "cyclic_group G" "group H"
  shows "cyclic_group H"
proof -
  interpret h: group_hom
    using assms
    by (simp add: group_hom_def group_hom_axioms_def is_group epi_def)
  obtain x where "x ∈ carrier G" and x: "carrier G = range (λn::int.
x [^] n)" and eq: "carrier H = h ` carrier G"
    using assms by (auto simp: cyclic_group epi_def)
  have "h ` carrier G = range (λn::int. h x [^]_H n)"
    by (metis (no_types, lifting) ‹x ∈ carrier G› h.hom_int_pow image_cong
image_image x)
  then show ?thesis
    using ‹x ∈ carrier G› eq h.cyclic_group by blast
qed

lemma isomorphic_group_cyclicity:
   "⟦G ≅ H; group G; group H⟧ ⟹ cyclic_group G ⟷ cyclic_group H"
  by (meson ex_in_conv group.cyclic_group_epimorphic_image group.iso_sym
is_iso_def iso_iff_mon_epi)


end


theory Multiplicative_Group
imports
  Complex_Main
  Group
  Coset
  UnivPoly
  Generated_Groups
  Elementary_Groups
begin
```

# 18 Simplification Rules for Polynomials

```
lemma (in ring_hom_cring) hom_sub[simp]:
  assumes "x ∈ carrier R" "y ∈ carrier R"
```

```
  shows "h (x ⊖ y) = h x ⊖S h y"
  using assms by (simp add: R.minus_eq S.minus_eq)

context UP_ring begin

lemma deg_nzero_nzero:
  assumes deg_p_nzero: "deg R p ≠ 0"
  shows "p ≠ 0P"
  using deg_zero deg_p_nzero by auto

lemma deg_add_eq:
  assumes c: "p ∈ carrier P" "q ∈ carrier P"
  assumes "deg R q ≠ deg R p"
  shows "deg R (p ⊕P q) = max (deg R p) (deg R q)"
proof -
  let ?m = "max (deg R p) (deg R q)"
  from assms have "coeff P p ?m = 0 ⟷ coeff P q ?m ≠ 0"
    by (metis deg_belowI lcoeff_nonzero[OF deg_nzero_nzero] linear max.absorb_iff2
max.absorb1)
  then have "coeff P (p ⊕P q) ?m ≠ 0"
    using assms by auto
  then have "deg R (p ⊕P q) ≥ ?m"
    using assms by (blast intro: deg_belowI)
  with deg_add[OF c] show ?thesis by arith
qed

lemma deg_minus_eq:
  assumes "p ∈ carrier P" "q ∈ carrier P" "deg R q ≠ deg R p"
  shows "deg R (p ⊖P q) = max (deg R p) (deg R q)"
  using assms by (simp add: deg_add_eq a_minus_def)

end

context UP_cring begin

lemma evalRR_add:
  assumes "p ∈ carrier P" "q ∈ carrier P"
  assumes x: "x ∈ carrier R"
  shows "eval R R id x (p ⊕P q) = eval R R id x p ⊕ eval R R id x q"
proof -
  interpret UP_pre_univ_prop R R id by unfold_locales simp
  interpret ring_hom_cring P R "eval R R id x" by unfold_locales (rule
eval_ring_hom[OF x])
  show ?thesis using assms by simp
qed

lemma evalRR_sub:
  assumes "p ∈ carrier P" "q ∈ carrier P"
  assumes x: "x ∈ carrier R"
```

```
  shows "eval R R id x (p ⊖P q) = eval R R id x p ⊖ eval R R id x q"
proof -
  interpret UP_pre_univ_prop R R id by unfold_locales simp
  interpret ring_hom_cring P R "eval R R id x" by unfold_locales (rule
eval_ring_hom[OF x])
  show ?thesis using assms by simp
qed

lemma evalRR_mult:
  assumes "p ∈ carrier P" "q ∈ carrier P"
  assumes x: "x ∈ carrier R"
  shows "eval R R id x (p ⊗P q) = eval R R id x p ⊗ eval R R id x q"
proof -
  interpret UP_pre_univ_prop R R id by unfold_locales simp
  interpret ring_hom_cring P R "eval R R id x" by unfold_locales (rule
eval_ring_hom[OF x])
  show ?thesis using assms by simp
qed

lemma evalRR_monom:
  assumes a: "a ∈ carrier R" and x: "x ∈ carrier R"
  shows "eval R R id x (monom P a d) = a ⊗ x [^] d"
proof -
  interpret UP_pre_univ_prop R R id by unfold_locales simp
  show ?thesis using assms by (simp add: eval_monom)
qed

lemma evalRR_one:
  assumes x: "x ∈ carrier R"
  shows "eval R R id x 1P = 1"
proof -
  interpret UP_pre_univ_prop R R id by unfold_locales simp
  interpret ring_hom_cring P R "eval R R id x" by unfold_locales (rule
eval_ring_hom[OF x])
  show ?thesis using assms by simp
qed

lemma carrier_evalRR:
  assumes x: "x ∈ carrier R" and "p ∈ carrier P"
  shows "eval R R id x p ∈ carrier R"
proof -
  interpret UP_pre_univ_prop R R id by unfold_locales simp
  interpret ring_hom_cring P R "eval R R id x" by unfold_locales (rule
eval_ring_hom[OF x])
  show ?thesis using assms by simp
qed

lemmas evalRR_simps = evalRR_add evalRR_sub evalRR_mult evalRR_monom
evalRR_one carrier_evalRR
```

**end**

# 19 Properties of the Euler $\varphi$-function

In this section we prove that for every positive natural number the equation $\sum_{d|n}^{n} \varphi(d) = n$ holds.

```
lemma dvd_div_ge_1:
  fixes a b :: nat
  assumes "a ≥ 1" "b dvd a"
  shows "a div b ≥ 1"
proof -
  from ‹b dvd a› obtain c where "a = b * c" ..
  with ‹a ≥ 1› show ?thesis by simp
qed

lemma dvd_nat_bounds:
 fixes n p :: nat
 assumes "p > 0" "n dvd p"
 shows "n > 0 ∧ n ≤ p"
 using assms by (simp add: dvd_pos_nat dvd_imp_le)


definition phi' :: "nat => nat"
  where "phi' m = card {x. 1 ≤ x ∧ x ≤ m ∧ coprime x m}"

notation (latex output)
  phi' (‹φ _›)

lemma phi'_nonzero:
  assumes "m > 0"
  shows "phi' m > 0"
proof -
  have "1 ∈ {x. 1 ≤ x ∧ x ≤ m ∧ coprime x m}" using assms by simp
  hence "card {x. 1 ≤ x ∧ x ≤ m ∧ coprime x m} > 0" by (auto simp: card_gt_0_iff)
  thus ?thesis unfolding phi'_def by simp
qed

lemma dvd_div_eq_1:
  fixes a b c :: nat
  assumes "c dvd a" "c dvd b" "a div c = b div c"
  shows "a = b" using assms dvd_mult_div_cancel[OF ‹c dvd a›] dvd_mult_div_cancel[OF
‹c dvd b›]
                by presburger

lemma dvd_div_eq_2:
  fixes a b c :: nat
  assumes "c>0" "a dvd c" "b dvd c" "c div a = c div b"
```

```
  shows "a = b"
  proof -
  have "a > 0" "a ≤ c" using dvd_nat_bounds[OF assms(1-2)] by auto
  have "a*(c div a) = c" using assms dvd_mult_div_cancel by fastforce
  also have "... = b*(c div a)" using assms dvd_mult_div_cancel by fastforce
  finally show "a = b" using ‹c>0› dvd_div_ge_1[OF _ ‹a dvd c›] by fastforce
qed

lemma div_mult_mono:
  fixes a b c :: nat
  assumes "a > 0" "a≤d"
  shows "a * b div d ≤ b"
proof -
  have "a*b div d ≤ b*a div a" using assms div_le_mono2 mult.commute[of
a b] by presburger
  thus ?thesis using assms by force
qed
```

We arrive at the main result of this section: For every positive natural number the equation $\sum_{d|n}^{n} \varphi(d) = n$ holds.

The outline of the proof for this lemma is as follows: We count the $n$ fractions $1/n$, …, $(n-1)/n$, $n/n$. We analyze the reduced form $a/d = m/n$ for any of those fractions. We want to know how many fractions $m/n$ have the reduced form denominator $d$. The condition $1 \leq m \leq n$ is equivalent to the condition $1 \leq a \leq d$. Therefore we want to know how many $a$ with $1 \leq a \leq d$ exist, s.t. `gcd a d = 1`. This number is exactly `φ d`.

Finally, by counting the fractions $m/n$ according to their reduced form denominator, we get:

```
(∑d | d dvd n. φ d) = n
```

. To formalize this proof in Isabelle, we analyze for an arbitrary divisor $d$ of $n$

- the set of reduced form numerators `{a. 1 ≤ a ∧ a ≤ d ∧ coprime a d}`
- the set of numerators $m$, for which $m/n$ has the reduced form denominator $d$, i.e. the set `{m ∈ {1..n}. n div gcd m n = d}`

We show that `λa. a * n div d` with the inverse `λa. a div gcd a n` is a bijection between theses sets, thus yielding the equality

```
φ d = card {m ∈ {1..n}. n div gcd m n = d}
```

This gives us

```
(∑d | d dvd n. φ d) = card (⋃d∈{d. d dvd n} {m ∈ {1..n}. n div gcd m
n = d})
```

and by showing $\{1..n\} \subseteq (\bigcup_{d\in\{d.\ d\ dvd\ n\}} \{m \in \{1..n\}.\ n\ div\ gcd\ m\ n = d\})$ (this is our counting argument) the thesis follows.

```
lemma sum_phi'_factors:
 fixes n :: nat
 assumes "n > 0"
 shows "(∑d | d dvd n. phi' d) = n"
proof -
  have "card {a. 1 ≤ a ∧ a ≤ d ∧ coprime a d} = card {m ∈ {1 .. n}.
n div gcd m n = d}"
    (is "card ?RF = card ?F")
    if "d dvd n" for d
  proof (rule card_bij_eq)
    from that obtain q where q: "n = d * q" ..
    have "a = b" if "a * n div d = b * n div d" for a b
    proof -
      from that have "a * (n div d) = b * (n div d)"
        using dvd_div_mult[OF ‹d dvd n›] by (fastforce simp add: mult.commute)
      then show ?thesis using dvd_div_ge_1[OF _ ‹d dvd n›] ‹n>0›
        by (simp add: mult.commute nat_mult_eq_cancel1)
    qed
    thus "inj_on (λa. a*n div d) ?RF" unfolding inj_on_def by blast
    have "a * n div d ∈ ?F" if a: "a∈?RF" for a
    proof -
      from that have "a * (n div d) ≥ 1" using ‹n>0› dvd_div_ge_1[OF
_ ‹d dvd n›] by simp
      hence ge_1: "a * n div d ≥ 1" by (simp add: ‹d dvd n› div_mult_swap)
      have le_n: "a * n div d ≤ n" using div_mult_mono a by simp
      have "gcd (a * n div d) n = n div d * gcd a d"
        by (simp add: gcd_mult_distrib_nat q ac_simps)
      hence "n div gcd (a * n div d) n = d*n div (d*(n div d))" using
a by simp
      then show ?thesis
        using ge_1 le_n by (fastforce simp add: ‹d dvd n›)
    qed
    thus "(λa. a*n div d) ‘ ?RF ⊆ ?F" by blast
    have "m = l" if A: "m ∈ ?F" "l ∈ ?F" "m div gcd m n = l div gcd l
n" for m l
    proof -
      from that have "gcd m n = gcd l n" using dvd_div_eq_2[OF assms]
by fastforce
      then show ?thesis using dvd_div_eq_1[of "gcd m n" m l] A(3) by
fastforce
    qed
    thus "inj_on (λa. a div gcd a n) ?F" unfolding inj_on_def by blast
    have "m div gcd m n ∈ ?RF" if "m ∈ ?F" for m
      using that dvd_div_ge_1  by (fastforce simp add: div_le_mono div_gcd_coprime)
    thus "(λa. a div gcd a n) ‘ ?F ⊆ ?RF" by blast
  qed force+
  hence phi'_eq: "⋀d. d dvd n ⟹ phi' d = card {m ∈ {1 .. n}. n div
```

```
gcd m n = d}"
    unfolding phi'_def by presburger
  have fin: "finite {d. d dvd n}" using dvd_nat_bounds[OF ‹n>0›] by force
  have "(∑d | d dvd n. phi' d)
                 = card (⋃d ∈ {d. d dvd n}. {m ∈ {1 .. n}. n div gcd
m n = d})"
    using card_UN_disjoint[OF fin, of "(λd. {m ∈ {1 .. n}. n div gcd m
n = d})"] phi'_eq
    by fastforce
  also have "(⋃d ∈ {d. d dvd n}. {m ∈ {1 .. n}. n div gcd m n = d}) =
{1 .. n}" (is "?L = ?R")
  proof
    show "?L ⊇ ?R"
    proof
      fix m assume m: "m ∈ ?R"
      thus "m ∈ ?L" using dvd_triv_right[of "n div gcd m n" "gcd m n"]
        by simp
    qed
  qed fastforce
  finally show ?thesis by force
qed
```

# 20 Order of an Element of a Group

```
context group begin

definition (in group) ord :: "'a ⇒ nat" where
  "ord x ≡ (@d. ∀n::nat. x [^] n = 1 ⟷ d dvd n)"

lemma (in group) pow_eq_id:
  assumes "x ∈ carrier G"
  shows "x [^] n = 1 ⟷ (ord x) dvd n"
proof (cases "∀n::nat. pow G x n = one G ⟶ n = 0")
  case True
  show ?thesis
    unfolding ord_def
    by (rule someI2 [where a=0]) (auto simp: True)
next
  case False
  define N where "N ≡ LEAST n::nat. x [^] n = 1 ∧ n > 0"
  have N: "x [^] N = 1 ∧ N > 0"
    using False
    apply (simp add: N_def)
    by (metis (mono_tags, lifting) LeastI)
  have eq0: "n = 0" if "x [^] n = 1" "n < N" for n
    using N_def not_less_Least that by fastforce
  show ?thesis
    unfolding ord_def
  proof (rule someI2 [where a = N], rule allI)
```

```
    fix n :: "nat"
    show "(x [^] n = 1) ⟷ (N dvd n)"
    proof (cases "n = 0")
      case False
      show ?thesis
        unfolding dvd_def
      proof safe
        assume 1: "x [^] n = 1"
        have "x [^] n = x [^] (n mod N + N * (n div N))"
          by simp
        also have "... = x [^] (n mod N) ⊗ x [^] (N * (n div N))"
          by (simp add: assms nat_pow_mult)
        also have "... = x [^] (n mod N)"
          by (metis N assms l_cancel_one nat_pow_closed nat_pow_one nat_pow_pow)
        finally have "x [^] (n mod N) = 1"
          by (simp add: "1")
        then have "n mod N = 0"
          using N eq0 mod_less_divisor by blast
        then show "∃k. n = N * k"
          by blast
      next
        fix k :: "nat"
        assume "n = N * k"
        with N show "x [^] (N * k) = 1"
          by (metis assms nat_pow_one nat_pow_pow)
      qed
    qed simp
  qed blast
qed

lemma (in group) pow_ord_eq_1 [simp]:
   "x ∈ carrier G ⟹ x [^] ord x = 1"
  by (simp add: pow_eq_id)

lemma (in group) int_pow_eq_id:
  assumes "x ∈ carrier G"
  shows "(pow G x i = one G ⟷ int (ord x) dvd i)"
proof (cases i rule: int_cases2)
  case (nonneg n)
  then show ?thesis
    by (simp add: int_pow_int pow_eq_id assms)
next
  case (nonpos n)
  then have "x [^] i = inv (x [^] n)"
    by (simp add: assms int_pow_int int_pow_neg)
  then show ?thesis
    by (simp add: assms pow_eq_id nonpos)
qed
```

```
lemma (in group) int_pow_eq:
   "x ∈ carrier G ⟹ (x [^] m = x [^] n) ⟷ int (ord x) dvd (n - m)"
  apply (simp flip: int_pow_eq_id)
  by (metis int_pow_closed int_pow_diff inv_closed r_inv right_cancel)

lemma (in group) ord_eq_0:
   "x ∈ carrier G ⟹ (ord x = 0 ⟷ (∀n::nat. n ≠ 0 ⟶ x [^] n ≠ 1))"
  by (auto simp: pow_eq_id)

lemma (in group) ord_unique:
   "x ∈ carrier G ⟹ ord x = d ⟷ (∀n. pow G x n = one G ⟷ d dvd
n)"
  by (meson dvd_antisym dvd_refl pow_eq_id)

lemma (in group) ord_eq_1:
   "x ∈ carrier G ⟹ (ord x = 1 ⟷ x = 1)"
  by (metis pow_eq_id nat_dvd_1_iff_1 nat_pow_eone)

lemma (in group) ord_id [simp]:
   "ord (one G) = 1"
  using ord_eq_1 by blast

lemma (in group) ord_inv [simp]:
   "x ∈ carrier G
       ⟹ ord (m_inv G x) = ord x"
  by (simp add: ord_unique pow_eq_id nat_pow_inv)

lemma (in group) ord_pow:
  assumes "x ∈ carrier G" "k dvd ord x" "k ≠ 0"
  shows "ord (pow G x k) = ord x div k"
proof -
  have "(x [^] k) [^] (ord x div k) = 1"
    using assms by (simp add: nat_pow_pow)
  moreover have "ord x dvd k * ord (x [^] k)"
    by (metis assms(1) pow_ord_eq_1 pow_eq_id nat_pow_closed nat_pow_pow)
  ultimately show ?thesis
    by (metis assms div_dvd_div dvd_antisym dvd_triv_left pow_eq_id nat_pow_closed
nonzero_mult_div_cancel_left)
qed

lemma (in group) ord_mul_divides:
  assumes eq: "x ⊗ y = y ⊗ x" and xy: "x ∈ carrier G" "y ∈ carrier
G"
  shows "ord (x ⊗ y) dvd (ord x * ord y)"
apply (simp add: xy flip: pow_eq_id eq)
  by (metis dvd_triv_left dvd_triv_right eq pow_eq_id one_closed pow_mult_distrib
r_one xy)

lemma (in comm_group) abelian_ord_mul_divides:
```

```
  "⟦x ∈ carrier G; y ∈ carrier G⟧
      ⟹ ord (x ⊗ y) dvd (ord x * ord y)"
  by (simp add: ord_mul_divides m_comm)

lemma ord_inj:
  assumes a: "a ∈ carrier G"
  shows "inj_on (λ x . a [^] x) {0 .. ord a - 1}"
proof -
  let ?M = "Max (ord ` carrier G)"
  have "finite {d ∈ {..?M}. a [^] d = 1}" by auto

  have *: False if A: "x < y" "x ∈ {0 .. ord a - 1}" "y ∈ {0 .. ord a
- 1}"
      "a [^] x = a [^] y" for x y
  proof -
    have "y - x < ord a" using that by auto
    moreover have "a [^] (y-x) = 1" using a A by (simp add: pow_eq_div2)
    ultimately have "min (y - x) (ord a) = ord a"
      using A(1) a pow_eq_id by auto
    with ‹y - x < ord a› show False by linarith
  qed
  show ?thesis
    unfolding inj_on_def by (metis nat_neq_iff *)
qed

lemma ord_inj':
  assumes a: "a ∈ carrier G"
  shows "inj_on (λ x . a [^] x) {1 .. ord a}"
proof (rule inj_onI, rule ccontr)
  fix x y :: nat
  assume A: "x ∈ {1 .. ord a}" "y ∈ {1 .. ord a}" "a [^] x = a [^] y"
"x≠y"
  then consider "x < ord a" "y < ord a" | "x = ord a" "y < ord a" | "y
= ord a" "x < ord a"
    by force
  then show False
  proof cases
    case 1
    then show ?thesis using ord_inj[OF assms] A unfolding inj_on_def
by fastforce
  next
    case 2
    hence "a [^] y = a [^] (0::nat)" using pow_ord_eq_1 A by (simp add:
a)
    hence "y=0" using ord_inj[OF assms] ‹y < ord a› unfolding inj_on_def
by force
    with A show ?thesis by fastforce
  next
    case 3
```

```
    hence "a [^] x = a [^] (0::nat)" using pow_ord_eq_1 A by (simp add:
a)
    hence "x=0" using ord_inj[OF assms] ‹x < ord a› unfolding inj_on_def
by force
    with A show ?thesis by fastforce
  qed
qed

lemma (in group) ord_ge_1:
  assumes finite: "finite (carrier G)" and a: "a ∈ carrier G"
  shows "ord a ≥ 1"
proof -
  have "((λn::nat. a [^] n) ` {0<..}) ⊆ carrier G"
    using a by blast
  then have "finite ((λn::nat. a [^] n) ` {0<..})"
    using finite_subset finite by auto
  then have "¬ inj_on (λn::nat. a [^] n) {0<..}"
    using finite_imageD infinite_Ioi by blast
  then obtain i j::nat where "i ≠ j" "a [^] i = a [^] j"
    by (auto simp: inj_on_def)
  then have "∃n::nat. n>0 ∧ a [^] n = 1"
    by (metis a diffs0_imp_equal pow_eq_div2 neq0_conv)
  then have "ord a ≠ 0"
    by (simp add: ord_eq_0 [OF a])
  then show ?thesis
    by simp
qed

lemma ord_elems:
  assumes "finite (carrier G)" "a ∈ carrier G"
  shows "{a[^]x | x. x ∈ (UNIV :: nat set)} = {a[^]x | x. x ∈ {0 .. ord
a - 1}}" (is "?L = ?R")
proof
  show "?R ⊆ ?L" by blast
  have "y ∈ {a[^]x | x. x ∈ {0 .. ord a - 1}}" if "y ∈ ?L" for y
  proof -
    from that obtain x::nat where x: "y = a[^]x" by auto
    define r q where "r = x mod ord a" and "q = x div ord a"
    then have "x = q * ord a + r"
      by (simp add: div_mult_mod_eq)
    hence "y = (a[^]ord a)[^]q ⊗ a[^]r"
      using x assms by (metis mult.commute nat_pow_mult nat_pow_pow)
    hence "y = a[^]r" using assms by (simp add: pow_ord_eq_1)
    have "r < ord a" using ord_ge_1[OF assms] by (simp add: r_def)
    hence "r ∈ {0 .. ord a - 1}" by (force simp: r_def)
    thus ?thesis using ‹y=a[^]r› by blast
  qed
  thus "?L ⊆ ?R" by auto
qed
```

```
lemma (in group)
  assumes "x ∈ carrier G"
  shows finite_cyclic_subgroup:
        "finite(carrier(subgroup_generated G {x})) ⟷ (∃n::nat. n ≠
0 ∧ x [^] n = 1)" (is "?fin ⟷ ?nat1")
    and infinite_cyclic_subgroup:
        "infinite(carrier(subgroup_generated G {x})) ⟷ (∀m n::nat.
x [^] m = x [^] n ⟶ m = n)" (is "¬ ?fin ⟷ ?nateq")
    and finite_cyclic_subgroup_int:
        "finite(carrier(subgroup_generated G {x})) ⟷ (∃i::int. i ≠
0 ∧ x [^] i = 1)" (is "?fin ⟷ ?int1")
    and infinite_cyclic_subgroup_int:
        "infinite(carrier(subgroup_generated G {x})) ⟷ (∀i j::int.
x [^] i = x [^] j ⟶ i = j)" (is "¬ ?fin ⟷ ?inteq")
proof -
  have 1: "¬ ?fin" if ?nateq
  proof -
    have "infinite (range (λn::nat. x [^] n))"
      using that range_inj_infinite [of "(λn::nat. x [^] n)"] by (auto
simp: inj_on_def)
    moreover have "range (λn::nat. x [^] n) ⊆ range (λi::int. x [^]
i)"
      apply clarify
      by (metis assms group.int_pow_neg int_pow_closed int_pow_neg_int
is_group local.inv_equality nat_pow_closed r_inv rangeI)
    ultimately show ?thesis
      using carrier_subgroup_generated_by_singleton [OF assms] finite_subset
by auto
  qed
  have 2: "m = n" if mn: "x [^] m = x [^] n" and eq [rule_format]: "?inteq"
for m n::nat
    using eq [of "int m" "int n"]
    by (simp add: int_pow_int mn)
  have 3: ?nat1 if non: "¬ ?inteq"
  proof -
    obtain i j::int where eq: "x [^] i = x [^] j" and "i ≠ j"
      using non by auto
    show ?thesis
    proof (cases i j rule: linorder_cases)
      case less
      then have [simp]: "x [^] (j - i) = 1"
        by (simp add: eq assms int_pow_diff)
      show ?thesis
        using less by (rule_tac x="nat (j-i)" in exI) auto
    next
      case greater
      then have [simp]: "x [^] (i - j) = 1"
        by (simp add: eq assms int_pow_diff)
```

```
      then show ?thesis
        using greater by (rule_tac x="nat (i-j)" in exI) auto
    qed (use ‹i ≠ j› in auto)
  qed
  have 4: "∃i::int. (i ≠ 0) ∧ x [^] i = 1" if "n ≠ 0" "x [^] n = 1"
for n::nat
    apply (rule_tac x="int n" in exI)
    by (simp add: int_pow_int that)
  have 5: "finite (carrier (subgroup_generated G {x}))" if "i ≠ 0" and
1: "x [^] i = 1" for i::int
  proof -
    obtain n::nat where n: "n > 0" "x [^] n = 1"
      using "1" "3" ‹i ≠ 0› by fastforce
    have "x [^] a ∈ ([^]) x ` {0..<n}" for a::int
    proof
      show "x [^] a = x [^] nat (a mod int n)"
        using n
        by simp (metis (no_types, lifting) assms dvd_minus_mod dvd_trans
int_pow_eq int_pow_eq_id int_pow_int)
      show "nat (a mod int n) ∈ {0..<n}"
        using n by (simp add: nat_less_iff)
    qed
    then have "carrier (subgroup_generated G {x}) ⊆ ([^]) x ` {0..<n}"
      using carrier_subgroup_generated_by_singleton [OF assms] by auto
    then show ?thesis
      using finite_surj by blast
  qed
  show "?fin ⟷ ?nat1" "¬ ?fin ⟷ ?nateq" "?fin ⟷ ?int1" "¬ ?fin
⟷ ?inteq"
    using 1 2 3 4 5 by meson+
qed

lemma (in group) finite_cyclic_subgroup_order:
   "x ∈ carrier G ⟹ finite(carrier(subgroup_generated G {x})) ⟷ ord
x ≠ 0"
  by (simp add: finite_cyclic_subgroup ord_eq_0)

lemma (in group) infinite_cyclic_subgroup_order:
   "x ∈ carrier G ⟹ infinite (carrier(subgroup_generated G {x})) ⟷
ord x = 0"
  by (simp add: finite_cyclic_subgroup_order)

lemma generate_pow_on_finite_carrier:
  assumes "finite (carrier G)" and a: "a ∈ carrier G"
  shows "generate G { a } = { a [^] k | k. k ∈ (UNIV :: nat set) }"
proof
  show "{ a [^] k | k. k ∈ (UNIV :: nat set) } ⊆ generate G { a }"
  proof
    fix b assume "b ∈ { a [^] k | k. k ∈ (UNIV :: nat set) }"
```

```
    then obtain k :: nat where "b = a [^] k" by blast
    hence "b = a [^] (int k)"
      by (simp add: int_pow_int)
    thus "b ∈ generate G { a }"
      unfolding generate_pow[OF a] by blast
  qed
next
  show "generate G { a } ⊆ { a [^] k | k. k ∈ (UNIV :: nat set) }"
  proof
    fix b assume "b ∈ generate G { a }"
    then obtain k :: int where k: "b = a [^] k"
      unfolding generate_pow[OF a] by blast
    show "b ∈ { a [^] k | k. k ∈ (UNIV :: nat set) }"
    proof (cases "k < 0")
      assume "¬ k < 0"
      hence "b = a [^] (nat k)"
        by (simp add: k)
      thus ?thesis by blast
    next
      assume "k < 0"
      hence b: "b = inv (a [^] (nat (- k)))"
        using k a by (auto simp: int_pow_neg)
      obtain m where m: "ord a * m ≥ nat (- k)"
        by (metis assms mult.left_neutral mult_le_mono1 ord_ge_1)
      hence "a [^] (ord a * m) = 1"
        by (metis a nat_pow_one nat_pow_pow pow_ord_eq_1)
      then obtain k' :: nat where "(a [^] (nat (- k))) ⊗ (a [^] k') =
1"
        using m a nat_le_iff_add nat_pow_mult by auto
      hence "b = a [^] k'"
        using b a by (metis inv_unique' nat_pow_closed nat_pow_comm)
      thus "b ∈ { a [^] k | k. k ∈ (UNIV :: nat set) }" by blast
    qed
  qed
qed

lemma ord_elems_inf_carrier:
 assumes "a ∈ carrier G" "ord a ≠ 0"
 shows "{a[^]x | x. x ∈ (UNIV :: nat set)} = {a[^]x | x. x ∈ {0 .. ord
a - 1}}" (is "?L = ?R")
proof
  show "?R ⊆ ?L" by blast
  have "y ∈ {a[^]x | x. x ∈ {0 .. ord a - 1}}" if "y ∈ ?L" for y
  proof -
    from that obtain x::nat where x: "y = a[^]x" by auto
    define r q where "r = x mod ord a" and "q = x div ord a"
    then have "x = q * ord a + r"
      by (simp add: div_mult_mod_eq)
    hence "y = (a[^]ord a)[^]q ⊗ a[^]r"
```

```
      using x assms by (metis mult.commute nat_pow_mult nat_pow_pow)
    hence "y = a[^]r" using assms by simp
    have "r < ord a" using assms by (simp add: r_def)
    hence "r ∈ {0 .. ord a - 1}" by (force simp: r_def)
    then show ?thesis using ‹y=a[^]r› by blast
  qed
  thus "?L ⊆ ?R" by auto
qed

lemma generate_pow_nat:
 assumes a: "a ∈ carrier G" and "ord a ≠ 0"
 shows "generate G { a } = { a [^] k | k. k ∈ (UNIV :: nat set) }"
proof
 show "{ a [^] k | k. k ∈ (UNIV :: nat set) } ⊆ generate G { a }"
 proof
   fix b assume "b ∈ { a [^] k | k. k ∈ (UNIV :: nat set) }"
   then obtain k :: nat where "b = a [^] k" by blast
   hence "b = a [^] (int k)"
     by (simp add: int_pow_int)
   thus "b ∈ generate G { a }"
     unfolding generate_pow[OF a] by blast
 qed
next
 show "generate G { a } ⊆ { a [^] k | k. k ∈ (UNIV :: nat set) }"
 proof
   fix b assume "b ∈ generate G { a }"
   then obtain k :: int where k: "b = a [^] k"
     unfolding generate_pow[OF a] by blast
   show "b ∈ { a [^] k | k. k ∈ (UNIV :: nat set) }"
   proof (cases "k < 0")
     assume "¬ k < 0"
     hence "b = a [^] (nat k)"
       by (simp add: k)
     thus ?thesis by blast
   next
     assume "k < 0"
     hence b: "b = inv (a [^] (nat (- k)))"
       using k a by (auto simp: int_pow_neg)
     obtain m where m: "ord a * m ≥ nat (- k)"
       by (metis assms(2) dvd_imp_le dvd_triv_right le_zero_eq mult_eq_0_iff
not_gr_zero)
     hence "a [^] (ord a * m) = 1"
       by (metis a nat_pow_one nat_pow_pow pow_ord_eq_1)
     then obtain k' :: nat where "(a [^] (nat (- k))) ⊗ (a [^] k') =
1"
       using m a nat_le_iff_add nat_pow_mult by auto
     hence "b = a [^] k'"
       using b a by (metis inv_unique' nat_pow_closed nat_pow_comm)
     thus "b ∈ { a [^] k | k. k ∈ (UNIV :: nat set) }" by blast
```

```
    qed
  qed
qed

lemma generate_pow_card:
  assumes a: "a ∈ carrier G"
  shows "ord a = card (generate G { a })"
proof (cases "ord a = 0")
  case True
  then have "infinite (carrier (subgroup_generated G {a}))"
    using infinite_cyclic_subgroup_order[OF a] by auto
  then have "infinite (generate G {a})"
    unfolding subgroup_generated_def
    using a by simp
  then show ?thesis
    using ‹ord a = 0› by auto
next
  case finite_subgroup: False
  then have "generate G { a } = (([^]) a) ` {0..ord a - 1}"
    using generate_pow_nat ord_elems_inf_carrier a by auto
  hence "card (generate G {a}) = card {0..ord a - 1}"
    using ord_inj[OF a] card_image by metis
  also have "... = ord a" using finite_subgroup by auto
  finally show ?thesis..
qed

lemma (in group) cyclic_order_is_ord:
 assumes "g ∈ carrier G"
 shows "ord g = order (subgroup_generated G {g})"
 unfolding order_def subgroup_generated_def
 using assms generate_pow_card by simp

lemma ord_dvd_group_order:
  assumes "a ∈ carrier G" shows "(ord a) dvd (order G)"
  using lagrange[OF generate_is_subgroup[of "{a}"]] assms
  unfolding generate_pow_card[OF assms]
  by (metis dvd_triv_right empty_subsetI insert_subset)

lemma (in group) pow_order_eq_1:
  assumes "a ∈ carrier G" shows "a [^] order G = 1"
  using assms by (metis nat_pow_pow ord_dvd_group_order pow_ord_eq_1 dvdE
nat_pow_one)

lemma dvd_gcd:
  fixes a b :: nat
  obtains q where "a * (b div gcd a b) = b*q"
proof
  have "a * (b div gcd a b) = (a div gcd a b) * b" by (simp add:  div_mult_swap
dvd_div_mult)
```

```
  also have "... = b * (a div gcd a b)" by simp
  finally show "a * (b div gcd a b) = b * (a div gcd a b) " .
qed

lemma (in group) ord_le_group_order:
  assumes finite: "finite (carrier G)" and a: "a ∈ carrier G"
  shows "ord a ≤ order G"
  by (simp add: a dvd_imp_le local.finite ord_dvd_group_order order_gt_0_iff_finite)

lemma (in group) ord_pow_gen:
  assumes "x ∈ carrier G"
  shows "ord (pow G x k) = (if k = 0 then 1 else ord x div gcd (ord x)
k)"
proof -
  have "ord (x [^] k) = ord x div gcd (ord x) k"
    if "0 < k"
  proof -
    have "(d dvd k * n) = (d div gcd (d) k dvd n)" for d n
      using that by (simp add: div_dvd_iff_mult gcd_mult_distrib_nat mult.commute)
    then show ?thesis
      using that by (auto simp add: assms ord_unique nat_pow_pow pow_eq_id)
  qed
  then show ?thesis by auto
qed

lemma (in group)
  assumes finite': "finite (carrier G)" "a ∈ carrier G"
  shows pow_ord_eq_ord_iff: "group.ord G (a [^] k) = ord a ⟷ coprime
k (ord a)" (is "?L ⟷ ?R")
    using assms ord_ge_1 [OF assms]
    by (auto simp: div_eq_dividend_iff ord_pow_gen coprime_iff_gcd_eq_1
gcd.commute split: if_split_asm)

lemma element_generates_subgroup:
  assumes finite[simp]: "finite (carrier G)"
  assumes a[simp]: "a ∈ carrier G"
  shows "subgroup {a [^] i | i. i ∈ {0 .. ord a - 1}} G"
  using generate_is_subgroup[of "{ a }"] assms(2)
        generate_pow_on_finite_carrier[OF assms]
  unfolding ord_elems[OF assms] by auto

end
```

# 21 Number of Roots of a Polynomial

```
definition mult_of :: "('a, 'b) ring_scheme ⇒ 'a monoid" where
  "mult_of R ≡ ⦇ carrier = carrier R - {0R}, mult = mult R, one = 1R⦈"

lemma carrier_mult_of [simp]: "carrier (mult_of R) = carrier R - {0R}"
```

```
  by (simp add: mult_of_def)

lemma mult_mult_of [simp]: "mult (mult_of R) = mult R"
 by (simp add: mult_of_def)

lemma nat_pow_mult_of: "([^]mult_of R) = (([^]R) :: _ ⇒ nat ⇒ _)"
  by (simp add: mult_of_def fun_eq_iff nat_pow_def)

lemma one_mult_of [simp]: "𝟭mult_of R = 𝟭R"
  by (simp add: mult_of_def)

lemmas mult_of_simps = carrier_mult_of mult_mult_of nat_pow_mult_of one_mult_of

context field
begin

lemma mult_of_is_Units: "mult_of R = units_of R"
  unfolding mult_of_def units_of_def using field_Units by auto

lemma m_inv_mult_of:
"⋀x. x ∈ carrier (mult_of R) ⟹ m_inv (mult_of R) x = m_inv R x"
  using mult_of_is_Units units_of_inv unfolding units_of_def
  by simp

lemma (in field) field_mult_group: "group (mult_of R)"
  proof (rule groupI)
  show "∃y∈carrier (mult_of R). y ⊗mult_of R x = 𝟭mult_of R"
    if "x ∈ carrier (mult_of R)" for x
    using group.l_inv_ex mult_of_is_Units that units_group by fastforce
qed (auto simp: m_assoc dest: integral)

lemma finite_mult_of: "finite (carrier R) ⟹ finite (carrier (mult_of
R))"
  by simp

lemma order_mult_of: "finite (carrier R) ⟹ order (mult_of R) = order
R - 1"
  unfolding order_def carrier_mult_of by (simp add: card.remove)

end



lemma (in monoid) Units_pow_closed :
  fixes d :: nat
  assumes "x ∈ Units G"
  shows "x [^] d ∈ Units G"
    by (metis assms group.is_monoid monoid.nat_pow_closed units_group
units_of_carrier units_of_pow)
```

```
lemma (in ring) r_right_minus_eq[simp]:
  assumes "a ∈ carrier R" "b ∈ carrier R"
  shows "a ⊖ b = 𝟬 ⟷ a = b"
  using assms by (metis a_minus_def add.inv_closed minus_equality r_neg)

context UP_cring begin

lemma is_UP_cring: "UP_cring R" by (unfold_locales)
lemma is_UP_ring:
  shows "UP_ring R" by (unfold_locales)

end

context UP_domain begin


lemma roots_bound:
  assumes f [simp]: "f ∈ carrier P"
  assumes f_not_zero: "f ≠ 𝟬P"
  assumes finite: "finite (carrier R)"
  shows "finite {a ∈ carrier R . eval R R id a f = 𝟬} ∧
         card {a ∈ carrier R . eval R R id a f = 𝟬} ≤ deg R f" using
f f_not_zero
proof (induction "deg R f" arbitrary: f)
  case 0
  have "⋀x. eval R R id x f ≠ 𝟬"
  proof -
    fix x
    have "(⨁i∈{..deg R f}. id (coeff P f i) ⊗ x [^] i) ≠ 𝟬"
      using 0 lcoeff_nonzero_nonzero[where p = f] by simp
    thus "eval R R id x f ≠ 𝟬" using 0 unfolding eval_def P_def by simp
  qed
  then have *: "{a ∈ carrier R. eval R R (λa. a) a f = 𝟬} = {}"
    by (auto simp: id_def)
  show ?case by (simp add: *)
next
  case (Suc x)
  show ?case
  proof (cases "∃ a ∈ carrier R . eval R R id a f = 𝟬")
    case True
    then obtain a where a_carrier[simp]: "a ∈ carrier R" and a_root:
"eval R R id a f = 𝟬" by blast
    have R_not_triv: "carrier R ≠ {𝟬}"
      by (metis R.one_zeroI R.zero_not_one)
    obtain q  where q: "(q ∈ carrier P)" and
      f: "f = (monom P 𝟭R 1 ⊖ P monom P a 0) ⊗P q ⊕P monom P (eval R
R id a f) 0"
     using remainder_theorem[OF Suc.prems(1) a_carrier R_not_triv] by
```

```
auto
    hence lin_fac: "f = (monom P 1R 1 ⊖ P monom P a 0) ⊗P q" using q
by (simp add: a_root)
    have deg: "deg R (monom P 1R 1 ⊖ P monom P a 0) = 1"
      using a_carrier by (simp add: deg_minus_eq)
    hence mon_not_zero: "(monom P 1R 1 ⊖ P monom P a 0) ≠ 0P"
      by (fastforce simp del: r_right_minus_eq)
    have q_not_zero: "q ≠ 0P" using Suc by (auto simp add : lin_fac)
    hence "deg R q = x" using Suc deg deg_mult[OF mon_not_zero q_not_zero
_ q]
      by (simp add : lin_fac)
    hence q_IH: "finite {a ∈ carrier R . eval R R id a q = 0}
                ∧ card {a ∈ carrier R . eval R R id a q = 0} ≤ x" us-
ing Suc q q_not_zero by blast
    have subs: "{a ∈ carrier R . eval R R id a f = 0}
                ⊆ {a ∈ carrier R . eval R R id a q = 0} ∪ {a}" (is "?L
⊆ ?R ∪ {a}")
      using a_carrier ‹q ∈ _›
      by (auto simp: evalRR_simps lin_fac R.integral_iff)
    have "{a ∈ carrier R . eval R R id a f = 0} ⊆ insert a {a ∈ carrier
R . eval R R id a q = 0}"
     using subs by auto
    hence "card {a ∈ carrier R . eval R R id a f = 0} ≤
           card (insert a {a ∈ carrier R . eval R R id a q = 0})" us-
ing q_IH by (blast intro: card_mono)
    also have "... ≤ deg R f" using q_IH ‹Suc x = _›
      by (simp add: card_insert_if)
    finally show ?thesis using q_IH ‹Suc x = _› using finite by force
  next
    case False
    hence "card {a ∈ carrier R. eval R R id a f = 0} = 0" using finite
by auto
    also have "... ≤  deg R f" by simp
    finally show ?thesis using finite by auto
  qed
qed

end

lemma (in domain) num_roots_le_deg :
  fixes p d :: nat
  assumes finite: "finite (carrier R)"
  assumes d_neq_zero: "d ≠ 0"
  shows "card {x ∈ carrier R. x [^] d = 1} ≤ d"
proof -
  let ?f = "monom (UP R) 1R d ⊖ (UP R) monom (UP R) 1R 0"
  have one_in_carrier: "1 ∈ carrier R" by simp
  interpret R: UP_domain R "UP R" by (unfold_locales)
  have "deg R ?f = d"
```

```
    using d_neq_zero by (simp add: R.deg_minus_eq)
  hence f_not_zero: "?f ≠ 0UP R" using  d_neq_zero by (auto simp add
: R.deg_nzero_nzero)
  have roots_bound: "finite {a ∈ carrier R . eval R R id a ?f = 0} ∧
                  card {a ∈ carrier R . eval R R id a ?f = 0} ≤ deg
R ?f"
                  using finite by (intro R.roots_bound[OF _ f_not_zero])
simp
  have subs: "{x ∈ carrier R. x [^] d = 1} ⊆ {a ∈ carrier R . eval R
R id a ?f = 0}"
    by (auto simp: R.evalRR_simps)
  then have "card {x ∈ carrier R. x [^] d = 1} ≤
        card {a ∈ carrier R. eval R R id a ?f = 0}" using finite by (simp
add : card_mono)
  thus ?thesis using ‹deg R ?f = d› roots_bound by linarith
qed
```

## 22 The Multiplicative Group of a Field

In this section we show that the multiplicative group of a finite field is generated by a single element, i.e. it is cyclic. The proof is inspired by the first proof given in the survey [2].

```
context field begin

lemma num_elems_of_ord_eq_phi':
  assumes finite: "finite (carrier R)" and dvd: "d dvd order (mult_of
R)"
      and exists: "∃a∈carrier (mult_of R). group.ord (mult_of R) a =
d"
  shows "card {a ∈ carrier (mult_of R). group.ord (mult_of R) a = d}
= phi' d"
proof -
  note mult_of_simps[simp]
  have finite': "finite (carrier (mult_of R))" using finite by (rule
finite_mult_of)

  interpret G:group "mult_of R" rewrites "([^]mult_of R) = (([^]) :: _ ⇒
nat ⇒ _)" and "1mult_of R = 1"
    by (rule field_mult_group) simp_all

  from exists
  obtain a where a: "a ∈ carrier (mult_of R)" and ord_a: "group.ord (mult_of
R) a = d"
    by (auto simp add: card_gt_0_iff)

  have set_eq1: "{a[^]n| n. n ∈ {1 .. d}} = {x ∈ carrier (mult_of R).
x [^] d = 1}"
  proof (rule card_seteq)
```

```
    show "finite {x ∈ carrier (mult_of R). x [^] d = 1}" using finite
by auto

    show "{a[^]n| n. n ∈ {1 ..d}} ⊆ {x ∈ carrier (mult_of R). x[^]d
= 1}"
    proof
      fix x assume "x ∈ {a[^]n | n. n ∈ {1 .. d}}"
      then obtain n where n: "x = a[^]n ∧ n ∈ {1 .. d}" by auto
      have "x[^]d =(a[^]d)[^]n" using n a ord_a by (simp add:nat_pow_pow
mult.commute)
      hence "x[^]d = 1" using ord_a G.pow_ord_eq_1[OF a] by fastforce
      thus "x ∈ {x ∈ carrier (mult_of R). x[^]d = 1}" using G.nat_pow_closed[OF
a] n by blast
    qed

    show "card {x ∈ carrier (mult_of R). x [^] d = 1} ≤ card {a[^]n |
n. n ∈ {1 .. d}}"
    proof -
      have *: "{a[^]n | n. n ∈ {1 .. d }} = ((λ n. a[^]n) ‘ {1 .. d})"
by auto
      have "0 < order (mult_of R)" unfolding order_mult_of[OF finite]
        using card_mono[OF finite, of "{0, 1}"] by (simp add: order_def)
      have "card {x ∈ carrier (mult_of R). x [^] d = 1} ≤ card {x ∈ carrier
R. x [^] d = 1}"
        using finite by (auto intro: card_mono)
      also have "... ≤ d" using ‹0 < order (mult_of R)› num_roots_le_deg[OF
finite, of d]
        by (simp add : dvd_pos_nat[OF _ ‹d dvd order (mult_of R)›])
      finally show ?thesis using G.ord_inj'[OF a] ord_a * by (simp add:
card_image)
    qed
  qed

  have set_eq2: "{x ∈ carrier (mult_of R) . group.ord (mult_of R) x =
d}
                = (λ n . a[^]n) ‘ {n ∈ {1 .. d}. group.ord (mult_of R)
(a[^]n) = d}" (is "?L = ?R")
  proof
    have "x ∈ ?R" if x: "x ∈ (carrier (mult_of R)) ∧ group.ord (mult_of
R) x = d" for x
    proof -
      from that have "x ∈ {x ∈ carrier (mult_of R). x [^] d = 1}"
        by (simp add: G.pow_ord_eq_1[of x, symmetric])
      then obtain n where n: "x = a[^]n ∧ n ∈ {1 .. d}" using set_eq1
by blast
      then show ?thesis using x by fast
    qed
    thus "?L ⊆ ?R" by blast
    show "?R ⊆ ?L" using a by (auto simp add: carrier_mult_of[symmetric]
```

```
simp del: carrier_mult_of)
  qed
  have "inj_on (λ n . a[^]n) {n ∈ {1 .. d}. group.ord (mult_of R) (a[^]n)
= d}"
    using G.ord_inj'[OF a, unfolded ord_a] unfolding inj_on_def by fast
  hence "card ((λn. a[^]n) ` {n ∈ {1 .. d}. group.ord (mult_of R) (a[^]n)
= d})
        = card {k ∈ {1 .. d}. group.ord (mult_of R) (a[^]k) = d}"
        using card_image by blast
  thus ?thesis using set_eq2 G.pow_ord_eq_ord_iff[OF finite' ‹a ∈ _›,
unfolded ord_a]
    by (simp add: phi'_def)
qed

end


theorem (in field) finite_field_mult_group_has_gen :
  assumes finite: "finite (carrier R)"
  shows "∃ a ∈ carrier (mult_of R) . carrier (mult_of R) = {a[^]i | i::nat
. i ∈ UNIV}"
proof -
  note mult_of_simps[simp]
  have finite': "finite (carrier (mult_of R))" using finite by (rule
finite_mult_of)

  interpret G: group "mult_of R" rewrites
      "([^]mult_of R) = (([^]) :: _ ⇒ nat ⇒ _)" and "1mult_of R = 1"
    by (rule field_mult_group) (simp_all add: fun_eq_iff nat_pow_def)

  let ?N = "λ x . card {a ∈ carrier (mult_of R). group.ord (mult_of R)
a  = x}"
  have "0 < order R - 1" unfolding order_def using card_mono[OF finite,
of "{0, 1}"] by simp
  then have *: "0 < order (mult_of R)" using assms by (simp add: order_mult_of)
  have fin: "finite {d. d dvd order (mult_of R) }" using dvd_nat_bounds[OF
*] by force

  have "(∑d | d dvd order (mult_of R). ?N d)
      = card (UN d:{d . d dvd order (mult_of R) }. {a ∈ carrier (mult_of
R). group.ord (mult_of R) a  = d})"
      (is "_ = card ?U")
    using fin finite by (subst card_UN_disjoint) auto
  also have "?U = carrier (mult_of R)"
  proof
    have "x ∈ ?U" if x: "x ∈ carrier (mult_of R)" for x
    proof -
      from that have x': "x∈carrier (mult_of R)" by simp
      then have "group.ord (mult_of R) x dvd order (mult_of R)"
```

```
        using G.ord_dvd_group_order by blast
      then show ?thesis
        using dvd_nat_bounds[of "order (mult_of R)" "group.ord (mult_of
R) x"] x by blast
    qed
    thus "carrier (mult_of R) ⊆ ?U" by blast
  qed auto
  also have "card ... = order (mult_of R)"
    using order_mult_of finite' by (simp add: order_def)
  finally have sum_Ns_eq: "(∑d | d dvd order (mult_of R). ?N d) = order
(mult_of R)" .

  have "card {a ∈ carrier (mult_of R). group.ord (mult_of R) a = d} ≤
phi' d"
    if d: "d dvd order (mult_of R)" for d
  proof (cases "card {a ∈ carrier (mult_of R). group.ord (mult_of R) a
= d} = 0")
    case True
    thus ?thesis by presburger
  next
    case False
    hence "∃a ∈ carrier (mult_of R). group.ord (mult_of R) a = d" by
(auto simp: card_eq_0_iff)
    thus ?thesis using num_elems_of_ord_eq_phi'[OF finite d] by auto
  qed
  hence all_le: "⋀i. i ∈ {d. d dvd order (mult_of R) }
        ⟹ (λi. card {a ∈ carrier (mult_of R). group.ord (mult_of R)
a = i}) i ≤ (λi. phi' i) i" by fast
  hence le: "(∑i | i dvd order (mult_of R). ?N i)
            ≤ (∑i | i dvd order (mult_of R). phi' i)"
            using sum_mono[of "{d .  d dvd order (mult_of R)}"
                  "λi. card {a ∈ carrier (mult_of R). group.ord (mult_of
R) a = i}"] by presburger
  have "order (mult_of R) = (∑d | d dvd order (mult_of R). phi' d)"
using *
    by (simp add: sum_phi'_factors)
  hence eq: "(∑i | i dvd order (mult_of R). ?N i)
          = (∑i | i dvd order (mult_of R). phi' i)" using le sum_Ns_eq
by presburger
  have "⋀i. i ∈ {d. d dvd order (mult_of R) } ⟹ ?N i = (λi. phi' i)
i"
  proof (rule ccontr)
    fix i
    assume i1: "i ∈ {d. d dvd order (mult_of R)}" and "?N i ≠ phi' i"
    hence "?N i = 0"
      using num_elems_of_ord_eq_phi'[OF finite, of i] by (auto simp: card_eq_0_iff)
    moreover  have "0 < i" using * i1 by (simp add: dvd_nat_bounds[of
"order (mult_of R)" i])
    ultimately have "?N i < phi' i" using phi'_nonzero by presburger
```

```
    hence "(∑i | i dvd order (mult_of R). ?N i)
         < (∑i | i dvd order (mult_of R). phi' i)"
      using sum_strict_mono_ex1[OF fin, of "?N" "λ i . phi' i"]
            i1 all_le by auto
    thus False using eq by force
  qed
  hence "?N (order (mult_of R)) > 0" using * by (simp add: phi'_nonzero)
  then obtain a where a: "a ∈ carrier (mult_of R)" and a_ord: "group.ord
(mult_of R) a = order (mult_of R)"
    by (auto simp add: card_gt_0_iff)
  hence set_eq: "{a[^]i | i::nat. i ∈ UNIV} = (λx. a[^]x) ` {0 .. group.ord
(mult_of R) a - 1}"
    using G.ord_elems[OF finite'] by auto
  have card_eq: "card ((λx. a[^]x) ` {0 .. group.ord (mult_of R) a - 1})
= card {0 .. group.ord (mult_of R) a - 1}"
    by (intro card_image G.ord_inj finite' a)
  hence "card ((λ x . a[^]x) ` {0 .. group.ord (mult_of R) a - 1}) = card
{0 ..order (mult_of R) - 1}"
    using assms by (simp add: card_eq a_ord)
  hence card_R_minus_1: "card {a[^]i | i::nat. i ∈ UNIV} =  order (mult_of
R)"
    using * by (subst set_eq) auto
  have **: "{a[^]i | i::nat. i ∈ UNIV} ⊆ carrier (mult_of R)"
    using G.nat_pow_closed[OF a] by auto
  with _ have "carrier (mult_of R) = {a[^]i|i::nat. i ∈ UNIV}"
    by (rule card_seteq[symmetric]) (simp_all add: card_R_minus_1 finite
order_def del: UNIV_I)
  thus ?thesis using a by blast
qed

end


theory Group_Action
imports Bij Coset Congruence
begin
```

# 23 Group Actions

```
locale group_action =
  fixes G (structure) and E and φ
  assumes group_hom: "group_hom G (BijGroup E) φ"

definition
  orbit :: "[_, 'a ⇒ 'b ⇒ 'b, 'b] ⇒ 'b set"
  where "orbit G φ x = {(φ g) x | g. g ∈ carrier G}"

definition
  orbits :: "[_, 'b set, 'a ⇒ 'b ⇒ 'b] ⇒ ('b set) set"
```

```
  where "orbits G E φ = {orbit G φ x | x. x ∈ E}"

definition
  stabilizer :: "[_, 'a ⇒ 'b ⇒ 'b, 'b] ⇒ 'a set"
  where "stabilizer G φ x = {g ∈ carrier G. (φ g) x = x}"

definition
  invariants :: "['b set, 'a ⇒ 'b ⇒ 'b, 'a] ⇒ 'b set"
  where "invariants E φ g = {x ∈ E. (φ g) x = x}"

definition
  normalizer :: "[_, 'a set] ⇒ 'a set"
  where "normalizer G H =
         stabilizer G (λg. λH ∈ {H. H ⊆ carrier G}. g <#G H #>G (invG
g)) H"

locale faithful_action = group_action +
  assumes faithful: "inj_on φ (carrier G)"

locale transitive_action = group_action +
  assumes unique_orbit: "⟦ x ∈ E; y ∈ E ⟧ ⟹ ∃g ∈ carrier G. (φ g)
x = y"
```

## 23.1 Prelimineries

Some simple lemmas to make group action's properties more explicit

```
lemma (in group_action) id_eq_one: "(λx ∈ E. x) = φ 1"
  by (metis BijGroup_def group_hom group_hom.hom_one select_convs(2))

lemma (in group_action) bij_prop0:
  "⋀ g. g ∈ carrier G ⟹ (φ g) ∈ Bij E"
  by (metis BijGroup_def group_hom group_hom.hom_closed partial_object.select_convs(1))

lemma (in group_action) surj_prop:
  "⋀ g. g ∈ carrier G ⟹ (φ g) ` E = E"
  using bij_prop0 by (simp add: Bij_def bij_betw_def)

lemma (in group_action) inj_prop:
  "⋀ g. g ∈ carrier G ⟹ inj_on (φ g) E"
  using bij_prop0 by (simp add: Bij_def bij_betw_def)

lemma (in group_action) bij_prop1:
  "⋀ g y. ⟦ g ∈ carrier G; y ∈ E ⟧ ⟹  ∃!x ∈ E. (φ g) x = y"
proof -
  fix g y assume "g ∈ carrier G" "y ∈ E"
  hence "∃x ∈ E. (φ g) x = y"
    using surj_prop by force
  moreover have "⋀ x1 x2. ⟦ x1 ∈ E; x2 ∈ E ⟧ ⟹ (φ g) x1 = (φ g) x2
⟹ x1 = x2"
```

```
    using inj_prop by (meson ‹g ∈ carrier G› inj_on_eq_iff)
  ultimately show "∃!x ∈ E. (φ g) x = y"
    by blast
qed

lemma (in group_action) composition_rule:
  assumes "x ∈ E" "g1 ∈ carrier G" "g2 ∈ carrier G"
  shows "φ (g1 ⊗ g2) x = (φ g1) (φ g2 x)"
proof -
  have "φ (g1 ⊗ g2) x = ((φ g1) ⊗BijGroup E (φ g2)) x"
    using assms(2) assms(3) group_hom group_hom.hom_mult by fastforce
  also have " ... = (compose E (φ g1) (φ g2)) x"
    unfolding BijGroup_def by (simp add: assms bij_prop0)
  finally show "φ (g1 ⊗ g2) x = (φ g1) (φ g2 x)"
    by (simp add: assms(1) compose_eq)
qed

lemma (in group_action) element_image:
  assumes "g ∈ carrier G" and "x ∈ E" and "(φ g) x = y"
  shows "y ∈ E"
  using surj_prop assms by blast
```

## 23.2 Orbits

We prove here that orbits form an equivalence relation

```
lemma (in group_action) orbit_sym_aux:
  assumes "g ∈ carrier G"
    and "x ∈ E"
    and "(φ g) x = y"
  shows "(φ (inv g)) y = x"
proof -
  interpret group G
    using group_hom group_hom.axioms(1) by auto
  have "y ∈ E"
    using element_image assms by simp
  have "inv g ∈ carrier G"
    by (simp add: assms(1))

  have "(φ (inv g)) y = (φ (inv g)) ((φ g) x)"
    using assms(3) by simp
  also have " ... = compose E (φ (inv g)) (φ g) x"
    by (simp add: assms(2) compose_eq)
  also have " ... = ((φ (inv g)) ⊗BijGroup E (φ g)) x"
    by (simp add: BijGroup_def assms(1) bij_prop0)
  also have " ... = (φ ((inv g) ⊗ g)) x"
    by (metis ‹inv g ∈ carrier G› assms(1) group_hom group_hom.hom_mult)
  finally show "(φ (inv g)) y = x"
    by (metis assms(1) assms(2) id_eq_one l_inv restrict_apply)
qed
```

```
lemma (in group_action) orbit_refl:
  "x ∈ E ⟹ x ∈ orbit G φ x"
proof -
  assume "x ∈ E" hence "(φ 1) x = x"
    using id_eq_one by (metis restrict_apply')
  thus "x ∈ orbit G φ x" unfolding orbit_def
    using group.is_monoid group_hom group_hom.axioms(1) by force
qed

lemma (in group_action) orbit_sym:
  assumes "x ∈ E" and "y ∈ E" and "y ∈ orbit G φ x"
  shows "x ∈ orbit G φ y"
proof -
  have "∃ g ∈ carrier G. (φ g) x = y"
    using assms by (auto simp: orbit_def)
  then obtain g where g: "g ∈ carrier G ∧ (φ g) x = y" by blast
  hence "(φ (inv g)) y = x"
    using orbit_sym_aux by (simp add: assms(1))
  thus ?thesis
    using g group_hom group_hom.axioms(1) orbit_def by fastforce
qed

lemma (in group_action) orbit_trans:
  assumes "x ∈ E" "y ∈ E" "z ∈ E"
    and "y ∈ orbit G φ x" "z ∈ orbit G φ y"
  shows "z ∈ orbit G φ x"
proof -
  interpret group G
    using group_hom group_hom.axioms(1) by auto
  obtain g1 where g1: "g1 ∈ carrier G ∧ (φ g1) x = y"
    using assms by (auto simp: orbit_def)
  obtain g2 where g2: "g2 ∈ carrier G ∧ (φ g2) y = z"
    using assms by (auto simp: orbit_def)
  have "(φ (g2 ⊗ g1)) x = ((φ g2) ⊗BijGroup E (φ g1)) x"
    using g1 g2 group_hom group_hom.hom_mult by fastforce
  also have " ... = (φ g2) ((φ g1) x)"
    using composition_rule assms(1) calculation g1 g2 by auto
  finally have "(φ (g2 ⊗ g1)) x = z"
    by (simp add: g1 g2)
  thus ?thesis
    using g1 g2 orbit_def by force
qed

lemma (in group_action) orbits_as_classes:
  "classes⦇ carrier = E, eq = λx. λy. y ∈ orbit G φ x ⦈ = orbits G E φ"
  unfolding eq_classes_def eq_class_of_def orbits_def orbit_def
  using element_image by auto
```

```
theorem (in group_action) orbit_partition:
  "partition E (orbits G E φ)"
proof -
  have "equivalence ⦇ carrier = E, eq = λx. λy. y ∈ orbit G φ x ⦈"
  unfolding equivalence_def apply simp
  using orbit_refl orbit_sym orbit_trans by blast
  thus ?thesis using equivalence.partition_from_equivalence orbits_as_classes
by fastforce
qed

corollary (in group_action) orbits_coverture:
  "⋃ (orbits G E φ) = E"
  using partition.partition_coverture[OF orbit_partition] by simp

corollary (in group_action) disjoint_union:
  assumes "orb1 ∈ (orbits G E φ)" "orb2 ∈ (orbits G E φ)"
  shows "(orb1 = orb2) ∨ (orb1 ∩ orb2) = {}"
  using partition.disjoint_union[OF orbit_partition] assms by auto

corollary (in group_action) disjoint_sum:
  assumes "finite E"
  shows "(∑orb∈(orbits G E φ). ∑x∈orb. f x) = (∑x∈E. f x)"
  using partition.disjoint_sum[OF orbit_partition] assms by auto
```

#### 23.2.1 Transitive Actions

Transitive actions have only one orbit

```
lemma (in transitive_action) all_equivalent:
  "⟦ x ∈ E; y ∈ E ⟧ ⟹ x .=⦇carrier = E, eq = λx y. y ∈ orbit G φ x⦈ y"
proof -
  assume "x ∈ E" "y ∈ E"
  hence "∃ g ∈ carrier G. (φ g) x = y"
    using unique_orbit  by blast
  hence "y ∈ orbit G φ x"
    using orbit_def by fastforce
  thus "x .=⦇carrier = E, eq = λx y. y ∈ orbit G φ x⦈ y" by simp
qed

proposition (in transitive_action) one_orbit:
  assumes "E ≠ {}"
  shows "card (orbits G E φ) = 1"
proof -
  have "orbits G E φ ≠ {}"
    using assms orbits_coverture by auto
  moreover have "⋀ orb1 orb2. ⟦ orb1 ∈ (orbits G E φ); orb2 ∈ (orbits
G E φ) ⟧ ⟹ orb1 = orb2"
  proof -
    fix orb1 orb2 assume orb1: "orb1 ∈ (orbits G E φ)"
                 and orb2: "orb2 ∈ (orbits G E φ)"
```

```
    then obtain x y where x: "orb1 = orbit G φ x" and x_E: "x ∈ E"
                     and y: "orb2 = orbit G φ y" and y_E: "y ∈ E"
      unfolding orbits_def by blast
    hence "x ∈ orbit G φ y" using all_equivalent by auto
    hence "orb1 ∩ orb2 ≠ {}" using x y x_E orbit_refl by auto
    thus "orb1 = orb2" using disjoint_union[of orb1 orb2] orb1 orb2 by
auto
  qed
  ultimately show "card (orbits G E φ) = 1"
    by (meson is_singletonI' is_singleton_altdef)
qed
```

## 23.3 Stabilizers

We show that stabilizers are subgroups from the acting group

```
lemma (in group_action) stabilizer_subset:
  "stabilizer G φ x ⊆ carrier G"
  by (metis (no_types, lifting) mem_Collect_eq stabilizer_def subsetI)

lemma (in group_action) stabilizer_m_closed:
  assumes "x ∈ E" "g1 ∈ (stabilizer G φ x)" "g2 ∈ (stabilizer G φ x)"
  shows "(g1 ⊗ g2) ∈ (stabilizer G φ x)"
proof -
  interpret group G
    using group_hom group_hom.axioms(1) by auto

  have "φ g1 x = x"
    using assms stabilizer_def by fastforce
  moreover have "φ g2 x = x"
    using assms stabilizer_def by fastforce
  moreover have g1: "g1 ∈ carrier G"
    by (meson assms contra_subsetD stabilizer_subset)
  moreover have g2: "g2 ∈ carrier G"
    by (meson assms contra_subsetD stabilizer_subset)
  ultimately have "φ (g1 ⊗ g2) x = x"
    using composition_rule assms by simp

  thus ?thesis
    by (simp add: g1 g2 stabilizer_def)
qed

lemma (in group_action) stabilizer_one_closed:
  assumes "x ∈ E"
  shows "1 ∈ (stabilizer G φ x)"
proof -
  have "φ 1 x = x"
    by (metis assms id_eq_one restrict_apply')
  thus ?thesis
    using group_def group_hom group_hom.axioms(1) stabilizer_def by fastforce
```

```
qed

lemma (in group_action) stabilizer_m_inv_closed:
  assumes "x ∈ E" "g ∈ (stabilizer G φ x)"
  shows "(inv g) ∈ (stabilizer G φ x)"
proof -
  interpret group G
    using group_hom group_hom.axioms(1) by auto

  have "φ g x = x"
    using assms(2) stabilizer_def by fastforce
  moreover have g: "g ∈ carrier G"
    using assms(2) stabilizer_subset by blast
  moreover have inv_g: "inv g ∈ carrier G"
    by (simp add: g)
  ultimately have "φ (inv g) x = x"
    using assms(1) orbit_sym_aux by blast

  thus ?thesis by (simp add: inv_g stabilizer_def)
qed

theorem (in group_action) stabilizer_subgroup:
  assumes "x ∈ E"
  shows "subgroup (stabilizer G φ x) G"
  unfolding subgroup_def
  using stabilizer_subset stabilizer_m_closed stabilizer_one_closed
        stabilizer_m_inv_closed assms by simp
```

## 23.4 The Orbit-Stabilizer Theorem

In this subsection, we prove the Orbit-Stabilizer theorem. Our approach is to show the existence of a bijection between "rcosets (stabilizer G phi x)" and "orbit G phi x". Then we use Lagrange's theorem to find the cardinal of the first set.

### 23.4.1 Rcosets - Supporting Lemmas

```
corollary (in group_action) stab_rcosets_not_empty:
  assumes "x ∈ E" "R ∈ rcosets (stabilizer G φ x)"
  shows "R ≠ {}"
  using subgroup.rcosets_non_empty[OF stabilizer_subgroup[OF assms(1)]
assms(2)] by simp

corollary (in group_action) diff_stabilizes:
  assumes "x ∈ E" "R ∈ rcosets (stabilizer G φ x)"
  shows "⋀g1 g2. ⟦ g1 ∈ R; g2 ∈ R ⟧ ⟹ g1 ⊗ (inv g2) ∈ stabilizer G
φ x"
  using group.diff_neutralizes[of G "stabilizer G φ x" R] stabilizer_subgroup[OF
assms(1)]
```

```
      assms(2) group_hom group_hom.axioms(1) by blast
```

### 23.4.2 Bijection Between Rcosets and an Orbit - Definition and Supporting Lemmas

```
definition
  orb_stab_fun :: "[_, ('a ⇒ 'b ⇒ 'b), 'a set, 'b] ⇒ 'b"
  where "orb_stab_fun G φ R x = (φ (inv_G (SOME h. h ∈ R))) x"

lemma (in group_action) orbit_stab_fun_is_well_defined0:
  assumes "x ∈ E" "R ∈ rcosets (stabilizer G φ x)"
  shows "⋀g1 g2. ⟦ g1 ∈ R; g2 ∈ R ⟧ ⟹ (φ (inv g1)) x = (φ (inv g2))
x"
proof -
  fix g1 g2 assume g1: "g1 ∈ R" and g2: "g2 ∈ R"
  have R_carr: "R ⊆ carrier G"
    using subgroup.rcosets_carrier[OF stabilizer_subgroup[OF assms(1)]]
          assms(2) group_hom group_hom.axioms(1) by auto
  from R_carr have g1_carr: "g1 ∈ carrier G" using g1 by blast
  from R_carr have g2_carr: "g2 ∈ carrier G" using g2 by blast

  have "g1 ⊗ (inv g2) ∈ stabilizer G φ x"
    using diff_stabilizes[of x R g1 g2] assms g1 g2 by blast
  hence "φ (g1 ⊗ (inv g2)) x = x"
    by (simp add: stabilizer_def)
  hence "(φ (inv g1)) x = (φ (inv g1)) (φ (g1 ⊗ (inv g2)) x)" by simp
  also have " ... = φ ((inv g1) ⊗ (g1 ⊗ (inv g2))) x"
    using group_def assms(1) composition_rule g1_carr g2_carr
          group_hom group_hom.axioms(1) monoid.m_closed by fastforce
  also have " ... = φ (((inv g1) ⊗ g1) ⊗ (inv g2)) x"
    using group_def g1_carr g2_carr group_hom group_hom.axioms(1) monoid.m_assoc
by fastforce
  finally show "(φ (inv g1)) x = (φ (inv g2)) x"
    using group_def g1_carr g2_carr group.l_inv group_hom group_hom.axioms(1)
by fastforce
qed

lemma (in group_action) orbit_stab_fun_is_well_defined1:
  assumes "x ∈ E" "R ∈ rcosets (stabilizer G φ x)"
  shows "⋀g. g ∈ R ⟹ (φ (inv (SOME h. h ∈ R))) x = (φ (inv g)) x"
  by (meson assms orbit_stab_fun_is_well_defined0 someI_ex)

lemma (in group_action) orbit_stab_fun_is_inj:
  assumes "x ∈ E"
    and "R1 ∈ rcosets (stabilizer G φ x)"
    and "R2 ∈ rcosets (stabilizer G φ x)"
    and "φ (inv (SOME h. h ∈ R1)) x = φ (inv (SOME h. h ∈ R2)) x"
  shows "R1 = R2"
proof -
```

```
  have "(∃g1. g1 ∈ R1) ∧ (∃g2. g2 ∈ R2)"
    using assms(1-3) stab_rcosets_not_empty by auto
  then obtain g1 g2 where g1: "g1 ∈ R1" and g2: "g2 ∈ R2" by blast
  hence g12_carr: "g1 ∈ carrier G ∧ g2 ∈ carrier G"
    using subgroup.rcosets_carrier assms(1-3) group_hom
          group_hom.axioms(1) stabilizer_subgroup by blast

  then obtain r1 r2 where r1: "r1 ∈ carrier G" "R1 = (stabilizer G φ
x) #> r1"
                    and r2: "r2 ∈ carrier G" "R2 = (stabilizer G φ
x) #> r2"
    using assms(1-3) unfolding RCOSETS_def by blast
  then obtain s1 s2 where s1: "s1 ∈ (stabilizer G φ x)" "g1 = s1 ⊗ r1"
                    and s2: "s2 ∈ (stabilizer G φ x)" "g2 = s2 ⊗ r2"
    using g1 g2 unfolding r_coset_def by blast

  have "φ (inv g1) x = φ (inv (SOME h. h ∈ R1)) x"
    using orbit_stab_fun_is_well_defined1[OF assms(1) assms(2) g1] by
simp
  also have " ... = φ (inv (SOME h. h ∈ R2)) x"
    using assms(4) by simp
  finally have "φ (inv g1) x = φ (inv g2) x"
    using orbit_stab_fun_is_well_defined1[OF assms(1) assms(3) g2] by
simp

  hence "φ g2 (φ (inv g1) x) = φ g2 (φ (inv g2) x)" by simp
  also have " ... = φ (g2 ⊗ (inv g2)) x"
    using assms(1) composition_rule g12_carr group_hom group_hom.axioms(1)
by fastforce
  finally have "φ g2 (φ (inv g1) x) = x"
    using g12_carr assms(1) group.r_inv group_hom group_hom.axioms(1)
          id_eq_one restrict_apply by metis
  hence "φ (g2 ⊗ (inv g1)) x = x"
    using assms(1) composition_rule g12_carr group_hom group_hom.axioms(1)
by fastforce
  hence "g2 ⊗ (inv g1) ∈ (stabilizer G φ x)"
    using g12_carr group.subgroup_self group_hom group_hom.axioms(1)
          mem_Collect_eq stabilizer_def subgroup_def by (metis (mono_tags,
lifting))
  then obtain s where s: "s ∈ (stabilizer G φ x)" "s = g2 ⊗ (inv g1)"
by blast

  let ?h = "s ⊗ g1"
  have "?h = s ⊗ (s1 ⊗ r1)" by (simp add: s1)
  hence "?h = (s ⊗ s1) ⊗ r1"
    using stabilizer_subgroup[OF assms(1)] group_def group_hom
          group_hom.axioms(1) monoid.m_assoc r1 s s1 subgroup.mem_carrier
by fastforce
  hence inR1: "?h ∈ (stabilizer G φ x) #> r1" unfolding r_coset_def
```

```
    using stabilizer_subgroup[OF assms(1)] assms(1) s s1 stabilizer_m_closed
by auto

  have "?h = g2" using s stabilizer_subgroup[OF assms(1)] g12_carr group.inv_solve_right
                      group_hom group_hom.axioms(1) subgroup.mem_carrier
by metis
  hence inR2: "?h ∈ (stabilizer G φ x) #> r2"
    using g2 r2 by blast

  have "R1 ∩ R2 ≠ {}" using inR1 inR2 r1 r2 by blast
  thus ?thesis
    using stabilizer_subgroup group.rcos_disjoint[of G "stabilizer G φ
x"] assms group_hom group_hom.axioms(1)
    unfolding disjnt_def pairwise_def  by blast
qed

lemma (in group_action) orbit_stab_fun_is_surj:
  assumes "x ∈ E" "y ∈ orbit G φ x"
  shows "∃R ∈ rcosets (stabilizer G φ x). φ (inv (SOME h. h ∈ R)) x
= y"
proof -
  have "∃g ∈ carrier G. (φ g) x = y"
    using assms(2) unfolding orbit_def by blast
  then obtain g where g: "g ∈ carrier G ∧ (φ g) x = y" by blast

  let ?R = "(stabilizer G φ x) #> (inv g)"
  have R: "?R ∈ rcosets (stabilizer G φ x)"
    unfolding RCOSETS_def using g group_hom group_hom.axioms(1) by fastforce
  moreover have "𝟭 ⊗ (inv g) ∈ ?R"
    unfolding r_coset_def using assms(1) stabilizer_one_closed by auto
  ultimately have "φ (inv (SOME h. h ∈ ?R)) x = φ (inv (𝟭 ⊗ (inv g)))
x"
    using orbit_stab_fun_is_well_defined1[OF assms(1)] by simp
  also have " ... = (φ g) x"
    using group_def g group_hom group_hom.axioms(1) monoid.l_one by fastforce
  finally have "φ (inv (SOME h. h ∈ ?R)) x = y"
    using g by simp
  thus ?thesis using R by blast
qed

proposition (in group_action) orbit_stab_fun_is_bij:
  assumes "x ∈ E"
  shows "bij_betw (λR. (φ (inv (SOME h. h ∈ R))) x) (rcosets (stabilizer
G φ x)) (orbit G φ x)"
  unfolding bij_betw_def
proof
  show "inj_on (λR. φ (inv (SOME h. h ∈ R)) x) (rcosets stabilizer G
φ x)"
    using orbit_stab_fun_is_inj[OF assms(1)] by (simp add: inj_on_def)
```

```
next
  have "⋀R. R ∈ (rcosets stabilizer G φ x) ⟹ φ (inv (SOME h. h ∈ R))
x ∈ orbit G φ x "
  proof -
    fix R assume R: "R ∈ (rcosets stabilizer G φ x)"
    then obtain g where g: "g ∈ R"
      using assms stab_rcosets_not_empty by auto
    hence "φ (inv (SOME h. h ∈ R)) x = φ (inv g) x"
      using R  assms orbit_stab_fun_is_well_defined1 by blast
    thus "φ (inv (SOME h. h ∈ R)) x ∈ orbit G φ x" unfolding orbit_def
      using subgroup.rcosets_carrier group_hom group_hom.axioms(1)
            g R assms stabilizer_subgroup by fastforce
  qed
  moreover have "orbit G φ x ⊆ (λR. φ (inv (SOME h. h ∈ R)) x) ‘ (rcosets
stabilizer G φ x)"
    using assms orbit_stab_fun_is_surj by fastforce
  ultimately show "(λR. φ (inv (SOME h. h ∈ R)) x) ‘ (rcosets stabilizer
G φ x) = orbit G φ x "
    using assms set_eq_subset by blast
qed
```

### 23.4.3 The Theorem

```
theorem (in group_action) orbit_stabilizer_theorem:
  assumes "x ∈ E"
  shows "card (orbit G φ x) * card (stabilizer G φ x) = order G"
proof -
  have "card (rcosets stabilizer G φ x) = card (orbit G φ x)"
    using orbit_stab_fun_is_bij[OF assms(1)] bij_betw_same_card by blast
  moreover have "card (rcosets stabilizer G φ x) * card (stabilizer G
φ x) = order G"
    using stabilizer_subgroup assms group.lagrange group_hom group_hom.axioms(1)
by blast
  ultimately show ?thesis by auto
qed
```

## 23.5 The Burnside's Lemma

### 23.5.1 Sums and Cardinals

```
lemma card_as_sums:
  assumes "A = {x ∈ B. P x}" "finite B"
  shows "card A = (∑x∈B. (if P x then 1 else 0))"
proof -
  have "A ⊆ B" using assms(1) by blast
  have "card A = (∑x∈A. 1)" by simp
  also have " ... = (∑x∈A. (if P x then 1 else 0))"
    by (simp add: assms(1))
  also have " ... = (∑x∈A. (if P x then 1 else 0)) + (∑x∈(B - A). (if
P x then 1 else 0))"
```

```
    using assms(1) by auto
  finally show "card A = (∑x∈B. (if P x then 1 else 0))"
    using ‹A ⊆ B› add.commute assms(2) sum.subset_diff by metis
qed

lemma sum_invertion:
  "⟦ finite A; finite B ⟧ ⟹ (∑x∈A. ∑y∈B. f x y) = (∑y∈B. ∑x∈A.
f x y)"
proof (induct set: finite)
  case empty thus ?case by simp
next
  case (insert x A')
  have "(∑x∈insert x A'. ∑y∈B. f x y) = (∑y∈B. f x y) + (∑x∈A'.
∑y∈B. f x y)"
    by (simp add: insert.hyps)
  also have " ... = (∑y∈B. f x y) + (∑y∈B. ∑x∈A'. f x y)"
    using insert.hyps by (simp add: insert.prems)
  also have " ... = (∑y∈B. (f x y) + (∑x∈A'. f x y))"
    by (simp add: sum.distrib)
  finally have "(∑x∈insert x A'. ∑y∈B. f x y) = (∑y∈B. ∑x∈insert
x A'. f x y)"
    using sum.swap by blast
  thus ?case by simp
qed

lemma (in group_action) card_stablizer_sum:
  assumes "finite (carrier G)" "orb ∈ (orbits G E φ)"
  shows "(∑x ∈ orb. card (stabilizer G φ x)) = order G"
proof -
  obtain x where x:"x ∈ E" and orb:"orb = orbit G φ x"
    using assms(2) unfolding orbits_def by blast
  have "⋀y. y ∈ orb ⟹ card (stabilizer G φ x) = card (stabilizer G
φ y)"
  proof -
    fix y assume "y ∈ orb"
    hence y: "y ∈ E ∧ y ∈ orbit G φ x"
      using x orb assms(2) orbits_coverture by auto
    hence same_orbit: "(orbit G φ x) = (orbit G φ y)"
      using disjoint_union[of "orbit G φ x" "orbit G φ y"] orbit_refl
x
      unfolding orbits_def by auto
    have "card (orbit G φ x) * card (stabilizer G φ x) =
          card (orbit G φ y) * card (stabilizer G φ y)"
      using y assms(1) x orbit_stabilizer_theorem by simp
    hence "card (orbit G φ x) * card (stabilizer G φ x) =
           card (orbit G φ x) * card (stabilizer G φ y)" using same_orbit
by simp
    moreover have "orbit G φ x ≠ {} ∧ finite (orbit G φ x)"
      using y orbit_def[of G φ x] assms(1) by auto
```

```
    hence "card (orbit G φ x) > 0"
      by (simp add: card_gt_0_iff)
    ultimately show "card (stabilizer G φ x) = card (stabilizer G φ y)"
by auto
  qed
  hence "(∑x ∈ orb. card (stabilizer G φ x)) = (∑y ∈ orb. card (stabilizer
G φ x))" by auto
  also have " ... = card (stabilizer G φ x) * (∑y ∈ orb. 1)" by simp
  also have " ... = card (stabilizer G φ x) * card (orbit G φ x)"
    using orb by auto
  finally show "(∑x ∈ orb. card (stabilizer G φ x)) = order G"
    by (metis mult.commute orbit_stabilizer_theorem x)
qed
```

### 23.5.2 The Lemma

```
theorem (in group_action) burnside:
  assumes "finite (carrier G)" "finite E"
  shows "card (orbits G E φ) * order G = (∑g ∈ carrier G. card(invariants
E φ g))"
proof -
  have "(∑g ∈ carrier G. card(invariants E φ g)) =
        (∑g ∈ carrier G. ∑x ∈ E. (if (φ g) x = x then 1 else 0))"
    by (simp add: assms(2) card_as_sums invariants_def)
  also have " ... = (∑x ∈ E. ∑g ∈ carrier G. (if (φ g) x = x then 1
else 0))"
    using sum_invertion[where ?f = "λ g x. (if (φ g) x = x then 1 else
0)"] assms by auto
  also have " ... = (∑x ∈ E. card (stabilizer G φ x))"
    by (simp add: assms(1) card_as_sums stabilizer_def)
  also have " ... = (∑orbit ∈ (orbits G E φ). ∑x ∈ orbit. card (stabilizer
G φ x))"
    using disjoint_sum orbits_coverture assms(2) by metis
  also have " ... = (∑orbit ∈ (orbits G E φ). order G)"
    by (simp add: assms(1) card_stablizer_sum)
  finally have "(∑g ∈ carrier G. card(invariants E φ g)) = card (orbits
G E φ) * order G" by simp
  thus ?thesis by simp
qed
```

## 23.6 Action by Conjugation

### 23.6.1 Action Over Itself

A Group Acts by Conjugation Over Itself

```
lemma (in group) conjugation_is_inj:
  assumes "g ∈ carrier G" "h1 ∈ carrier G" "h2 ∈ carrier G"
    and "g ⊗ h1 ⊗ (inv g) = g ⊗ h2 ⊗ (inv g)"
    shows "h1 = h2"
```

```
  using assms by auto

lemma (in group) conjugation_is_surj:
  assumes "g ∈ carrier G" "h ∈ carrier G"
  shows "g ⊗ ((inv g) ⊗ h ⊗ g) ⊗ (inv g) = h"
  using assms m_assoc inv_closed inv_inv m_closed monoid_axioms r_inv
r_one
  by metis

lemma (in group) conjugation_is_bij:
  assumes "g ∈ carrier G"
  shows "bij_betw (λh ∈ carrier G. g ⊗ h ⊗ (inv g)) (carrier G) (carrier
G)"
         (is "bij_betw ?φ (carrier G) (carrier G)")
  unfolding bij_betw_def
proof
  show "inj_on ?φ (carrier G)"
    using conjugation_is_inj by (simp add: assms inj_on_def)
next
  have S: "⋀ h. h ∈ carrier G ⟹ (inv g) ⊗ h ⊗ g ∈ carrier G"
    using assms by blast
  have "⋀ h. h ∈ carrier G ⟹ ?φ ((inv g) ⊗ h ⊗ g) = h"
    using assms by (simp add: conjugation_is_surj)
  hence "carrier G ⊆ ?φ ` carrier G"
    using S image_iff by fastforce
  moreover have "⋀ h. h ∈ carrier G ⟹ ?φ h ∈ carrier G"
    using assms by simp
  hence "?φ ` carrier G ⊆ carrier G" by blast
  ultimately show "?φ ` carrier G = carrier G" by blast
qed

lemma(in group) conjugation_is_hom:
  "(λg. λh ∈ carrier G. g ⊗ h ⊗ inv g) ∈ hom G (BijGroup (carrier G))"
  unfolding hom_def
proof -
  let ?ψ = "λg. λh. g ⊗ h ⊗ inv g"
  let ?φ = "λg. restrict (?ψ g) (carrier G)"


  have Step0: "⋀ g. g ∈ carrier G ⟹ (?φ g) ∈ Bij (carrier G)"
    using Bij_def conjugation_is_bij by fastforce
  hence Step1: "?φ: carrier G → carrier (BijGroup (carrier G))"
    unfolding BijGroup_def by simp


  have "⋀ g1 g2. ⟦ g1 ∈ carrier G; g2 ∈ carrier G ⟧ ⟹
                  (⋀ h. h ∈ carrier G ⟹ ?ψ (g1 ⊗ g2) h = (?φ g1) ((?φ
g2) h))"
  proof -
```

```
    fix g1 g2 h assume g1: "g1 ∈ carrier G" and g2: "g2 ∈ carrier G"
and h: "h ∈ carrier G"
    have "inv (g1 ⊗ g2) = (inv g2) ⊗ (inv g1)"
      using g1 g2 by (simp add: inv_mult_group)
    thus "?ψ (g1 ⊗ g2) h  = (?φ g1) ((?φ g2) h)"
      by (simp add: g1 g2 h m_assoc)
  qed
  hence "⋀ g1 g2. ⟦ g1 ∈ carrier G; g2 ∈ carrier G ⟧ ⟹
        (λ h ∈ carrier G. ?ψ (g1 ⊗ g2) h) = (λ h ∈ carrier G. (?φ g1)
((?φ g2) h))" by auto
  hence Step2: "⋀ g1 g2. ⟦ g1 ∈ carrier G; g2 ∈ carrier G ⟧ ⟹
               ?φ (g1 ⊗ g2) = (?φ g1) ⊗BijGroup (carrier G) (?φ g2)"
    unfolding BijGroup_def by (simp add: Step0 compose_def)


  thus "?φ ∈ {h: carrier G → carrier (BijGroup (carrier G)).
              (∀x ∈ carrier G. ∀y ∈ carrier G. h (x ⊗ y) = h x ⊗BijGroup (carrier G)
h y)}"
    using Step1 Step2 by auto
qed

theorem (in group) action_by_conjugation:
  "group_action G (carrier G) (λg. (λh ∈ carrier G. g ⊗ h ⊗ (inv g)))"
  unfolding group_action_def group_hom_def using conjugation_is_hom
  by (simp add: group_BijGroup group_hom_axioms.intro is_group)
```

### 23.6.2 Action Over The Set of Subgroups

A Group Acts by Conjugation Over The Set of Subgroups

```
lemma (in group) subgroup_conjugation_is_inj_aux:
  assumes "g ∈ carrier G" "H1 ⊆ carrier G" "H2 ⊆ carrier G"
    and "g <# H1 #> (inv g) = g <# H2 #> (inv g)"
    shows "H1 ⊆ H2"
proof
  fix h1 assume h1: "h1 ∈ H1"
  hence "g ⊗ h1 ⊗ (inv g) ∈ g <# H1 #> (inv g)"
    unfolding l_coset_def r_coset_def using assms by blast
  hence "g ⊗ h1 ⊗ (inv g) ∈ g <# H2 #> (inv g)"
    using assms by auto
  hence "∃h2 ∈ H2. g ⊗ h1 ⊗ (inv g) = g ⊗ h2 ⊗ (inv g)"
      unfolding l_coset_def r_coset_def by blast
  then obtain h2 where "h2 ∈ H2 ∧ g ⊗ h1 ⊗ (inv g) = g ⊗ h2 ⊗ (inv
g)" by blast
  thus "h1 ∈ H2"
    using assms conjugation_is_inj h1 by blast
qed

lemma (in group) subgroup_conjugation_is_inj:
  assumes "g ∈ carrier G" "H1 ⊆ carrier G" "H2 ⊆ carrier G"
```

```
    and "g <# H1 #> (inv g) = g <# H2 #> (inv g)"
    shows "H1 = H2"
  using subgroup_conjugation_is_inj_aux assms set_eq_subset by metis

lemma (in group) subgroup_conjugation_is_surj0:
  assumes "g ∈ carrier G" "H ⊆ carrier G"
  shows "g <# ((inv g) <# H #> g) #> (inv g) = H"
  using coset_assoc assms coset_mult_assoc l_coset_subset_G lcos_m_assoc
  by (simp add: lcos_mult_one)

lemma (in group) subgroup_conjugation_is_surj1:
  assumes "g ∈ carrier G" "subgroup H G"
  shows "subgroup ((inv g) <# H #> g) G"
proof
  show "𝟭 ∈ inv g <# H #> g"
  proof -
    have "𝟭 ∈ H" by (simp add: assms(2) subgroup.one_closed)
    hence "inv g ⊗ 𝟭 ⊗ g ∈ inv g <# H #> g"
      unfolding l_coset_def r_coset_def by blast
    thus "𝟭 ∈ inv g <# H #> g" using assms by simp
  qed
next
  show "inv g <# H #> g ⊆ carrier G"
  proof
    fix x assume "x ∈ inv g <# H #> g"
    hence "∃h ∈ H. x = (inv g) ⊗ h ⊗ g"
      unfolding r_coset_def l_coset_def by blast
    hence "∃h ∈ (carrier G). x = (inv g) ⊗ h ⊗ g"
      by (meson assms subgroup.mem_carrier)
    thus "x ∈ carrier G" using assms by blast
  qed
next
  show " ⋀ x y. ⟦ x ∈ inv g <# H #> g; y ∈ inv g <# H #> g ⟧ ⟹ x ⊗
y ∈ inv g <# H #> g"
  proof -
    fix x y assume "x ∈ inv g <# H #> g"  "y ∈ inv g <# H #> g"
    then obtain h1 h2 where h12: "h1 ∈ H" "h2 ∈ H" and "x = (inv g)
⊗ h1 ⊗ g ∧ y = (inv g) ⊗ h2 ⊗ g"
      unfolding l_coset_def r_coset_def by blast
    hence "x ⊗ y = ((inv g) ⊗ h1 ⊗ g) ⊗ ((inv g) ⊗ h2 ⊗ g)" by blast
    also have "... = ((inv g) ⊗ h1 ⊗ (g ⊗ inv g) ⊗ h2 ⊗ g)"
      using h12 assms inv_closed  m_assoc m_closed subgroup.mem_carrier
[OF ‹subgroup H G›] by presburger
    also have "... = ((inv g) ⊗ (h1 ⊗ h2) ⊗ g)"
      by (simp add: h12 assms m_assoc subgroup.mem_carrier [OF ‹subgroup
H G›])
    finally have "∃ h ∈ H. x ⊗ y = (inv g) ⊗ h ⊗ g"
      by (meson assms(2) h12 subgroup_def)
    thus "x ⊗ y ∈ inv g <# H #> g"
```

```
      unfolding l_coset_def r_coset_def by blast
  qed
next
  show "⋀x. x ∈ inv g <# H #> g ⟹ inv x ∈ inv g <# H #> g"
  proof -
    fix x assume "x ∈ inv g <# H #> g"
    hence "∃h ∈ H. x = (inv g) ⊗ h ⊗ g"
      unfolding r_coset_def l_coset_def by blast
    then obtain h where h: "h ∈ H ∧ x = (inv g) ⊗ h ⊗ g" by blast
    hence "x ⊗ (inv g) ⊗ (inv h) ⊗ g = 1"
      using assms m_assoc monoid_axioms by (simp add: subgroup.mem_carrier)
    hence "inv x = (inv g) ⊗ (inv h) ⊗ g"
      using assms h inv_mult_group m_assoc monoid_axioms by (simp add:
subgroup.mem_carrier)
    moreover have "inv h ∈ H"
      by (simp add: assms h subgroup.m_inv_closed)
    ultimately show "inv x ∈ inv g <# H #> g" unfolding r_coset_def l_coset_def
by blast
  qed
qed

lemma (in group) subgroup_conjugation_is_surj2:
  assumes "g ∈ carrier G" "subgroup H G"
  shows "subgroup (g <# H #> (inv g)) G"
  using subgroup_conjugation_is_surj1 by (metis assms inv_closed inv_inv)

lemma (in group) subgroup_conjugation_is_bij:
  assumes "g ∈ carrier G"
  shows "bij_betw (λH ∈ {H. subgroup H G}. g <# H #> (inv g)) {H. subgroup
H G} {H. subgroup H G}"
         (is "bij_betw ?φ {H. subgroup H G} {H. subgroup H G}")
  unfolding bij_betw_def
proof
  show "inj_on ?φ {H. subgroup H G}"
    using subgroup_conjugation_is_inj assms inj_on_def subgroup.subset
    by (metis (mono_tags, lifting) inj_on_restrict_eq mem_Collect_eq)
next
  have "⋀H. H ∈ {H. subgroup H G} ⟹ ?φ ((inv g) <# H #> g) = H"
    by (simp add: assms subgroup.subset subgroup_conjugation_is_surj0
                  subgroup_conjugation_is_surj1 is_group)
  hence "⋀H. H ∈ {H. subgroup H G} ⟹ ∃H' ∈ {H. subgroup H G}. ?φ H'
= H"
    using assms subgroup_conjugation_is_surj1 by fastforce
  thus "?φ ` {H. subgroup H G} = {H. subgroup H G}"
    using subgroup_conjugation_is_surj2 assms by auto
qed

lemma (in group) subgroup_conjugation_is_hom:
  "(λg. λH ∈ {H. subgroup H G}. g <# H #> (inv g)) ∈ hom G (BijGroup
```

```
{H. subgroup H G})"
  unfolding hom_def
proof -

  let ?ψ = "λg. λH. g <# H #> (inv g)"
  let ?φ = "λg. restrict (?ψ g) {H. subgroup H G}"

  have Step0: "⋀ g. g ∈ carrier G ⟹ (?φ g) ∈ Bij {H. subgroup H G}"
    using Bij_def subgroup_conjugation_is_bij by fastforce
  hence Step1: "?φ: carrier G → carrier (BijGroup {H. subgroup H G})"
    unfolding BijGroup_def by simp

  have "⋀ g1 g2. ⟦ g1 ∈ carrier G; g2 ∈ carrier G ⟧ ⟹
                  (⋀ H. H ∈ {H. subgroup H G} ⟹ ?ψ (g1 ⊗ g2) H = (?φ
g1) ((?φ g2) H))"
  proof -
    fix g1 g2 H assume g1: "g1 ∈ carrier G" and g2: "g2 ∈ carrier G"
and H': "H ∈ {H. subgroup H G}"
    hence H: "subgroup H G" by simp
    have "(?φ g1) ((?φ g2) H) = g1 <# (g2 <# H #> (inv g2)) #> (inv g1)"
      by (simp add: H g2 subgroup_conjugation_is_surj2)
    also have " ... = g1 <# (g2 <# H) #> ((inv g2) ⊗ (inv g1))"
      by (simp add: H coset_mult_assoc g1 g2 group.coset_assoc
                    is_group l_coset_subset_G subgroup.subset)
    also have " ... = g1 <# (g2 <# H) #> inv (g1 ⊗ g2)"
      using g1 g2 by (simp add: inv_mult_group)
    finally have "(?φ g1) ((?φ g2) H) = ?ψ (g1 ⊗ g2) H"
      by (simp add: H g1 g2 lcos_m_assoc subgroup.subset)
    thus "?ψ (g1 ⊗ g2) H = (?φ g1) ((?φ g2) H)" by auto
  qed
  hence "⋀ g1 g2. ⟦ g1 ∈ carrier G; g2 ∈ carrier G ⟧ ⟹
         (λH ∈ {H. subgroup H G}. ?ψ (g1 ⊗ g2) H) = (λH ∈ {H. subgroup
H G}. (?φ g1) ((?φ g2) H))"
    by (meson restrict_ext)
  hence Step2: "⋀ g1 g2. ⟦ g1 ∈ carrier G; g2 ∈ carrier G ⟧ ⟹
                ?φ (g1 ⊗ g2) = (?φ g1) ⊗BijGroup {H. subgroup H G} (?φ
g2)"
    unfolding BijGroup_def by (simp add: Step0 compose_def)

  show "?φ ∈ {h: carrier G → carrier (BijGroup {H. subgroup H G}).
              ∀x∈carrier G. ∀y∈carrier G. h (x ⊗ y) = h x ⊗BijGroup {H. subgroup H G}
h y}"
    using Step1 Step2 by auto
qed

theorem (in group) action_by_conjugation_on_subgroups_set:
  "group_action G {H. subgroup H G} (λg. λH ∈ {H. subgroup H G}. g <#
H #> (inv g))"
  unfolding group_action_def group_hom_def using subgroup_conjugation_is_hom
```

```
  by (simp add: group_BijGroup group_hom_axioms.intro is_group)
```

### 23.6.3 Action Over The Power Set

A Group Acts by Conjugation Over The Power Set

```
lemma (in group) subset_conjugation_is_bij:
  assumes "g ∈ carrier G"
  shows "bij_betw (λH ∈ {H. H ⊆ carrier G}. g <# H #> (inv g)) {H. H
⊆ carrier G} {H. H ⊆ carrier G}"
         (is "bij_betw ?φ {H. H ⊆ carrier G} {H. H ⊆ carrier G}")
  unfolding bij_betw_def
proof
  show "inj_on ?φ {H. H ⊆ carrier G}"
    using subgroup_conjugation_is_inj assms inj_on_def
    by (metis (mono_tags, lifting) inj_on_restrict_eq mem_Collect_eq)
next
  have "⋀H. H ∈ {H. H ⊆ carrier G} ⟹ ?φ ((inv g) <# H #> g) = H"
    by (simp add: assms l_coset_subset_G r_coset_subset_G subgroup_conjugation_is_surj0)
  hence "⋀H. H ∈ {H. H ⊆ carrier G} ⟹ ∃H' ∈ {H. H ⊆ carrier G}. ?φ
H' = H"
    by (metis assms l_coset_subset_G mem_Collect_eq r_coset_subset_G subgroup_def
subgroup_self)
  hence "{H. H ⊆ carrier G} ⊆ ?φ ` {H. H ⊆ carrier G}" by blast
  moreover have "?φ ` {H. H ⊆ carrier G} ⊆ {H. H ⊆ carrier G}"
    by clarsimp (meson assms contra_subsetD inv_closed l_coset_subset_G
r_coset_subset_G)
  ultimately show "?φ ` {H. H ⊆ carrier G} = {H. H ⊆ carrier G}" by
simp
qed

lemma (in group) subset_conjugation_is_hom:
  "(λg. λH ∈ {H. H ⊆ carrier G}. g <# H #> (inv g)) ∈ hom G (BijGroup
{H. H ⊆ carrier G})"
  unfolding hom_def
proof -

  let ?ψ = "λg. λH. g <# H #> (inv g)"
  let ?φ = "λg. restrict (?ψ g) {H. H ⊆ carrier G}"

  have Step0: "⋀ g. g ∈ carrier G ⟹ (?φ g) ∈ Bij {H. H ⊆ carrier G}"
    using Bij_def subset_conjugation_is_bij by fastforce
  hence Step1: "?φ: carrier G → carrier (BijGroup {H. H ⊆ carrier G})"
    unfolding BijGroup_def by simp

  have "⋀ g1 g2. ⟦ g1 ∈ carrier G; g2 ∈ carrier G ⟧ ⟹
                  (⋀ H. H ∈ {H. H ⊆ carrier G} ⟹ ?ψ (g1 ⊗ g2) H =
(?φ g1) ((?φ g2) H))"
  proof -
    fix g1 g2 H assume g1: "g1 ∈ carrier G" and g2: "g2 ∈ carrier G"
```

```
and H: "H ∈ {H. H ⊆ carrier G}"
    hence "(?φ g1) ((?φ g2) H) = g1 <# (g2 <# H #> (inv g2)) #> (inv
g1)"
      using l_coset_subset_G r_coset_subset_G by auto
    also have " ... = g1 <# (g2 <# H) #> ((inv g2) ⊗ (inv g1))"
      using H coset_assoc coset_mult_assoc g1 g2 l_coset_subset_G by auto
    also have " ... = g1 <# (g2 <# H) #> inv (g1 ⊗ g2)"
      using g1 g2 by (simp add: inv_mult_group)
    finally have "(?φ g1) ((?φ g2) H) = ?ψ (g1 ⊗ g2) H"
      using H g1 g2 lcos_m_assoc by force
    thus "?ψ (g1 ⊗ g2) H = (?φ g1) ((?φ g2) H)" by auto
  qed
  hence "⋀ g1 g2. ⟦ g1 ∈ carrier G; g2 ∈ carrier G ⟧ ⟹
        (λH ∈ {H. H ⊆ carrier G}. ?ψ (g1 ⊗ g2) H) = (λH ∈ {H. H ⊆ carrier
G}. (?φ g1) ((?φ g2) H))"
    by (meson restrict_ext)
  hence Step2: "⋀ g1 g2. ⟦ g1 ∈ carrier G; g2 ∈ carrier G ⟧ ⟹
              ?φ (g1 ⊗ g2) = (?φ g1) ⊗BijGroup {H. H ⊆ carrier G} (?φ
g2)"
    unfolding BijGroup_def by (simp add: Step0 compose_def)

  show "?φ ∈ {h: carrier G → carrier (BijGroup {H. H ⊆ carrier G}).
             ∀x∈carrier G. ∀y∈carrier G. h (x ⊗ y) = h x ⊗BijGroup {H. H ⊆ carrier G}
h y}"
    using Step1 Step2 by auto
qed

theorem (in group) action_by_conjugation_on_power_set:
  "group_action G {H. H ⊆ carrier G} (λg. λH ∈ {H. H ⊆ carrier G}. g
<# H #> (inv g))"
  unfolding group_action_def group_hom_def using subset_conjugation_is_hom
  by (simp add: group_BijGroup group_hom_axioms.intro is_group)

corollary (in group) normalizer_imp_subgroup:
  assumes "H ⊆ carrier G"
  shows "subgroup (normalizer G H) G"
  unfolding normalizer_def
  using group_action.stabilizer_subgroup[OF action_by_conjugation_on_power_set]
assms by auto
```

## 23.7 Subgroup of an Acting Group

A Subgroup of an Acting Group Induces an Action

```
lemma (in group_action) induced_homomorphism:
  assumes "subgroup H G"
  shows "φ ∈ hom (G ⦇carrier := H⦈) (BijGroup E)"
  unfolding hom_def apply simp
proof -
  have S0: "H ⊆ carrier G" by (meson assms subgroup_def)
```

```
  hence "φ: H → carrier (BijGroup E)"
    by (simp add: BijGroup_def bij_prop0 subset_eq)
  thus "φ: H → carrier (BijGroup E) ∧ (∀x ∈ H. ∀y ∈ H. φ (x ⊗ y) =
φ x ⊗BijGroup E φ y)"
    by (simp add: S0  group_hom group_hom.hom_mult rev_subsetD)
qed

theorem (in group_action) induced_action:
  assumes "subgroup H G"
  shows "group_action (G (|carrier := H|)) E φ"
  unfolding group_action_def group_hom_def
  using induced_homomorphism assms group.subgroup_imp_group group_BijGroup
        group_hom group_hom.axioms(1) group_hom_axioms_def by blast

end
```

# 24 The Zassenhaus Lemma

```
theory Zassenhaus
  imports Coset Group_Action
begin
```

Proves the second isomorphism theorem and the Zassenhaus lemma.

## 24.1 Lemmas about normalizer

```
lemma (in group) subgroup_in_normalizer:
  assumes "subgroup H G"
  shows "normal H (G(|carrier:= (normalizer G H)|))"
proof(intro group.normal_invI)
  show "Group.group (G(|carrier := normalizer G H|))"
    by (simp add: assms group.normalizer_imp_subgroup is_group subgroup_imp_group
subgroup.subset)
  have K:"H ⊆ (normalizer G H)" unfolding normalizer_def
  proof
    fix x assume xH: "x ∈ H"
    from xH have xG : "x ∈ carrier G" using subgroup.subset assms by
auto
    have "x <# H = H"
      by (metis ‹x ∈ H› assms group.lcos_mult_one is_group
          l_repr_independence one_closed subgroup.subset)
    moreover have "H #> inv x = H"
      by (simp add: xH assms is_group subgroup.rcos_const subgroup.m_inv_closed)
    ultimately have "x <# H #> (inv x) = H" by simp
    thus " x ∈ stabilizer G (λg. λH∈{H. H ⊆ carrier G}. g <# H #> inv
g) H"
      using assms xG subgroup.subset unfolding stabilizer_def by auto
  qed
  thus "subgroup H (G(|carrier:= (normalizer G H)|))"
```

```
    using subgroup_incl normalizer_imp_subgroup assms by (simp add: subgroup.subset)
  show  " ⋀x h. x ∈ carrier (G(|carrier := normalizer G H|)) ⟹ h ∈ H
⟹
             x ⊗G(|carrier := normalizer G H|) h
               ⊗G(|carrier := normalizer G H|) invG(|carrier := normalizer G H|)
x ∈ H"
    proof-
    fix x h assume xnorm : "x ∈ carrier (G(|carrier := normalizer G H|))"
and hH : "h ∈ H"
    have xnormalizer:"x ∈ normalizer G H" using xnorm by simp
    moreover have hnormalizer:"h ∈ normalizer G H" using hH K by auto
    ultimately have "x ⊗G(|carrier := normalizer G H|) h = x ⊗ h" by simp
    moreover have " invG(|carrier := normalizer G H|) x =  inv x"
      using xnormalizer
      by (simp add: assms normalizer_imp_subgroup subgroup.subset m_inv_consistent)
    ultimately  have xhxegal: "x ⊗G(|carrier := normalizer G H|) h
               ⊗G(|carrier := normalizer G H|) invG(|carrier := normalizer G H|)
x
                  = x ⊗h ⊗ inv x"
      using  hnormalizer by simp
    have  "x ⊗h ⊗ inv x ∈ (x <# H #> inv x)"
      unfolding l_coset_def r_coset_def using hH  by auto
    moreover have "x <# H #> inv x = H"
      using xnormalizer assms subgroup.subset[OF assms]
      unfolding normalizer_def stabilizer_def by auto
    ultimately have "x ⊗h ⊗ inv x ∈ H" by simp
    thus  " x ⊗G(|carrier := normalizer G H|) h
               ⊗G(|carrier := normalizer G H|) invG(|carrier := normalizer G H|)
x ∈ H"
      using xhxegal hH xnorm by simp
  qed
qed


lemma (in group) normal_imp_subgroup_normalizer:
  assumes "subgroup H G"
    and "N ◁ (G(|carrier := H|))"
  shows "subgroup H (G(|carrier := normalizer G N|))"
proof-
  have N_carrierG : "N ⊆ carrier(G)"
    using assms normal_imp_subgroup subgroup.subset
    using incl_subgroup by blast
  {have "H ⊆ normalizer G N" unfolding normalizer_def stabilizer_def
    proof
      fix x assume xH : "x ∈ H"
      hence xcarrierG : "x ∈ carrier(G)" using assms subgroup.subset
by auto
      have "   N #> x = x <# N" using assms xH
        unfolding r_coset_def l_coset_def normal_def normal_axioms_def
```

```
subgroup_imp_group by auto
      hence "x <# N #> inv x =(N #> x) #> inv x"
        by simp
      also have "... = N #> 1"
        using  assms r_inv xcarrierG coset_mult_assoc[OF N_carrierG] by
simp
      finally have "x <# N #> inv x = N" by (simp add: N_carrierG)
      thus "x ∈ {g ∈ carrier G. (λH∈{H. H ⊆ carrier G}. g <# H #> inv
g) N = N}"
        using xcarrierG by (simp add : N_carrierG)
    qed}
  thus "subgroup H (G(|carrier := normalizer G N|))"
    using subgroup_incl[OF assms(1) normalizer_imp_subgroup]
      assms normal_imp_subgroup subgroup.subset
    by (metis  group.incl_subgroup is_group)
qed
```

## 24.2 Second Isomorphism Theorem

```
lemma (in group) mult_norm_subgroup:
  assumes "normal N G"
    and "subgroup H G"
  shows "subgroup (N<#>H) G" unfolding subgroup_def
proof-
  have  A :"N <#> H ⊆ carrier G"
    using assms  setmult_subset_G by (simp add: normal_imp_subgroup subgroup.subset)

  have B :"⋀ x y. ⟦x ∈ (N <#> H); y ∈ (N <#> H)⟧ ⟹ (x ⊗ y) ∈ (N<#>H)"
  proof-
    fix x y assume B1a: "x ∈ (N <#> H)"  and B1b: "y ∈ (N <#> H)"
    obtain n1 h1 where B2:"n1 ∈ N ∧ h1 ∈ H ∧ n1⊗h1 = x"
      using set_mult_def B1a by (metis (no_types, lifting) UN_E singletonD)
    obtain n2 h2 where B3:"n2 ∈ N ∧ h2 ∈ H ∧ n2⊗h2 = y"
      using set_mult_def B1b by (metis (no_types, lifting) UN_E singletonD)
    have "N #> h1 = h1 <# N"
      using normalI B2 assms normal.coset_eq subgroup.subset by blast
    hence "h1⊗n2 ∈ N #> h1"
      using B2 B3 assms l_coset_def by fastforce
    from this obtain y2 where y2_def:"y2 ∈ N" and y2_prop:"y2⊗h1 = h1⊗n2"
      using singletonD by (metis (no_types, lifting) UN_E r_coset_def)
    have "⋀a. a ∈ N ⟹ a ∈ carrier G"  "⋀a. a ∈ H ⟹ a ∈ carrier
G"
      by (meson assms normal_def subgroup.mem_carrier)+
    then have "x⊗y =  n1 ⊗ y2 ⊗ h1 ⊗ h2" using y2_def B2 B3
      by (metis (no_types) B2 B3 ‹⋀a. a ∈ N ⟹ a ∈ carrier G› m_assoc
m_closed y2_def y2_prop)
    moreover have B4 :"n1 ⊗ y2 ∈N"
      using B2 y2_def assms normal_imp_subgroup by (metis subgroup_def)
    moreover have "h1 ⊗ h2 ∈H" using B2 B3 assms by (simp add: subgroup.m_closed)
```

```
    hence "(n1 ⊗ y2) ⊗ (h1 ⊗ h2) ∈(N<#>H) "
      using B4  unfolding set_mult_def by auto
    hence "n1 ⊗ y2 ⊗ h1 ⊗ h2 ∈(N<#>H)"
      using m_assoc B2 B3 assms  normal_imp_subgroup by (metis B4 subgroup.mem_carrier)
    ultimately show  "x ⊗ y ∈ N <#> H" by auto
  qed
  have C :"⋀ x. x∈(N<#>H)  ⟹ (inv x)∈(N<#>H)"

  proof-
    fix x assume C1 : "x ∈ (N<#>H)"
    obtain n h where C2:"n ∈ N ∧ h ∈ H ∧ n⊗h = x"
      using set_mult_def C1 by (metis (no_types, lifting) UN_E singletonD)
    have C3 :"inv(n⊗h) = inv(h)⊗inv(n)"
      by (meson C2  assms inv_mult_group normal_imp_subgroup subgroup.mem_carrier)
    hence "... ⊗h ∈ N"
      using assms C2
      by (meson normal.inv_op_closed1 normal_def subgroup.m_inv_closed
subgroup.mem_carrier)
    hence  C4:"(inv h ⊗ inv n ⊗ h) ⊗ inv h ∈ (N<#>H)"
      using   C2 assms subgroup.m_inv_closed[of H G h] unfolding set_mult_def
by auto
    have "inv h ⊗ inv n ⊗ h ⊗ inv h = inv h ⊗ inv n"
      using  subgroup.subset[OF assms(2)]
      by (metis A C1 C2 C3 inv_closed inv_solve_right m_closed subsetCE)
    thus "inv(x)∈N<#>H" using C4 C2 C3 by simp
  qed

  have D : "𝟭 ∈ N <#> H"
  proof-
    have D1 : "𝟭 ∈ N"
      using assms by (simp add: normal_def subgroup.one_closed)
     have D2 :"𝟭 ∈ H"
      using assms by (simp add: subgroup.one_closed)
    thus "𝟭 ∈ (N <#> H)"
      using set_mult_def D1 assms by fastforce
  qed
  thus "(N <#> H ⊆ carrier G ∧ (∀x y. x ∈ N <#> H ⟶ y ∈ N <#> H ⟶
x ⊗ y ∈ N <#> H)) ∧
    𝟭 ∈ N <#> H ∧ (∀x. x ∈ N <#> H ⟶ inv x ∈ N <#> H)" using A B C
D assms by blast
qed


lemma (in group) mult_norm_sub_in_sub:
  assumes "normal N (G⦇carrier:=K⦈)"
  assumes "subgroup H (G⦇carrier:=K⦈)"
  assumes "subgroup K G"
  shows  "subgroup (N<#>H) (G⦇carrier:=K⦈)"
proof-
```

```
  have Hyp:"subgroup (N <#>G(|carrier := K|) H) (G(|carrier := K|))"
    using group.mult_norm_subgroup[where ?G = "G(|carrier := K|)"] assms
subgroup_imp_group by auto
  have "H ⊆ carrier(G(|carrier := K|))" using assms subgroup.subset by
blast
  also have "... ⊆ K" by simp
  finally have Incl1:"H ⊆ K" by simp
  have "N ⊆ carrier(G(|carrier := K|))" using assms normal_imp_subgroup
subgroup.subset by blast
  also have "... ⊆ K" by simp
  finally have Incl2:"N ⊆ K" by simp
  have "(N <#>G(|carrier := K|) H) = (N <#> H)"
    using set_mult_consistent by simp
  thus "subgroup (N<#>H) (G(|carrier:=K|))" using Hyp by auto
qed


lemma (in group) subgroup_of_normal_set_mult:
  assumes "normal N G"
and "subgroup H G"
shows "subgroup H (G(|carrier := N <#> H|))"
proof-
  have "1 ∈ N" using normal_imp_subgroup assms(1) subgroup_def by blast
  hence "1 <# H ⊆ N <#> H" unfolding set_mult_def l_coset_def by blast
  hence H_incl : "H ⊆ N <#> H"
    by (metis assms(2) lcos_mult_one subgroup_def)
  show "subgroup H (G(|carrier := N <#> H|))"
  using subgroup_incl[OF assms(2) mult_norm_subgroup[OF assms(1) assms(2)]
H_incl] .
qed


lemma (in group) normal_in_normal_set_mult:
  assumes "normal N G"
and "subgroup H G"
shows "normal N (G(|carrier := N <#> H|))"
proof-
  have "1 ∈ H" using  assms(2) subgroup_def by blast
  hence "N #> 1  ⊆ N <#> H" unfolding set_mult_def r_coset_def by blast
  hence N_incl : "N ⊆ N <#> H"
    by (metis assms(1) normal_imp_subgroup coset_mult_one subgroup_def)
  thus "normal N (G(|carrier := N <#> H|))"
    using normal_Int_subgroup[OF mult_norm_subgroup[OF assms] assms(1)]
    by (simp add : inf_absorb1)
qed


proposition (in group) weak_snd_iso_thme:
  assumes "subgroup  H G"
```

```
    and "N⊲G"
  shows "(G⦇carrier := N<#>H⦈ Mod N ≅ G⦇carrier:=H⦈ Mod (N∩H))"
proof-
  define f where "f =  (#>) N"
  have GroupNH : "Group.group (G⦇carrier := N<#>H⦈)"
    using subgroup_imp_group assms mult_norm_subgroup by simp
  have  HcarrierNH :"H ⊆ carrier(G⦇carrier := N<#>H⦈)"
    using assms subgroup_of_normal_set_mult subgroup.subset by blast
  hence HNH :"H ⊆ N<#>H" by simp
  have op_hom : "f ∈ hom (G⦇carrier := H⦈) (G⦇carrier := N <#> H⦈ Mod
N)" unfolding hom_def
  proof
    have "⋀x . x ∈ carrier (G⦇carrier :=H⦈) ⟹
       (#>G⦇carrier := N <#> H⦈) N x ∈  carrier (G⦇carrier := N <#> H⦈ Mod
N)"
    proof-
      fix x assume  "x ∈ carrier (G⦇carrier :=H⦈)"
      hence xH : "x ∈ H" by simp
      hence "(#>G⦇carrier := N <#> H⦈) N x ∈ rcosetsG⦇carrier := N <#> H⦈
N"
        using HcarrierNH RCOSETS_def[where ?G = "G⦇carrier := N <#> H⦈"]
by blast
      thus "(#>G⦇carrier := N <#> H⦈) N x ∈  carrier (G⦇carrier := N <#>
H⦈ Mod N)"
        unfolding FactGroup_def by simp
    qed
    hence "(#>G⦇carrier := N <#> H⦈) N ∈ carrier (G⦇carrier :=H⦈) →
           carrier (G⦇carrier := N <#> H⦈ Mod N)" by auto
    hence "f ∈ carrier (G⦇carrier :=H⦈) → carrier (G⦇carrier := N <#>
H⦈ Mod N)"
      unfolding r_coset_def f_def  by simp
    moreover have "⋀x y. x∈carrier (G⦇carrier := H⦈) ⟹ y∈carrier (G⦇carrier
:= H⦈) ⟹
                  f (x ⊗G⦇carrier := H⦈ y) =  f(x) ⊗G⦇carrier := N <#> H⦈ Mod N
f(y)"
    proof-
      fix x y assume "x∈carrier (G⦇carrier := H⦈)" "y∈carrier (G⦇carrier
:= H⦈)"
      hence xHyH :"x ∈ H" "y ∈ H" by auto
      have Nxeq :"N #>G⦇carrier := N<#>H⦈ x = N #>x" unfolding r_coset_def
by simp
      have Nyeq :"N #>G⦇carrier := N<#>H⦈ y = N #>y" unfolding r_coset_def
by simp

      have "x ⊗G⦇carrier := H⦈ y =x ⊗G⦇carrier := N<#>H⦈ y" by simp
      hence "N #>G⦇carrier := N<#>H⦈ x ⊗G⦇carrier := H⦈ y
             = N #>G⦇carrier := N<#>H⦈ x ⊗G⦇carrier := N<#>H⦈ y" by simp
      also have "... = (N #>G⦇carrier := N<#>H⦈ x) <#>G⦇carrier := N<#>H⦈
```

```
                    (N #>G⦇carrier := N<#>H⦈ y)"
        using normal.rcos_sum[OF normal_in_normal_set_mult[OF assms(2)
assms(1)], of x y]
              xHyH assms HcarrierNH by auto
      finally show "f (x ⊗G⦇carrier := H⦈ y) =  f(x) ⊗G⦇carrier := N <#> H⦈ Mod N
f(y)"
        unfolding  FactGroup_def r_coset_def f_def  using Nxeq Nyeq  by
auto
    qed
    hence "(∀x∈carrier (G⦇carrier := H⦈). ∀y∈carrier (G⦇carrier := H⦈).
           f (x ⊗G⦇carrier := H⦈ y) =  f(x) ⊗G⦇carrier := N <#> H⦈ Mod N
f(y))" by blast
    ultimately show  " f ∈ carrier (G⦇carrier := H⦈) → carrier (G⦇carrier
:= N <#> H⦈ Mod N) ∧
    (∀x∈carrier (G⦇carrier := H⦈). ∀y∈carrier (G⦇carrier := H⦈).
     f (x ⊗G⦇carrier := H⦈ y) =  f(x) ⊗G⦇carrier := N <#> H⦈ Mod N f(y))"
      by auto
  qed
  hence homomorphism : "group_hom (G⦇carrier := H⦈) (G⦇carrier := N <#>
H⦈ Mod N) f"
    unfolding group_hom_def group_hom_axioms_def using subgroup_imp_group[OF
assms(1)]
              normal.factorgroup_is_group[OF normal_in_normal_set_mult[OF
assms(2) assms(1)]] by auto
  moreover have im_f :  "(f  ` carrier(G⦇carrier:=H⦈)) = carrier(G⦇carrier
:= N <#> H⦈ Mod N)"
  proof
    show  "f ` carrier (G⦇carrier := H⦈) ⊆ carrier (G⦇carrier := N <#>
H⦈ Mod N)"
      using op_hom unfolding hom_def using funcset_image by blast
  next
    show "carrier (G⦇carrier := N <#> H⦈ Mod N) ⊆ f ` carrier (G⦇carrier
:= H⦈)"
    proof
      fix x assume p : " x ∈ carrier (G⦇carrier := N <#> H⦈ Mod N)"
      hence "x ∈ ⋃{y. ∃x∈carrier (G⦇carrier := N <#> H⦈). y = {N #>G⦇carrier := N <#> H⦈
x}}"
        unfolding FactGroup_def RCOSETS_def by auto
      hence hyp :"∃y. ∃h∈carrier (G⦇carrier := N <#> H⦈). y = {N #>G⦇carrier := N <#> H⦈
h} ∧ x ∈ y"
        using Union_iff by blast
      from hyp obtain nh where nhNH:"nh ∈carrier (G⦇carrier := N <#>
H⦈)"
                        and "x ∈ {N #>G⦇carrier := N <#> H⦈ nh}"
        by blast
      hence K: "x = (#>G⦇carrier := N <#> H⦈) N nh" by simp
      have "nh ∈ N <#> H" using nhNH by simp
      from this obtain n h where nN : "n ∈ N" and hH : " h ∈ H" and
```

```
nhnh: "n ⊗ h = nh"
        unfolding set_mult_def by blast
      have  "x = (#>G(|carrier := N <#> H|)) N (n ⊗ h)" using K nhnh by simp
      hence  "x = (#>) N (n ⊗ h)" using K nhnh unfolding r_coset_def
by auto
      also have "... = (N #> n) #>h"
        using coset_mult_assoc hH nN assms subgroup.subset normal_imp_subgroup
        by (metis subgroup.mem_carrier)
      finally have "x = (#>) N h"
        using coset_join2[of n N] nN assms by (simp add: normal_imp_subgroup
subgroup.mem_carrier)
      thus "x ∈ f ` carrier (G(|carrier := H|))" using hH unfolding f_def
by simp
    qed
  qed
  moreover have ker_f :"kernel (G(|carrier := H|)) (G(|carrier := N<#>H|)
Mod N) f  = N∩H"
    unfolding kernel_def f_def
    proof-
      have "{x ∈ carrier (G(|carrier := H|)). N #> x = 1G(|carrier := N <#> H|) Mod N}
=
            {x ∈ carrier (G(|carrier := H|)). N #> x = N}" unfolding FactGroup_def
by simp
      also have "... = {x ∈ carrier (G(|carrier := H|)). x ∈ N}"
        using coset_join1
        by (metis (no_types, lifting) assms group.subgroup_self incl_subgroup
is_group
          normal_imp_subgroup subgroup.mem_carrier subgroup.rcos_const
subgroup_imp_group)
      also have "... =N ∩ (carrier(G(|carrier := H|)))" by auto
      finally show "{x ∈ carrier (G(|carrier := H|)). N#>x = 1G(|carrier := N <#> H|) Mod N}
= N ∩ H"
        by simp
    qed
    ultimately have "(G(|carrier := H|) Mod N ∩ H) ≅ (G(|carrier := N <#>
H|) Mod N)"
      using group_hom.FactGroup_iso[OF homomorphism im_f] by auto
    hence "G(|carrier := N <#> H|) Mod N ≅ G(|carrier := H|) Mod N ∩ H"
      by (simp add: group.iso_sym assms normal.factorgroup_is_group normal_Int_subgroup)
    thus "G(|carrier := N <#> H|) Mod N ≅ G(|carrier := H|) Mod N ∩ H" by
auto
qed


theorem (in group) snd_iso_thme:
  assumes "subgroup H G"
    and "subgroup N G"
    and "subgroup H (G(|carrier:= (normalizer G N)|))"
  shows "(G(|carrier:= N<#>H|) Mod N)  ≅ (G(|carrier:= H|) Mod (H∩N))"
```

```
proof-
  have "G(|carrier := normalizer G N, carrier := H|)
       = G(|carrier := H|)"  by simp
  hence "G(|carrier := normalizer G N, carrier := H|) Mod N ∩ H =
         G(|carrier := H|) Mod N ∩ H" by auto
  moreover have "G(|carrier := normalizer G N,
                    carrier := N <#>G(|carrier := normalizer G N|) H|) =
               G(|carrier := N <#>G(|carrier := normalizer G N|) H|)" by simp
  hence "G(|carrier := normalizer G N,
          carrier := N <#>G(|carrier := normalizer G N|) H|) Mod N =
          G(|carrier := N <#>G(|carrier := normalizer G N|) H|) Mod N" by auto
  hence "G(|carrier := normalizer G N,
          carrier := N <#>G(|carrier := normalizer G N|) H|) Mod N  ≅
         G(|carrier := normalizer G N, carrier := H|) Mod N ∩ H =
          (G(|carrier:= N<#>H|) Mod N)  ≅
         G(|carrier := normalizer G N, carrier := H|) Mod N ∩ H"
    using subgroup.subset[OF assms(3)]
          subgroup.subset[OF normal_imp_subgroup[OF subgroup_in_normalizer[OF
assms(2)]]]
    by simp
  ultimately have "G(|carrier := normalizer G N,
                    carrier := N <#>G(|carrier := normalizer G N|) H|) Mod N
≅
                  G(|carrier := normalizer G N, carrier := H|) Mod N ∩ H
=
                 (G(|carrier:= N<#>H|) Mod N)  ≅  G(|carrier := H|) Mod N
∩ H" by auto
  moreover have "G(|carrier := normalizer G N,
                    carrier := N <#>G(|carrier := normalizer G N|) H|) Mod N
≅
                  G(|carrier := normalizer G N, carrier := H|) Mod N ∩ H"
    using group.weak_snd_iso_thme[OF subgroup_imp_group[OF normalizer_imp_subgroup[OF
          subgroup.subset[OF assms(2)]]] assms(3) subgroup_in_normalizer[OF
assms(2)]]
    by simp
  moreover have "H∩N = N∩H" using assms  by auto
  ultimately show "(G(|carrier:= N<#>H|) Mod N)  ≅  G(|carrier := H|) Mod
H ∩ N" by auto
qed


corollary (in group) snd_iso_thme_recip :
  assumes "subgroup H G"
    and "subgroup N G"
    and "subgroup H (G(|carrier:= (normalizer G N)|))"
  shows "(G(|carrier:= H<#>N|) Mod N)  ≅ (G(|carrier:= H|) Mod (H∩N))"
  by (metis assms commut_normal_subgroup group.subgroup_in_normalizer
is_group subgroup.subset
```

```
    normalizer_imp_subgroup snd_iso_thme)
```

## 24.3 The Zassenhaus Lemma

```
lemma (in group) distinc:
  assumes "subgroup  H G"
    and "H1◁G(|carrier := H|)"
    and  "subgroup K G"
    and "K1◁G(|carrier:=K|)"
  shows "subgroup (H∩K) (G(|carrier:=(normalizer G (H1<#>(H∩K1))) |))"
proof (intro subgroup_incl[OF subgroups_Inter_pair[OF assms(1) assms(3)]])
  show "subgroup (normalizer G (H1 <#> H ∩ K1)) G"
    using normalizer_imp_subgroup assms normal_imp_subgroup subgroup.subset
    by (metis group.incl_subgroup is_group setmult_subset_G subgroups_Inter_pair)
next
  show "H ∩ K ⊆ normalizer G (H1 <#> H ∩ K1)" unfolding normalizer_def
stabilizer_def
  proof
    fix x assume xHK : "x ∈ H ∩ K"
    hence xG : "{x} ⊆ carrier G" "{inv x} ⊆ carrier G"
      using subgroup.subset assms inv_closed xHK by auto
    have allG : "H ⊆ carrier G" "K ⊆ carrier G" "H1 ⊆ carrier G"  "K1
⊆ carrier G"
      using assms subgroup.subset normal_imp_subgroup incl_subgroup ap-
ply blast+ .
    have HK1: "H ∩ K1 ⊆ carrier G"
      by (simp add: allG(1) le_infI1)
    have HK1_normal: "H∩K1 ◁ (G(|carrier :=  H ∩ K|))" using normal_inter[OF
assms(3)assms(1)assms(4)]
      by (simp add : inf_commute)
    have "H ∩ K ⊆ normalizer G (H ∩ K1)"
      using subgroup.subset[OF normal_imp_subgroup_normalizer[OF subgroups_Inter_pair[OF
            assms(1)assms(3)]HK1_normal]] by auto
    hence "x <# (H ∩ K1) #> inv x = (H ∩ K1)"
      using xHK subgroup.subset[OF subgroups_Inter_pair[OF assms(1) incl_subgroup[OF
assms(3)
                                                            normal_imp_subgroup[OF
assms(4)]]]]
      unfolding normalizer_def stabilizer_def by auto
    moreover have "H ⊆  normalizer G H1"
      using subgroup.subset[OF normal_imp_subgroup_normalizer[OF assms(1)assms(2)]]
by auto
    hence "x <# H1 #> inv x = H1"
      using xHK subgroup.subset[OF  incl_subgroup[OF assms(1) normal_imp_subgroup[OF
assms(2)]]]
      unfolding normalizer_def stabilizer_def by auto
    ultimately have "H1 <#> H ∩ K1 = (x <# H1 #> inv x) <#> (x <#  H ∩
K1 #> inv x)" by auto
    also have "... = ({x} <#> H1) <#> {inv x} <#> ({x} <#>  H ∩ K1 <#>
```

```
{inv x})"
      by (simp add : r_coset_eq_set_mult l_coset_eq_set_mult)
    also have "... = ({x} <#> H1 <#> {inv x} <#> {x}) <#>  (H ∩ K1 <#>
{inv x})"
      using HK1 allG(3) set_mult_assoc setmult_subset_G xG(1) by auto
    also have "... = ({x} <#> H1 <#> {1}) <#>  (H ∩ K1 <#> {inv x})"
      using allG xG coset_mult_assoc by (simp add: r_coset_eq_set_mult
setmult_subset_G)
    also have "... =({x} <#> H1) <#>  (H ∩ K1 <#> {inv x})"
      using coset_mult_one r_coset_eq_set_mult[of G H1 1] set_mult_assoc[OF
xG(1) allG(3)] allG
      by auto
    also have "... = {x} <#> (H1 <#> H ∩ K1) <#> {inv x}"
      using allG xG set_mult_assoc setmult_subset_G by (metis inf.coboundedI2)
    finally have "H1 <#> H ∩ K1 = x <# (H1 <#> H ∩ K1) #> inv x"
      using xG setmult_subset_G allG by (simp add: l_coset_eq_set_mult
r_coset_eq_set_mult)
    thus "x ∈ {g ∈ carrier G. (λH∈{H. H ⊆ carrier G}. g <# H #> inv g)
(H1 <#> H ∩ K1)
                                                                        =
H1 <#> H ∩ K1}"
      using xG allG setmult_subset_G[OF allG(3), where ?K = "H∩K1"] xHK
      by auto
  qed
qed

lemma (in group) preliminary1:
  assumes "subgroup  H G"
    and "H1◁G(|carrier := H|)"
    and  "subgroup K G"
    and "K1◁G(|carrier:=K|)"
  shows " (H∩K) ∩ (H1<#>(H∩K1)) = (H1∩K)<#>(H∩K1)"
proof
  have all_inclG : "H ⊆ carrier G" "H1 ⊆ carrier G" "K ⊆ carrier G"
"K1 ⊆ carrier G"
    using assms subgroup.subset normal_imp_subgroup incl_subgroup ap-
ply blast+.
  show "H ∩ K ∩ (H1 <#> H ∩ K1) ⊆ H1 ∩ K <#> H ∩ K1"
  proof
    fix x assume x_def : "x ∈ (H ∩ K) ∩ (H1 <#> (H ∩ K1))"
    from x_def have x_incl : "x ∈ H" "x ∈ K" "x ∈ (H1 <#> (H ∩ K1))"
by auto
    then obtain h1 hk1 where h1hk1_def : "h1 ∈ H1" "hk1 ∈ H ∩ K1" "h1
⊗ hk1 = x"
      using assms unfolding set_mult_def by blast
    hence "hk1 ∈ H ∩ K" using subgroup.subset[OF normal_imp_subgroup[OF
assms(4)]] by auto
    hence "inv hk1 ∈ H ∩ K" using subgroup.m_inv_closed[OF subgroups_Inter_pair]
assms by auto
```

```
    moreover have "h1 ⊗ hk1 ∈ H ∩ K" using x_incl h1hk1_def by auto
    ultimately have "h1 ⊗ hk1 ⊗ inv hk1 ∈ H ∩ K"
      using subgroup.m_closed[OF subgroups_Inter_pair] assms by auto
    hence "h1 ∈ H ∩ K" using  h1hk1_def assms subgroup.subset incl_subgroup
normal_imp_subgroup
      by (metis Int_iff contra_subsetD inv_solve_right m_closed)
    hence "h1 ∈ H1 ∩ H ∩ K" using h1hk1_def by auto
    hence "h1 ∈ H1 ∩ K" using subgroup.subset[OF normal_imp_subgroup[OF
assms(2)]] by auto
    hence "h1 ⊗ hk1 ∈ (H1∩K)<#>(H∩K1)"
      using h1hk1_def unfolding set_mult_def by auto
    thus " x ∈ (H1∩K)<#>(H∩K1)" using h1hk1_def x_def by auto
  qed
  show "H1 ∩ K <#> H ∩ K1 ⊆ H ∩ K ∩ (H1 <#> H ∩ K1)"
  proof-
    have "H1 ∩ K ⊆ H ∩ K" using subgroup.subset[OF normal_imp_subgroup[OF
assms(2)]] by auto
    moreover have "H ∩ K1 ⊆ H ∩ K"
      using subgroup.subset[OF normal_imp_subgroup[OF assms(4)]] by auto
    ultimately have "H1 ∩ K <#> H ∩ K1 ⊆ H ∩ K" unfolding set_mult_def
      using subgroup.m_closed[OF subgroups_Inter_pair [OF assms(1)assms(3)]]
by blast
    moreover have "H1 ∩ K ⊆ H1" by auto
    hence "H1 ∩ K <#> H ∩ K1 ⊆ (H1 <#> H ∩ K1)" unfolding set_mult_def
by auto
    ultimately show "H1 ∩ K <#> H ∩ K1 ⊆ H ∩ K ∩ (H1 <#> H ∩ K1)" by
auto
  qed
qed

lemma (in group) preliminary2:
  assumes "subgroup  H G"
    and "H1⊲G(|carrier := H|)"
    and  "subgroup K G"
    and "K1⊲G(|carrier:=K|)"
  shows "(H1<#>(H∩K1)) ⊲ G(|carrier:=(H1<#>(H∩K))|)"
proof-
  have all_inclG : "H ⊆ carrier G" "H1 ⊆ carrier G" "K ⊆ carrier G"
"K1 ⊆ carrier G"
    using assms subgroup.subset normal_imp_subgroup incl_subgroup ap-
ply blast+.
  have subH1:"subgroup (H1 <#> H ∩ K) (G(|carrier := H|))"
    using mult_norm_sub_in_sub[OF assms(2)subgroup_incl[OF subgroups_Inter_pair[OF
assms(1)assms(3)]
          assms(1)]] assms by auto
  have "Group.group (G(|carrier:=(H1<#>(H∩K))|))"
    using  subgroup_imp_group[OF incl_subgroup[OF assms(1) subH1]].
  moreover have subH2 : "subgroup (H1 <#> H ∩ K1) (G(|carrier := H|))"
    using mult_norm_sub_in_sub[OF assms(2) subgroup_incl[OF subgroups_Inter_pair[OF
```

```
          assms(1) incl_subgroup[OF assms(3)normal_imp_subgroup[OF assms(4)]]]]]
assms by auto
  hence "(H∩K1) ⊆ (H∩K)"
    using assms subgroup.subset normal_imp_subgroup monoid.cases_scheme
    by (metis inf.mono  partial_object.simps(1) partial_object.update_convs(1)
subset_refl)
  hence incl:"(H1<#>(H∩K1)) ⊆ H1<#>(H∩K)" using assms subgroup.subset
normal_imp_subgroup
    unfolding set_mult_def by blast
  hence "subgroup (H1 <#> H ∩ K1) (G(|carrier := (H1<#>(H∩K))|))"
    using assms subgroup_incl[OF incl_subgroup[OF assms(1)subH2]incl_subgroup[OF
assms(1)
          subH1]] normal_imp_subgroup subgroup.subset unfolding set_mult_def
by blast
  moreover have " (⋀ x. x∈carrier (G(|carrier := H1 <#> H ∩ K|)) ⟹
        H1 <#> H∩K1 #>G(|carrier := H1 <#> H∩K|) x = x <#G(|carrier := H1 <#> H∩K|)
(H1 <#> H∩K1))"
  proof-
    fix x assume  "x ∈carrier (G(|carrier := H1 <#> H ∩ K|))"
    hence x_def : "x ∈ H1 <#> H ∩ K" by simp
    from this obtain h1 hk where h1hk_def :"h1 ∈ H1" "hk ∈ H ∩ K" "h1
⊗ hk = x"
      unfolding set_mult_def by blast
    have HK1: "H ∩ K1 ⊆ carrier G"
      by (simp add: all_inclG(1) le_infI1)
    have xH : "x ∈ H" using subgroup.subset[OF subH1] using x_def by
auto
    hence allG : "h1 ∈ carrier G" "hk ∈ carrier G" "x ∈ carrier G"
      using assms subgroup.subset h1hk_def normal_imp_subgroup incl_subgroup
apply blast+.
    hence "x <#G(|carrier := H1 <#> H∩K|) (H1 <#> H∩K1) =h1 ⊗ hk <# (H1 <#>
H∩K1)"
      using subgroup.subset xH h1hk_def by (simp add: l_coset_def)
    also have "... = h1 <# (hk <# (H1 <#> H∩K1))"
      using lcos_m_assoc[OF subgroup.subset[OF incl_subgroup[OF assms(1)
subH1]]allG(1)allG(2)]
      by (metis allG(1) allG(2) assms(1) incl_subgroup lcos_m_assoc subH2
subgroup.subset)
    also have "... = h1 <# (hk <# H1 <#> H∩K1)"
      using set_mult_assoc all_inclG allG by (simp add: l_coset_eq_set_mult
inf.coboundedI1)
    also have "... = h1 <# (hk <# H1 #> 1 <#> H∩K1 #> 1)"
      using coset_mult_one allG all_inclG l_coset_subset_G
      by (simp add: inf.coboundedI2 setmult_subset_G)
    also have "... = h1 <# (hk <# H1 #> inv hk #> hk <#> H∩K1 #> inv hk
#> hk)"
      using all_inclG allG coset_mult_assoc l_coset_subset_G
      by (simp add: inf.coboundedI1 setmult_subset_G)
    finally have "x <#G(|carrier := H1 <#> H ∩ K|) (H1 <#> H ∩ K1)
```

```
              = h1 <# ((hk <# H1 #> inv hk) <#> (hk <# H∩K1 #> inv
hk) #> hk)"
      using rcos_assoc_lcos allG all_inclG HK1
      by (simp add: l_coset_subset_G r_coset_subset_G setmult_rcos_assoc)
    moreover have "H ⊆  normalizer G H1"
      using assms h1hk_def subgroup.subset[OF normal_imp_subgroup_normalizer]
by simp
    hence "⋀g. g ∈ H ⟹  g ∈ {g ∈ carrier G. (λH∈{H. H ⊆ carrier G}.
g <# H #> inv g) H1 = H1}"
      using all_inclG assms unfolding normalizer_def stabilizer_def by
auto
    hence "⋀g. g ∈ H ⟹  g <# H1 #> inv g = H1" using all_inclG by
simp
    hence "(hk <# H1 #> inv hk) = H1" using h1hk_def all_inclG by simp
    moreover have "H∩K ⊆ normalizer G (H∩K1)"
      using normal_inter[OF assms(3)assms(1)assms(4)] assms subgroups_Inter_pair
            subgroup.subset[OF normal_imp_subgroup_normalizer] by (simp
add: inf_commute)
    hence "⋀g. g∈H∩K ⟹ g∈{g∈carrier G. (λH∈{H. H ⊆ carrier G}. g
<# H #> inv g) (H∩K1) = H∩K1}"
      using all_inclG assms unfolding normalizer_def stabilizer_def by
auto
    hence "⋀g. g ∈ H∩K ⟹  g <# (H∩K1) #> inv g = H∩K1"
      using subgroup.subset[OF subgroups_Inter_pair[OF assms(1) incl_subgroup[OF
            assms(3)normal_imp_subgroup[OF assms(4)]]]] by auto
    hence "(hk <# H∩K1 #> inv hk) = H∩K1" using h1hk_def by simp
    ultimately have "x <#G⦇carrier := H1 <#> H ∩ K⦈ (H1 <#> H ∩ K1) = h1
<#(H1 <#> (H ∩ K1)#> hk)"
      by auto
    also have "... = h1 <# H1 <#> ((H ∩ K1)#> hk)"
      using set_mult_assoc[where ?M = "{h1}" and ?H = "H1" and ?K =
"(H ∩ K1)#> hk"] allG all_inclG
      by (simp add: l_coset_eq_set_mult inf.coboundedI2 r_coset_subset_G
setmult_rcos_assoc)
    also have "... = H1 <#> ((H ∩ K1)#> hk)"
      using coset_join3 allG incl_subgroup[OF assms(1)normal_imp_subgroup[OF
assms(2)]] h1hk_def
      by auto
    finally have eq1 : "x <#G⦇carrier := H1 <#> H ∩ K⦈ (H1 <#> H ∩ K1) =
H1 <#> (H ∩ K1) #> hk"
      by (simp add: allG(2) all_inclG inf.coboundedI2 setmult_rcos_assoc)
    have "H1 <#> H ∩ K1 #>G⦇carrier := H1 <#> H ∩ K⦈ x = H1 <#> H ∩ K1 #>
(h1 ⊗ hk)"
      using subgroup.subset xH h1hk_def by (simp add: r_coset_def)
    also have "... = H1 <#> H ∩ K1 #> h1 #> hk"
      using coset_mult_assoc by (simp add: allG all_inclG inf.coboundedI2
setmult_subset_G)
    also have"... =  H ∩ K1 <#> H1 #> h1 #> hk"
      using commut_normal_subgroup[OF assms(1)assms(2)subgroup_incl[OF
```

```
subgroups_Inter_pair[OF
          assms(1)incl_subgroup[OF assms(3)normal_imp_subgroup[OF assms(4)]]]assms(1)]]
by simp
    also have "... = H ∩ K1 <#> H1  #> hk"
      using coset_join2[OF allG(1)incl_subgroup[OF assms(1)normal_imp_subgroup]
            h1hk_def(1)] all_inclG allG assms by (metis inf.coboundedI2
setmult_rcos_assoc)
    finally  have "H1 <#> H ∩ K1 #>G(|carrier := H1 <#> H ∩ K|) x =H1 <#> H
∩ K1  #> hk"
      using commut_normal_subgroup[OF assms(1)assms(2)subgroup_incl[OF
subgroups_Inter_pair[OF
          assms(1)incl_subgroup[OF assms(3)normal_imp_subgroup[OF assms(4)]]]assms(1)]]
by simp
    thus " H1 <#> H ∩ K1 #>G(|carrier := H1 <#> H ∩ K|) x =
             x <#G(|carrier := H1 <#> H ∩ K|) (H1 <#> H ∩ K1)" using eq1 by
simp
  qed
  ultimately show "H1 <#> H ∩ K1 ◁ G(|carrier := H1 <#> H ∩ K|)"
    unfolding normal_def normal_axioms_def by auto
qed


proposition (in group)  Zassenhaus_1:
  assumes "subgroup  H G"
    and "H1◁G(|carrier := H|)"
    and  "subgroup K G"
    and "K1◁G(|carrier:=K|)"
  shows "(G(|carrier:= H1 <#> (H∩K)|) Mod (H1<#>H∩K1)) ≅ (G(|carrier:= (H∩K)|)
Mod  ((H1∩K)<#>(H∩K1)))"
proof-
  define N  and N1 where "N = (H∩K)" and "N1 =H1<#>(H∩K1)"
  have normal_N_N1 : "subgroup N (G(|carrier:=(normalizer G N1)|))"
    by (simp add: N1_def N_def assms distinc normal_imp_subgroup)
  have Hp:"(G(|carrier:= N<#>N1|) Mod N1)  ≅ (G(|carrier:= N|) Mod (N∩N1))"
  by (metis N1_def N_def assms incl_subgroup inf_le1 mult_norm_sub_in_sub
       normal_N_N1 normal_imp_subgroup snd_iso_thme_recip subgroup_incl
subgroups_Inter_pair)
  have H_simp: "N<#>N1 = H1<#> (H∩K)"
  proof-
    have H1_incl_G : "H1 ⊆ carrier G"
      using assms normal_imp_subgroup incl_subgroup subgroup.subset by
blast
    have K1_incl_G :"K1 ⊆ carrier G"
      using assms normal_imp_subgroup incl_subgroup subgroup.subset by
blast
    have "N<#>N1=  (H∩K)<#> (H1<#>(H∩K1))" by (auto simp add: N_def N1_def)
    also have "... = ((H∩K)<#>H1) <#>(H∩K1)"
      using set_mult_assoc[where ?M = "H∩K"] K1_incl_G H1_incl_G assms
      by (simp add: inf.coboundedI2 subgroup.subset)
```

```
    also have "... = (H1<#>(H∩K))<#>(H∩K1)"
      using commut_normal_subgroup assms subgroup_incl subgroups_Inter_pair
by auto
    also have "... =  H1 <#> ((H∩K)<#>(H∩K1))"
      using set_mult_assoc K1_incl_G H1_incl_G assms
      by (simp add: inf.coboundedI2 subgroup.subset)
    also have " ((H∩K)<#>(H∩K1)) = (H∩K)"
    proof (intro set_mult_subgroup_idem[where ?H = "H∩K" and ?N="H∩K1",
             OF subgroups_Inter_pair[OF assms(1) assms(3)]])
      show "subgroup (H ∩ K1) (G(|carrier := H ∩ K|))"
        using subgroup_incl[where ?I = "H∩K1" and ?J = "H∩K",OF subgroups_Inter_pair[OF
assms(1)
              incl_subgroup[OF assms(3) normal_imp_subgroup]] subgroups_Inter_pair]
assms
              normal_imp_subgroup by (metis inf_commute normal_inter)
    qed
    hence " H1 <#> ((H∩K)<#>(H∩K1)) =  H1 <#> ((H∩K))"
      by simp
    thus "N <#> N1 = H1 <#> H ∩ K"
      by (simp add: calculation)
  qed

  have "N∩N1 = (H1∩K)<#>(H∩K1)"
    using preliminary1 assms N_def N1_def by simp
  thus  "(G(|carrier:= H1 <#> (H∩K)|) Mod N1)  ≅ (G(|carrier:= N|) Mod  ((H1∩K)<#>(H∩K1)))"
    using H_simp Hp by auto
qed


theorem (in group) Zassenhaus:
  assumes "subgroup  H G"
    and "H1◁G(|carrier := H|)"
    and  "subgroup K G"
    and "K1◁G(|carrier:=K|)"
  shows "(G(|carrier:= H1 <#> (H∩K)|) Mod (H1<#>(H∩K1)))  ≅
         (G(|carrier:= K1 <#> (H∩K)|) Mod (K1<#>(K∩H1)))"
proof-
  define Gmod1 Gmod2 Gmod3 Gmod4
    where "Gmod1 = (G(|carrier:= H1 <#> (H∩K)|) Mod (H1<#>(H∩K1))) "
      and "Gmod2 = (G(|carrier:= K1 <#> (K∩H)|) Mod (K1<#>(K∩H1)))"
      and "Gmod3 = (G(|carrier:= (H∩K)|) Mod  ((H1∩K)<#>(H∩K1)))"
      and "Gmod4 = (G(|carrier:= (K∩H)|) Mod  ((K1∩H)<#>(K∩H1)))"
  have Hyp :  "Gmod1  ≅ Gmod3" "Gmod2  ≅  Gmod4"
    using Zassenhaus_1 assms Gmod1_def Gmod2_def Gmod3_def Gmod4_def by
auto
  have Hp : "Gmod3 = G(|carrier:= (K∩H)|) Mod ((K∩H1)<#>(K1∩H))"
    by (simp add: Gmod3_def inf_commute)
  have "(K∩H1)<#>(K1∩H) = (K1∩H)<#>(K∩H1)"
  proof (intro commut_normal_subgroup[OF subgroups_Inter_pair[OF assms(1)assms(3)]])
```

```
    show "K1 ∩ H ◁ G⦇carrier := H ∩ K⦈"
      using normal_inter[OF assms(3)assms(1)assms(4)] by (simp add: inf_commute)
   next
    show "subgroup (K ∩ H1) (G⦇carrier := H ∩ K⦈)"
      using subgroup_incl by (simp add: assms inf_commute normal_imp_subgroup
normal_inter)
  qed
  hence  "Gmod3  = Gmod4" using Hp Gmod4_def by simp
  hence "Gmod1 ≅ Gmod2"
    by (metis assms group.iso_sym iso_trans Hyp normal.factorgroup_is_group
Gmod2_def preliminary2)
  thus ?thesis using Gmod1_def Gmod2_def by (simp add: inf_commute)
qed

end
```

# 25 Divisibility in monoids and rings

```
theory Divisibility
  imports "HOL-Combinatorics.List_Permutation" Coset Group
begin
```

# 26 Factorial Monoids

## 26.1 Monoids with Cancellation Law

```
locale monoid_cancel = monoid +
  assumes l_cancel: "⟦c ⊗ a = c ⊗ b; a ∈ carrier G; b ∈ carrier G; c
∈ carrier G⟧ ⟹ a = b"
    and r_cancel: "⟦a ⊗ c = b ⊗ c; a ∈ carrier G; b ∈ carrier G; c ∈
carrier G⟧ ⟹ a = b"

lemma (in monoid) monoid_cancelI:
  assumes l_cancel: "⋀a b c. ⟦c ⊗ a = c ⊗ b; a ∈ carrier G; b ∈ carrier
G; c ∈ carrier G⟧ ⟹ a = b"
    and r_cancel: "⋀a b c. ⟦a ⊗ c = b ⊗ c; a ∈ carrier G; b ∈ carrier
G; c ∈ carrier G⟧ ⟹ a = b"
  shows "monoid_cancel G"
    by standard fact+

lemma (in monoid_cancel) is_monoid_cancel: "monoid_cancel G" ..

sublocale group ⊆ monoid_cancel
  by standard simp_all


locale comm_monoid_cancel = monoid_cancel + comm_monoid
```

```
lemma comm_monoid_cancelI:
  fixes G (structure)
  assumes "comm_monoid G"
  assumes cancel: "⋀a b c. ⟦a ⊗ c = b ⊗ c; a ∈ carrier G; b ∈ carrier
G; c ∈ carrier G⟧ ⟹ a = b"
  shows "comm_monoid_cancel G"
proof -
  interpret comm_monoid G by fact
  show "comm_monoid_cancel G"
    by unfold_locales (metis assms(2) m_ac(2))+
qed

lemma (in comm_monoid_cancel) is_comm_monoid_cancel: "comm_monoid_cancel
G"
  by intro_locales

sublocale comm_group ⊆ comm_monoid_cancel ..
```

## 26.2 Products of Units in Monoids

```
lemma (in monoid) prod_unit_l:
  assumes abunit[simp]: "a ⊗ b ∈ Units G"
    and aunit[simp]: "a ∈ Units G"
    and carr[simp]: "a ∈ carrier G"  "b ∈ carrier G"
  shows "b ∈ Units G"
proof -
  have c: "inv (a ⊗ b) ⊗ a ∈ carrier G" by simp

  have "(inv (a ⊗ b) ⊗ a) ⊗ b = inv (a ⊗ b) ⊗ (a ⊗ b)"
    by (simp add: m_assoc)
  also have "... = 1" by simp
  finally have li: "(inv (a ⊗ b) ⊗ a) ⊗ b = 1" .

  have "1 = inv a ⊗ a" by (simp add: Units_l_inv[symmetric])
  also have "... = inv a ⊗ 1 ⊗ a" by simp
  also have "... = inv a ⊗ ((a ⊗ b) ⊗ inv (a ⊗ b)) ⊗ a"
    by (simp add: Units_r_inv[OF abunit, symmetric] del: Units_r_inv)
  also have "... = ((inv a ⊗ a) ⊗ b) ⊗ inv (a ⊗ b) ⊗ a"
    by (simp add: m_assoc del: Units_l_inv)
  also have "... = b ⊗ inv (a ⊗ b) ⊗ a" by simp
  also have "... = b ⊗ (inv (a ⊗ b) ⊗ a)" by (simp add: m_assoc)
  finally have ri: "b ⊗ (inv (a ⊗ b) ⊗ a) = 1 " by simp

  from c li ri show "b ∈ Units G" by (auto simp: Units_def)
qed

lemma (in monoid) prod_unit_r:
  assumes abunit[simp]: "a ⊗ b ∈ Units G"
    and bunit[simp]: "b ∈ Units G"
```

```
    and carr[simp]: "a ∈ carrier G"  "b ∈ carrier G"
  shows "a ∈ Units G"
proof -
  have c: "b ⊗ inv (a ⊗ b) ∈ carrier G" by simp

  have "a ⊗ (b ⊗ inv (a ⊗ b)) = (a ⊗ b) ⊗ inv (a ⊗ b)"
    by (simp add: m_assoc del: Units_r_inv)
  also have "... = 1" by simp
  finally have li: "a ⊗ (b ⊗ inv (a ⊗ b)) = 1" .

  have "1 = b ⊗ inv b" by (simp add: Units_r_inv[symmetric])
  also have "... = b ⊗ 1 ⊗ inv b" by simp
  also have "... = b ⊗ (inv (a ⊗ b) ⊗ (a ⊗ b)) ⊗ inv b"
    by (simp add: Units_l_inv[OF abunit, symmetric] del: Units_l_inv)
  also have "... = (b ⊗ inv (a ⊗ b) ⊗ a) ⊗ (b ⊗ inv b)"
    by (simp add: m_assoc del: Units_l_inv)
  also have "... = b ⊗ inv (a ⊗ b) ⊗ a" by simp
  finally have ri: "(b ⊗ inv (a ⊗ b)) ⊗ a = 1 " by simp

  from c li ri show "a ∈ Units G" by (auto simp: Units_def)
qed

lemma (in comm_monoid) unit_factor:
  assumes abunit: "a ⊗ b ∈ Units G"
    and [simp]: "a ∈ carrier G"  "b ∈ carrier G"
  shows "a ∈ Units G"
  using abunit[simplified Units_def]
proof clarsimp
  fix i
  assume [simp]: "i ∈ carrier G"

  have carr': "b ⊗ i ∈ carrier G" by simp

  have "(b ⊗ i) ⊗ a = (i ⊗ b) ⊗ a" by (simp add: m_comm)
  also have "... = i ⊗ (b ⊗ a)" by (simp add: m_assoc)
  also have "... = i ⊗ (a ⊗ b)" by (simp add: m_comm)
  also assume "i ⊗ (a ⊗ b) = 1"
  finally have li': "(b ⊗ i) ⊗ a = 1" .

  have "a ⊗ (b ⊗ i) = a ⊗ b ⊗ i" by (simp add: m_assoc)
  also assume "a ⊗ b ⊗ i = 1"
  finally have ri': "a ⊗ (b ⊗ i) = 1" .

  from carr' li' ri'
  show "a ∈ Units G" by (simp add: Units_def, fast)
qed
```

## 26.3 Divisibility and Association

### 26.3.1 Function definitions

```
definition factor :: "[_, 'a, 'a] ⇒ bool" (infix ‹dividesı› 65)
  where "a divides_G b ⟷ (∃c∈carrier G. b = a ⊗_G c)"

definition associated :: "[_, 'a, 'a] ⇒ bool" (infix ‹∼ı› 55)
  where "a ∼_G b ⟷ a divides_G b ∧ b divides_G a"

abbreviation "division_rel G ≡ (|carrier = carrier G, eq = (∼_G), le =
(divides_G)|)"

definition properfactor :: "[_, 'a, 'a] ⇒ bool"
  where "properfactor G a b ⟷ a divides_G b ∧ ¬(b divides_G a)"

definition irreducible :: "[_, 'a] ⇒ bool"
  where "irreducible G a ⟷ a ∉ Units G ∧ (∀b∈carrier G. properfactor
G b a ⟶ b ∈ Units G)"

definition prime :: "[_, 'a] ⇒ bool"
  where "prime G p ⟷
    p ∉ Units G ∧
    (∀a∈carrier G. ∀b∈carrier G. p divides_G (a ⊗_G b) ⟶ p divides_G
a ∨ p divides_G b)"
```

### 26.3.2 Divisibility

```
lemma dividesI:
  fixes G (structure)
  assumes carr: "c ∈ carrier G"
    and p: "b = a ⊗ c"
  shows "a divides b"
  unfolding factor_def using assms by fast

lemma dividesI' [intro]:
  fixes G (structure)
  assumes p: "b = a ⊗ c"
    and carr: "c ∈ carrier G"
  shows "a divides b"
  using assms by (fast intro: dividesI)

lemma dividesD:
  fixes G (structure)
  assumes "a divides b"
  shows "∃c∈carrier G. b = a ⊗ c"
  using assms unfolding factor_def by fast

lemma dividesE [elim]:
  fixes G (structure)
```

```
  assumes d: "a divides b"
    and elim: "⋀c. ⟦b = a ⊗ c; c ∈ carrier G⟧ ⟹ P"
  shows "P"
proof -
  from dividesD[OF d] obtain c where "c ∈ carrier G" and "b = a ⊗ c"
by auto
  then show P by (elim elim)
qed

lemma (in monoid) divides_refl[simp, intro!]:
  assumes carr: "a ∈ carrier G"
  shows "a divides a"
  by (intro dividesI[of "𝟭"]) (simp_all add: carr)

lemma (in monoid) divides_trans [trans]:
  assumes dvds: "a divides b" "b divides c"
    and acarr: "a ∈ carrier G"
  shows "a divides c"
  using dvds[THEN dividesD] by (blast intro: dividesI m_assoc acarr)

lemma (in monoid) divides_mult_lI [intro]:
  assumes  "a divides b" "a ∈ carrier G" "c ∈ carrier G"
  shows "(c ⊗ a) divides (c ⊗ b)"
  by (metis assms factor_def m_assoc)

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
  show "c ⊗ a divides c ⊗ b ⟹ a divides b"
    using carr monoid.m_assoc monoid_axioms monoid_cancel.l_cancel monoid_cancel_axioms
by fastforce
  show "a divides b ⟹ c ⊗ a divides c ⊗ b"
  using carr(1) carr(3) by blast
qed

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)"
  using carr ab by (metis divides_mult_lI m_comm)

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"
  using carr by (simp add: m_comm[of a c] m_comm[of b c])

lemma (in monoid) divides_prod_r:
  assumes ab: "a divides b"
```

```
    and carr: "a ∈ carrier G" "c ∈ carrier G"
  shows "a divides (b ⊗ c)"
  using ab carr by (fast intro: m_assoc)

lemma (in comm_monoid) divides_prod_l:
  assumes "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier G" "a divides
b"
  shows "a divides (c ⊗ b)"
  using assms  by (simp add: divides_prod_r m_comm)

lemma (in monoid) unit_divides:
  assumes uunit: "u ∈ Units G"
    and acarr: "a ∈ carrier G"
  shows "u divides a"
proof (intro dividesI[of "(inv u) ⊗ a"], fast intro: uunit acarr)
  from uunit acarr have xcarr: "inv u ⊗ a ∈ carrier G" by fast
  from uunit acarr have "u ⊗ (inv u ⊗ a) = (u ⊗ inv u) ⊗ a"
    by (fast intro: m_assoc[symmetric])
  also have "... = 1 ⊗ a" by (simp add: Units_r_inv[OF uunit])
  also from acarr have "... = a" by simp
  finally show "a = u ⊗ (inv u ⊗ a)" ..
qed

lemma (in comm_monoid) divides_unit:
  assumes udvd: "a divides u"
    and  carr: "a ∈ carrier G"  "u ∈ Units G"
  shows "a ∈ Units G"
  using udvd carr by (blast intro: unit_factor)

lemma (in comm_monoid) Unit_eq_dividesone:
  assumes ucarr: "u ∈ carrier G"
  shows "u ∈ Units G = u divides 1"
  using ucarr by (fast dest: divides_unit intro: unit_divides)
```

### 26.3.3 Association

```
lemma associatedI:
  fixes G (structure)
  assumes "a divides b" "b divides a"
  shows "a ∼ b"
  using assms by (simp add: associated_def)

lemma (in monoid) associatedI2:
  assumes uunit[simp]: "u ∈ Units G"
    and a: "a = b ⊗ u"
    and bcarr: "b ∈ carrier G"
  shows "a ∼ b"
  using uunit bcarr
  unfolding a
```

```
  apply (intro associatedI)
  apply (metis Units_closed divides_mult_lI one_closed r_one unit_divides)
  by blast

lemma (in monoid) associatedI2':
  assumes "a = b ⊗ u"
    and "u ∈ Units G"
    and "b ∈ carrier G"
  shows "a ∼ b"
  using assms by (intro associatedI2)

lemma associatedD:
  fixes G (structure)
  assumes "a ∼ b"
  shows "a divides b"
  using assms by (simp add: associated_def)

lemma (in monoid_cancel) associatedD2:
  assumes assoc: "a ∼ b"
    and carr: "a ∈ carrier G" "b ∈ carrier G"
  shows "∃u∈Units G. a = b ⊗ u"
  using assoc
  unfolding associated_def
proof clarify
  assume "b divides a"
  then obtain u where ucarr: "u ∈ carrier G" and a: "a = b ⊗ u"
    by (rule dividesE)

  assume "a divides b"
  then obtain u' where u'carr: "u' ∈ carrier G" and b: "b = a ⊗ u'"
    by (rule dividesE)
  note carr = carr ucarr u'carr

  from carr have "a ⊗ 1 = a" by simp
  also have "... = b ⊗ u" by (simp add: a)
  also have "... = a ⊗ u' ⊗ u" by (simp add: b)
  also from carr have "... = a ⊗ (u' ⊗ u)" by (simp add: m_assoc)
  finally have "a ⊗ 1 = a ⊗ (u' ⊗ u)" .
  with carr have u1: "1 = u' ⊗ u" by (fast dest: l_cancel)

  from carr have "b ⊗ 1 = b" by simp
  also have "... = a ⊗ u'" by (simp add: b)
  also have "... = b ⊗ u ⊗ u'" by (simp add: a)
  also from carr have "... = b ⊗ (u ⊗ u')" by (simp add: m_assoc)
  finally have "b ⊗ 1 = b ⊗ (u ⊗ u')" .
  with carr have u2: "1 = u ⊗ u'" by (fast dest: l_cancel)

  from u'carr u1[symmetric] u2[symmetric] have "∃u'∈carrier G. u' ⊗
u = 1 ∧ u ⊗ u' = 1"
```

```
    by fast
  then have "u ∈ Units G"
    by (simp add: Units_def ucarr)
  with ucarr a show "∃u∈Units G. a = b ⊗ u" by fast
qed

lemma associatedE:
  fixes G (structure)
  assumes assoc: "a ∼ b"
    and e: "⟦a divides b; b divides a⟧ ⟹ P"
  shows "P"
proof -
  from assoc have "a divides b" "b divides a"
    by (simp_all add: associated_def)
  then show P by (elim e)
qed

lemma (in monoid_cancel) associatedE2:
  assumes assoc: "a ∼ b"
    and e: "⋀u. ⟦a = b ⊗ u; u ∈ Units G⟧ ⟹ P"
    and carr: "a ∈ carrier G"  "b ∈ carrier G"
  shows "P"
proof -
  from assoc and carr have "∃u∈Units G. a = b ⊗ u"
    by (rule associatedD2)
  then obtain u where "u ∈ Units G"  "a = b ⊗ u"
    by auto
  then show P by (elim e)
qed

lemma (in monoid) associated_refl [simp, intro!]:
  assumes "a ∈ carrier G"
  shows "a ∼ a"
  using assms by (fast intro: associatedI)

lemma (in monoid) associated_sym [sym]:
  assumes "a ∼ b"
  shows "b ∼ a"
  using assms by (iprover intro: associatedI elim: associatedE)

lemma (in monoid) associated_trans [trans]:
  assumes "a ∼ b"  "b ∼ c"
    and "a ∈ carrier G" "c ∈ carrier G"
  shows "a ∼ c"
  using assms by (iprover intro: associatedI divides_trans elim: associatedE)

lemma (in monoid) division_equiv [intro, simp]: "equivalence (division_rel
G)"
  apply unfold_locales
```

```
    apply simp_all
   apply (metis associated_def)
  apply (iprover intro: associated_trans)
  done
```

### 26.3.4 Division and associativity

```
lemmas divides_antisym = associatedI

lemma (in monoid) divides_cong_l [trans]:
  assumes "x ∼ x'" "x' divides y" "x ∈ carrier G"
  shows "x divides y"
  by (meson assms associatedD divides_trans)

lemma (in monoid) divides_cong_r [trans]:
  assumes "x divides y" "y ∼ y'" "x ∈ carrier G"
  shows "x divides y'"
  by (meson assms associatedD divides_trans)

lemma (in monoid) division_weak_partial_order [simp, intro!]:
  "weak_partial_order (division_rel G)"
  apply unfold_locales
      apply (simp_all add: associated_sym divides_antisym)
     apply (metis associated_trans)
   apply (metis divides_trans)
  by (meson associated_def divides_trans)
```

### 26.3.5 Multiplication and associativity

```
lemma (in monoid) mult_cong_r:
  assumes "b ∼ b'" "a ∈ carrier G"  "b ∈ carrier G"  "b' ∈ carrier G"
  shows "a ⊗ b ∼ a ⊗ b'"
  by (meson assms associated_def divides_mult_lI)

lemma (in comm_monoid) mult_cong_l:
  assumes "a ∼ a'" "a ∈ carrier G"  "a' ∈ carrier G"  "b ∈ carrier G"
  shows "a ⊗ b ∼ a' ⊗ b"
  using assms m_comm mult_cong_r by auto

lemma (in monoid_cancel) assoc_l_cancel:
  assumes "a ∈ carrier G"  "b ∈ carrier G"  "b' ∈ carrier G" "a ⊗ b
∼ a ⊗ b'"
  shows "b ∼ b'"
  by (meson assms associated_def divides_mult_l)

lemma (in comm_monoid_cancel) assoc_r_cancel:
  assumes "a ⊗ b ∼ a' ⊗ b" "a ∈ carrier G"  "a' ∈ carrier G"  "b ∈
carrier G"
  shows "a ∼ a'"
  using assms assoc_l_cancel m_comm by presburger
```

### 26.3.6 Units

```
lemma (in monoid_cancel) assoc_unit_l [trans]:
  assumes "a ∼ b"
    and "b ∈ Units G"
    and "a ∈ carrier G"
  shows "a ∈ Units G"
  using assms by (fast elim: associatedE2)

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"
  using aunit bcarr associated_sym[OF asc] by (blast intro: assoc_unit_l)

lemma (in comm_monoid) Units_cong:
  assumes aunit: "a ∈ Units G" and asc: "a ∼ b"
    and bcarr: "b ∈ carrier G"
  shows "b ∈ Units G"
  using assms by (blast intro: divides_unit elim: associatedE)

lemma (in monoid) Units_assoc:
  assumes units: "a ∈ Units G"  "b ∈ Units G"
  shows "a ∼ b"
  using units by (fast intro: associatedI unit_divides)

lemma (in monoid) Units_are_ones: "Units G {.=}(division_rel G) {1}"
proof -
  have "a .∈division_rel G {1}" if "a ∈ Units G" for a
  proof -
    have "a ∼ 1"
      by (rule associatedI) (simp_all add: Units_closed that unit_divides)
    then show ?thesis
      by (simp add: elem_def)
  qed
  moreover have "1 .∈division_rel G Units G"
    by (simp add: equivalence.mem_imp_elem)
  ultimately show ?thesis
    by (auto simp: set_eq_def)
qed

lemma (in comm_monoid) Units_Lower: "Units G = Lower (division_rel G)
(carrier G)"
  apply (auto simp add: Units_def Lower_def)
   apply (metis Units_one_closed unit_divides unit_factor)
  apply (metis Unit_eq_dividesone Units_r_inv_ex m_ac(2) one_closed)
  done

lemma (in monoid_cancel) associated_iff:
```

```
  assumes "a ∈ carrier G" "b ∈ carrier G"
  shows "a ∼ b ⟷ (∃c ∈ Units G. a = b ⊗ c)"
  using assms associatedI2' associatedD2 by auto
```

### 26.3.7 Proper factors

```
lemma properfactorI:
  fixes G (structure)
  assumes "a divides b"
    and "¬(b divides a)"
  shows "properfactor G a b"
  using assms unfolding properfactor_def by simp

lemma properfactorI2:
  fixes G (structure)
  assumes advdb: "a divides b"
    and neq: "¬(a ∼ b)"
  shows "properfactor G a b"
proof (rule properfactorI, rule advdb, rule notI)
  assume "b divides a"
  with advdb have "a ∼ b" by (rule associatedI)
  with neq show "False" by fast
qed

lemma (in comm_monoid_cancel) properfactorI3:
  assumes p: "p = a ⊗ b"
    and nunit: "b ∉ Units G"
    and carr: "a ∈ carrier G"  "b ∈ carrier G"
  shows "properfactor G a p"
  unfolding p
  using carr
  apply (intro properfactorI, fast)
proof (clarsimp, elim dividesE)
  fix c
  assume ccarr: "c ∈ carrier G"
  note [simp] = carr ccarr

  have "a ⊗ 1 = a" by simp
  also assume "a = a ⊗ b ⊗ c"
  also have "... = a ⊗ (b ⊗ c)" by (simp add: m_assoc)
  finally have "a ⊗ 1 = a ⊗ (b ⊗ c)" .

  then have rinv: "1 = b ⊗ c" by (intro l_cancel[of "a" "1" "b ⊗ c"],
simp+)
  also have "... = c ⊗ b" by (simp add: m_comm)
  finally have linv: "1 = c ⊗ b" .

  from ccarr linv[symmetric] rinv[symmetric] have "b ∈ Units G"
    unfolding Units_def by fastforce
```

```
  with nunit show False ..
qed

lemma properfactorE:
  fixes G (structure)
  assumes pf: "properfactor G a b"
    and r: "⟦a divides b; ¬(b divides a)⟧ ⟹ P"
  shows "P"
  using pf unfolding properfactor_def by (fast intro: r)

lemma properfactorE2:
  fixes G (structure)
  assumes pf: "properfactor G a b"
    and elim: "⟦a divides b; ¬(a ∼ b)⟧ ⟹ P"
  shows "P"
  using pf unfolding properfactor_def by (fast elim: elim associatedE)

lemma (in monoid) properfactor_unitE:
  assumes uunit: "u ∈ Units G"
    and pf: "properfactor G a u"
    and acarr: "a ∈ carrier G"
  shows "P"
  using pf unit_divides[OF uunit acarr] by (fast elim: properfactorE)

lemma (in monoid) properfactor_divides:
  assumes pf: "properfactor G a b"
  shows "a divides b"
  using pf by (elim properfactorE)

lemma (in monoid) properfactor_trans1 [trans]:
  assumes "a divides b"  "properfactor G b c" "a ∈ carrier G"  "c ∈ carrier
G"
  shows "properfactor G a c"
  by (meson divides_trans properfactorE properfactorI assms)

lemma (in monoid) properfactor_trans2 [trans]:
  assumes "properfactor G a b"  "b divides c" "a ∈ carrier G"  "b ∈ carrier
G"
  shows "properfactor G a c"
  by (meson divides_trans properfactorE properfactorI assms)

lemma properfactor_lless:
  fixes G (structure)
  shows "properfactor G = lless (division_rel G)"
  by (force simp: lless_def properfactor_def associated_def)

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"
  using pf
  unfolding properfactor_lless
proof -
  interpret weak_partial_order "division_rel G" ..
  from x'x have "x' .=division_rel G x" by simp
  also assume "x ⊏division_rel G y"
  finally show "x' ⊏division_rel G y" by (simp add: carr)
qed

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'"
  using pf
  unfolding properfactor_lless
proof -
  interpret weak_partial_order "division_rel G" ..
  assume "x ⊏division_rel G y"
  also from yy'
  have "y .=division_rel G y'" by simp
  finally show "x ⊏division_rel G y'" by (simp add: carr)
qed

lemma (in monoid_cancel) properfactor_mult_lI [intro]:
  assumes ab: "properfactor G a b"
    and carr: "a ∈ carrier G" "c ∈ carrier G"
  shows "properfactor G (c ⊗ a) (c ⊗ b)"
  using ab carr by (fastforce elim: properfactorE intro: properfactorI)

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"
  using carr by (fastforce elim: properfactorE intro: properfactorI)

lemma (in comm_monoid_cancel) properfactor_mult_rI [intro]:
  assumes ab: "properfactor G a b"
    and carr: "a ∈ carrier G" "c ∈ carrier G"
  shows "properfactor G (a ⊗ c) (b ⊗ c)"
  using ab carr by (fastforce elim: properfactorE intro: properfactorI)

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"
  using carr by (fastforce elim: properfactorE intro: properfactorI)

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)"
  by (intro properfactor_trans2[OF ab] divides_prod_r) simp_all

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)"
  by (intro properfactor_trans2[OF ab] divides_prod_l) simp_all
```

## 26.4 Irreducible Elements and Primes

### 26.4.1 Irreducible elements

```
lemma irreducibleI:
  fixes G (structure)
  assumes "a ∉ Units G"
    and "⋀b. ⟦b ∈ carrier G; properfactor G b a⟧ ⟹ b ∈ Units G"
  shows "irreducible G a"
  using assms unfolding irreducible_def by blast

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"
  using assms unfolding irreducible_def by blast

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"
  using assms by (fast elim: irreducibleE)

lemma (in monoid_cancel) irreducible_cong [trans]:
  assumes "irreducible G a" "a ∼ a'" "a ∈ carrier G"  "a' ∈ carrier
G"
  shows "irreducible G a'"
proof -
  have "a' divides a"
    by (meson ‹a ∼ a'› associated_def)
  then show ?thesis
    by (metis (no_types) assms assoc_unit_l irreducibleE irreducibleI
monoid.properfactor_trans2 monoid_axioms)
qed
```

```
lemma (in monoid) irreducible_prod_rI:
  assumes "irreducible G a" "b ∈ Units G" "a ∈ carrier G"  "b ∈ carrier
G"
  shows "irreducible G (a ⊗ b)"
  using assms
  by (metis (no_types, lifting) associatedI2' irreducible_def monoid.m_closed
monoid_axioms prod_unit_r properfactor_cong_r)

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)"
  by (metis aunit birr carr irreducible_prod_rI m_comm)

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
  using irr
proof (elim irreducibleE)
  assume abnunit: "a ⊗ b ∉ Units G"
    and isunit[rule_format]: "∀ba. ba ∈ carrier G ∧ properfactor G ba
(a ⊗ b) ⟶ ba ∈ Units G"
  show P
  proof (cases "a ∈ Units G")
    case aunit: True
    have "irreducible G b"
    proof (rule irreducibleI, rule notI)
      assume "b ∈ Units G"
      with aunit have "(a ⊗ b) ∈ Units G" by fast
      with abnunit show "False" ..
    next
      fix c
      assume ccarr: "c ∈ carrier G"
        and "properfactor G c b"
      then have "properfactor G c (a ⊗ b)" by (simp add: properfactor_prod_l[of
c b a])
      with ccarr show "c ∈ Units G" by (fast intro: isunit)
    qed
    with aunit show "P" by (rule e2)
  next
    case anunit: False
    with carr have "properfactor G b (b ⊗ a)" by (fast intro: properfactorI3)
    then have bf: "properfactor G b (a ⊗ b)" by (subst m_comm[of a b],
simp+)
    then have bunit: "b ∈ Units G" by (intro isunit, simp)
```

```
    have "irreducible G a"
    proof (rule irreducibleI, rule notI)
      assume "a ∈ Units G"
      with bunit have "(a ⊗ b) ∈ Units G" by fast
      with abnunit show "False" ..
    next
      fix c
      assume ccarr: "c ∈ carrier G"
        and "properfactor G c a"
      then have "properfactor G c (a ⊗ b)"
        by (simp add: properfactor_prod_r[of c a b])
      with ccarr show "c ∈ Units G" by (fast intro: isunit)
    qed
    from this bunit show "P" by (rule e1)
  qed
qed

lemma divides_irreducible_condition:
  assumes "irreducible G r" and "a ∈ carrier G"
  shows "a divides_G r ⟹ a ∈ Units G ∨ a ∼_G r"
  using assms unfolding irreducible_def properfactor_def associated_def
  by (cases "r divides_G a", auto)
```

### 26.4.2 Prime elements

```
lemma primeI:
  fixes G (structure)
  assumes "p ∉ Units G"
    and "⋀a b. ⟦a ∈ carrier G; b ∈ carrier G; p divides (a ⊗ b)⟧ ⟹
p divides a ∨ p divides b"
  shows "prime G p"
  using assms unfolding prime_def by blast

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"
  using pprime unfolding prime_def by (blast dest: e)

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"
  using assms by (blast elim: primeE)
```

```
lemma (in monoid_cancel) prime_cong [trans]:
  assumes "prime G p"
    and pp': "p ∼ p'" "p ∈ carrier G"  "p' ∈ carrier G"
  shows "prime G p'"
  using assms
  by (auto simp: prime_def assoc_unit_l) (metis pp' associated_sym divides_cong_l)

lemma (in comm_monoid_cancel) prime_irreducible:
  assumes "prime G p"
  shows "irreducible G p"
proof (rule irreducibleI)
  show "p ∉ Units G"
    using assms unfolding prime_def by simp
next
  fix b assume A: "b ∈ carrier G" "properfactor G b p"
  then obtain c where c: "c ∈ carrier G" "p = b ⊗ c"
    unfolding properfactor_def factor_def by auto
  hence "p divides c"
    using A assms unfolding prime_def properfactor_def by auto
  then obtain b' where b': "b' ∈ carrier G" "c = p ⊗ b'"
    unfolding factor_def by auto
  hence "1 = b ⊗ b'"
    by (metis A(1) l_cancel m_closed m_lcomm one_closed r_one c)
  thus "b ∈ Units G"
    using A(1) Units_one_closed b'(1) unit_factor by presburger
qed

lemma (in comm_monoid_cancel) prime_pow_divides_iff:
  assumes "p ∈ carrier G" "a ∈ carrier G" "b ∈ carrier G" and "prime
G p" and "¬ (p divides a)"
  shows "(p [^] (n :: nat)) divides (a ⊗ b) ⟷ (p [^] n) divides b"
proof
  assume "(p [^] n) divides b" thus "(p [^] n) divides (a ⊗ b)"
    using divides_prod_l[of "p [^] n" b a] assms by simp
next
  assume "(p [^] n) divides (a ⊗ b)" thus "(p [^] n) divides b"
  proof (induction n)
    case 0 with ‹b ∈ carrier G› show ?case
      by (simp add: unit_divides)
  next
    case (Suc n)
    hence "(p [^] n) divides (a ⊗ b)" and "(p [^] n) divides b"
      using assms(1) divides_prod_r by auto
    with ‹(p [^] (Suc n)) divides (a ⊗ b)› obtain c d
      where c: "c ∈ carrier G" and "b = (p [^] n) ⊗ c"
        and d: "d ∈ carrier G" and "a ⊗ b = (p [^] (Suc n)) ⊗ d"
      using assms by blast
    hence "(p [^] n) ⊗ (a ⊗ c) = (p [^] n) ⊗ (p ⊗ d)"
      using assms by (simp add: m_assoc m_lcomm)
```

```
    hence "a ⊗ c = p ⊗ d"
      using c d assms(1) assms(2) l_cancel by blast
    with ‹¬ (p divides a)› and ‹prime G p› have "p divides c"
      by (metis assms(2) c d dividesI' prime_divides)
    with ‹b = (p [^] n) ⊗ c› show ?case
      using assms(1) c by simp
  qed
qed
```

## 26.5 Factorization and Factorial Monoids

### 26.5.1 Function definitions

```
definition factors :: "('a, _) monoid_scheme ⇒ 'a list ⇒ 'a ⇒ bool"
  where "factors G fs a ⟷ (∀x ∈ (set fs). irreducible G x) ∧ foldr
(⊗_G) fs 1_G = a"

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

abbreviation list_assoc :: "('a, _) monoid_scheme ⇒ 'a list ⇒ 'a list
⇒ bool" (infix ‹[∼]ı› 44)
  where "list_assoc G ≡ list_all2 (∼_G)"

definition essentially_equal :: "('a, _) monoid_scheme ⇒ 'a list ⇒ 'a
list ⇒ bool"
  where "essentially_equal G fs1 fs2 ⟷ (∃fs1'. fs1 <~~> fs1' ∧ fs1'
[∼]_G fs2)"


locale factorial_monoid = comm_monoid_cancel +
  assumes factors_exist: "⟦a ∈ carrier G; a ∉ Units G⟧ ⟹ ∃fs. set fs
⊆ carrier G ∧ factors G fs a"
    and factors_unique:
      "⟦factors G fs a; factors G fs' a; a ∈ carrier G; a ∉ Units G;
        set fs ⊆ carrier G; set fs' ⊆ carrier G⟧ ⟹ essentially_equal
G fs fs'"
```

### 26.5.2 Comparing lists of elements

Association on lists

```
lemma (in monoid) listassoc_refl [simp, intro]:
  assumes "set as ⊆ carrier G"
  shows "as [∼] as"
  using assms by (induct as) simp_all

lemma (in monoid) listassoc_sym [sym]:
  assumes "as [∼] bs"
```

```
    and "set as ⊆ carrier G"
    and "set bs ⊆ carrier G"
  shows "bs [∼] as"
  using assms
proof (induction as arbitrary: bs)
  case Cons
  then show ?case
    by (induction bs) (use associated_sym in auto)
qed auto

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"
  using assms
  apply (simp add: list_all2_conv_all_nth set_conv_nth, safe)
  by (metis (mono_tags, lifting) associated_trans nth_mem subsetCE)

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"
  using assms
  by (fastforce simp add: list_all2_conv_all_nth set_conv_nth intro: irreducible_cong)
```

Permutations

```
lemma perm_map [intro]:
  assumes p: "a <~~> b"
  shows "map f a <~~> map f b"
  using p by simp

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 -
  from m have ‹length a = length b›
    by (rule map_eq_imp_length_eq)
  from p have ‹mset c = mset b›
    by simp
  then obtain p where ‹p permutes {..<length b}› ‹permute_list p b =
c›
    by (rule mset_eq_permutation)
  with ‹length a = length b› have ‹p permutes {..<length a}›
    by simp
  moreover define d where ‹d = permute_list p a›
  ultimately have ‹mset a = mset d› ‹map f d = map f c›
    using m ‹p permutes {..<length b}› ‹permute_list p b = c›
```

```
    by (auto simp flip: permute_list_map)
  then show ?thesis
    by auto
qed

lemma (in monoid) perm_assoc_switch:
  assumes a:"as [∼] bs" and p: "bs <~~> cs"
  shows "∃bs'. as <~~> bs' ∧ bs' [∼] cs"
proof -
  from p have ‹mset cs = mset bs›
    by simp
  then obtain p where ‹p permutes {..<length bs}› ‹permute_list p bs
= cs›
    by (rule mset_eq_permutation)
  moreover define bs' where ‹bs' = permute_list p as›
  ultimately have ‹as <~~> bs'› and ‹bs' [∼] cs›
    using a by (auto simp add: list_all2_permute_list_iff list_all2_lengthD)
  then show ?thesis by blast
qed

lemma (in monoid) perm_assoc_switch_r:
  assumes p: "as <~~> bs" and a:"bs [∼] cs"
  shows "∃bs'. as [∼] bs' ∧ bs' <~~> cs"
  using a p by (rule list_all2_reorder_left_invariance)

declare perm_sym [sym]

lemma perm_setP:
  assumes perm: "as <~~> bs"
    and as: "P (set as)"
  shows "P (set bs)"
  using assms by (metis set_mset_mset)

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"
  using ex unfolding essentially_equal_def by fast

lemma (in monoid) essentially_equalE:
  assumes ee: "essentially_equal G fs1 fs2"
    and e: "⋀fs1'. ⟦fs1 <~~> fs1'; fs1' [∼] fs2⟧ ⟹ P"
  shows "P"
```

```
  using ee unfolding essentially_equal_def by (fast intro: e)

lemma (in monoid) ee_refl [simp,intro]:
  assumes carr: "set as ⊆ carrier G"
  shows "essentially_equal G as as"
  using carr by (fast intro: essentially_equalI)

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"
  using ee
proof (elim essentially_equalE)
  fix fs
  assume "as <~~> fs"  "fs [∼] bs"
  from perm_assoc_switch_r [OF this] obtain fs' where a: "as [∼] fs'"
and p: "fs' <~~> bs"
    by blast
  from p have "bs <~~> fs'" by (rule perm_sym)
  with a[symmetric] carr show ?thesis
    by (iprover intro: essentially_equalI perm_closed)
qed

lemma (in monoid) ee_trans [trans]:
  assumes ab: "essentially_equal G as bs" and bc: "essentially_equal
G bs cs"
    and ascarr: "set as ⊆ carrier G"
    and bscarr: "set bs ⊆ carrier G"
    and cscarr: "set cs ⊆ carrier G"
  shows "essentially_equal G as cs"
  using ab bc
proof (elim essentially_equalE)
  fix abs bcs
  assume "abs [∼] bs" and pb: "bs <~~> bcs"
  from perm_assoc_switch [OF this] obtain bs' where p: "abs <~~> bs'"
and a: "bs' [∼] bcs"
    by blast
  assume "as <~~> abs"
  with p have pp: "as <~~> bs'" by simp
  from pp ascarr have c1: "set bs' ⊆ carrier G" by (rule perm_closed)
  from pb bscarr have c2: "set bcs ⊆ carrier G" by (rule perm_closed)
  assume "bcs [∼] cs"
  then have "bs' [∼] cs"
    using a c1 c2 cscarr listassoc_trans by blast
  with pp show ?thesis
    by (rule essentially_equalI)
qed
```

### 26.5.3 Properties of lists of elements

Multiplication of factors in a list

```
lemma (in monoid) multlist_closed [simp, intro]:
  assumes ascarr: "set fs ⊆ carrier G"
  shows "foldr (⊗) fs 1 ∈ carrier G"
  using ascarr by (induct fs) simp_all

lemma  (in comm_monoid) multlist_dividesI:
  assumes "f ∈ set fs" and "set fs ⊆ carrier G"
  shows "f divides (foldr (⊗) fs 1)"
  using assms
proof (induction fs)
  case (Cons a fs)
  then have f: "f ∈ carrier G"
    by blast
  show ?case
    using Cons.IH Cons.prems(1) Cons.prems(2) divides_prod_l f by auto
qed auto

lemma (in comm_monoid_cancel) multlist_listassoc_cong:
  assumes "fs [∼] fs'"
    and "set fs ⊆ carrier G" and "set fs' ⊆ carrier G"
  shows "foldr (⊗) fs 1 ∼ foldr (⊗) fs' 1"
  using assms
proof (induct fs arbitrary: fs')
  case (Cons a as fs')
  then show ?case
  proof (induction fs')
    case (Cons b bs)
    then have p: "a ⊗ foldr (⊗) as 1 ∼ b ⊗ foldr (⊗) as 1"
      by (simp add: mult_cong_l)
    then have "foldr (⊗) as 1 ∼ foldr (⊗) bs 1"
      using Cons by auto
    with Cons have "b ⊗ foldr (⊗) as 1 ∼ b ⊗ foldr (⊗) bs 1"
      by (simp add: mult_cong_r)
    then show ?case
      using Cons.prems(3) Cons.prems(4) monoid.associated_trans monoid_axioms
p by force
  qed auto
qed auto

lemma (in comm_monoid) multlist_perm_cong:
  assumes prm: "as <~~> bs"
    and ascarr: "set as ⊆ carrier G"
  shows "foldr (⊗) as 1 = foldr (⊗) bs 1"
proof -
  from prm have ‹mset (rev as) = mset (rev bs)›
    by simp
```

```
  moreover note one_closed
  ultimately have ‹fold (⊗) (rev as) 1 = fold (⊗) (rev bs) 1›
    by (rule fold_permuted_eq) (use ascarr in ‹auto intro: m_lcomm›)
  then show ?thesis
    by (simp add: foldr_conv_fold)
qed

lemma (in comm_monoid_cancel) multlist_ee_cong:
  assumes "essentially_equal G fs fs'"
    and "set fs ⊆ carrier G" and "set fs' ⊆ carrier G"
  shows "foldr (⊗) fs 1 ∼ foldr (⊗) fs' 1"
  using assms
  by (metis essentially_equal_def multlist_listassoc_cong multlist_perm_cong
perm_closed)
```

### 26.5.4 Factorization in irreducible elements

```
lemma wfactorsI:
  fixes G (structure)
  assumes "∀f∈set fs. irreducible G f"
    and "foldr (⊗) fs 1 ∼ a"
  shows "wfactors G fs a"
  using assms unfolding wfactors_def by simp

lemma wfactorsE:
  fixes G (structure)
  assumes wf: "wfactors G fs a"
    and e: "⟦∀f∈set fs. irreducible G f; foldr (⊗) fs 1 ∼ a⟧ ⟹ P"
  shows "P"
  using wf unfolding wfactors_def by (fast dest: e)

lemma (in monoid) factorsI:
  assumes "∀f∈set fs. irreducible G f"
    and "foldr (⊗) fs 1 = a"
  shows "factors G fs a"
  using assms unfolding factors_def by simp

lemma factorsE:
  fixes G (structure)
  assumes f: "factors G fs a"
    and e: "⟦∀f∈set fs. irreducible G f; foldr (⊗) fs 1 = a⟧ ⟹ P"
  shows "P"
  using f unfolding factors_def by (simp add: e)

lemma (in monoid) factors_wfactors:
  assumes "factors G as a" and "set as ⊆ carrier G"
  shows "wfactors G as a"
  using assms by (blast elim: factorsE intro: wfactorsI)
```

```
lemma (in monoid) wfactors_factors:
  assumes "wfactors G as a" and "set as ⊆ carrier G"
  shows "∃a'. factors G as a' ∧ a' ∼ a"
  using assms by (blast elim: wfactorsE intro: factorsI)

lemma (in monoid) factors_closed [dest]:
  assumes "factors G fs a" and "set fs ⊆ carrier G"
  shows "a ∈ carrier G"
  using assms by (elim factorsE, clarsimp)

lemma (in monoid) nunit_factors:
  assumes anunit: "a ∉ Units G"
    and fs: "factors G as a"
  shows "length as > 0"
proof -
  from anunit Units_one_closed have "a ≠ 1" by auto
  with fs show ?thesis by (auto elim: factorsE)
qed

lemma (in monoid) unit_wfactors [simp]:
  assumes aunit: "a ∈ Units G"
  shows "wfactors G [] a"
  using aunit by (intro wfactorsI) (simp, simp add: Units_assoc)

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 (cases fs)
  case fs: (Cons f fs')
  from carr have fcarr[simp]: "f ∈ carrier G" and carr'[simp]: "set
fs' ⊆ carrier G"
    by (simp_all add: fs)

  from fs wf have "irreducible G f" by (simp add: wfactors_def)
  then have fnunit: "f ∉ Units G" by (fast elim: irreducibleE)

  from fs wf have a: "f ⊗ foldr (⊗) fs' 1 ∼ a" by (simp add: wfactors_def)

  note aunit
  also from fs wf
  have a: "f ⊗ foldr (⊗) fs' 1 ∼ a" by (simp add: wfactors_def)
  have "a ∼ f ⊗ foldr (⊗) fs' 1"
    by (simp add: Units_closed[OF aunit] a[symmetric])
  finally have "f ⊗ foldr (⊗) fs' 1 ∈ Units G" by simp
  then have "f ∈ Units G" by (intro unit_factor[of f], simp+)
  with fnunit show ?thesis by contradiction
qed
```

Comparing wfactors

```
lemma (in comm_monoid_cancel) wfactors_listassoc_cong_l:
  assumes fact: "wfactors G fs a"
    and asc: "fs [∼] fs'"
    and carr: "a ∈ carrier G"  "set fs ⊆ carrier G"  "set fs' ⊆ carrier
G"
  shows "wfactors G fs' a"
proof -
  { from asc[symmetric] have "foldr (⊗) fs' 1 ∼ foldr (⊗) fs 1"
      by (simp add: multlist_listassoc_cong carr)
    also assume "foldr (⊗) fs 1 ∼ a"
    finally have "foldr (⊗) fs' 1 ∼ a" by (simp add: carr) }
  then show ?thesis
  using fact
  by (meson asc carr(2) carr(3) irrlist_listassoc_cong wfactors_def)
qed

lemma (in comm_monoid) wfactors_perm_cong_l:
  assumes "wfactors G fs a"
    and "fs <~~> fs'"
    and "set fs ⊆ carrier G"
  shows "wfactors G fs' a"
  using assms irrlist_perm_cong multlist_perm_cong wfactors_def by fastforce

lemma (in comm_monoid_cancel) wfactors_ee_cong_l [trans]:
  assumes ee: "essentially_equal G as bs"
    and bfs: "wfactors G bs b"
    and carr: "b ∈ carrier G"  "set as ⊆ carrier G"  "set bs ⊆ carrier
G"
  shows "wfactors G as b"
  using ee
proof (elim essentially_equalE)
  fix fs
  assume prm: "as <~~> fs"
  with carr have fscarr: "set fs ⊆ carrier G"
    using perm_closed by blast

  note bfs
  also assume [symmetric]: "fs [∼] bs"
  also (wfactors_listassoc_cong_l)
  have ‹mset fs = mset as› using prm by simp
  finally (wfactors_perm_cong_l)
  show "wfactors G as b" by (simp add: carr fscarr)
qed

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'"
  using fac
proof (elim wfactorsE, intro wfactorsI)
  assume "foldr (⊗) fs 1 ∼ a" also note aa'
  finally show "foldr (⊗) fs 1 ∼ a'" by simp
qed
```

### 26.5.5 Essentially equal factorizations

```
lemma (in comm_monoid_cancel) unitfactor_ee:
  assumes uunit: "u ∈ Units G"
    and carr: "set as ⊆ carrier G"
  shows "essentially_equal G (as[0 := (as!0 ⊗ u)]) as"
    (is "essentially_equal G ?as' as")
proof -
  have "as[0 := as ! 0 ⊗ u] [∼] as"
  proof (cases as)
    case (Cons a as')
    then show ?thesis
      using associatedI2 carr uunit by auto
  qed auto
  then show ?thesis
    using essentially_equal_def by blast
qed

lemma (in comm_monoid_cancel) factors_cong_unit:
  assumes u: "u ∈ Units G"
    and a: "a ∉ Units G"
    and afs: "factors G as a"
    and ascarr: "set as ⊆ carrier G"
  shows "factors G (as[0 := (as!0 ⊗ u)]) (a ⊗ u)"
    (is "factors G ?as' ?a'")
proof (cases as)
  case Nil
  then show ?thesis
    using afs a nunit_factors by auto
next
  case (Cons b bs)
  have *: "∀f∈set as. irreducible G f" "foldr (⊗) as 1 = a"
    using afs  by (auto simp: factors_def)
  show ?thesis
  proof (intro factorsI)
    show "foldr (⊗) (as[0 := as ! 0 ⊗ u]) 1 = a ⊗ u"
      using Cons u ascarr * by (auto simp add: m_ac Units_closed)
    show "∀f∈set (as[0 := as ! 0 ⊗ u]). irreducible G f"
      using Cons u ascarr * by (force intro: irreducible_prod_rI)
  qed
qed
```

```
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"
  using afs bfs
proof (elim wfactorsE)
  from prm have [simp]: "set bs ⊆ carrier G" by (simp add: perm_closed)
  assume "foldr (⊗) as 1 ∼ a"
  then have "a ∼ foldr (⊗) as 1"
    by (simp add: associated_sym)
  also from prm
  have "foldr (⊗) as 1 = foldr (⊗) bs 1" by (rule multlist_perm_cong,
simp)
  also assume "foldr (⊗) bs 1 ∼ b"
  finally show "a ∼ b" by simp
qed

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"
  using afs bfs
proof (elim wfactorsE)
  assume "foldr (⊗) as 1 ∼ a"
  then have "a ∼ foldr (⊗) as 1" by (simp add: associated_sym)
  also from assoc
  have "foldr (⊗) as 1 ∼ foldr (⊗) bs 1" by (rule multlist_listassoc_cong,
simp+)
  also assume "foldr (⊗) bs 1 ∼ b"
  finally show "a ∼ b" by simp
qed

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"
  using ee
proof (elim essentially_equalE)
  fix fs
  assume prm: "as <~~> fs"
```

```
  then have as'carr[simp]: "set fs ⊆ carrier G"
    by (simp add: perm_closed)
  from afs prm have afs': "wfactors G fs a"
    by (rule wfactors_perm_cong_l) simp
  assume "fs [∼] bs"
  from this afs' bfs show "a ∼ b"
    by (rule listassoc_wfactorsD) simp_all
qed

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"
  using assms by (blast intro: factors_wfactors dest: ee_wfactorsD)

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 -
  note carr[simp] = factors_closed[OF afs ascarr] ascarr[THEN subsetD]
    factors_closed[OF bfs bscarr] bscarr[THEN subsetD]

  from ab carr obtain u where uunit: "u ∈ Units G" and a: "a = b ⊗ u"
    by (elim associatedE2)

  from uunit bscarr have ee: "essentially_equal G (bs[0 := (bs!0 ⊗ u)])
bs"
    (is "essentially_equal G ?bs' bs")
    by (rule unitfactor_ee)

  from bscarr uunit have bs'carr: "set ?bs' ⊆ carrier G"
    by (cases bs) (simp_all add: Units_closed)

  from uunit bnunit bfs bscarr have fac: "factors G ?bs' (b ⊗ u)"
    by (rule factors_cong_unit)

  from afs fac[simplified a[symmetric]] ascarr bs'carr anunit
  have "essentially_equal G as ?bs'"
    by (blast intro: factors_unique)
  also note ee
  finally show "essentially_equal G as bs"
    by (simp add: ascarr bscarr bs'carr)
qed

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"
  using assms
proof (cases "a ∈ Units G")
  case aunit: True
  also note asc
  finally have bunit: "b ∈ Units G" by simp

  from aunit asf ascarr have e: "as = []"
    by (rule unit_wfactors_empty)
  from bunit bsf bscarr have e': "bs = []"
    by (rule unit_wfactors_empty)

  have "essentially_equal G [] []"
    by (fast intro: essentially_equalI)
  then show ?thesis
    by (simp add: e e')
next
  case anunit: False
  have bnunit: "b ∉ Units G"
  proof clarify
    assume "b ∈ Units G"
    also note asc[symmetric]
    finally have "a ∈ Units G" by simp
    with anunit show False ..
  qed

  from wfactors_factors[OF asf ascarr] obtain a' where fa': "factors
G as a'" and a': "a' ∼ a"
    by blast
  from fa' ascarr have a'carr[simp]: "a' ∈ carrier G"
    by fast

  have a'nunit: "a' ∉ Units G"
  proof clarify
    assume "a' ∈ Units G"
    also note a'
    finally have "a ∈ Units G" by simp
    with anunit
    show "False" ..
  qed

  from wfactors_factors[OF bsf bscarr] obtain b' where fb': "factors
G bs b'" and b': "b' ∼ b"
    by blast
```

```
  from fb' bscarr have b'carr[simp]: "b' ∈ carrier G"
    by fast

  have b'nunit: "b' ∉ Units G"
  proof clarify
    assume "b' ∈ Units G"
    also note b'
    finally have "b ∈ Units G" by simp
    with bnunit show False ..
  qed

  note a'
  also note asc
  also note b'[symmetric]
  finally have "a' ∼ b'" by simp
  from this fa' a'nunit fb' b'nunit ascarr bscarr show "essentially_equal
G as bs"
    by (rule ee_factorsI)
qed

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"
  using assms by (fast intro: ee_wfactorsI ee_wfactorsD)

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 (cases "a ∈ Units G")
  case True
  then have "wfactors G [] a" by (rule unit_wfactors)
  then show ?thesis by (intro exI) force
next
  case False
  with factors_exist [OF acarr] obtain fs where fscarr: "set fs ⊆ carrier
G" and f: "factors G fs a"
    by blast
  from f have "wfactors G fs a" by (rule factors_wfactors) fact
  with fscarr show ?thesis by fast
qed

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"
  unfolding wfactors_def using assms by blast
```

```
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'"
  using assms by (fast intro: ee_wfactorsI[of a a])

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)"
  using assms unfolding factors_def by simp

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)"
  using assms unfolding wfactors_def by (simp add: mult_cong_r)

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 -
  have "foldr (⊗) (fa @ fb) 1 = foldr (⊗) fa 1 ⊗ foldr (⊗) fb 1" if
"set fa ⊆ carrier G"
    "Ball (set fa) (irreducible G)"
    using that bscarr by (induct fa) (simp_all add: m_assoc)
  then show ?thesis
    using assms unfolding factors_def by force
qed

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)"
  using wfactors_factors[OF asf ascarr] and wfactors_factors[OF bsf bscarr]
proof clarsimp
  fix a' b'
  assume asf': "factors G as a'" and a'a: "a' ∼ a"
    and bsf': "factors G bs b'" and b'b: "b' ∼ b"
  from asf' have a'carr: "a' ∈ carrier G" by (rule factors_closed) fact
  from bsf' have b'carr: "b' ∈ carrier G" by (rule factors_closed) fact

  note carr = acarr bcarr a'carr b'carr ascarr bscarr
```

```
  from asf' bsf' have "factors G (as @ bs) (a' ⊗ b')"
    by (rule factors_mult) fact+

  with carr have abf': "wfactors G (as @ bs) (a' ⊗ b')"
    by (intro factors_wfactors) simp_all
  also from b'b carr have trb: "a' ⊗ b' ∼ a' ⊗ b"
    by (intro mult_cong_r)
  also from a'a carr have tra: "a' ⊗ b ∼ a ⊗ b"
    by (intro mult_cong_l)
  finally show "wfactors G (as @ bs) (a ⊗ b)"
    by (simp add: carr)
qed

lemma (in comm_monoid) factors_dividesI:
  assumes "factors G fs a"
    and "f ∈ set fs"
    and "set fs ⊆ carrier G"
  shows "f divides a"
  using assms by (fast elim: factorsE intro: multlist_dividesI)

lemma (in comm_monoid) wfactors_dividesI:
  assumes p: "wfactors G fs a"
    and fscarr: "set fs ⊆ carrier G" and acarr: "a ∈ carrier G"
    and f: "f ∈ set fs"
  shows "f divides a"
  using wfactors_factors[OF p fscarr]
proof clarsimp
  fix a'
  assume fsa': "factors G fs a'" and a'a: "a' ∼ a"
  with fscarr have a'carr: "a' ∈ carrier G"
    by (simp add: factors_closed)

  from fsa' fscarr f have "f divides a'"
    by (fast intro: factors_dividesI)
  also note a'a
  finally show "f divides a"
    by (simp add: f fscarr[THEN subsetD] acarr a'carr)
qed
```

#### 26.5.6 Factorial monoids and wfactors

```
lemma (in comm_monoid_cancel) factorial_monoidI:
  assumes wfactors_exists: "⋀a. ⟦ a ∈ carrier G; a ∉ Units G ⟧ ⟹ ∃fs.
set fs ⊆ carrier G ∧ wfactors G fs a"
    and wfactors_unique:
      "⋀a fs fs'. ⟦a ∈ carrier G; set fs ⊆ carrier G; set fs' ⊆ carrier
G;
        wfactors G fs a; wfactors G fs' a⟧ ⟹ essentially_equal G fs
fs'"
```

```
  shows "factorial_monoid G"
proof
  fix a
  assume acarr: "a ∈ carrier G" and anunit: "a ∉ Units G"
  from wfactors_exists[OF acarr anunit]
  obtain as where ascarr: "set as ⊆ carrier G" and afs: "wfactors G
as a"
    by blast
  from wfactors_factors [OF afs ascarr] obtain a' where afs': "factors
G as a'" and a'a: "a' ∼ a"
    by blast
  from afs' ascarr have a'carr: "a' ∈ carrier G"
    by fast
  have a'nunit: "a' ∉ Units G"
  proof clarify
    assume "a' ∈ Units G"
    also note a'a
    finally have "a ∈ Units G" by (simp add: acarr)
    with anunit show False ..
  qed

  from a'carr acarr a'a obtain u where uunit: "u ∈ Units G" and a':
"a' = a ⊗ u"
    by (blast elim: associatedE2)

  note [simp] = acarr Units_closed[OF uunit] Units_inv_closed[OF uunit]
  have "a = a ⊗ 1" by simp
  also have "... = a ⊗ (u ⊗ inv u)" by (simp add: uunit)
  also have "... = a' ⊗ inv u" by (simp add: m_assoc[symmetric] a'[symmetric])
  finally have a: "a = a' ⊗ inv u" .

  from ascarr uunit have cr: "set (as[0:=(as!0 ⊗ inv u)]) ⊆ carrier
G"
    by (cases as) auto
  from afs' uunit a'nunit acarr ascarr have "factors G (as[0:=(as!0 ⊗
inv u)]) a"
    by (simp add: a factors_cong_unit)
  with cr show "∃fs. set fs ⊆ carrier G ∧ factors G fs a"
    by fast
qed (blast intro: factors_wfactors wfactors_unique)
```

## 26.6 Factorizations as Multisets

Gives useful operations like intersection

```
abbreviation "assocs G x ≡ eq_closure_of (division_rel G) {x}"

definition "fmset G as = mset (map (assocs G) as)"
```

Helper lemmas

```
lemma (in monoid) assocs_repr_independence:
  assumes "y ∈ assocs G x" "x ∈ carrier G"
  shows "assocs G x = assocs G y"
  using assms
  by (simp add: eq_closure_of_def elem_def) (use associated_sym associated_trans
in ‹blast+›)

lemma (in monoid) assocs_self:
  assumes "x ∈ carrier G"
  shows "x ∈ assocs G x"
  using assms by (fastforce intro: closure_ofI2)

lemma (in monoid) assocs_repr_independenceD:
  assumes repr: "assocs G x = assocs G y" and ycarr: "y ∈ carrier G"
  shows "y ∈ assocs G x"
  unfolding repr using ycarr by (intro assocs_self)

lemma (in comm_monoid) assocs_assoc:
  assumes "a ∈ assocs G b" "b ∈ carrier G"
  shows "a ∼ b"
  using assms by (elim closure_ofE2) simp

lemmas (in comm_monoid) assocs_eqD = assocs_repr_independenceD[THEN assocs_assoc]
```

### 26.6.1 Comparing multisets

```
lemma (in monoid) fmset_perm_cong:
  assumes prm: "as <~~> bs"
  shows "fmset G as = fmset G bs"
  using perm_map[OF prm] unfolding fmset_def by blast

lemma (in comm_monoid_cancel) eqc_listassoc_cong:
  assumes "as [∼] bs" and "set as ⊆ carrier G" and "set bs ⊆ carrier
G"
  shows "map (assocs G) as = map (assocs G) bs"
  using assms
proof (induction as arbitrary: bs)
  case Nil
  then show ?case by simp
next
  case (Cons a as)
  then show ?case
  proof (clarsimp simp add: Cons_eq_map_conv list_all2_Cons1)
    fix z zs
    assume zzs: "a ∈ carrier G" "set as ⊆ carrier G" "bs = z # zs" "a
∼ z"
      "as [∼] zs" "z ∈ carrier G" "set zs ⊆ carrier G"
    then show "assocs G a = assocs G z"
      apply (simp add: eq_closure_of_def elem_def)
```

```
      using ‹a ∈ carrier G› ‹z ∈ carrier G› ‹a ∼ z› associated_sym
associated_trans by blast+
  qed
qed

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"
  using assms unfolding fmset_def by (simp add: eqc_listassoc_cong)

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"
  using ee
  thm essentially_equal_def
proof (elim essentially_equalE)
  fix as'
  assume prm: "as <~~> as'"
  from prm ascarr have as'carr: "set as' ⊆ carrier G"
    by (rule perm_closed)
  from prm have "fmset G as = fmset G as'"
    by (rule fmset_perm_cong)
  also assume "as' [∼] bs"
  with as'carr bscarr have "fmset G as' = fmset G bs"
    by (simp add: fmset_listassoc_cong)
  finally show "fmset G as = fmset G bs" .
qed

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 -
  from mset have "mset (map (assocs G) bs) = mset (map (assocs G) as)"
    by (simp add: fmset_def)
  then obtain p where ‹p permutes {..<length (map (assocs G) as)}›
    ‹permute_list p (map (assocs G) as) = map (assocs G) bs›
    by (rule mset_eq_permutation)
  then have ‹p permutes {..<length as}›
    ‹map (assocs G) (permute_list p as) = map (assocs G) bs›
    by (simp_all add: permute_list_map)
  moreover define as' where ‹as' = permute_list p as›
  ultimately have tp: "as <~~> as'" and tm: "map (assocs G) as' = map
(assocs G) bs"
    by simp_all
  from tp show ?thesis
  proof (rule essentially_equalI)
```

```
    from tm tp ascarr have as'carr: "set as' ⊆ carrier G"
      using perm_closed by blast
    from tm as'carr[THEN subsetD] bscarr[THEN subsetD] show "as' [∼]
bs"
      by (induct as' arbitrary: bs) (simp, fastforce dest: assocs_eqD[THEN
associated_sym])
  qed
qed

lemma (in comm_monoid_cancel) ee_is_fmset:
  assumes "set as ⊆ carrier G" and "set bs ⊆ carrier G"
  shows "essentially_equal G as bs = (fmset G as = fmset G bs)"
  using assms by (fast intro: ee_fmset fmset_ee)
```

### 26.6.2 Interpreting multisets as factorizations

```
lemma (in monoid) mset_fmsetEx:
  assumes elems: "⋀X. X ∈ set_mset Cs ⟹ ∃x. P x ∧ X = assocs G x"
  shows "∃cs. (∀c ∈ set cs. P c) ∧ fmset G cs = Cs"
proof -
  from surjE[OF surj_mset] obtain Cs' where Cs: "Cs = mset Cs'"
    by blast
  have "∃cs. (∀c ∈ set cs. P c) ∧ mset (map (assocs G) cs) = Cs"
    using elems unfolding Cs
  proof (induction Cs')
    case (Cons a Cs')
    then obtain c cs where csP: "∀x∈set cs. P x" and mset: "mset (map
(assocs G) cs) = mset Cs'"
            and cP: "P c" and a: "a = assocs G c"
      by force
    then have tP: "∀x∈set (c#cs). P x"
      by simp
    show ?case
      using tP mset a by fastforce
  qed auto
  then show ?thesis by (simp add: fmset_def)
qed

lemma (in monoid) mset_wfactorsEx:
  assumes elems: "⋀X. X ∈ set_mset Cs ⟹ ∃x. (x ∈ carrier G ∧ irreducible
G x) ∧ X = assocs G x"
  shows "∃c cs. c ∈ carrier G ∧ set cs ⊆ carrier G ∧ wfactors G cs c
∧ fmset G cs = Cs"
proof -
  have "∃cs. (∀c∈set cs. c ∈ carrier G ∧ irreducible G c) ∧ fmset G
cs = Cs"
    by (intro mset_fmsetEx, rule elems)
  then obtain cs where p[rule_format]: "∀c∈set cs. c ∈ carrier G ∧ irreducible
G c"
```

```
    and Cs[symmetric]: "fmset G cs = Cs" by auto
  from p have cscarr: "set cs ⊆ carrier G" by fast
  from p have "∃c. c ∈ carrier G ∧ wfactors G cs c"
    by (intro wfactors_prod_exists) auto
  then obtain c where ccarr: "c ∈ carrier G" and cfs: "wfactors G cs
c" by auto
  with cscarr Cs show ?thesis by fast
qed
```

### 26.6.3 Multiplication on multisets

```
lemma (in factorial_monoid) mult_wfactors_fmset:
  assumes afs: "wfactors G as a"
    and bfs: "wfactors G bs b"
    and cfs: "wfactors G cs (a ⊗ b)"
    and carr: "a ∈ carrier G"  "b ∈ carrier G"
              "set as ⊆ carrier G"  "set bs ⊆ carrier G"  "set cs ⊆ carrier
G"
  shows "fmset G cs = fmset G as + fmset G bs"
proof -
  from assms have "wfactors G (as @ bs) (a ⊗ b)"
    by (intro wfactors_mult)
  with carr cfs have "essentially_equal G cs (as@bs)"
    by (intro ee_wfactorsI[of "a⊗b" "a⊗b"]) simp_all
  with carr have "fmset G cs = fmset G (as@bs)"
    by (intro ee_fmset) simp_all
  also have "fmset G (as@bs) = fmset G as + fmset G bs"
    by (simp add: fmset_def)
  finally show "fmset G cs = fmset G as + fmset G bs" .
qed

lemma (in factorial_monoid) mult_factors_fmset:
  assumes afs: "factors G as a"
    and bfs: "factors G bs b"
    and cfs: "factors G cs (a ⊗ b)"
    and "set as ⊆ carrier G"  "set bs ⊆ carrier G"  "set cs ⊆ carrier
G"
  shows "fmset G cs = fmset G as + fmset G bs"
  using assms by (blast intro: factors_wfactors mult_wfactors_fmset)

lemma (in comm_monoid_cancel) fmset_wfactors_mult:
  assumes mset: "fmset G cs = fmset G as + fmset G bs"
    and carr: "a ∈ carrier G"  "b ∈ carrier G"  "c ∈ carrier G"
      "set as ⊆ carrier G"  "set bs ⊆ carrier G"  "set cs ⊆ carrier G"
    and fs: "wfactors G as a"  "wfactors G bs b"  "wfactors G cs c"
  shows "c ∼ a ⊗ b"
proof -
  from carr fs have m: "wfactors G (as @ bs) (a ⊗ b)"
    by (intro wfactors_mult)
```

```
  from mset have "fmset G cs = fmset G (as@bs)"
    by (simp add: fmset_def)
  then have "essentially_equal G cs (as@bs)"
    by (rule fmset_ee) (simp_all add: carr)
  then show "c ∼ a ⊗ b"
    by (rule ee_wfactorsD[of "cs" "as@bs"]) (simp_all add: assms m)
qed
```

### 26.6.4 Divisibility on multisets

```
lemma (in factorial_monoid) divides_fmsubset:
  assumes ab: "a divides b"
    and afs: "wfactors G as a"
    and bfs: "wfactors G bs b"
    and carr: "a ∈ carrier G"  "b ∈ carrier G"  "set as ⊆ carrier G"
"set bs ⊆ carrier G"
  shows "fmset G as ⊆# fmset G bs"
  using ab
proof (elim dividesE)
  fix c
  assume ccarr: "c ∈ carrier G"
  from wfactors_exist [OF this]
  obtain cs where cscarr: "set cs ⊆ carrier G" and cfs: "wfactors G
cs c"
    by blast
  note carr = carr ccarr cscarr

  assume "b = a ⊗ c"
  with afs bfs cfs carr have "fmset G bs = fmset G as + fmset G cs"
    by (intro mult_wfactors_fmset[OF afs cfs]) simp_all
  then show ?thesis by simp
qed

lemma (in comm_monoid_cancel) fmsubset_divides:
  assumes msubset: "fmset G as ⊆# fmset G bs"
    and afs: "wfactors G as a"
    and bfs: "wfactors G bs b"
    and acarr: "a ∈ carrier G"
    and bcarr: "b ∈ carrier G"
    and ascarr: "set as ⊆ carrier G"
    and bscarr: "set bs ⊆ carrier G"
  shows "a divides b"
proof -
  from afs have airr: "∀a ∈ set as. irreducible G a" by (fast elim:
wfactorsE)
  from bfs have birr: "∀b ∈ set bs. irreducible G b" by (fast elim:
wfactorsE)
```

```
  have "∃c cs. c ∈ carrier G ∧ set cs ⊆ carrier G ∧ wfactors G cs c
∧ fmset G cs = fmset G bs - fmset G as"
  proof (intro mset_wfactorsEx, simp)
    fix X
    assume "X ∈# fmset G bs - fmset G as"
    then have "X ∈# fmset G bs" by (rule in_diffD)
    then have "X ∈ set (map (assocs G) bs)" by (simp add: fmset_def)
    then have "∃x. x ∈ set bs ∧ X = assocs G x" by (induct bs) auto
    then obtain x where xbs: "x ∈ set bs" and X: "X = assocs G x" by
auto
    with bscarr have xcarr: "x ∈ carrier G" by fast
    from xbs birr have xirr: "irreducible G x" by simp

    from xcarr and xirr and X show "∃x. x ∈ carrier G ∧ irreducible
G x ∧ X = assocs G x"
      by fast
  qed
  then obtain c cs
    where ccarr: "c ∈ carrier G"
      and cscarr: "set cs ⊆ carrier G"
      and csf: "wfactors G cs c"
      and csmset: "fmset G cs = fmset G bs - fmset G as" by auto

  from csmset msubset
  have "fmset G bs = fmset G as + fmset G cs"
    by (simp add: multiset_eq_iff subseteq_mset_def)
  then have basc: "b ∼ a ⊗ c"
    by (rule fmset_wfactors_mult) fact+
  then show ?thesis
  proof (elim associatedE2)
    fix u
    assume "u ∈ Units G"  "b = a ⊗ c ⊗ u"
    with acarr ccarr show "a divides b"
      by (fast intro: dividesI[of "c ⊗ u"] m_assoc)
  qed (simp_all add: acarr bcarr ccarr)
qed

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)"
  using assms
  by (blast intro: divides_fmsubset fmsubset_divides)
```

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 (rule properfactorI)
  show "a divides b"
    using assms asubb fmsubset_divides by blast
  show "¬ b divides a"
    by (meson anb assms asubb factorial_monoid.divides_fmsubset factorial_monoid_axioms
subset_mset.antisym)
qed

lemma (in factorial_monoid) properfactor_fmset:
  assumes "properfactor G a b"
    and "wfactors G as a"
    and "wfactors G bs b"
    and "a ∈ carrier G"
    and "b ∈ carrier G"
    and "set as ⊆ carrier G"
    and "set bs ⊆ carrier G"
  shows "fmset G as ⊆# fmset G bs"
  using assms
  by (meson divides_as_fmsubset properfactor_divides)

lemma (in factorial_monoid) properfactor_fmset_ne:
  assumes pf: "properfactor G a b"
    and "wfactors G as a"
    and "wfactors G bs b"
    and "a ∈ carrier G"
    and "b ∈ carrier G"
    and "set as ⊆ carrier G"
    and "set bs ⊆ carrier G"
  shows "fmset G as ≠ fmset G bs"
  using properfactorE [OF pf] assms divides_as_fmsubset by force
```

## 26.7 Irreducible Elements are Prime

```
lemma (in factorial_monoid) irreducible_prime:
  assumes pirr: "irreducible G p" and pcarr: "p ∈ carrier G"
  shows "prime G p"
  using pirr
proof (elim irreducibleE, intro primeI)
  fix a b
```

```
  assume acarr: "a ∈ carrier G"  and bcarr: "b ∈ carrier G"
    and pdvdab: "p divides (a ⊗ b)"
    and pnunit: "p ∉ Units G"
  assume irreduc[rule_format]:
    "∀b. b ∈ carrier G ∧ properfactor G b p ⟶ b ∈ Units G"
  from pdvdab obtain c where ccarr: "c ∈ carrier G" and abpc: "a ⊗ b
= p ⊗ c"
    by (rule dividesE)
  obtain as where ascarr: "set as ⊆ carrier G" and afs: "wfactors G
as a"
    using wfactors_exist [OF acarr] by blast
  obtain bs where bscarr: "set bs ⊆ carrier G" and bfs: "wfactors G
bs b"
    using wfactors_exist [OF bcarr] by blast
  obtain cs where cscarr: "set cs ⊆ carrier G" and cfs: "wfactors G
cs c"
    using wfactors_exist [OF ccarr] by blast
  note carr[simp] = pcarr acarr bcarr ccarr ascarr bscarr cscarr
  from pirr cfs  abpc have "wfactors G (p # cs) (a ⊗ b)"
    by (simp add: wfactors_mult_single)
  moreover have  "wfactors G (as @ bs) (a ⊗ b)"
    by (rule wfactors_mult [OF afs bfs]) fact+
  ultimately have "essentially_equal G (p # cs) (as @ bs)"
    by (rule wfactors_unique) simp+
  then obtain ds where "p # cs <~~> ds" and dsassoc: "ds [∼] (as @ bs)"
    by (fast elim: essentially_equalE)
  then have "p ∈ set ds"
    by (metis ‹mset (p # cs) = mset ds› insert_iff list.set(2) perm_set_eq)

  with dsassoc obtain p' where "p' ∈ set (as@bs)" and pp': "p ∼ p'"
    unfolding list_all2_conv_all_nth set_conv_nth by force
  then consider "p' ∈ set as" | "p' ∈ set bs" by auto
  then show "p divides a ∨ p divides b"
    using afs bfs divides_cong_l pp' wfactors_dividesI
    by (meson acarr ascarr bcarr bscarr pcarr)
qed


— A version using factors, more complicated
lemma (in factorial_monoid) factors_irreducible_prime:
  assumes pirr: "irreducible G p" and pcarr: "p ∈ carrier G"
  shows "prime G p"
proof (rule primeI)
  show "p ∉ Units G"
    by (meson irreducibleE pirr)
  have irreduc: "⋀b. ⟦b ∈ carrier G; properfactor G b p⟧ ⟹ b ∈ Units
G"
    using pirr by (auto simp: irreducible_def)
  show "p divides a ∨ p divides b"
```

```
    if acarr: "a ∈ carrier G" and bcarr: "b ∈ carrier G" and pdvdab:
"p divides (a ⊗ b)" for a b
  proof -
    from pdvdab obtain c where ccarr: "c ∈ carrier G" and abpc: "a ⊗
b = p ⊗ c"
      by (rule dividesE)
    note [simp] = pcarr acarr bcarr ccarr

    show "p divides a ∨ p divides b"
    proof (cases "a ∈ Units G")
      case True
      then have "p divides b"
        by (metis acarr associatedI2' associated_def bcarr divides_trans
m_comm pcarr pdvdab)
      then show ?thesis ..
    next
      case anunit: False
      show ?thesis
      proof (cases "b ∈ Units G")
        case True
        then have "p divides a"
          by (meson acarr bcarr divides_unit irreducible_prime pcarr pdvdab
pirr prime_def)
        then show ?thesis ..
      next
        case bnunit: False
        then have cnunit: "c ∉ Units G"
          by (metis abpc acarr anunit bcarr ccarr irreducible_prodE irreducible_prod_rI
pcarr pirr)
        then have abnunit: "a ⊗ b ∉ Units G"
          using acarr anunit bcarr unit_factor by blast
        obtain as where ascarr: "set as ⊆ carrier G" and afac: "factors
G as a"
          using factors_exist [OF acarr anunit] by blast
        obtain bs where bscarr: "set bs ⊆ carrier G" and bfac: "factors
G bs b"
          using factors_exist [OF bcarr bnunit] by blast
        obtain cs where cscarr: "set cs ⊆ carrier G" and cfac: "factors
G cs c"
          using factors_exist [OF ccarr cnunit] by auto
        note [simp] = ascarr bscarr cscarr
        from pirr cfac abpc have abfac': "factors G (p # cs) (a ⊗ b)"
          by (simp add: factors_mult_single)
        from afac and bfac have "factors G (as @ bs) (a ⊗ b)"
          by (rule factors_mult) fact+
        with abfac' have "essentially_equal G (p # cs) (as @ bs)"
          using abnunit factors_unique by auto
        then obtain ds where "p # cs <~~> ds" and dsassoc: "ds [∼] (as
@ bs)"
```

```
          by (fast elim: essentially_equalE)
        then have "p ∈ set ds"
          by (metis list.set_intros(1) set_mset_mset)
        with dsassoc obtain p' where "p' ∈ set (as@bs)" and pp': "p
∼ p'"
          unfolding list_all2_conv_all_nth set_conv_nth by force
        then consider "p' ∈ set as" | "p' ∈ set bs" by auto
        then show "p divides a ∨ p divides b"
          by (meson afac bfac divides_cong_l factors_dividesI pp' ascarr
bscarr pcarr)
      qed
    qed
  qed
qed
```

## 26.8 Greatest Common Divisors and Lowest Common Multiples

### 26.8.1 Definitions

```
definition isgcd :: "[('a,_) monoid_scheme, 'a, 'a, 'a] ⇒ bool"
    (‹(‹notation=‹mixfix gcdof››_ gcdofı _ _)› [81,81,81] 80)
  where "x gcdof_G a b ⟷ x divides_G a ∧ x divides_G b ∧
    (∀y∈carrier G. (y divides_G a ∧ y divides_G b ⟶ y divides_G x))"

definition islcm :: "[_, 'a, 'a, 'a] ⇒ bool"
    (‹(‹notation=‹mixfix lcmof››_ lcmofı _ _)› [81,81,81] 80)
  where "x lcmof_G a b ⟷ a divides_G x ∧ b divides_G x ∧
    (∀y∈carrier G. (a divides_G y ∧ b divides_G y ⟶ x divides_G y))"

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

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

definition "SomeGcd G A = Lattice.inf (division_rel G) A"


locale gcd_condition_monoid = comm_monoid_cancel +
  assumes gcdof_exists: "⟦a ∈ carrier G; b ∈ carrier G⟧ ⟹ ∃c. c ∈ carrier
G ∧ c gcdof a b"

locale primeness_condition_monoid = comm_monoid_cancel +
  assumes irreducible_prime: "⟦a ∈ carrier G; irreducible G a⟧ ⟹ prime
G a"

locale divisor_chain_condition_monoid = comm_monoid_cancel +
  assumes division_wellfounded: "wf {(x, y). x ∈ carrier G ∧ y ∈ carrier
G ∧ properfactor G x y}"
```

### 26.8.2 Connections to Lattice.thy

```
lemma gcdof_greatestLower:
  fixes G (structure)
  assumes carr[simp]: "a ∈ carrier G"  "b ∈ carrier G"
  shows "(x ∈ carrier G ∧ x gcdof a b) = greatest (division_rel G) x
(Lower (division_rel G) {a, b})"
  by (auto simp: isgcd_def greatest_def Lower_def elem_def)

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})"
  by (auto simp: islcm_def least_def Upper_def elem_def)

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"
  by (simp add: somegcd_def meet_def inf_def gcdof_greatestLower[OF carr])

lemma (in monoid) isgcd_divides_l:
  assumes "a divides b"
    and "a ∈ carrier G"  "b ∈ carrier G"
  shows "a gcdof a b"
  using assms unfolding isgcd_def by fast

lemma (in monoid) isgcd_divides_r:
  assumes "b divides a"
    and "a ∈ carrier G"  "b ∈ carrier G"
  shows "b gcdof a b"
  using assms unfolding isgcd_def by fast
```

### 26.8.3 Existence of gcd and lcm

```
lemma (in factorial_monoid) gcdof_exists:
  assumes acarr: "a ∈ carrier G"
    and bcarr: "b ∈ carrier G"
  shows "∃c. c ∈ carrier G ∧ c gcdof a b"
proof -
  from wfactors_exist [OF acarr]
  obtain as where ascarr: "set as ⊆ carrier G" and afs: "wfactors G
as a"
    by blast
  from afs have airr: "∀a ∈ set as. irreducible G a"
    by (fast elim: wfactorsE)

  from wfactors_exist [OF bcarr]
  obtain bs where bscarr: "set bs ⊆ carrier G" and bfs: "wfactors G
```

```
bs b"
    by blast
  from bfs have birr: "∀b ∈ set bs. irreducible G b"
    by (fast elim: wfactorsE)

  have "∃c cs. c ∈ carrier G ∧ set cs ⊆ carrier G ∧ wfactors G cs c
∧
    fmset G cs = fmset G as ∩# fmset G bs"
  proof (intro mset_wfactorsEx)
    fix X
    assume "X ∈# fmset G as ∩# fmset G bs"
    then have "X ∈# fmset G as" by simp
    then have "X ∈ set (map (assocs G) as)"
      by (simp add: fmset_def)
    then have "∃x. X = assocs G x ∧ x ∈ set as"
      by (induct as) auto
    then obtain x where X: "X = assocs G x" and xas: "x ∈ set as"
      by blast
    with ascarr have xcarr: "x ∈ carrier G"
      by blast
    from xas airr have xirr: "irreducible G x"
      by simp
    from xcarr and xirr and X show "∃x. (x ∈ carrier G ∧ irreducible
G x) ∧ X = assocs G x"
      by blast
  qed
  then obtain c cs
    where ccarr: "c ∈ carrier G"
      and cscarr: "set cs ⊆ carrier G"
      and csirr: "wfactors G cs c"
      and csmset: "fmset G cs = fmset G as ∩# fmset G bs"
    by auto

  have "c gcdof a b"
  proof (simp add: isgcd_def, safe)
    from csmset
    have "fmset G cs ⊆# fmset G as"
      by simp
    then show "c divides a" by (rule fmsubset_divides) fact+
  next
    from csmset have "fmset G cs ⊆# fmset G bs"
      by simp
    then show "c divides b"
      by (rule fmsubset_divides) fact+
  next
    fix y
    assume "y ∈ carrier G"
    from wfactors_exist [OF this]
    obtain ys where yscarr: "set ys ⊆ carrier G" and yfs: "wfactors
```

```
G ys y"
      by blast

    assume "y divides a"
    then have ya: "fmset G ys ⊆# fmset G as"
      by (rule divides_fmsubset) fact+

    assume "y divides b"
    then have yb: "fmset G ys ⊆# fmset G bs"
      by (rule divides_fmsubset) fact+

    from ya yb csmset have "fmset G ys ⊆# fmset G cs"
      by (simp add: subset_mset_def)
    then show "y divides c"
      by (rule fmsubset_divides) fact+
  qed
  with ccarr show "∃c. c ∈ carrier G ∧ c gcdof a b"
    by fast
qed

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 -
  from wfactors_exist [OF acarr]
  obtain as where ascarr: "set as ⊆ carrier G" and afs: "wfactors G
as a"
    by blast
  from afs have airr: "∀a ∈ set as. irreducible G a"
    by (fast elim: wfactorsE)

  from wfactors_exist [OF bcarr]
  obtain bs where bscarr: "set bs ⊆ carrier G" and bfs: "wfactors G
bs b"
    by blast
  from bfs have birr: "∀b ∈ set bs. irreducible G b"
    by (fast elim: wfactorsE)

  have "∃c cs. c ∈ carrier G ∧ set cs ⊆ carrier G ∧ wfactors G cs c
∧
    fmset G cs = (fmset G as - fmset G bs) + fmset G bs"
  proof (intro mset_wfactorsEx)
    fix X
    assume "X ∈# (fmset G as - fmset G bs) + fmset G bs"
    then have "X ∈# fmset G as ∨ X ∈# fmset G bs"
      by (auto dest: in_diffD)
    then consider "X ∈ set_mset (fmset G as)" | "X ∈ set_mset (fmset
G bs)"
```

```
      by fast
    then show "∃x. (x ∈ carrier G ∧ irreducible G x) ∧ X = assocs G
x"
    proof cases
      case 1
      then have "X ∈ set (map (assocs G) as)" by (simp add: fmset_def)
      then have "∃x. x ∈ set as ∧ X = assocs G x" by (induct as) auto
      then obtain x where xas: "x ∈ set as" and X: "X = assocs G x"
by auto
      with ascarr have xcarr: "x ∈ carrier G" by fast
      from xas airr have xirr: "irreducible G x" by simp
      from xcarr and xirr and X show ?thesis by fast
    next
      case 2
      then have "X ∈ set (map (assocs G) bs)" by (simp add: fmset_def)
      then have "∃x. x ∈ set bs ∧ X = assocs G x" by (induct as) auto
      then obtain x where xbs: "x ∈ set bs" and X: "X = assocs G x"
by auto
      with bscarr have xcarr: "x ∈ carrier G" by fast
      from xbs birr have xirr: "irreducible G x" by simp
      from xcarr and xirr and X show ?thesis by fast
    qed
  qed
  then obtain c cs
    where ccarr: "c ∈ carrier G"
      and cscarr: "set cs ⊆ carrier G"
      and csirr: "wfactors G cs c"
      and csmset: "fmset G cs = fmset G as - fmset G bs + fmset G bs"
    by auto

  have "c lcmof a b"
  proof (simp add: islcm_def, safe)
    from csmset have "fmset G as ⊆# fmset G cs"
      by (simp add: subseteq_mset_def, force)
    then show "a divides c"
      by (rule fmsubset_divides) fact+
  next
    from csmset have "fmset G bs ⊆# fmset G cs"
      by (simp add: subset_mset_def)
    then show "b divides c"
      by (rule fmsubset_divides) fact+
  next
    fix y
    assume "y ∈ carrier G"
    from wfactors_exist [OF this]
    obtain ys where yscarr: "set ys ⊆ carrier G" and yfs: "wfactors
G ys y"
      by blast
```

```
    assume "a divides y"
    then have ya: "fmset G as ⊆# fmset G ys"
      by (rule divides_fmsubset) fact+

    assume "b divides y"
    then have yb: "fmset G bs ⊆# fmset G ys"
      by (rule divides_fmsubset) fact+

    from ya yb csmset have "fmset G cs ⊆# fmset G ys"
      using subset_eq_diff_conv subset_mset.le_diff_conv2 by fastforce
    then show "c divides y"
      by (rule fmsubset_divides) fact+
  qed
  with ccarr show "∃c. c ∈ carrier G ∧ c lcmof a b"
    by fast
qed
```

## 26.9 Conditions for Factoriality

### 26.9.1 Gcd condition

```
lemma (in gcd_condition_monoid) division_weak_lower_semilattice [simp]:
  "weak_lower_semilattice (division_rel G)"
proof -
  interpret weak_partial_order "division_rel G" ..
  show ?thesis
  proof (unfold_locales, simp_all)
    fix x y
    assume carr: "x ∈ carrier G"  "y ∈ carrier G"
    from gcdof_exists [OF this] obtain z where zcarr: "z ∈ carrier G"
and isgcd: "z gcdof x y"
      by blast
    with carr have "greatest (division_rel G) z (Lower (division_rel
G) {x, y})"
      by (subst gcdof_greatestLower[symmetric], simp+)
    then show "∃z. greatest (division_rel G) z (Lower (division_rel G)
{x, y})"
      by fast
  qed
qed

lemma (in gcd_condition_monoid) gcdof_cong_l:
  assumes "a' ∼ a" "a gcdof b c" "a' ∈ carrier G" and carr': "a ∈ carrier
G"  "b ∈ carrier G"  "c ∈ carrier G"
  shows "a' gcdof b c"
proof -
  interpret weak_lower_semilattice "division_rel G" by simp
  have "is_glb (division_rel G) a' {b, c}"
    by (subst greatest_Lower_cong_l[of _ a]) (simp_all add: assms gcdof_greatestLower[symme
  then have "a' ∈ carrier G ∧ a' gcdof b c"
```

```
    by (simp add: gcdof_greatestLower carr')
  then show ?thesis ..
qed

lemma (in gcd_condition_monoid) gcd_closed [simp]:
  assumes "a ∈ carrier G" "b ∈ carrier G"
  shows "somegcd G a b ∈ carrier G"
proof -
  interpret weak_lower_semilattice "division_rel G" by simp
  show ?thesis
  using  assms meet_closed by (simp add: somegcd_meet)
qed

lemma (in gcd_condition_monoid) gcd_isgcd:
  assumes "a ∈ carrier G"  "b ∈ carrier G"
  shows "(somegcd G a b) gcdof a b"
proof -
  interpret weak_lower_semilattice "division_rel G"
    by simp
  from assms have "somegcd G a b ∈ carrier G ∧ (somegcd G a b) gcdof
a b"
    by (simp add: gcdof_greatestLower inf_of_two_greatest meet_def somegcd_meet)
  then show "(somegcd G a b) gcdof a b"
    by simp
qed

lemma (in gcd_condition_monoid) gcd_exists:
  assumes "a ∈ carrier G"  "b ∈ carrier G"
  shows "∃x∈carrier G. x = somegcd G a b"
proof -
  interpret weak_lower_semilattice "division_rel G"
    by simp
  show ?thesis
    by (metis assms gcd_closed)
qed

lemma (in gcd_condition_monoid) gcd_divides_l:
  assumes "a ∈ carrier G" "b ∈ carrier G"
  shows "(somegcd G a b) divides a"
proof -
  interpret weak_lower_semilattice "division_rel G"
    by simp
  show ?thesis
    by (metis assms gcd_isgcd isgcd_def)
qed

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

```
proof -
  interpret weak_lower_semilattice "division_rel G"
    by simp
  show ?thesis
    by (metis assms gcd_isgcd isgcd_def)
qed

lemma (in gcd_condition_monoid) gcd_divides:
  assumes "z divides x" "z divides y"
    and L: "x ∈ carrier G"  "y ∈ carrier G"  "z ∈ carrier G"
  shows "z divides (somegcd G x y)"
proof -
  interpret weak_lower_semilattice "division_rel G"
    by simp
  show ?thesis
    by (metis gcd_isgcd isgcd_def assms)
qed

lemma (in gcd_condition_monoid) gcd_cong_l:
  assumes "x ∼ x'" "x ∈ carrier G"  "x' ∈ carrier G"  "y ∈ carrier G"
  shows "somegcd G x y ∼ somegcd G x' y"
proof -
  interpret weak_lower_semilattice "division_rel G"
    by simp
  show ?thesis
    using somegcd_meet assms
    by (metis eq_object.select_convs(1) meet_cong_l partial_object.select_convs(1))
qed

lemma (in gcd_condition_monoid) gcd_cong_r:
  assumes "y ∼ y'" "x ∈ carrier G"  "y ∈ carrier G" "y' ∈ carrier G"
  shows "somegcd G x y ∼ somegcd G x y'"
proof -
  interpret weak_lower_semilattice "division_rel G" by simp
  show ?thesis
    by (meson associated_def assms gcd_closed gcd_divides gcd_divides_l
gcd_divides_r monoid.divides_trans monoid_axioms)
qed

lemma (in gcd_condition_monoid) gcdI:
  assumes dvd: "a divides b"  "a divides c"
    and others: "⋀y. ⟦y∈carrier G; y divides b; y divides c⟧ ⟹ y divides
a"
    and acarr: "a ∈ carrier G" and bcarr: "b ∈ carrier G" and ccarr:
"c ∈ carrier G"
  shows "a ∼ somegcd G b c"
proof -
  have "∃a. a ∈ carrier G ∧ a gcdof b c"
    by (simp add: bcarr ccarr gcdof_exists)
```

```
  moreover have "⋀x. x ∈ carrier G ∧ x gcdof b c ⟹ a ∼ x"
    by (simp add: acarr associated_def dvd isgcd_def others)
  ultimately show ?thesis
    unfolding somegcd_def by (blast intro: someI2_ex)
qed

lemma (in gcd_condition_monoid) gcdI2:
  assumes "a gcdof b c" and "a ∈ carrier G" and "b ∈ carrier G" and
"c ∈ carrier G"
  shows "a ∼ somegcd G b c"
  using assms unfolding isgcd_def
  by (simp add: gcdI)

lemma (in gcd_condition_monoid) SomeGcd_ex:
  assumes "finite A"  "A ⊆ carrier G"  "A ≠ {}"
  shows "∃x ∈ carrier G. x = SomeGcd G A"
proof -
  interpret weak_lower_semilattice "division_rel G"
    by simp
  show ?thesis
    using finite_inf_closed by (simp add: assms SomeGcd_def)
qed

lemma (in gcd_condition_monoid) gcd_assoc:
  assumes "a ∈ carrier G"  "b ∈ carrier G"  "c ∈ carrier G"
  shows "somegcd G (somegcd G a b) c ∼ somegcd G a (somegcd G b c)"
proof -
  interpret weak_lower_semilattice "division_rel G"
    by simp
  show ?thesis
    unfolding associated_def
    by (meson assms divides_trans gcd_divides gcd_divides_l gcd_divides_r
gcd_exists)
qed

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 -
  let ?d = "somegcd G a b"
  let ?e = "somegcd G (c ⊗ a) (c ⊗ b)"
  note carr[simp] = acarr bcarr ccarr
  have dcarr: "?d ∈ carrier G" by simp
  have ecarr: "?e ∈ carrier G" by simp
  note carr = carr dcarr ecarr

  have "?d divides a" by (simp add: gcd_divides_l)
  then have cd'ca: "c ⊗ ?d divides (c ⊗ a)" by (simp add: divides_mult_lI)
```

```
  have "?d divides b" by (simp add: gcd_divides_r)
  then have cd'cb: "c ⊗ ?d divides (c ⊗ b)" by (simp add: divides_mult_lI)

  from cd'ca cd'cb have cd'e: "c ⊗ ?d divides ?e"
    by (rule gcd_divides) simp_all
  then obtain u where ucarr[simp]: "u ∈ carrier G" and e_cdu: "?e =
c ⊗ ?d ⊗ u"
    by blast

  note carr = carr ucarr

  have "?e divides c ⊗ a" by (rule gcd_divides_l) simp_all
  then obtain x where xcarr: "x ∈ carrier G" and ca_ex: "c ⊗ a = ?e
⊗ x"
    by blast
  with e_cdu have ca_cdux: "c ⊗ a = c ⊗ ?d ⊗ u ⊗ x"
    by simp

  from ca_cdux xcarr have "c ⊗ a = c ⊗ (?d ⊗ u ⊗ x)"
    by (simp add: m_assoc)
  then have "a = ?d ⊗ u ⊗ x"
    by (rule l_cancel[of c a]) (simp add: xcarr)+
  then have du'a: "?d ⊗ u divides a"
    by (rule dividesI[OF xcarr])

  have "?e divides c ⊗ b" by (intro gcd_divides_r) simp_all
  then obtain x where xcarr: "x ∈ carrier G" and cb_ex: "c ⊗ b = ?e
⊗ x"
    by blast
  with e_cdu have cb_cdux: "c ⊗ b = c ⊗ ?d ⊗ u ⊗ x"
    by simp

  from cb_cdux xcarr have "c ⊗ b = c ⊗ (?d ⊗ u ⊗ x)"
    by (simp add: m_assoc)
  with xcarr have "b = ?d ⊗ u ⊗ x"
    by (intro l_cancel[of c b]) simp_all
  then have du'b: "?d ⊗ u divides b"
    by (intro dividesI[OF xcarr])

  from du'a du'b carr have du'd: "?d ⊗ u divides ?d"
    by (intro gcd_divides) simp_all
  then have uunit: "u ∈ Units G"
  proof (elim dividesE)
    fix v
    assume vcarr[simp]: "v ∈ carrier G"
    assume d: "?d = ?d ⊗ u ⊗ v"
    have "?d ⊗ 1 = ?d ⊗ u ⊗ v" by simp fact
    also have "?d ⊗ u ⊗ v = ?d ⊗ (u ⊗ v)" by (simp add: m_assoc)
```

```
    finally have "?d ⊗ 1 = ?d ⊗ (u ⊗ v)" .
    then have i2: "1 = u ⊗ v" by (rule l_cancel) simp_all
    then have i1: "1 = v ⊗ u" by (simp add: m_comm)
    from vcarr i1[symmetric] i2[symmetric] show "u ∈ Units G"
      by (auto simp: Units_def)
  qed

  from e_cdu uunit have "somegcd G (c ⊗ a) (c ⊗ b) ∼ c ⊗ somegcd G
a b"
    by (intro associatedI2[of u]) simp_all
  from this[symmetric] show "c ⊗ somegcd G a b ∼ somegcd G (c ⊗ a)
(c ⊗ b)"
    by simp
qed

lemma (in monoid) assoc_subst:
  assumes ab: "a ∼ b"
    and cP: "∀a b. a ∈ carrier G ∧ b ∈ carrier G ∧ a ∼ b
      ⟶ f a ∈ carrier G ∧ f b ∈ carrier G ∧ f a ∼ f b"
    and carr: "a ∈ carrier G"  "b ∈ carrier G"
  shows "f a ∼ f b"
  using assms by auto

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 -
  have "c = c ⊗ 1" by simp
  also from abrelprime[symmetric]
  have "... ∼ c ⊗ somegcd G a b"
    by (rule assoc_subst) (simp add: mult_cong_r)+
  also have "... ∼ somegcd G (c ⊗ a) (c ⊗ b)"
    by (rule gcd_mult) fact+
  finally have c: "c ∼ somegcd G (c ⊗ a) (c ⊗ b)"
    by simp

  from carr have a: "a ∼ somegcd G a (c ⊗ a)"
    by (fast intro: gcdI divides_prod_l)

  have "somegcd G a (b ⊗ c) ∼ somegcd G a (c ⊗ b)"
    by (simp add: m_comm)
  also from a have "... ∼ somegcd G (somegcd G a (c ⊗ a)) (c ⊗ b)"
    by (rule assoc_subst) (simp add: gcd_cong_l)+
  also from gcd_assoc have "... ∼ somegcd G a (somegcd G (c ⊗ a) (c ⊗
b))"
    by (rule assoc_subst) simp+
  also from c[symmetric] have "... ∼ somegcd G a c"
```

```
    by (rule assoc_subst) (simp add: gcd_cong_r)+
  also note acrelprime
  finally show "somegcd G a (b ⊗ c) ∼ 1"
    by simp
qed

lemma (in gcd_condition_monoid) primeness_condition: "primeness_condition_monoid
G"
proof -
  have *: "p divides a ∨ p divides b"
    if pcarr[simp]: "p ∈ carrier G" and acarr[simp]: "a ∈ carrier G" and
bcarr[simp]: "b ∈ carrier G"
      and pirr: "irreducible G p" and pdvdab: "p divides a ⊗ b"
    for p a b
  proof -
    from pirr have pnunit: "p ∉ Units G"
      and r: "⋀b. ⟦b ∈ carrier G; properfactor G b p⟧ ⟹ b ∈ Units G"
      by (fast elim: irreducibleE)+
    show "p divides a ∨ p divides b"
    proof (rule ccontr, clarsimp)
      assume npdvda: "¬ p divides a" and npdvdb: "¬ p divides b"
      have "1 ∼ somegcd G p a"
      proof (intro gcdI unit_divides)
        show "⋀y. ⟦y ∈ carrier G; y divides p; y divides a⟧ ⟹ y ∈ Units
G"
          by (meson divides_trans npdvda pcarr properfactorI r)
      qed auto
      with pcarr acarr have pa: "somegcd G p a ∼ 1"
        by (fast intro: associated_sym[of "1"] gcd_closed)
      have "1 ∼ somegcd G p b"
      proof (intro gcdI unit_divides)
        show "⋀y. ⟦y ∈ carrier G; y divides p; y divides b⟧ ⟹ y ∈ Units
G"
          by (meson divides_trans npdvdb pcarr properfactorI r)
      qed auto
      with pcarr bcarr have pb: "somegcd G p b ∼ 1"
        by (fast intro: associated_sym[of "1"] gcd_closed)
      have "p ∼ somegcd G p (a ⊗ b)"
        using pdvdab by (simp add: gcdI2 isgcd_divides_l)
      also from pa pb pcarr acarr bcarr have "somegcd G p (a ⊗ b) ∼ 1"
        by (rule relprime_mult)
      finally have "p ∼ 1"
        by simp
      with pcarr have "p ∈ Units G"
        by (fast intro: assoc_unit_l)
      with pnunit show False ..
    qed
  qed
  show ?thesis
```

```
    by unfold_locales (metis * primeI irreducibleE)
qed


sublocale gcd_condition_monoid ⊆ primeness_condition_monoid
  by (rule primeness_condition)
```

### 26.9.2 Divisor chain condition

```
lemma (in divisor_chain_condition_monoid) wfactors_exist:
  assumes acarr: "a ∈ carrier G"
  shows "∃as. set as ⊆ carrier G ∧ wfactors G as a"
proof -
  have r: "a ∈ carrier G ⟹ (∃as. set as ⊆ carrier G ∧ wfactors G as
a)"
    using division_wellfounded
  proof (induction rule: wf_induct_rule)
    case (less x)
    then have xcarr: "x ∈ carrier G"
      by auto
    show ?case
    proof (cases "x ∈ Units G")
      case True
      then show ?thesis
        by (metis bot.extremum list.set(1) unit_wfactors)
    next
      case xnunit: False
      show ?thesis
      proof (cases "irreducible G x")
        case True
        then show ?thesis
          by (rule_tac x="[x]" in exI) (simp add: wfactors_def xcarr)
      next
        case False
        then obtain y where ycarr: "y ∈ carrier G" and ynunit: "y ∉
Units G" and pfyx: "properfactor G y x"
          by (meson irreducible_def xnunit)
        obtain ys where yscarr: "set ys ⊆ carrier G" and yfs: "wfactors
G ys y"
          using less ycarr pfyx by blast
        then obtain z where zcarr: "z ∈ carrier G" and x: "x = y ⊗ z"
          by (meson dividesE pfyx properfactorE2)
        from zcarr ycarr have "properfactor G z x"
          using m_comm properfactorI3 x ynunit by blast
        with less zcarr obtain zs where zscarr: "set zs ⊆ carrier G"
and zfs: "wfactors G zs z"
          by blast
        from yscarr zscarr have xscarr: "set (ys@zs) ⊆ carrier G"
          by simp
```

```
        have "wfactors G (ys@zs) (y⊗z)"
          using xscarr ycarr yfs zcarr zfs by auto
        then have "wfactors G (ys@zs) x"
          by (simp add: x)
        with xscarr show "∃xs. set xs ⊆ carrier G ∧ wfactors G xs x"
          by fast
      qed
    qed
  qed
  from acarr show ?thesis by (rule r)
qed
```

### 26.9.3 Primeness condition

```
lemma (in comm_monoid_cancel) multlist_prime_pos:
  assumes aprime: "prime G a" and carr: "a ∈ carrier G"
     and as: "set as ⊆ carrier G" "a divides (foldr (⊗) as 1)"
   shows "∃i<length as. a divides (as!i)"
  using as
proof (induction as)
  case Nil
  then show ?case
    by simp (meson Units_one_closed aprime carr divides_unit primeE)
next
  case (Cons x as)
  then have "x ∈ carrier G"  "set as ⊆ carrier G" and "a divides x ⊗
foldr (⊗) as 1"
    by auto
  with carr aprime have "a divides x ∨ a divides foldr (⊗) as 1"
    by (intro prime_divides) simp+
  then show ?case
    using Cons.IH Cons.prems(1) by force
qed

proposition (in primeness_condition_monoid) wfactors_unique:
  assumes "wfactors G as a"  "wfactors G as' a"
    and "a ∈ carrier G"  "set as ⊆ carrier G"  "set as' ⊆ carrier G"
  shows "essentially_equal G as as'"
  using assms
proof (induct as arbitrary: a as')
  case Nil
  then have "a ∼ 1"
    by (simp add: perm_wfactorsD)
  then have "as' = []"
    using Nil.prems assoc_unit_l unit_wfactors_empty by blast
  then show ?case
    by auto
next
  case (Cons ah as)
```

```
  then have ahdvda: "ah divides a"
    using wfactors_dividesI by auto
    then obtain a' where a'carr: "a' ∈ carrier G" and a: "a = ah ⊗ a'"
      by blast
    have carr_ah: "ah ∈ carrier G" "set as ⊆ carrier G"
      using Cons.prems by fastforce+
    have "ah ⊗ foldr (⊗) as 1 ∼ a"
      by (rule wfactorsE[OF ‹wfactors G (ah # as) a›]) auto
    then have "foldr (⊗) as 1 ∼ a'"
      by (metis Cons.prems(4) a a'carr assoc_l_cancel insert_subset list.set(2)
monoid.multlist_closed monoid_axioms)
    then
    have a'fs: "wfactors G as a'"
      by (meson Cons.prems(1) set_subset_Cons subset_iff wfactorsE wfactorsI)
    then have ahirr: "irreducible G ah"
      by (meson Cons.prems(1) list.set_intros(1) wfactorsE)
    with Cons have ahprime: "prime G ah"
      by (simp add: irreducible_prime)
    note ahdvda
    also have "a divides (foldr (⊗) as' 1)"
      by (meson Cons.prems(2) associatedE wfactorsE)
    finally have "ah divides (foldr (⊗) as' 1)"
      using Cons.prems(4) by auto
    with ahprime have "∃i<length as'. ah divides as'!i"
      by (intro multlist_prime_pos) (use Cons.prems in auto)
    then obtain i where len: "i<length as'" and ahdvd: "ah divides as'!i"
      by blast
    then obtain x where "x ∈ carrier G" and asi: "as'!i = ah ⊗ x"
      by blast
    have irrasi: "irreducible G (as'!i)"
      using nth_mem[OF len] wfactorsE
      by (metis Cons.prems(2))
    have asicarr[simp]: "as'!i ∈ carrier G"
      using len ‹set as' ⊆ carrier G› nth_mem by blast
    have asiah: "as'!i ∼ ah"
      by (metis ‹ah ∈ carrier G› ‹x ∈ carrier G› asi irrasi ahprime
associatedI2 irreducible_prodE primeE)
    note setparts = set_take_subset[of i as'] set_drop_subset[of "Suc
i" as']
    have "∃aa_1. aa_1 ∈ carrier G ∧ wfactors G (take i as') aa_1"
      using Cons
      by (metis setparts(1) subset_trans in_set_takeD wfactorsE wfactors_prod_exists)
    then obtain aa_1 where aa1carr [simp]: "aa_1 ∈ carrier G" and aa1fs:
"wfactors G (take i as') aa_1"
      by auto
    obtain aa_2 where aa2carr [simp]: "aa_2 ∈ carrier G"
      and aa2fs: "wfactors G (drop (Suc i) as') aa_2"
      by (metis Cons.prems(2) Cons.prems(5) subset_code(1) in_set_dropD
wfactors_def wfactors_prod_exists)
```

```
    have set_drop: "set (drop (Suc i) as') ⊆ carrier G"
      using Cons.prems(5) setparts(2) by blast
    moreover have set_take: "set (take i as') ⊆ carrier G"
      using  Cons.prems(5) setparts by auto
    moreover have v1: "wfactors G (take i as' @ drop (Suc i) as') (aa_1
⊗ aa_2)"
      using aa1fs aa2fs ‹set as' ⊆ carrier G› by (force simp add: dest:
in_set_takeD in_set_dropD)
    ultimately have v1': "wfactors G (as'!i # take i as' @ drop (Suc i)
as') (as'!i ⊗ (aa_1 ⊗ aa_2))"
      using irrasi wfactors_mult_single
        by (simp add: irrasi v1 wfactors_mult_single)
    have "wfactors G (as'!i # drop (Suc i) as') (as'!i ⊗ aa_2)"
      by (simp add: aa2fs irrasi set_drop wfactors_mult_single)
    with len  aa1carr aa2carr aa1fs
    have v2: "wfactors G (take i as' @ as'!i # drop (Suc i) as') (aa_1
⊗ (as'!i ⊗ aa_2))"
      using wfactors_mult  by (simp add: set_take set_drop)
    from len have as': "as' = (take i as' @ as'!i # drop (Suc i) as')"
      by (simp add: Cons_nth_drop_Suc)
    have eer: "essentially_equal G (take i as' @ as'!i # drop (Suc i)
as') as'"
      using Cons.prems(5) as' by auto
    with v2 aa1carr aa2carr nth_mem[OF len] have "aa_1 ⊗ (as'!i ⊗ aa_2)
∼ a"
      using Cons.prems as' comm_monoid_cancel.ee_wfactorsD is_comm_monoid_cancel
by fastforce
    then have t1: "as'!i ⊗ (aa_1 ⊗ aa_2) ∼ a"
      by (metis aa1carr aa2carr asicarr m_lcomm)
    from asiah have "ah ⊗ (aa_1 ⊗ aa_2) ∼ as'!i ⊗ (aa_1 ⊗ aa_2)"
      by (simp add: ‹ah ∈ carrier G› associated_sym mult_cong_l)
    also note t1
    finally have "ah ⊗ (aa_1 ⊗ aa_2) ∼ a"
      using Cons.prems(3) carr_ah aa1carr aa2carr by blast
    with aa1carr aa2carr a'carr nth_mem[OF len] have a': "aa_1 ⊗ aa_2
∼ a'"
      using a assoc_l_cancel carr_ah(1) by blast
    note v1
    also note a'
    finally have "wfactors G (take i as' @ drop (Suc i) as') a'"
      by (simp add: a'carr set_drop set_take)
    from a'fs this have "essentially_equal G as (take i as' @ drop (Suc
i) as')"
      using Cons.hyps a'carr carr_ah(2) set_drop set_take by auto
    then obtain bs where ‹mset as = mset bs› ‹bs [∼] take i as' @ drop
(Suc i) as'›
      by (auto simp add: essentially_equal_def)
    with carr_ah have ‹mset (ah # as) = mset (ah # bs)› ‹ah # bs [∼]
```

```
ah # take i as' @ drop (Suc i) as'›
      by simp_all
    then have ee1: "essentially_equal G (ah # as) (ah # take i as' @
drop (Suc i) as')"
      unfolding essentially_equal_def by blast
    have ee2: "essentially_equal G (ah # take i as' @ drop (Suc i) as')
      (as' ! i # take i as' @ drop (Suc i) as')"
    proof (intro essentially_equalI)
      show "ah # take i as' @ drop (Suc i) as' <~~> ah # take i as' @
drop (Suc i) as'"
        by simp
    next
      show "ah # take i as' @ drop (Suc i) as' [∼] as' ! i # take i as'
@ drop (Suc i) as'"
        by (simp add: asiah associated_sym set_drop set_take)
    qed

    note ee1
    also note ee2
    also have "essentially_equal G (as' ! i # take i as' @ drop (Suc i)
as')
                                   (take i as' @ as' ! i # drop (Suc i)
as')"
      by (metis Cons.prems(5) as' essentially_equalI listassoc_refl perm_append_Cons)
    finally have "essentially_equal G (ah # as) (take i as' @ as' ! i #
drop (Suc i) as')"
      using Cons.prems(4) set_drop set_take by auto
    then show ?case
      using as' by auto
qed
```

### 26.9.4 Application to factorial monoids

Number of factors for wellfoundedness

```
definition factorcount :: "_ ⇒ 'a ⇒ nat"
  where "factorcount G a =
    (THE c. ∀as. set as ⊆ carrier G ∧ wfactors G as a ⟶ c = length
as)"

lemma (in monoid) ee_length:
  assumes ee: "essentially_equal G as bs"
  shows "length as = length bs"
  by (rule essentially_equalE[OF ee]) (metis list_all2_conv_all_nth perm_length)

lemma (in factorial_monoid) factorcount_exists:
  assumes carr[simp]: "a ∈ carrier G"
  shows "∃c. ∀as. set as ⊆ carrier G ∧ wfactors G as a ⟶ c = length
as"
proof -
```

```
  have "∃as. set as ⊆ carrier G ∧ wfactors G as a"
    by (intro wfactors_exist) simp
  then obtain as where ascarr[simp]: "set as ⊆ carrier G" and afs: "wfactors
G as a"
    by (auto simp del: carr)
  have "∀as'. set as' ⊆ carrier G ∧ wfactors G as' a ⟶ length as =
length as'"
    by (metis afs ascarr assms ee_length wfactors_unique)
  then show "∃c. ∀as'. set as' ⊆ carrier G ∧ wfactors G as' a ⟶ c
= length as'" ..
qed

lemma (in factorial_monoid) factorcount_unique:
  assumes afs: "wfactors G as a"
    and acarr[simp]: "a ∈ carrier G" and ascarr: "set as ⊆ carrier G"
  shows "factorcount G a = length as"
proof -
  have "∃ac. ∀as. set as ⊆ carrier G ∧ wfactors G as a ⟶ ac = length
as"
    by (rule factorcount_exists) simp
  then obtain ac where alen: "∀as. set as ⊆ carrier G ∧ wfactors G as
a ⟶ ac = length as"
    by auto
  then have ac: "ac = factorcount G a"
    unfolding factorcount_def using ascarr by (blast intro: theI2 afs)
  from ascarr afs have "ac = length as"
    by (simp add: alen)
  with ac show ?thesis
    by simp
qed

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 (rule dividesE[OF dvd])
  fix c
  from assms have "∃as. set as ⊆ carrier G ∧ wfactors G as a"
    by blast
  then obtain as where ascarr: "set as ⊆ carrier G" and afs: "wfactors
G as a"
    by blast
  with acarr have fca: "factorcount G a = length as"
    by (intro factorcount_unique)

  assume ccarr: "c ∈ carrier G"
  then have "∃cs. set cs ⊆ carrier G ∧ wfactors G cs c"
    by blast
```

```
  then obtain cs where cscarr: "set cs ⊆ carrier G" and cfs: "wfactors
G cs c"
    by blast

  note [simp] = acarr bcarr ccarr ascarr cscarr
  assume b: "b = a ⊗ c"
  from afs cfs have "wfactors G (as@cs) (a ⊗ c)"
    by (intro wfactors_mult) simp_all
  with b have "wfactors G (as@cs) b"
    by simp
  then have "factorcount G b = length (as@cs)"
    by (intro factorcount_unique) simp_all
  then have "factorcount G b = length as + length cs"
    by simp
  with fca show ?thesis
    by simp
qed

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"
  using assms
  by (auto simp: associated_def factorial_monoid.divides_fcount factorial_monoid_axioms
le_antisym)

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 (rule properfactorE[OF pf], elim dividesE)
  fix c
  from assms have "∃as. set as ⊆ carrier G ∧ wfactors G as a"
    by blast
  then obtain as where ascarr: "set as ⊆ carrier G" and afs: "wfactors
G as a"
    by blast
  with acarr have fca: "factorcount G a = length as"
    by (intro factorcount_unique)

  assume ccarr: "c ∈ carrier G"
  then have "∃cs. set cs ⊆ carrier G ∧ wfactors G cs c"
    by blast
  then obtain cs where cscarr: "set cs ⊆ carrier G" and cfs: "wfactors
G cs c"
    by blast

  assume b: "b = a ⊗ c"
```

```
  have "wfactors G (as@cs) (a ⊗ c)"
    by (rule wfactors_mult) fact+
  with b have "wfactors G (as@cs) b"
    by simp
  with ascarr cscarr bcarr have "factorcount G b = length (as@cs)"
    by (simp add: factorcount_unique)
  then have fcb: "factorcount G b = length as + length cs"
    by simp

  assume nbdvda: "¬ b divides a"
  have "c ∉ Units G"
  proof
    assume cunit:"c ∈ Units G"
    have "b ⊗ inv c = a ⊗ c ⊗ inv c"
      by (simp add: b)
    also from ccarr acarr cunit have "... = a ⊗ (c ⊗ inv c)"
      by (fast intro: m_assoc)
    also from ccarr cunit have "... = a ⊗ 1" by simp
    also from acarr have "... = a" by simp
    finally have "a = b ⊗ inv c" by simp
    with ccarr cunit have "b divides a"
      by (fast intro: dividesI[of "inv c"])
    with nbdvda show False by simp
  qed
  with cfs have "length cs > 0"
    by (metis Units_one_closed assoc_unit_r ccarr foldr.simps(1) id_apply
length_greater_0_conv wfactors_def)
  with fca fcb show ?thesis
    by simp
qed

sublocale factorial_monoid ⊆ divisor_chain_condition_monoid
  apply unfold_locales
  apply (rule wfUNIVI)
  apply (rule measure_induct[of "factorcount G"])
  using properfactor_fcount by auto

sublocale factorial_monoid ⊆ primeness_condition_monoid
  by standard (rule irreducible_prime)


lemma (in factorial_monoid) primeness_condition: "primeness_condition_monoid
G" ..

lemma (in factorial_monoid) gcd_condition [simp]: "gcd_condition_monoid
G"
  by standard (rule gcdof_exists)
```

```
sublocale factorial_monoid ⊆ gcd_condition_monoid
  by standard (rule gcdof_exists)

lemma (in factorial_monoid) division_weak_lattice [simp]: "weak_lattice
(division_rel G)"
proof -
  interpret weak_lower_semilattice "division_rel G"
    by simp
  show "weak_lattice (division_rel G)"
  proof (unfold_locales, simp_all)
    fix x y
    assume carr: "x ∈ carrier G"  "y ∈ carrier G"
    from lcmof_exists [OF this] obtain z where zcarr: "z ∈ carrier G"
and isgcd: "z lcmof x y"
      by blast
    with carr have "least (division_rel G) z (Upper (division_rel G)
{x, y})"
      by (simp add: lcmof_leastUpper[symmetric])
    then show "∃z. least (division_rel G) z (Upper (division_rel G) {x,
y})"
      by blast
  qed
qed
```

## 26.10 Factoriality Theorems

```
theorem factorial_condition_one:
  "divisor_chain_condition_monoid G ∧ primeness_condition_monoid G ⟷
factorial_monoid G"
proof (rule iffI, clarify)
  assume dcc: "divisor_chain_condition_monoid G"
    and pc: "primeness_condition_monoid G"
  interpret divisor_chain_condition_monoid "G" by (rule dcc)
  interpret primeness_condition_monoid "G" by (rule pc)
  show "factorial_monoid G"
    by (fast intro: factorial_monoidI wfactors_exist wfactors_unique)
next
  assume "factorial_monoid G"
  then interpret factorial_monoid "G" .
  show "divisor_chain_condition_monoid G ∧ primeness_condition_monoid
G"
    by rule unfold_locales
qed

theorem factorial_condition_two:
  "divisor_chain_condition_monoid G ∧ gcd_condition_monoid G ⟷ factorial_monoid
G"
proof (rule iffI, clarify)
  assume dcc: "divisor_chain_condition_monoid G"
```

```
    and gc: "gcd_condition_monoid G"
  interpret divisor_chain_condition_monoid "G" by (rule dcc)
  interpret gcd_condition_monoid "G" by (rule gc)
  show "factorial_monoid G"
    by (simp add: factorial_condition_one[symmetric], rule, unfold_locales)
next
  assume "factorial_monoid G"
  then interpret factorial_monoid "G" .
  show "divisor_chain_condition_monoid G ∧ gcd_condition_monoid G"
    by rule unfold_locales
qed

end


theory QuotRing
imports RingHom
begin
```

# 27 Quotient Rings

## 27.1 Multiplication on Cosets

```
definition rcoset_mult :: "[('a, _) ring_scheme, 'a set, 'a set, 'a set]
⇒ 'a set"
    (‹(‹open_block notation=‹mixfix rcoset_mult››[mod _:] _ ⨂ı _)›
[81,81,81] 80)
  where "rcoset_mult R I A B = (⋃a∈A. ⋃b∈B. I +>R (a ⊗R b))"
```

rcoset_mult fulfils the properties required by congruences

```
lemma (in ideal) rcoset_mult_add:
  assumes "x ∈ carrier R" "y ∈ carrier R"
  shows "[mod I:] (I +> x) ⨂ (I +> y) = I +> (x ⊗ y)"
proof -
  have 1: "z ∈ I +> x ⊗ y"
    if x'rcos: "x' ∈ I +> x" and y'rcos: "y' ∈ I +> y" and zrcos: "z
∈ I +> x' ⊗ y'" for z x' y'
  proof -
    from that
    obtain hx hy hz where hxI: "hx ∈ I" and x': "x' = hx ⊕ x" and hyI:
"hy ∈ I" and y': "y' = hy ⊕ y"
      and hzI: "hz ∈ I" and z: "z = hz ⊕ (x' ⊗ y')"
      by (auto simp: a_r_coset_def r_coset_def)
    note carr = assms hxI[THEN a_Hcarr] hyI[THEN a_Hcarr] hzI[THEN a_Hcarr]
    from z  x' y' have "z = hz ⊕ ((hx ⊕ x) ⊗ (hy ⊕ y))" by simp
    also from carr have "... = (hz ⊕ (hx ⊗ (hy ⊕ y)) ⊕ x ⊗ hy) ⊕ x
⊗ y" by algebra
    finally have z2: "z = (hz ⊕ (hx ⊗ (hy ⊕ y)) ⊕ x ⊗ hy) ⊕ x ⊗ y" .
    from hxI hyI hzI carr have "hz ⊕ (hx ⊗ (hy ⊕ y)) ⊕ x ⊗ hy ∈ I"
```

```
      by (simp add: I_l_closed I_r_closed)
    with z2 show ?thesis
      by (auto simp add: a_r_coset_def r_coset_def)
  qed
  have 2: "∃a∈I +> x. ∃b∈I +> y. z ∈ I +> a ⊗ b" if "z ∈ I +> x ⊗ y"
for z
    using assms a_rcos_self that by blast
  show ?thesis
    unfolding rcoset_mult_def using assms
    by (auto simp: intro!: 1 2)
qed
```

## 27.2 Quotient Ring Definition

```
definition FactRing :: "[('a,'b) ring_scheme, 'a set] ⇒ ('a set) ring"
    (infixl ‹Quot› 65)
  where "FactRing R I =
    (|carrier = a_rcosets_R I, mult = rcoset_mult R I,
      one = (I +>_R 1_R), zero = I, add = set_add R|)"

lemmas FactRing_simps = FactRing_def A_RCOSETS_defs a_r_coset_def[symmetric]
```

## 27.3 Factorization over General Ideals

The quotient is a ring

```
lemma (in ideal) quotient_is_ring: "ring (R Quot I)"
proof (rule ringI)
  show "abelian_group (R Quot I)"
    by (rule comm_group_abelian_groupI)
      (simp add: FactRing_def a_factorgroup_is_comm_group[unfolded A_FactGroup_def'])
  show "Group.monoid (R Quot I)"
    by (rule monoidI)
      (auto simp add: FactRing_simps rcoset_mult_add m_assoc)
qed (auto simp: FactRing_simps rcoset_mult_add a_rcos_sum l_distr r_distr)
```

This is a ring homomorphism

```
lemma (in ideal) rcos_ring_hom: "((+>) I) ∈ ring_hom R (R Quot I)"
  by (simp add: ring_hom_memI FactRing_def a_rcosetsI[OF a_subset] rcoset_mult_add
a_rcos_sum)

lemma (in ideal) rcos_ring_hom_ring: "ring_hom_ring R (R Quot I) ((+>)
I)"
  by (simp add: local.ring_axioms quotient_is_ring rcos_ring_hom ring_hom_ringI2)
```

The quotient of a cring is also commutative

```
lemma (in ideal) quotient_is_cring:
  assumes "cring R"
  shows "cring (R Quot I)"
```

```
proof -
  interpret cring R by fact
  show ?thesis
    apply (intro cring.intro comm_monoid.intro comm_monoid_axioms.intro
quotient_is_ring)
     apply (rule ring.axioms[OF quotient_is_ring])
    apply (auto simp add: FactRing_simps rcoset_mult_add m_comm)
    done
qed
```

Cosets as a ring homomorphism on crings

```
lemma (in ideal) rcos_ring_hom_cring:
  assumes "cring R"
  shows "ring_hom_cring R (R Quot I) ((+>) I)"
proof -
  interpret cring R by fact
  show ?thesis
    apply (rule ring_hom_cringI)
      apply (rule rcos_ring_hom_ring)
     apply (rule is_cring)
    apply (rule quotient_is_cring)
    apply (rule is_cring)
    done
qed
```

## 27.4 Factorization over Prime Ideals

The quotient ring generated by a prime ideal is a domain

```
lemma (in primeideal) quotient_is_domain: "domain (R Quot I)"
proof -
  have 1: "I +> 1 = I ⟹ False"
    using I_notcarr a_rcos_self one_imp_carrier by blast
  have 2: "I +> x = I"
    if  carr: "x ∈ carrier R" "y ∈ carrier R"
    and a: "I +> x ⊗ y = I"
    and b: "I +> y ≠ I" for x y
    by (metis I_prime a a_rcos_const a_rcos_self b m_closed that)
  show ?thesis
    apply (intro domain.intro quotient_is_cring is_cring domain_axioms.intro)
     apply (metis "1" FactRing_def monoid.simps(2) ring.simps(1))
    apply (simp add: FactRing_simps)
    apply (metis "2" rcoset_mult_add)
    done
qed
```

Generating right cosets of a prime ideal is a homomorphism on commutative rings

```
lemma (in primeideal) rcos_ring_hom_cring: "ring_hom_cring R (R Quot
I) ((+>) I)"
```

```
  by (rule rcos_ring_hom_cring) (rule is_cring)
```

## 27.5 Factorization over Maximal Ideals

In a commutative ring, the quotient ring over a maximal ideal is a field. The proof follows "W. Adkins, S. Weintraub: Algebra – An Approach via Module Theory"

```
proposition (in maximalideal) quotient_is_field:
  assumes "cring R"
  shows "field (R Quot I)"
proof -
  interpret cring R by fact
  have 1: "𝟬R Quot I ≠ 𝟭R Quot I"  — Quotient is not empty
  proof
    assume "𝟬R Quot I = 𝟭R Quot I"
    then have II1: "I = I +> 𝟭" by (simp add: FactRing_def)
    then have "I = carrier R"
      using a_rcos_self one_imp_carrier by blast
    with I_notcarr show False by simp
  qed
  have 2: "∃y∈carrier R. I +> a ⊗ y = I +> 𝟭" if IanI: "I +> a ≠ I" and
acarr: "a ∈ carrier R" for a
    — Existence of Inverse
  proof -
    — Helper ideal J
    define J :: "'a set" where "J = (carrier R #> a) <+> I"
    have idealJ: "ideal J R"
      using J_def acarr add_ideals cgenideal_eq_rcos cgenideal_ideal is_ideal
by auto
    have IinJ: "I ⊆ J"
    proof (clarsimp simp: J_def r_coset_def set_add_defs)
      fix x
      assume xI: "x ∈ I"
      have "x = 𝟬 ⊗ a ⊕ x"
        by (simp add: acarr xI)
      with xI show "∃xa∈carrier R. ∃k∈I. x = xa ⊗ a ⊕ k" by fast
    qed
    have JnI: "J ≠ I"
    proof -
      have "a ∉ I"
        using IanI a_rcos_const by blast
      moreover have "a ∈ J"
      proof (simp add: J_def r_coset_def set_add_defs)
        from acarr
        have "a = 𝟭 ⊗ a ⊕ 𝟬" by algebra
        with one_closed and additive_subgroup.zero_closed[OF is_additive_subgroup]
        show "∃x∈carrier R. ∃k∈I. a = x ⊗ a ⊕ k" by fast
      qed
      ultimately show ?thesis by blast
```

```
    qed
    then have Jcarr: "J = carrier R"
      using I_maximal IinJ additive_subgroup.a_subset idealJ ideal_def
by blast

    — Calculating an inverse for a
    from one_closed[folded Jcarr]
    obtain r i where rcarr: "r ∈ carrier R"
      and iI: "i ∈ I" and one: "1 = r ⊗ a ⊕ i"
      by (auto simp add: J_def r_coset_def set_add_defs)

    from one and rcarr and acarr and iI[THEN a_Hcarr]
    have rai1: "a ⊗ r = ⊖i ⊕ 1" by algebra

    — Lifting to cosets
    from iI have "⊖i ⊕ 1 ∈ I +> 1"
      by (intro a_rcosI, simp, intro a_subset, simp)
    with rai1 have "a ⊗ r ∈ I +> 1" by simp
    then have "I +> 1 = I +> a ⊗ r"
      by (rule a_repr_independence, simp) (rule a_subgroup)

    from rcarr and this[symmetric]
    show "∃r∈carrier R. I +> a ⊗ r = I +> 1" by fast
  qed
  show ?thesis
    apply (intro cring.cring_fieldI2 quotient_is_cring is_cring 1)
     apply (clarsimp simp add: FactRing_simps rcoset_mult_add 2)
    done
qed


lemma (in ring_hom_ring) trivial_hom_iff:
  "(h ` (carrier R) = { 0_S }) = (a_kernel R S h = carrier R)"
  using group_hom.trivial_hom_iff[OF a_group_hom] by (simp add: a_kernel_def)

lemma (in ring_hom_ring) trivial_ker_imp_inj:
  assumes "a_kernel R S h = { 0 }"
  shows "inj_on h (carrier R)"
  using group_hom.trivial_ker_imp_inj[OF a_group_hom] assms a_kernel_def[of
R S h] by simp


lemma (in ring_hom_ring) inj_iff_trivial_ker:
  shows "inj_on h (carrier R) ⟷ a_kernel R S h = { 0 }"
  using group_hom.inj_iff_trivial_ker[OF a_group_hom] a_kernel_def[of
R S h] by simp


corollary ring_hom_in_hom:
```

```
  assumes "h ∈ ring_hom R S" shows "h ∈ hom R S" and "h ∈ hom (add_monoid
R) (add_monoid S)"
  using assms unfolding ring_hom_def hom_def by auto


corollary set_add_hom:
  assumes "h ∈ ring_hom R S" "I ⊆ carrier R" and "J ⊆ carrier R"
  shows "h ` (I <+>R J) = h ` I <+>S h ` J"
  using set_mult_hom[OF ring_hom_in_hom(2)[OF assms(1)]] assms(2-3)
  unfolding a_kernel_def[of R S h] set_add_def by simp


corollary a_coset_hom:
  assumes "h ∈ ring_hom R S" "I ⊆ carrier R" "a ∈ carrier R"
  shows "h ` (a <+R I) = h a <+S (h ` I)" and "h ` (I +>R a) = (h ` I)
+>S h a"
  using assms coset_hom[OF ring_hom_in_hom(2)[OF assms(1)], of I a]
  unfolding a_l_coset_def l_coset_eq_set_mult
            a_r_coset_def r_coset_eq_set_mult
  by simp_all


corollary (in ring_hom_ring) set_add_ker_hom:
  assumes "I ⊆ carrier R"
  shows "h ` (I <+> (a_kernel R S h)) = h ` I" and "h ` ((a_kernel R
S h) <+> I) = h ` I"
  using group_hom.set_mult_ker_hom[OF a_group_hom] assms
  unfolding a_kernel_def[of R S h] set_add_def by simp+

lemma (in ring_hom_ring) non_trivial_field_hom_imp_inj:
  assumes R: "field R"
    and h: "h ` (carrier R) ≠ { 0S }"
  shows "inj_on h (carrier R)"
proof -
  from h have "a_kernel R S h ≠ carrier R"
    using trivial_hom_iff by linarith
  hence "a_kernel R S h = { 0 }"
    using field.all_ideals[OF R] kernel_is_ideal by blast
  thus "inj_on h (carrier R)"
    using trivial_ker_imp_inj by blast
qed

lemma "field R ⟹ cring R"
  using fieldE(1) by blast

lemma non_trivial_field_hom_is_inj:
  assumes "h ∈ ring_hom R S" and "field R" and "field S"
  shows "inj_on h (carrier R)"
proof -
```

```
  interpret ring_hom_cring R S h
    using assms(1) ring_hom_cring.intro[OF assms(2-3)[THEN fieldE(1)]]
    unfolding symmetric[OF ring_hom_cring_axioms_def] by simp

  have "h 𝟭R = 𝟭S" and "𝟭S ≠ 𝟬S"
    using domain.one_not_zero[OF field.axioms(1)[OF assms(3)]] by auto
  hence "h ` (carrier R) ≠ { 𝟬S }"
    using ring.kernel_zero ring.trivial_hom_iff by fastforce
  thus ?thesis
    using ring.non_trivial_field_hom_imp_inj[OF assms(2)] by simp
qed

lemma (in ring_hom_ring) img_is_add_subgroup:
  assumes "subgroup H (add_monoid R)"
  shows "subgroup (h ` H) (add_monoid S)"
proof -
  have "group ((add_monoid R) ⦇ carrier := H ⦈)"
    using assms R.add.subgroup_imp_group by blast
  moreover have "H ⊆ carrier R" by (simp add: R.add.subgroupE(1) assms)
  hence "h ∈ hom ((add_monoid R) ⦇ carrier := H ⦈) (add_monoid S)"
    unfolding hom_def by (auto simp add: subsetD)
  ultimately have "subgroup (h ` carrier ((add_monoid R) ⦇ carrier :=
H ⦈)) (add_monoid S)"
    using group_hom.img_is_subgroup[of "(add_monoid R) ⦇ carrier := H
⦈" "add_monoid S" h]
    using a_group_hom group_hom_axioms.intro group_hom_def by blast
  thus "subgroup (h ` H) (add_monoid S)" by simp
qed

lemma (in ring) ring_ideal_imp_quot_ideal:
  assumes "ideal I R"
    and A: "ideal J R"
  shows "ideal ((+>) I ` J) (R Quot I)"
proof (rule idealI)
  show "ring (R Quot I)"
    by (simp add: assms(1) ideal.quotient_is_ring)
next
  have "subgroup J (add_monoid R)"
    by (simp add: additive_subgroup.a_subgroup A ideal.axioms(1))
  moreover have "((+>) I) ∈ ring_hom R (R Quot I)"
    by (simp add: assms(1) ideal.rcos_ring_hom)
  ultimately show "subgroup ((+>) I ` J) (add_monoid (R Quot I))"
    using assms(1) ideal.rcos_ring_hom_ring ring_hom_ring.img_is_add_subgroup
by blast
next
  fix a x assume "a ∈ (+>) I ` J" "x ∈ carrier (R Quot I)"
  then obtain i j where i: "i ∈ carrier R" "x = I +> i"
    and j: "j ∈ J" "a = I +> j"
    unfolding FactRing_def using A_RCOSETS_def'[of R I] by auto
```

```
  hence "a ⊗R Quot I x = [mod I:] (I +> j) ⨂ (I +> i)"
    unfolding FactRing_def by simp
  hence "a ⊗R Quot I x = I +> (j ⊗ i)"
    using ideal.rcoset_mult_add[OF assms(1), of j i] i(1) j(1) A ideal.Icarr
by force
  thus "a ⊗R Quot I x ∈ (+>) I ‘ J"
    using A i(1) j(1) by (simp add: ideal.I_r_closed)

  have "x ⊗R Quot I a = [mod I:] (I +> i) ⨂ (I +> j)"
    unfolding FactRing_def i j by simp
  hence "x ⊗R Quot I a = I +> (i ⊗ j)"
    using ideal.rcoset_mult_add[OF assms(1), of i j] i(1) j(1) A ideal.Icarr
by force
  thus "x ⊗R Quot I a ∈ (+>) I ‘ J"
    using A i(1) j(1) by (simp add: ideal.I_l_closed)
qed

lemma (in ring_hom_ring) ideal_vimage:
  assumes "ideal I S"
  shows "ideal { r ∈ carrier R. h r ∈ I } R"
proof
  show "{ r ∈ carrier R. h r ∈ I } ⊆ carrier (add_monoid R)" by auto
  show "1add_monoid R ∈ { r ∈ carrier R. h r ∈ I }"
    by (simp add: additive_subgroup.zero_closed assms ideal.axioms(1))
next
  fix a b
  assume "a ∈ { r ∈ carrier R. h r ∈ I }"
     and "b ∈ { r ∈ carrier R. h r ∈ I }"
  hence a: "a ∈ carrier R" "h a ∈ I"
    and b: "b ∈ carrier R" "h b ∈ I" by auto
  hence "h (a ⊕ b) = (h a) ⊕S (h b)" using hom_add by blast
  moreover have "(h a) ⊕S (h b) ∈ I" using a b assms
    by (simp add: additive_subgroup.a_closed ideal.axioms(1))
  ultimately show "a ⊗add_monoid R b ∈ { r ∈ carrier R. h r ∈ I }"
    using a(1) b (1) by auto

  have "h (⊖ a) = ⊖S (h a)" by (simp add: a)
  moreover have "⊖S (h a) ∈ I"
    by (simp add: a(2) additive_subgroup.a_inv_closed assms ideal.axioms(1))
  ultimately show "invadd_monoid R a ∈ { r ∈ carrier R. h r ∈ I }"
    using a by (simp add: a_inv_def)
next
  fix a r
  assume "a ∈ { r ∈ carrier R. h r ∈ I }" and r: "r ∈ carrier R"
  hence a: "a ∈ carrier R" "h a ∈ I" by auto

  have "h a ⊗S h r ∈ I"
    using assms a r by (simp add: ideal.I_r_closed)
  thus "a ⊗ r ∈ { r ∈ carrier R. h r ∈ I }" by (simp add: a(1) r)
```

```
  have "h r ⊗S h a ∈ I"
    using assms a r by (simp add: ideal.I_l_closed)
  thus "r ⊗ a ∈ { r ∈ carrier R. h r ∈ I }" by (simp add: a(1) r)
qed

lemma (in ring) canonical_proj_vimage_in_carrier:
  assumes "ideal I R"
    and A: "J ⊆ carrier (R Quot I)"
  shows "⋃ J ⊆ carrier R"
proof
  fix j assume j: "j ∈ ⋃ J"
  then obtain j' where j': "j' ∈ J" "j ∈ j'"
    by blast
  then obtain r where r: "r ∈ carrier R" "j' = I +> r"
    using A j' unfolding FactRing_def using A_RCOSETS_def'[of R I] by
auto
  thus "j ∈ carrier R"
    using j' assms
    by (meson a_r_coset_subset_G additive_subgroup.a_subset contra_subsetD
ideal.axioms(1))
qed

lemma (in ring) canonical_proj_vimage_mem_iff:
  assumes "ideal I R" "J ⊆ carrier (R Quot I)"
    and a: "a ∈ carrier R"
  shows "(a ∈ ⋃ J) = (I +> a ∈ J)"
proof
  assume "a ∈ ⋃ J"
  then obtain j where j: "j ∈ J" "a ∈ j" by blast
  then obtain r where r: "r ∈ carrier R" "j = I +> r"
    using assms j unfolding FactRing_def using A_RCOSETS_def'[of R I]
by auto
  hence "I +> r = I +> a"
    using add.repr_independence[of a I r] j r
    by (metis a_r_coset_def additive_subgroup.a_subgroup assms(1) ideal.axioms(1))
  thus "I +> a ∈ J" using r j by simp
next
  assume "I +> a ∈ J"
  hence "𝟬 ⊕ a ∈ I +> a"
    using additive_subgroup.zero_closed[OF ideal.axioms(1)[OF assms(1)]]
          a_r_coset_def'[of R I a] by blast
  thus "a ∈ ⋃ J" using a ‹I +> a ∈ J› by auto
qed

corollary (in ring) quot_ideal_imp_ring_ideal:
  assumes "ideal I R"
  shows "ideal J (R Quot I) ⟹ ideal (⋃ J) R"
proof -
```

```
  assume A: "ideal J (R Quot I)"
  have "⋃ J = { r ∈ carrier R. I +> r ∈ J }"
    using canonical_proj_vimage_in_carrier[OF assms, of J]
          canonical_proj_vimage_mem_iff[OF assms, of J]
          additive_subgroup.a_subset[OF ideal.axioms(1)[OF A]] by blast
  thus "ideal (⋃ J) R"
    using ring_hom_ring.ideal_vimage[OF ideal.rcos_ring_hom_ring[OF assms]
A] by simp
qed

lemma (in ring) ideal_incl_iff:
  assumes "ideal I R" "ideal J R"
  shows "(I ⊆ J) = (J = (⋃ j ∈ J. I +> j))"
proof
  assume "J = (⋃ j ∈ J. I +> j)" hence "I +> 0 ⊆ J"
    using additive_subgroup.zero_closed[OF ideal.axioms(1)[OF assms(2)]]
by blast
  thus "I ⊆ J" using additive_subgroup.a_subset[OF ideal.axioms(1)[OF
assms(1)]] by simp
next
  assume A: "I ⊆ J" show "J = (⋃j∈J. I +> j)"
  proof
    show "J ⊆ (⋃ j ∈ J. I +> j)"
    proof
      fix j assume j: "j ∈ J"
      have "0 ∈ I" by (simp add: additive_subgroup.zero_closed assms(1)
ideal.axioms(1))
      hence "0 ⊕ j ∈ I +> j"
        using a_r_coset_def'[of R I j] by blast
      thus "j ∈ (⋃j∈J. I +> j)"
        using assms(2) j additive_subgroup.a_Hcarr ideal.axioms(1) by
fastforce
    qed
    show "(⋃ j ∈ J. I +> j) ⊆ J"
    proof
      fix x assume "x ∈ (⋃ j ∈ J. I +> j)"
      then obtain j where j: "j ∈ J" "x ∈ I +> j" by blast
      then obtain i where i: "i ∈ I" "x = i ⊕ j"
        using a_r_coset_def'[of R I j] by blast
      thus "x ∈ J"
        using assms(2) j A additive_subgroup.a_closed[OF ideal.axioms(1)[OF
assms(2)]] by blast
    qed
  qed
qed

theorem (in ring) quot_ideal_correspondence:
  assumes "ideal I R"
  shows "bij_betw (λJ. (+>) I ‘ J) { J. ideal J R ∧ I ⊆ J } { J . ideal
```

```
J (R Quot I) }"
proof (rule bij_betw_byWitness[where ?f' = "λX. ⋃ X"])
  show "∀J ∈ { J. ideal J R ∧ I ⊆ J }. (λX. ⋃ X) ((+>) I ` J) = J"
    using assms ideal_incl_iff by blast
  show "(λJ. (+>) I ` J) ` { J. ideal J R ∧ I ⊆ J } ⊆ { J. ideal J (R
Quot I) }"
    using assms ring_ideal_imp_quot_ideal by auto
  show "(λX. ⋃ X) ` { J. ideal J (R Quot I) } ⊆ { J. ideal J R ∧ I ⊆
J }"
  proof
    fix J assume "J ∈ ((λX. ⋃ X) ` { J. ideal J (R Quot I) })"
    then obtain J' where J': "ideal J' (R Quot I)" "J = ⋃ J'" by blast
    hence "ideal J R"
      using assms quot_ideal_imp_ring_ideal by auto
    moreover have "I ∈ J'"
      using additive_subgroup.zero_closed[OF ideal.axioms(1)[OF J'(1)]]
unfolding FactRing_def by simp
    ultimately show "J ∈ { J. ideal J R ∧ I ⊆ J }" using J'(2) by auto
  qed
  show "∀J' ∈ { J. ideal J (R Quot I) }. ((+>) I ` (⋃ J')) = J'"
  proof
    fix J' assume "J' ∈ { J. ideal J (R Quot I) }"
    hence subset: "J' ⊆ carrier (R Quot I) ∧ ideal J' (R Quot I)"
      using additive_subgroup.a_subset ideal_def by blast
    hence "((+>) I ` (⋃ J')) ⊆ J'"
      using canonical_proj_vimage_in_carrier canonical_proj_vimage_mem_iff
      by (meson assms contra_subsetD image_subsetI)
    moreover have "J' ⊆ ((+>) I ` (⋃ J'))"
    proof
      fix x assume "x ∈ J'"
      then obtain r where r: "r ∈ carrier R" "x = I +> r"
        using subset unfolding FactRing_def A_RCOSETS_def'[of R I] by
auto
      hence "r ∈ (⋃ J')"
        using ‹x ∈ J'› assms canonical_proj_vimage_mem_iff subset by
blast
      thus "x ∈ ((+>) I ` (⋃ J'))" using r(2) by blast
    qed
    ultimately show "((+>) I ` (⋃ J')) = J'" by blast
  qed
qed

lemma (in cring) quot_domain_imp_primeideal:
  assumes "ideal P R"
    and A: "domain (R Quot P)"
  shows "primeideal P R"
proof -
  show "primeideal P R"
  proof (rule primeidealI)
```

```
    show "ideal P R" using assms(1) .
    show "cring R" using is_cring .
  next
    show "carrier R ≠ P"
    proof (rule ccontr)
      assume "¬ carrier R ≠ P" hence "carrier R = P" by simp
      hence "⋀I. I ∈ carrier (R Quot P) ⟹ I = P"
        unfolding FactRing_def A_RCOSETS_def' apply simp
        using a_coset_join2 additive_subgroup.a_subgroup assms ideal.axioms(1)
by blast
      hence "𝟭(R Quot P) = 𝟬(R Quot P)"
        by (metis assms ideal.quotient_is_ring ring.ring_simprules(2)
ring.ring_simprules(6))
      thus False using domain.one_not_zero[OF A] by simp
    qed
  next
    fix a b assume a: "a ∈ carrier R" and b: "b ∈ carrier R" and ab:
"a ⊗ b ∈ P"
    hence "P +> (a ⊗ b) = 𝟬(R Quot P)" unfolding FactRing_def
      by (simp add: a_coset_join2 additive_subgroup.a_subgroup assms ideal.axioms(1))
    moreover have "(P +> a) ⊗(R Quot P) (P +> b) = P +> (a ⊗ b)" un-
folding FactRing_def
      using a b by (simp add: assms ideal.rcoset_mult_add)
    moreover have "P +> a ∈ carrier (R Quot P) ∧ P +> b ∈ carrier (R
Quot P)"
      by (simp add: a b FactRing_def a_rcosetsI additive_subgroup.a_subset
assms ideal.axioms(1))
    ultimately have "P +> a = 𝟬(R Quot P) ∨ P +> b = 𝟬(R Quot P)"
      using domain.integral[OF A, of "P +> a" "P +> b"] by auto
    thus "a ∈ P ∨ b ∈ P" unfolding FactRing_def apply simp
      using a b assms a_coset_join1 additive_subgroup.a_subgroup ideal.axioms(1)
by blast
  qed
qed

lemma (in cring) quot_domain_iff_primeideal:
  assumes "ideal P R"
  shows "domain (R Quot P) = primeideal P R"
  using quot_domain_imp_primeideal[OF assms] primeideal.quotient_is_domain[of
P R] by auto
```

## 27.6 Isomorphism

```
definition
  ring_iso :: "_ ⇒ _ ⇒ ('a ⇒ 'b) set"
  where "ring_iso R S = { h. h ∈ ring_hom R S ∧ bij_betw h (carrier R)
(carrier S) }"

definition
```

```
  is_ring_iso :: "_ ⇒ _ ⇒ bool" (infixr ‹≃› 60)
  where "R ≃ S = (ring_iso R S ≠ {})"

definition
  morphic_prop :: "_ ⇒ ('a ⇒ bool) ⇒ bool"
  where "morphic_prop R P =
           ((P 𝟭R) ∧
            (∀r ∈ carrier R. P r) ∧
            (∀r1 ∈ carrier R. ∀r2 ∈ carrier R. P (r1 ⊗R r2)) ∧
            (∀r1 ∈ carrier R. ∀r2 ∈ carrier R. P (r1 ⊕R r2)))"

lemma ring_iso_memI:
  fixes R (structure) and S (structure)
  assumes "⋀x. x ∈ carrier R ⟹ h x ∈ carrier S"
      and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ h (x ⊗ y) = h
x ⊗S h y"
      and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ h (x ⊕ y) = h
x ⊕S h y"
      and "h 𝟭 = 𝟭S"
      and "bij_betw h (carrier R) (carrier S)"
  shows "h ∈ ring_iso R S"
  by (auto simp add: ring_hom_memI assms ring_iso_def)

lemma ring_iso_memE:
  fixes R (structure) and S (structure)
  assumes "h ∈ ring_iso R S"
  shows "⋀x. x ∈ carrier R ⟹ h x ∈ carrier S"
   and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ h (x ⊗ y) = h x ⊗S
h y"
   and "⋀x y. ⟦ x ∈ carrier R; y ∈ carrier R ⟧ ⟹ h (x ⊕ y) = h x ⊕S
h y"
   and "h 𝟭 = 𝟭S"
   and "bij_betw h (carrier R) (carrier S)"
  using assms unfolding ring_iso_def ring_hom_def by auto

lemma morphic_propI:
  fixes R (structure)
  assumes "P 𝟭"
    and "⋀r. r ∈ carrier R ⟹ P r"
    and "⋀r1 r2. ⟦ r1 ∈ carrier R; r2 ∈ carrier R ⟧ ⟹ P (r1 ⊗ r2)"
    and "⋀r1 r2. ⟦ r1 ∈ carrier R; r2 ∈ carrier R ⟧ ⟹ P (r1 ⊕ r2)"
  shows "morphic_prop R P"
  unfolding morphic_prop_def using assms by auto

lemma morphic_propE:
  fixes R (structure)
  assumes "morphic_prop R P"
  shows "P 𝟭"
    and "⋀r. r ∈ carrier R ⟹ P r"
```

```
    and "⋀r1 r2. ⟦ r1 ∈ carrier R; r2 ∈ carrier R ⟧ ⟹ P (r1 ⊗ r2)"
    and "⋀r1 r2. ⟦ r1 ∈ carrier R; r2 ∈ carrier R ⟧ ⟹ P (r1 ⊕ r2)"
  using assms unfolding morphic_prop_def by auto


lemma (in ring) ring_hom_restrict:
  assumes "f ∈ ring_hom R S" and "⋀r. r ∈ carrier R ⟹ f r = g r" shows
"g ∈ ring_hom R S"
  using assms(2) ring_hom_memE[OF assms(1)] by (auto intro: ring_hom_memI)


lemma (in ring) ring_iso_restrict:
  assumes "f ∈ ring_iso R S" and "⋀r. r ∈ carrier R ⟹ f r = g r" shows
"g ∈ ring_iso R S"
proof -
  have hom: "g ∈ ring_hom R S"
    using ring_hom_restrict assms unfolding ring_iso_def by auto
  have "bij_betw g (carrier R) (carrier S)"
    using bij_betw_cong[of "carrier R" f g] ring_iso_memE(5)[OF assms(1)]
assms(2) by simp
  thus ?thesis
    using ring_hom_memE[OF hom] by (auto intro!: ring_iso_memI)
qed

lemma ring_iso_morphic_prop:
  assumes "f ∈ ring_iso R S"
    and "morphic_prop R P"
    and "⋀r. P r ⟹ f r = g r"
  shows "g ∈ ring_iso R S"
proof -
  have eq0: "⋀r. r ∈ carrier R ⟹ f r = g r"
   and eq1: "f 𝟭R = g 𝟭R"
   and eq2: "⋀r1 r2. ⟦ r1 ∈ carrier R; r2 ∈ carrier R ⟧ ⟹ f (r1 ⊗R
r2) = g (r1 ⊗R r2)"
   and eq3: "⋀r1 r2. ⟦ r1 ∈ carrier R; r2 ∈ carrier R ⟧ ⟹ f (r1 ⊕R
r2) = g (r1 ⊕R r2)"
    using assms(2-3) unfolding morphic_prop_def by auto
  show ?thesis
    apply (rule ring_iso_memI)
    using assms(1) eq0 ring_iso_memE(1) apply fastforce
    apply (metis assms(1) eq0 eq2 ring_iso_memE(2))
    apply (metis assms(1) eq0 eq3 ring_iso_memE(3))
    using assms(1) eq1 ring_iso_memE(4) apply fastforce
    using assms(1) bij_betw_cong eq0 ring_iso_memE(5) by blast
qed

lemma (in ring) ring_hom_imp_img_ring:
  assumes "h ∈ ring_hom R S"
  shows "ring (S ⦇ carrier := h ` (carrier R), zero := h 𝟬 ⦈)" (is "ring
```

```
?h_img")
proof -
  have "h ∈ hom (add_monoid R) (add_monoid S)"
    using assms unfolding hom_def ring_hom_def by auto
  hence "comm_group ((add_monoid S) ⦇  carrier := h ‘ (carrier R), one
:= h 𝟬 ⦈)"
    using add.hom_imp_img_comm_group[of h "add_monoid S"] by simp
  hence comm_group: "comm_group (add_monoid ?h_img)"
    by (auto intro: comm_monoidI simp add: monoid.defs)

  moreover have "h ∈ hom R S"
    using assms unfolding ring_hom_def hom_def by auto
  hence "monoid (S ⦇  carrier := h ‘ (carrier R), one := h 𝟭 ⦈)"
    using hom_imp_img_monoid[of h S] by simp
  hence monoid: "monoid ?h_img"
    using ring_hom_memE(4)[OF assms] unfolding monoid_def by (simp add:
monoid.defs)
  show ?thesis
  proof (rule ringI, simp_all add: comm_group_abelian_groupI[OF comm_group]
monoid)
    fix x y z assume "x ∈ h ‘ carrier R" "y ∈ h ‘ carrier R" "z ∈ h ‘
carrier R"
    then obtain r1 r2 r3
      where r1: "r1 ∈ carrier R" "x = h r1"
        and r2: "r2 ∈ carrier R" "y = h r2"
        and r3: "r3 ∈ carrier R" "z = h r3" by blast
    hence "(x ⊕S y) ⊗S z = h ((r1 ⊕ r2) ⊗ r3)"
      using ring_hom_memE[OF assms] by auto
    also have " ... = h ((r1 ⊗ r3) ⊕ (r2 ⊗ r3))"
      using l_distr[OF r1(1) r2(1) r3(1)] by simp
    also have " ... = (x ⊗S z) ⊕S (y ⊗S z)"
      using ring_hom_memE[OF assms] r1 r2 r3 by auto
    finally show "(x ⊕S y) ⊗S z = (x ⊗S z) ⊕S (y ⊗S z)" .

    have "z ⊗S (x ⊕S y) = h (r3 ⊗ (r1 ⊕ r2))"
      using ring_hom_memE[OF assms] r1 r2 r3 by auto
    also have " ... =  h ((r3 ⊗ r1) ⊕ (r3 ⊗ r2))"
      using r_distr[OF r1(1) r2(1) r3(1)] by simp
    also have " ... = (z ⊗S x) ⊕S (z ⊗S y)"
      using ring_hom_memE[OF assms] r1 r2 r3 by auto
    finally show "z ⊗S (x ⊕S y) = (z ⊗S x) ⊕S (z ⊗S y)" .
  qed
qed

lemma (in ring) ring_iso_imp_img_ring:
  assumes "h ∈ ring_iso R S"
  shows "ring (S ⦇ zero := h 𝟬 ⦈)"
proof -
  have "ring (S ⦇ carrier := h ‘ (carrier R), zero := h 𝟬 ⦈)"
```

```
    using ring_hom_imp_img_ring[of h S] assms unfolding ring_iso_def by
auto
  moreover have "h ` (carrier R) = carrier S"
    using assms unfolding ring_iso_def bij_betw_def by auto
  ultimately show ?thesis by simp
qed

lemma (in cring) ring_iso_imp_img_cring:
  assumes "h ∈ ring_iso R S"
  shows "cring (S ⦇ zero := h 𝟬 ⦈)" (is "cring ?h_img")
proof -
  note m_comm
  interpret h_img?: ring ?h_img
    using ring_iso_imp_img_ring[OF assms] .
  show ?thesis
  proof (unfold_locales)
    fix x y assume "x ∈ carrier ?h_img" "y ∈ carrier ?h_img"
    then obtain r1 r2
      where r1: "r1 ∈ carrier R" "x = h r1"
        and r2: "r2 ∈ carrier R" "y = h r2"
      using assms image_iff[where ?f = h and ?A = "carrier R"]
      unfolding ring_iso_def bij_betw_def by auto
    have "x ⊗(?h_img) y = h (r1 ⊗ r2)"
      using assms r1 r2 unfolding ring_iso_def ring_hom_def by auto
    also have " ... = h (r2 ⊗ r1)"
      using m_comm[OF r1(1) r2(1)] by simp
    also have " ... = y ⊗(?h_img) x"
      using assms r1 r2 unfolding ring_iso_def ring_hom_def by auto
    finally show "x ⊗(?h_img) y = y ⊗(?h_img) x" .
  qed
qed

lemma (in domain) ring_iso_imp_img_domain:
  assumes "h ∈ ring_iso R S"
  shows "domain (S ⦇ zero := h 𝟬 ⦈)" (is "domain ?h_img")
proof -
  note aux = m_closed integral one_not_zero one_closed zero_closed
  interpret h_img?: cring ?h_img
    using ring_iso_imp_img_cring[OF assms] .
  show ?thesis
  proof (unfold_locales)
    have "𝟭?h_img = 𝟬?h_img ⟹ h 𝟭 = h 𝟬"
      using ring_iso_memE(4)[OF assms] by simp
    moreover have "h 𝟭 ≠ h 𝟬"
      using ring_iso_memE(5)[OF assms] aux(3-4)
      unfolding bij_betw_def inj_on_def by force
    ultimately show "𝟭?h_img ≠ 𝟬?h_img"
      by auto
  next
```

```
    fix a b
    assume A: "a ⊗?h_img b = 0?h_img" "a ∈ carrier ?h_img" "b ∈ carrier
?h_img"
    then obtain r1 r2
      where r1: "r1 ∈ carrier R" "a = h r1"
        and r2: "r2 ∈ carrier R" "b = h r2"
      using assms image_iff[where ?f = h and ?A = "carrier R"]
      unfolding ring_iso_def bij_betw_def by auto
    hence "a ⊗?h_img b = h (r1 ⊗ r2)"
      using assms r1 r2 unfolding ring_iso_def ring_hom_def by auto
    hence "h (r1 ⊗ r2) = h 0"
      using A(1) by simp
    hence "r1 ⊗ r2 = 0"
      using ring_iso_memE(5)[OF assms] aux(1)[OF r1(1) r2(1)] aux(5)
      unfolding bij_betw_def inj_on_def by force
    hence "r1 = 0 ∨ r2 = 0"
      using aux(2)[OF _ r1(1) r2(1)] by simp
    thus "a = 0?h_img ∨ b = 0?h_img"
      unfolding r1 r2 by auto
  qed
qed

lemma (in field) ring_iso_imp_img_field:
  assumes "h ∈ ring_iso R S"
  shows "field (S ⦇ zero := h 0 ⦈)" (is "field ?h_img")
proof -
  interpret h_img?: domain ?h_img
    using ring_iso_imp_img_domain[OF assms] .
  show ?thesis
  proof (unfold_locales, auto simp add: Units_def)
    interpret field R using field_axioms .
    fix a assume a: "a ∈ carrier S" "a ⊗S h 0 = 1S"
    then obtain r where r: "r ∈ carrier R" "a = h r"
      using assms image_iff[where ?f = h and ?A = "carrier R"]
      unfolding ring_iso_def bij_betw_def by auto
    have "a ⊗S h 0 = h (r ⊗ 0)" unfolding r(2)
      using ring_iso_memE(2)[OF assms r(1)] by simp
    hence "h 1 = h 0"
      using ring_iso_memE(4)[OF assms] r(1) a(2) by simp
    thus False
      using ring_iso_memE(5)[OF assms]
      unfolding bij_betw_def inj_on_def by force
  next
    interpret field R using field_axioms .
    fix s assume s: "s ∈ carrier S" "s ≠ h 0"
    then obtain r where r: "r ∈ carrier R" "s = h r"
      using assms image_iff[where ?f = h and ?A = "carrier R"]
      unfolding ring_iso_def bij_betw_def by auto
    hence "r ≠ 0" using s(2) by auto
```

```
    hence inv_r: "inv r ∈ carrier R" "inv r ≠ 0" "r ⊗ inv r = 1" "inv
r ⊗ r = 1"
      using field_Units r(1) by auto
    have "h (inv r) ⊗S h r = h 1" and "h r ⊗S h (inv r) = h 1"
      using ring_iso_memE(2)[OF assms inv_r(1) r(1)] inv_r(3-4)
            ring_iso_memE(2)[OF assms r(1) inv_r(1)] by auto
    thus "∃s' ∈ carrier S. s' ⊗S s = 1S ∧ s ⊗S s' = 1S"
      using ring_iso_memE(1,4)[OF assms] inv_r(1) r(2) by auto
  qed
qed

lemma ring_iso_same_card: "R ≃ S ⟹ card (carrier R) = card (carrier
S)"
  using bij_betw_same_card unfolding is_ring_iso_def ring_iso_def by auto


lemma ring_iso_set_refl: "id ∈ ring_iso R R"
  by (rule ring_iso_memI) (auto)

corollary ring_iso_refl: "R ≃ R"
  using is_ring_iso_def ring_iso_set_refl by auto

lemma ring_iso_set_trans:
  "⟦ f ∈ ring_iso R S; g ∈ ring_iso S Q ⟧ ⟹ (g ∘ f) ∈ ring_iso R Q"
  unfolding ring_iso_def using bij_betw_trans ring_hom_trans by fastforce

corollary ring_iso_trans: "⟦ R ≃ S; S ≃ Q ⟧ ⟹ R ≃ Q"
  using ring_iso_set_trans unfolding is_ring_iso_def by blast

lemma ring_iso_set_sym:
  assumes "ring R" and h: "h ∈ ring_iso R S"
  shows "(inv_into (carrier R) h) ∈ ring_iso S R"
proof -
  have h_hom: "h ∈ ring_hom R S"
    and h_surj: "h ` (carrier R) = (carrier S)"
    and h_inj: "⋀x1 x2. ⟦ x1 ∈ carrier R; x2 ∈ carrier R ⟧ ⟹  h x1
= h x2 ⟹ x1 = x2"
    using h unfolding ring_iso_def bij_betw_def inj_on_def by auto

  have h_inv_bij: "bij_betw (inv_into (carrier R) h) (carrier S) (carrier
R)"
    by (simp add: bij_betw_inv_into h ring_iso_memE(5))

  have "inv_into (carrier R) h ∈ ring_hom S R"
    using ring_iso_memE [OF h] bij_betwE [OF h_inv_bij] ‹ring R›
    by (simp add: bij_betw_imp_inj_on bij_betw_inv_into_right inv_into_f_eq
ring.ring_simprules ring_hom_memI)
  moreover have "bij_betw (inv_into (carrier R) h) (carrier S) (carrier
R)"
```

```
    using h_inv_bij by force
    ultimately show "inv_into (carrier R) h ∈ ring_iso S R"
      by (simp add: ring_iso_def)
qed

corollary ring_iso_sym:
  assumes "ring R"
  shows "R ≃ S ⟹ S ≃ R"
  using assms ring_iso_set_sym unfolding is_ring_iso_def by auto

lemma (in ring_hom_ring) the_elem_simp [simp]:
  assumes x: "x ∈ carrier R"
  shows "the_elem (h ` ((a_kernel R S h) +> x)) = h x"
proof -
  from x have "h x ∈ h ` ((a_kernel R S h) +> x)"
    using homeq_imp_rcos by blast
  thus "the_elem (h ` ((a_kernel R S h) +> x)) = h x"
    by (metis (no_types, lifting) x empty_iff homeq_imp_rcos rcos_imp_homeq
the_elem_image_unique)
qed

lemma (in ring_hom_ring) the_elem_inj:
  assumes "X ∈ carrier (R Quot (a_kernel R S h))"
    and "Y ∈ carrier (R Quot (a_kernel R S h))"
    and Eq: "the_elem (h ` X) = the_elem (h ` Y)"
  shows "X = Y"
proof -
  from assms obtain x y where x: "x ∈ carrier R" "X = (a_kernel R S h)
+> x"
    and y: "y ∈ carrier R" "Y = (a_kernel R S h) +> y"
    unfolding FactRing_def A_RCOSETS_def' by auto
  hence "h x = h y" using Eq by simp
  hence "x ⊖ y ∈ (a_kernel R S h)"
    by (simp add: a_minus_def abelian_subgroup.a_rcos_module_imp
                  abelian_subgroup_a_kernel homeq_imp_rcos x(1) y(1))
  thus "X = Y"
    by (metis R.a_coset_add_inv1 R.minus_eq abelian_subgroup.a_rcos_const
        abelian_subgroup_a_kernel additive_subgroup.a_subset additive_subgroup_a_kernel
x y)
qed

lemma (in ring_hom_ring) quot_mem:
  "X ∈ carrier (R Quot (a_kernel R S h)) ⟹ ∃x ∈ carrier R. X = (a_kernel
R S h) +> x"
  unfolding FactRing_simps by (simp add: a_r_coset_def)

lemma (in ring_hom_ring) the_elem_wf:
  assumes "X ∈ carrier (R Quot (a_kernel R S h))"
  shows "∃y ∈ carrier S. (h ` X) = { y }"
```

```
proof -
  from assms obtain x where x: "x ∈ carrier R" and X: "X = (a_kernel
R S h) +> x"
    using quot_mem by blast
  have "h x' = h x" if "x' ∈ X" for x'
  proof -
    from X that have "x' ∈ (a_kernel R S h) +> x" by simp
    then obtain k where k: "k ∈ a_kernel R S h" "x' = k ⊕ x"
      by (metis R.add.inv_closed R.add.m_assoc R.l_neg R.r_zero
          abelian_subgroup.a_elemrcos_carrier
          abelian_subgroup.a_rcos_module_imp abelian_subgroup_a_kernel
x)
    hence "h x' = h k ⊕S h x"
      by (meson additive_subgroup.a_Hcarr additive_subgroup_a_kernel hom_add
x)
    also have " ... =  h x"
      using k by (auto simp add: x)
    finally show "h x' = h x" .
  qed
  moreover have "h x ∈ h ` X"
    by (simp add: X homeq_imp_rcos x)
  ultimately have "(h ` X) = { h x }"
    by blast
  thus "∃y ∈ carrier S. (h ` X) = { y }" using x by simp
qed

corollary (in ring_hom_ring) the_elem_wf':
  "X ∈ carrier (R Quot (a_kernel R S h)) ⟹ ∃r ∈ carrier R. (h ` X)
= { h r }"
  using the_elem_wf by (metis quot_mem the_elem_eq the_elem_simp)

lemma (in ring_hom_ring) the_elem_hom:
  "(λX. the_elem (h ` X)) ∈ ring_hom (R Quot (a_kernel R S h)) S"
proof (rule ring_hom_memI)
  show "⋀x. x ∈ carrier (R Quot a_kernel R S h) ⟹ the_elem (h ` x)
∈ carrier S"
    using the_elem_wf by fastforce

  show "the_elem (h ` 1R Quot a_kernel R S h) = 1S"
    unfolding FactRing_def  using the_elem_simp[of "1R"] by simp

  fix X Y
  assume "X ∈ carrier (R Quot a_kernel R S h)"
     and "Y ∈ carrier (R Quot a_kernel R S h)"
  then obtain x y where x: "x ∈ carrier R" "X = (a_kernel R S h) +> x"
                    and y: "y ∈ carrier R" "Y = (a_kernel R S h) +> y"
    using quot_mem by blast

  have "X ⊗R Quot a_kernel R S h Y = (a_kernel R S h) +> (x ⊗ y)"
```

```
    by (simp add: FactRing_def ideal.rcoset_mult_add kernel_is_ideal x
y)
  thus "the_elem (h ‘ (X ⊗R Quot a_kernel R S h Y)) = the_elem (h ‘ X) ⊗S
the_elem (h ‘ Y)"
    by (simp add: x y)

  have "X ⊕R Quot a_kernel R S h Y = (a_kernel R S h) +> (x ⊕ y)"
    using ideal.rcos_ring_hom kernel_is_ideal ring_hom_add x y by fastforce
  thus "the_elem (h ‘ (X ⊕R Quot a_kernel R S h Y)) = the_elem (h ‘ X) ⊕S
the_elem (h ‘ Y)"
    by (simp add: x y)
qed

lemma (in ring_hom_ring) the_elem_surj:
  "(λX. (the_elem (h ‘ X))) ‘ carrier (R Quot (a_kernel R S h)) = (h ‘
(carrier R))"
proof
  show "(λX. the_elem (h ‘ X)) ‘ carrier (R Quot a_kernel R S h) ⊆ h
‘ carrier R"
    using the_elem_wf' by fastforce
  show "h ‘ carrier R ⊆ (λX. the_elem (h ‘ X)) ‘ carrier (R Quot a_kernel
R S h)"
  proof
    fix y assume "y ∈ h ‘ carrier R"
    then obtain x where x: "x ∈ carrier R" "h x = y"
      by (metis image_iff)
    hence "the_elem (h ‘ ((a_kernel R S h) +> x)) = y" by simp
    moreover have "(a_kernel R S h) +> x ∈ carrier (R Quot (a_kernel
R S h))"
     unfolding FactRing_simps by (auto simp add: x a_r_coset_def)
    ultimately show "y ∈ (λX. (the_elem (h ‘ X))) ‘ carrier (R Quot (a_kernel
R S h))" by blast
  qed
qed

proposition (in ring_hom_ring) FactRing_iso_set_aux:
  "(λX. the_elem (h ‘ X)) ∈ ring_iso (R Quot (a_kernel R S h)) (S ⦇ carrier
:= h ‘ (carrier R) ⦈)"
proof -
  have *: "bij_betw (λX. the_elem (h ‘ X)) (carrier (R Quot a_kernel R
S h)) (h ‘ (carrier R))"
    unfolding bij_betw_def inj_on_def using the_elem_surj the_elem_inj
by simp
  have "(λX. the_elem (h ‘ X)): carrier (R Quot (a_kernel R S h)) → h
‘ (carrier R)"
    using the_elem_wf' by fastforce
  hence "(λX. the_elem (h ‘ X)) ∈ ring_hom (R Quot (a_kernel R S h))
(S ⦇ carrier := h ‘ (carrier R) ⦈)"
    using the_elem_hom the_elem_wf' unfolding ring_hom_def by simp
```

```
  with * show ?thesis unfolding ring_iso_def using the_elem_hom by simp
qed

theorem (in ring_hom_ring) FactRing_iso_set:
  assumes "h ‘ carrier R = carrier S"
  shows "(λX. the_elem (h ‘ X)) ∈ ring_iso (R Quot (a_kernel R S h))
S"
  using FactRing_iso_set_aux assms by auto

corollary (in ring_hom_ring) FactRing_iso:
  assumes "h ‘ carrier R = carrier S"
  shows "R Quot (a_kernel R S h) ≃ S"
  using FactRing_iso_set assms is_ring_iso_def by auto

corollary (in ring) FactRing_zeroideal:
  shows "R Quot { 0 } ≃ R" and "R ≃ R Quot { 0 }"
proof -
  have "ring_hom_ring R R id"
    using ring_axioms by (auto intro: ring_hom_ringI)
  moreover have "a_kernel R R id = { 0 }"
    unfolding a_kernel_def' by auto
  ultimately show "R Quot { 0 } ≃ R" and "R ≃ R Quot { 0 }"
    using ring_hom_ring.FactRing_iso[of R R id]
          ring_iso_sym[OF ideal.quotient_is_ring[OF zeroideal], of R]
by auto
qed

lemma (in ring_hom_ring) img_is_ring: "ring (S ⦇ carrier := h ‘ (carrier
R) ⦈)"
proof -
  let ?the_elem = "λX. the_elem (h ‘ X)"
  have FactRing_is_ring: "ring (R Quot (a_kernel R S h))"
    by (simp add: ideal.quotient_is_ring kernel_is_ideal)
  have "ring ((S ⦇ carrier := ?the_elem ‘ (carrier (R Quot (a_kernel R
S h))) ⦈)
                ⦇    zero := ?the_elem 0(R Quot (a_kernel R S h)) ⦈)"
    using ring.ring_iso_imp_img_ring[OF FactRing_is_ring, of ?the_elem
          "S ⦇ carrier := ?the_elem ‘ (carrier (R Quot (a_kernel R S h)))
⦈"]
          FactRing_iso_set_aux the_elem_surj by auto

  moreover
  have "0 ∈ (a_kernel R S h)"
    using a_kernel_def'[of R S h] by auto
  hence "1 ∈ (a_kernel R S h) +> 1"
    using a_r_coset_def'[of R "a_kernel R S h" 1] by force
  hence "1S ∈ (h ‘ ((a_kernel R S h) +> 1))"
    using hom_one by force
  hence "?the_elem 1(R Quot (a_kernel R S h)) = 1S"
```

```
    using the_elem_wf[of "(a_kernel R S h) +> 1"] by (simp add: FactRing_def)

  moreover
  have "0S ∈ (h ‘ (a_kernel R S h))"
    using a_kernel_def'[of R S h] hom_zero by force
  hence "0S ∈ (h ‘ 0(R Quot (a_kernel R S h)))"
    by (simp add: FactRing_def)
  hence "?the_elem 0(R Quot (a_kernel R S h)) = 0S"
    using the_elem_wf[OF ring.ring_simprules(2)[OF FactRing_is_ring]]
    by (metis singletonD the_elem_eq)

  ultimately
  have "ring ((S ⦇ carrier := h ‘ (carrier R) ⦈) ⦇ one := 1S, zero :=
0S ⦈)"
    using the_elem_surj by simp
  thus ?thesis
    by auto
qed

lemma (in ring_hom_ring) img_is_cring:
  assumes "cring S"
  shows "cring (S ⦇ carrier := h ‘ (carrier R) ⦈)"
proof -
  interpret ring "S ⦇ carrier := h ‘ (carrier R) ⦈"
    using img_is_ring .
  show ?thesis
    by unfold_locales (use assms in ‹auto simp: cring_def comm_monoid_def
comm_monoid_axioms_def›)
qed

lemma (in ring_hom_ring) img_is_domain:
  assumes "domain S"
  shows "domain (S ⦇ carrier := h ‘ (carrier R) ⦈)"
proof -
  interpret cring "S ⦇ carrier := h ‘ (carrier R) ⦈"
    using img_is_cring assms unfolding domain_def by simp
  show ?thesis
    apply unfold_locales
    using assms unfolding domain_def domain_axioms_def apply auto
    using hom_closed by blast
qed

proposition (in ring_hom_ring) primeideal_vimage:
  assumes R: "cring R"
    and A: "primeideal P S"
  shows "primeideal { r ∈ carrier R. h r ∈ P } R"
proof -
  from A have is_ideal: "ideal P S" unfolding primeideal_def by simp
  have "ring_hom_ring R (S Quot P) (((+>S) P) ∘ h)" (is "ring_hom_ring
```

```
?A ?B ?h")
    using ring_hom_trans[OF homh, of "(+>S) P" "S Quot P"]
          ideal.rcos_ring_hom_ring[OF is_ideal] R
    unfolding ring_hom_ring_def ring_hom_ring_axioms_def cring_def by
simp
  then interpret hom: ring_hom_ring R "S Quot P" "((+>S) P) ∘ h" by simp

  have "inj_on (λX. the_elem (?h ‘ X)) (carrier (R Quot (a_kernel R (S
Quot P) ?h)))"
    using hom.the_elem_inj unfolding inj_on_def by simp
  moreover
  have "ideal (a_kernel R (S Quot P) ?h) R"
    using hom.kernel_is_ideal by auto
  have hom’: "ring_hom_ring (R Quot (a_kernel R (S Quot P) ?h)) (S Quot
P) (λX. the_elem (?h ‘ X))"
    using hom.the_elem_hom hom.kernel_is_ideal
    by (meson hom.ring_hom_ring_axioms ideal.rcos_ring_hom_ring ring_hom_ring_axioms_def
ring_hom_ring_def)

  ultimately
  have "primeideal (a_kernel R (S Quot P) ?h) R"
    using ring_hom_ring.inj_on_domain[OF hom’] primeideal.quotient_is_domain[OF
A]
          cring.quot_domain_imp_primeideal[OF R hom.kernel_is_ideal] by
simp

  moreover have "a_kernel R (S Quot P) ?h = { r ∈ carrier R. h r ∈ P
}"
  proof
    show "a_kernel R (S Quot P) ?h ⊆ { r ∈ carrier R. h r ∈ P }"
    proof
      fix r assume "r ∈ a_kernel R (S Quot P) ?h"
      hence r: "r ∈ carrier R" "P +>S (h r) = P"
        unfolding a_kernel_def kernel_def FactRing_def by auto
      hence "h r ∈ P"
        using S.a_rcosI R.l_zero S.l_zero additive_subgroup.a_subset[OF
ideal.axioms(1)[OF is_ideal]]
              additive_subgroup.zero_closed[OF ideal.axioms(1)[OF is_ideal]]
hom_closed by metis
      thus "r ∈ { r ∈ carrier R. h r ∈ P }" using r by simp
    qed
  next
    show "{ r ∈ carrier R. h r ∈ P } ⊆ a_kernel R (S Quot P) ?h"
    proof
      fix r assume "r ∈ { r ∈ carrier R. h r ∈ P }"
      hence r: "r ∈ carrier R" "h r ∈ P" by simp_all
      hence "?h r = P"
        by (simp add: S.a_coset_join2 additive_subgroup.a_subgroup ideal.axioms(1)
is_ideal)
```

```
      thus "r ∈ a_kernel R (S Quot P) ?h"
        unfolding a_kernel_def kernel_def FactRing_def using r(1) by auto
    qed
  qed
  ultimately show "primeideal { r ∈ carrier R. h r ∈ P } R" by simp
qed

end


theory IntRing
imports "HOL-Computational_Algebra.Primes" QuotRing Lattice
begin
```

# 28 The Ring of Integers

## 28.1 Some properties of `int`

```
lemma dvds_eq_abseq:
  fixes k :: int
  shows "l dvd k ∧ k dvd l ⟷ |l| = |k|"
  by (metis dvd_if_abs_eq lcm.commute lcm_proj1_iff_int)
```

## 28.2 $\mathcal{Z}$: The Set of Integers as Algebraic Structure

```
abbreviation int_ring :: "int ring" ("𝒵")
  where "int_ring ≡ ⦇carrier = UNIV, mult = (*), one = 1, zero = 0, add
= (+)⦈"

lemma int_Zcarr [intro!, simp]: "k ∈ carrier 𝒵"
  by simp

lemma int_is_cring: "cring 𝒵"
proof (rule cringI)
  show "abelian_group 𝒵"
    by (rule abelian_groupI) (auto intro: left_minus)
  show "Group.comm_monoid 𝒵"
    by (simp add: Group.monoid.intro monoid.monoid_comm_monoidI)
qed (auto simp: distrib_right)
```

## 28.3 Interpretations

Since definitions of derived operations are global, their interpretation needs to be done as early as possible — that is, with as few assumptions as possible.

```
interpretation int: monoid 𝒵
  rewrites "carrier 𝒵 = UNIV"
    and "mult 𝒵 x y = x * y"
    and "one 𝒵 = 1"
```

```
    and "pow 𝒵 x n = x^n"
proof -
  — Specification
  show "monoid 𝒵" by standard auto
  then interpret int: monoid 𝒵 .

  — Carrier
  show "carrier 𝒵 = UNIV" by simp

  — Operations
  show "mult 𝒵 x y = x * y" for x y by simp
  show "one 𝒵 = 1" by simp
  show "pow 𝒵 x n = x^n" by (induct n) simp_all
qed

interpretation int: comm_monoid 𝒵
  rewrites "finprod 𝒵 f A = prod f A"
proof -
  — Specification
  show "comm_monoid 𝒵" by standard auto
  then interpret int: comm_monoid 𝒵 .

  — Operations
  show "finprod 𝒵 f A = prod f A"
    by (induct A rule: infinite_finite_induct) auto
qed

interpretation int: abelian_monoid 𝒵
  rewrites int_carrier_eq: "carrier 𝒵 = UNIV"
    and int_zero_eq: "zero 𝒵 = 0"
    and int_add_eq: "add 𝒵 x y = x + y"
    and int_finsum_eq: "finsum 𝒵 f A = sum f A"
proof -
  — Specification
  show "abelian_monoid 𝒵" by standard auto
  then interpret int: abelian_monoid 𝒵 .

  — Carrier
  show "carrier 𝒵 = UNIV" by simp

  — Operations
  show "add 𝒵 x y = x + y" for x y by simp
  show zero: "zero 𝒵 = 0" by simp
  show "finsum 𝒵 f A = sum f A"
    by (induct A rule: infinite_finite_induct) auto
qed

interpretation int: abelian_group 𝒵
```

```
  rewrites "carrier 𝒵 = UNIV"
    and "zero 𝒵 = 0"
    and "add 𝒵 x y = x + y"
    and "finsum 𝒵 f A = sum f A"
    and int_a_inv_eq: "a_inv 𝒵 x = - x"
    and int_a_minus_eq: "a_minus 𝒵 x y = x - y"
proof -
  — Specification
  show "abelian_group 𝒵"
  proof (rule abelian_groupI)
    fix x
    assume "x ∈ carrier 𝒵"
    then show "∃y ∈ carrier 𝒵. y ⊕𝒵 x = 𝟬𝒵"
      by simp arith
  qed auto
  then interpret int: abelian_group 𝒵 .
  — Operations
  have add: "add 𝒵 x y = x + y" for x y by simp
  have zero: "zero 𝒵 = 0" by simp
  show a_inv: "a_inv 𝒵 x = - x" for x
  proof -
    have "add 𝒵 (- x) x = zero 𝒵"
      by (simp add: add zero)
    then show ?thesis
      by (simp add: int.minus_equality)
  qed
  show "a_minus 𝒵 x y = x - y"
    by (simp add: int.minus_eq add a_inv)
qed (simp add: int_carrier_eq int_zero_eq int_add_eq int_finsum_eq)+

interpretation int: "domain" 𝒵
  rewrites "carrier 𝒵 = UNIV"
    and "zero 𝒵 = 0"
    and "add 𝒵 x y = x + y"
    and "finsum 𝒵 f A = sum f A"
    and "a_inv 𝒵 x = - x"
    and "a_minus 𝒵 x y = x - y"
proof -
  show "domain 𝒵"
    by unfold_locales (auto simp: distrib_right distrib_left)
qed (simp add: int_carrier_eq int_zero_eq int_add_eq int_finsum_eq int_a_inv_eq
int_a_minus_eq)+
```

Removal of occurrences of UNIV in interpretation result — experimental.

```
lemma UNIV:
  "x ∈ UNIV ⟷ True"
  "A ⊆ UNIV ⟷ True"
  "(∀x ∈ UNIV. P x) ⟷ (∀x. P x)"
```

```
  "(∃x ∈ UNIV. P x) ⟷ (∃x. P x)"
  "(True ⟶ Q) ⟷ Q"
  "(True ⟹ PROP R) ≡ PROP R"
  by simp_all

interpretation int :
  partial_order "⦇carrier = UNIV::int set, eq = (=), le = (≤)⦈"
  rewrites "carrier ⦇carrier = UNIV::int set, eq = (=), le = (≤)⦈ = UNIV"
    and "le ⦇carrier = UNIV::int set, eq = (=), le = (≤)⦈ x y = (x ≤
y)"
    and "lless ⦇carrier = UNIV::int set, eq = (=), le = (≤)⦈ x y = (x
< y)"
proof -
  show "partial_order ⦇carrier = UNIV::int set, eq = (=), le = (≤)⦈"
    by standard simp_all
  show "carrier ⦇carrier = UNIV::int set, eq = (=), le = (≤)⦈ = UNIV"
    by simp
  show "le ⦇carrier = UNIV::int set, eq = (=), le = (≤)⦈ x y = (x ≤ y)"
    by simp
  show "lless ⦇carrier = UNIV::int set, eq = (=), le = (≤)⦈ x y = (x
< y)"
    by (simp add: lless_def) auto
qed

interpretation int :
  lattice "⦇carrier = UNIV::int set, eq = (=), le = (≤)⦈"
  rewrites "join ⦇carrier = UNIV::int set, eq = (=), le = (≤)⦈ x y = max
x y"
    and "meet ⦇carrier = UNIV::int set, eq = (=), le = (≤)⦈ x y = min
x y"
proof -
  let ?Z = "⦇carrier = UNIV::int set, eq = (=), le = (≤)⦈"
  show "lattice ?Z"
    apply unfold_locales
    apply (simp_all add: least_def Upper_def greatest_def Lower_def)
    apply arith+
    done
  then interpret int: lattice "?Z" .
  show "join ?Z x y = max x y"
    by (metis int.le_iff_meet iso_tuple_UNIV_I join_comm linear max.absorb_iff2
max_def)
  show "meet ?Z x y = min x y"
    using int.meet_le int.meet_left int.meet_right by auto
qed

interpretation int :
  total_order "⦇carrier = UNIV::int set, eq = (=), le = (≤)⦈"
  by standard clarsimp
```

## 28.4 Generated Ideals of $\mathcal{Z}$

```
lemma int_Idl: "Idl⇘𝒵⇙ {a} = {x * a | x. True}"
  by (simp_all add: cgenideal_def int.cgenideal_eq_genideal[symmetric])

lemma multiples_principalideal: "principalideal {x * a | x. True } 𝒵"
  by (metis UNIV_I int.cgenideal_eq_genideal int.cgenideal_is_principalideal
int_Idl)

lemma prime_primeideal:
  assumes prime: "Factorial_Ring.prime p"
  shows "primeideal (Idl⇘𝒵⇙ {p}) 𝒵"
proof (rule primeidealI)
  show "ideal (Idl⇘𝒵⇙ {p}) 𝒵"
    by (rule int.genideal_ideal, simp)
  show "cring 𝒵"
    by (rule int_is_cring)
  have False if "UNIV = {v::int. ∃x. v = x * p}"
  proof -
    from that
    obtain i where "1 = i * p"
      by (blast intro:  elim: )
    then show False
      using prime by (auto simp add: abs_mult zmult_eq_1_iff)
  qed
  then show "carrier 𝒵 ≠ Idl⇘𝒵⇙ {p}"
    by (auto simp add: int.cgenideal_eq_genideal[symmetric] cgenideal_def)
  have "p dvd a ∨ p dvd b" if "a * b = x * p" for a b x
    by (simp add: prime prime_dvd_multD that)
  then show "⋀a b. ⟦a ∈ carrier 𝒵; b ∈ carrier 𝒵; a ⊗⇘𝒵⇙ b ∈ Idl⇘𝒵⇙ {p}⟧
          ⟹ a ∈ Idl⇘𝒵⇙ {p} ∨ b ∈ Idl⇘𝒵⇙ {p}"
    by (auto simp add: int.cgenideal_eq_genideal[symmetric] cgenideal_def
dvd_def mult.commute)
qed
```

## 28.5 Ideals and Divisibility

```
lemma int_Idl_subset_ideal: "Idl⇘𝒵⇙ {k} ⊆ Idl⇘𝒵⇙ {l} = (k ∈ Idl⇘𝒵⇙ {l})"
  by (rule int.Idl_subset_ideal') simp_all

lemma Idl_subset_eq_dvd: "Idl⇘𝒵⇙ {k} ⊆ Idl⇘𝒵⇙ {l} ⟷ l dvd k"
  by (subst int_Idl_subset_ideal) (auto simp: dvd_def int_Idl)

lemma dvds_eq_Idl: "l dvd k ∧ k dvd l ⟷ Idl⇘𝒵⇙ {k} = Idl⇘𝒵⇙ {l}"
proof -
  have a: "l dvd k ⟷ (Idl⇘𝒵⇙ {k} ⊆ Idl⇘𝒵⇙ {l})"
    by (rule Idl_subset_eq_dvd[symmetric])
  have b: "k dvd l ⟷ (Idl⇘𝒵⇙ {l} ⊆ Idl⇘𝒵⇙ {k})"
    by (rule Idl_subset_eq_dvd[symmetric])
```

```
  have "l dvd k ∧ k dvd l ⟷ Idl_Z {k} ⊆ Idl_Z {l} ∧ Idl_Z {l} ⊆ Idl_Z
{k}"
    by (subst a, subst b, simp)
  also have "Idl_Z {k} ⊆ Idl_Z {l} ∧ Idl_Z {l} ⊆ Idl_Z {k} ⟷ Idl_Z {k}
= Idl_Z {l}"
    by blast
  finally show ?thesis .
qed

lemma Idl_eq_abs: "Idl_Z {k} = Idl_Z {l} ⟷ |l| = |k|"
  apply (subst dvds_eq_abseq[symmetric])
  apply (rule dvds_eq_Idl[symmetric])
  done
```

## 28.6 Ideals and the Modulus

```
definition ZMod :: "int ⇒ int ⇒ int set"
  where "ZMod k r = (Idl_Z {k}) +>_Z r"

lemmas ZMod_defs =
  ZMod_def genideal_def

lemma rcos_zfact:
  assumes kIl: "k ∈ ZMod l r"
  shows "∃x. k = x * l + r"
proof -
  from kIl[unfolded ZMod_def] have "∃xl∈Idl_Z {l}. k = xl + r"
    by (simp add: a_r_coset_defs)
  then obtain xl where xl: "xl ∈ Idl_Z {l}" and k: "k = xl + r"
    by auto
  from xl obtain x where "xl = x * l"
    by (auto simp: int_Idl)
  with k have "k = x * l + r"
    by simp
  then show "∃x. k = x * l + r" ..
qed

lemma ZMod_imp_zmod:
  assumes zmods: "ZMod m a = ZMod m b"
  shows "a mod m = b mod m"
proof -
  interpret ideal "Idl_Z {m}" Z
    by (rule int.genideal_ideal) fast
  from zmods have "b ∈ ZMod m a"
    unfolding ZMod_def by (simp add: a_repr_independenceD)
  then have "∃x. b = x * m + a"
    by (rule rcos_zfact)
  then obtain x where "b = x * m + a"
    by fast
```

```
  then have "b mod m = (x * m + a) mod m"
    by simp
  also have "... = ((x * m) mod m) + (a mod m)"
    by (simp add: mod_add_eq)
  also have "... = a mod m"
    by simp
  finally have "b mod m = a mod m" .
  then show "a mod m = b mod m" ..
qed

lemma ZMod_mod: "ZMod m a = ZMod m (a mod m)"
proof -
  interpret ideal "Idl𝒵 {m}" 𝒵
    by (rule int.genideal_ideal) fast
  show ?thesis
    unfolding ZMod_def
    apply (rule a_repr_independence'[symmetric])
    apply (simp add: int_Idl a_r_coset_defs)
  proof -
    have "a = m * (a div m) + (a mod m)"
      by (simp add: mult_div_mod_eq [symmetric])
    then have "a = (a div m) * m + (a mod m)"
      by simp
    then show "∃h. (∃x. h = x * m) ∧ a = h + a mod m"
      by fast
  qed simp
qed

lemma zmod_imp_ZMod:
  assumes modeq: "a mod m = b mod m"
  shows "ZMod m a = ZMod m b"
proof -
  have "ZMod m a = ZMod m (a mod m)"
    by (rule ZMod_mod)
  also have "... = ZMod m (b mod m)"
    by (simp add: modeq[symmetric])
  also have "... = ZMod m b"
    by (rule ZMod_mod[symmetric])
  finally show ?thesis .
qed

corollary ZMod_eq_mod: "ZMod m a = ZMod m b ⟷ a mod m = b mod m"
  apply (rule iffI)
  apply (erule ZMod_imp_zmod)
  apply (erule zmod_imp_ZMod)
  done
```

### 28.7 Factorization

```
definition ZFact :: "int ⇒ int set ring"
  where "ZFact k = 𝒵 Quot (Idl𝒵 {k})"

lemmas ZFact_defs = ZFact_def FactRing_def

lemma ZFact_is_cring: "cring (ZFact k)"
  by (simp add: ZFact_def ideal.quotient_is_cring int.cring_axioms int.genideal_ideal)

lemma ZFact_zero: "carrier (ZFact 0) = (⋃a. {{a}})"
  by (simp add: ZFact_defs A_RCOSETS_defs r_coset_def int.genideal_zero)

lemma ZFact_one: "carrier (ZFact 1) = {UNIV}"
  unfolding ZFact_defs A_RCOSETS_defs r_coset_def ring_record_simps int.genideal_one
proof
  have "⋀a b::int. ∃x. b = x + a"
    by presburger
  then show "(⋃a::int. {⋃h. {h + a}}) ⊆ {UNIV}"
    by force
  then show "{UNIV} ⊆ (⋃a::int. {⋃h. {h + a}})"
    by (metis (no_types, lifting) UNIV_I UN_I singletonD singletonI subset_iff)
qed

lemma ZFact_prime_is_domain:
  assumes pprime: "Factorial_Ring.prime p"
  shows "domain (ZFact p)"
  by (simp add: ZFact_def pprime prime_primeideal primeideal.quotient_is_domain)

end


theory Weak_Morphisms
  imports QuotRing

begin
```

## 29 Weak Morphisms

The definition of ring isomorphism, as well as the definition of group isomorphism, doesn't enforce any algebraic constraint to the structure of the schemes involved. This seems unnatural, but it turns out to be very useful: in order to prove that a scheme B satisfy certain algebraic constraints, it's sufficient to prove those for a scheme A and show the existence of an isomorphism between A and B. In this section, we explore this idea in a different way: given a scheme A and a function f, we build a scheme B with an algebraic structure of same strength as A where f is an homomorphism from A to B.

## 29.1 Definitions

```
locale weak_group_morphism = normal H G for f and H and G (structure)
+
  assumes inj_mod_subgroup: "⟦ a ∈ carrier G; b ∈ carrier G ⟧ ⟹ f a
= f b ⟷ a ⊗ (inv b) ∈ H"

locale weak_ring_morphism = ideal I R for f and I and R (structure) +
  assumes inj_mod_ideal: "⟦ a ∈ carrier R; b ∈ carrier R ⟧ ⟹ f a =
f b ⟷ a ⊖ b ∈ I"


definition image_group :: "('a ⇒ 'b) ⇒ ('a, 'c) monoid_scheme ⇒ 'b monoid"
  where "image_group f G ≡
          ⦇ carrier = f ` (carrier G),
              mult = (λa b. f ((inv_into (carrier G) f a) ⊗G (inv_into
(carrier G) f b))),
               one = f 1G ⦈"

definition image_ring :: "('a ⇒ 'b) ⇒ ('a, 'c) ring_scheme ⇒ 'b ring"
  where "image_ring f R ≡ monoid.extend (image_group f R)
          ⦇ zero = f 0R,
             add = (λa b. f ((inv_into (carrier R) f a) ⊕R (inv_into
(carrier R) f b))) ⦈"
```

## 29.2 Weak Group Morphisms

```
lemma image_group_carrier: "carrier (image_group f G) = f ` (carrier
G)"
  unfolding image_group_def by simp

lemma image_group_one: "one (image_group f G) = f 1G"
  unfolding image_group_def by simp

lemma weak_group_morphismsI:
  assumes "H ◁ G" and "⋀a b. ⟦ a ∈ carrier G; b ∈ carrier G ⟧ ⟹ f
a = f b ⟷ a ⊗G (invG b) ∈ H"
  shows "weak_group_morphism f H G"
  using assms unfolding weak_group_morphism_def weak_group_morphism_axioms_def
by auto

lemma image_group_truncate:
  fixes R :: "('a, 'b) monoid_scheme"
  shows "monoid.truncate (image_group f R) = image_group f (monoid.truncate
R)"
  by (simp add: image_group_def monoid.defs)

lemma image_ring_truncate: "monoid.truncate (image_ring f R) = image_group
f R"
  by (simp add: image_ring_def monoid.defs)
```

```
lemma (in ring) weak_add_group_morphism:
  assumes "weak_ring_morphism f I R" shows "weak_group_morphism f I (add_monoid
R)"
proof -
  have is_normal: "I ◁ (add_monoid R)"
    using ideal_is_normal[OF  weak_ring_morphism.axioms(1)[OF assms]]
.
  show ?thesis
    using weak_group_morphism.intro[OF is_normal]
          weak_ring_morphism.inj_mod_ideal[OF assms]
    unfolding weak_group_morphism_axioms_def a_minus_def a_inv_def
    by auto
qed

lemma (in group) weak_group_morphism_range:
  assumes "weak_group_morphism f H G" and "a ∈ carrier G" shows "f `
(H #> a) = { f a }"
proof -
  interpret H: subgroup H G
    using weak_group_morphism.axioms(1)[OF assms(1)] unfolding normal_def
by simp
  show ?thesis
  proof
    show "{ f a } ⊆ f ` (H #> a)"
      using H.one_closed assms(2) unfolding r_coset_def by force
  next
    show "f ` (H #> a) ⊆ { f a }"
    proof
      fix b assume "b ∈ f ` (H #> a)" then obtain h where "h ∈ H" "h
∈ carrier G" "b = f (h ⊗ a)"
        unfolding r_coset_def using H.subset by auto
      thus "b ∈ { f a }"
        using weak_group_morphism.inj_mod_subgroup[OF assms(1)] assms(2)
        by (metis inv_solve_right m_closed singleton_iff)
    qed
  qed
qed

lemma (in group) vimage_eq_rcoset:
  assumes "weak_group_morphism f H G" and "a ∈ carrier G"
  shows "{ b ∈ carrier G. f b = f a } = H #> a" and "{ b ∈ carrier G.
f b = f a } = a <# H"
proof -
  interpret H: normal H G
    using weak_group_morphism.axioms(1)[OF assms(1)] by simp
  show "{ b ∈ carrier G. f b = f a } = H #> a"
  proof
    show "H #> a ⊆ { b ∈ carrier G. f b = f a }"
```

```
      using r_coset_subset_G[OF H.subset assms(2)] weak_group_morphism_range[OF
assms] by auto
  next
    show "{ b ∈ carrier G. f b = f a } ⊆ H #> a"
    proof
      fix b assume b: "b ∈ { b ∈ carrier G. f b = f a }" then obtain
h where "h ∈ H" "b ⊗ (inv a) = h"
        using weak_group_morphism.inj_mod_subgroup[OF assms(1)] assms(2)
by fastforce
      thus "b ∈ H #> a"
        using H.rcos_module[OF is_group] b assms(2) by blast
    qed
  qed
  thus "{ b ∈ carrier G. f b = f a } = a <# H"
    by (simp add: assms(2) H.coset_eq)
qed

lemma (in group) weak_group_morphism_ker:
  assumes "weak_group_morphism f H G" shows "kernel G (image_group f
G) f = H"
  using vimage_eq_rcoset(1)[OF assms one_closed] weak_group_morphism.axioms(1)[OF
assms(1)]
  by (simp add: image_group_def kernel_def normal_def subgroup.subset)

lemma (in group) weak_group_morphism_inv_into:
  assumes "weak_group_morphism f H G" and "a ∈ carrier G"
  obtains h h' where "h  ∈ H" "inv_into (carrier G) f (f a) = h ⊗ a"
                 and "h' ∈ H" "inv_into (carrier G) f (f a) = a ⊗ h'"
proof -
  have "inv_into (carrier G) f (f a) ∈ { b ∈ carrier G. f b = f a }"
    using assms(2) by (auto simp add: inv_into_into f_inv_into_f)
  thus thesis
    using that vimage_eq_rcoset[OF assms] unfolding r_coset_def l_coset_def
by blast
qed

proposition (in group) weak_group_morphism_is_iso:
  assumes "weak_group_morphism f H G" shows "(λx. the_elem (f ‘ x)) ∈
iso (G Mod H) (image_group f G)"
proof (auto simp add: iso_def hom_def image_group_def)
  interpret H: normal H G
    using weak_group_morphism.axioms(1)[OF assms] .

  show "⋀x. x ∈ carrier (G Mod H) ⟹ the_elem (f ‘ x) ∈ f ‘ carrier
G"
    unfolding FactGroup_def RCOSETS_def using weak_group_morphism_range[OF
assms] by auto

  thus  "bij_betw (λx. the_elem (f ‘ x)) (carrier (G Mod H)) (f ‘ carrier
```

```
G)"
    unfolding bij_betw_def
  proof (auto)
    fix a assume "a ∈ carrier G"
    hence "the_elem (f ‘ (H #> a)) = f a" and "H #> a ∈ carrier (G Mod
H)"
      using weak_group_morphism_range[OF assms] unfolding FactGroup_def
RCOSETS_def by auto
    thus "f a ∈ (λx. the_elem (f ‘ x)) ‘ carrier (G Mod H)"
      using image_iff by fastforce
  next
    show "inj_on (λx. the_elem (f ‘ x)) (carrier (G Mod H))"
    proof (rule inj_onI)
      fix x y assume "x ∈ (carrier (G Mod H))" and "y ∈ (carrier (G Mod
H))"
      then obtain a b where a: "a ∈ carrier G" "x = H #> a" and b: "b
∈ carrier G" "y = H #> b"
        unfolding FactGroup_def RCOSETS_def by auto
      assume "the_elem (f ‘ x) = the_elem (f ‘ y)"
      hence "a ⊗ (inv b) ∈ H"
        using weak_group_morphism.inj_mod_subgroup[OF assms]
              weak_group_morphism_range[OF assms] a b by auto
      thus "x = y"
        using a(1) b(1) unfolding a b
        by (meson H.rcos_const H.subset group.coset_mult_inv1 is_group)
    qed
  qed

  fix x y assume "x ∈ carrier (G Mod H)" "y ∈ carrier (G Mod H)"
  then obtain a b where a: "a ∈ carrier G" "x = H #> a" and b: "b ∈
carrier G" "y = H #> b"
    unfolding FactGroup_def RCOSETS_def by auto

  show "the_elem (f ‘ (x <#> y)) = f (inv_into (carrier G) f (the_elem
(f ‘ x)) ⊗
                                       inv_into (carrier G) f (the_elem
(f ‘ y)))"
  proof (simp add: weak_group_morphism_range[OF assms] a b)
    obtain h1 h2
      where h1: "h1 ∈ H" "inv_into (carrier G) f (f a) = a ⊗ h1"
        and h2: "h2 ∈ H" "inv_into (carrier G) f (f b) = h2 ⊗ b"
      using weak_group_morphism_inv_into[OF assms] a(1) b(1) by metis
    have "the_elem (f ‘ ((H #> a) <#> (H #> b))) = the_elem (f ‘ (H #>
(a ⊗ b)))"
      by (simp add: a b H.rcos_sum)
    hence "the_elem (f ‘ ((H #> a) <#> (H #> b))) = f (a ⊗ b)"
      using weak_group_morphism_range[OF assms] a(1) b(1) by auto
    moreover
    have "(a ⊗ h1) ⊗ (h2 ⊗ b) = a ⊗ (h1 ⊗ h2 ⊗ b)"
```

```
      by (simp add: a(1) b(1) h1(1) h2(1) H.subset m_assoc)
    hence "(a ⊗ h1) ⊗ (h2 ⊗ b) ∈ a <# (H #> b)"
      using h1(1) h2(1) unfolding l_coset_def r_coset_def by auto
    hence "(a ⊗ h1) ⊗ (h2 ⊗ b) ∈ (a ⊗ b) <# H"
      by (simp add: H.subset H.coset_eq a(1) b(1) lcos_m_assoc)
    hence "f (inv_into (carrier G) f (f a) ⊗ inv_into (carrier G) f (f
b)) = f (a ⊗ b)"
      using vimage_eq_rcoset(2)[OF assms] a(1) b(1) unfolding h1 h2 by
auto
    ultimately
    show "the_elem (f ‘ ((H #> a) <#> (H #> b))) = f (inv_into (carrier
G) f (f a) ⊗
                                                        inv_into (carrier
G) f (f b))"
      by simp
  qed
qed

corollary (in group) image_group_is_group:
  assumes "weak_group_morphism f H G" shows "group (image_group f G)"
proof -
  interpret H: normal H G
    using weak_group_morphism.axioms(1)[OF assms] .

  have "group ((image_group f G) ⦇ one := the_elem (f ‘ 𝟭G Mod H) ⦈)"
    using group.iso_imp_img_group[OF H.factorgroup_is_group weak_group_morphism_is_iso[OF
assms]] .
  moreover have "𝟭G Mod H = H #> 𝟭"
    unfolding FactGroup_def using H.subset by force
  hence "the_elem (f ‘ 𝟭G Mod H) = f 𝟭"
    using weak_group_morphism_range[OF assms one_closed] by simp
  ultimately show ?thesis by (simp add: image_group_def)
qed

corollary (in group) weak_group_morphism_is_hom:
  assumes "weak_group_morphism f H G" shows "f ∈ hom G (image_group f
G)"
proof -
  interpret H: normal H G
    using weak_group_morphism.axioms(1)[OF assms] .

  have the_elem_hom: "(λx. the_elem (f ‘ x)) ∈ hom (G Mod H) (image_group
f G)"
    using weak_group_morphism_is_iso[OF assms] by (simp add: iso_def)
  have hom: "(λx. the_elem (f ‘ x)) ∘ (#>) H ∈ hom G (image_group f G)"
    using hom_compose[OF H.r_coset_hom_Mod the_elem_hom]
    using Group.hom_compose H.r_coset_hom_Mod the_elem_hom by blast
  have restrict: "⋀a. a ∈ carrier G ⟹ ((λx. the_elem (f ‘ x)) ∘ (#>)
H) a = f a"
```

```
    using weak_group_morphism_range[OF assms] by auto
  show ?thesis
    using hom_restrict[OF hom restrict] by simp
qed

corollary (in group) weak_group_morphism_group_hom:
  assumes "weak_group_morphism f H G" shows "group_hom G (image_group
f G) f"
  using image_group_is_group[OF assms] weak_group_morphism_is_hom[OF assms]
group_axioms
  unfolding group_hom_def group_hom_axioms_def by simp
```

## 29.3 Weak Ring Morphisms

```
lemma image_ring_carrier: "carrier (image_ring f R) = f ` (carrier R)"
  unfolding image_ring_def image_group_def by (simp add: monoid.defs)

lemma image_ring_one: "one (image_ring f R) = f 𝟭R"
  unfolding image_ring_def image_group_def by (simp add: monoid.defs)

lemma image_ring_zero: "zero (image_ring f R) = f 𝟬R"
  unfolding image_ring_def image_group_def by (simp add: monoid.defs)

lemma weak_ring_morphismI:
  assumes "ideal I R" and "⋀a b. ⟦ a ∈ carrier R; b ∈ carrier R ⟧ ⟹
f a = f b ⟷ a ⊖R b ∈ I"
  shows "weak_ring_morphism f I R"
  using assms unfolding weak_ring_morphism_def weak_ring_morphism_axioms_def
by auto

lemma (in ring) weak_ring_morphism_range:
  assumes "weak_ring_morphism f I R" and "a ∈ carrier R" shows "f `
(I +> a) = { f a }"
  using add.weak_group_morphism_range[OF weak_add_group_morphism[OF assms(1)]
assms(2)]
  unfolding a_r_coset_def .

lemma (in ring) vimage_eq_a_rcoset:
  assumes "weak_ring_morphism f I R" and "a ∈ carrier R" shows "{ b
∈ carrier R. f b = f a } = I +> a"
  using add.vimage_eq_rcoset[OF weak_add_group_morphism[OF assms(1)] assms(2)]
  unfolding a_r_coset_def by simp

lemma (in ring) weak_ring_morphism_ker:
  assumes "weak_ring_morphism f I R" shows "a_kernel R (image_ring f
R) f = I"
  using add.weak_group_morphism_ker[OF weak_add_group_morphism[OF assms]]
  unfolding kernel_def a_kernel_def' image_ring_def image_group_def by
(simp add: monoid.defs)
```

```
lemma (in ring) weak_ring_morphism_inv_into:
  assumes "weak_ring_morphism f I R" and "a ∈ carrier R"
  obtains i where "i ∈ I" "inv_into (carrier R) f (f a) = i ⊕ a"
  using add.weak_group_morphism_inv_into(1)[OF weak_add_group_morphism[OF
assms(1)] assms(2)] by auto

proposition (in ring) weak_ring_morphism_is_iso:
  assumes "weak_ring_morphism f I R" shows "(λx. the_elem (f ` x)) ∈
ring_iso (R Quot I) (image_ring f R)"
proof (rule ring_iso_memI)
  show "bij_betw (λx. the_elem (f ` x)) (carrier (R Quot I)) (carrier
(image_ring f R))"
   and add_hom: "⋀x y. ⟦ x ∈ carrier (R Quot I); y ∈ carrier (R Quot
I) ⟧ ⟹
              the_elem (f ` (x ⊕R Quot I y)) = the_elem (f ` x) ⊕image_ring f R
the_elem (f ` y)"
    using add.weak_group_morphism_is_iso[OF weak_add_group_morphism[OF
assms]]
    unfolding iso_def hom_def FactGroup_def FactRing_def A_RCOSETS_def
set_add_def
    by (auto simp add: image_ring_def image_group_def monoid.defs)
next
  interpret I: ideal I R
    using weak_ring_morphism.axioms(1)[OF assms] .

  show "the_elem (f ` 𝟭R Quot I) = 𝟭image_ring f R"
   and "⋀x. x ∈ carrier (R Quot I) ⟹ the_elem (f ` x) ∈ carrier (image_ring
f R)"
    using weak_ring_morphism_range[OF assms] one_closed I.Icarr
    by (auto simp add: image_ring_def image_group_def monoid.defs FactRing_def
A_RCOSETS_def')

  fix x y assume "x ∈ carrier (R Quot I)" "y ∈ carrier (R Quot I)"
  then obtain a b where a: "a ∈ carrier R" "x = I +> a" and b: "b ∈
carrier R" "y = I +> b"
    unfolding FactRing_def A_RCOSETS_def' by auto
  hence prod: "x ⊗R Quot I y = I +> (a ⊗ b)"
    unfolding FactRing_def by (simp add: I.rcoset_mult_add)

  show "the_elem (f ` (x ⊗R Quot I y)) = the_elem (f ` x) ⊗image_ring f R
the_elem (f ` y)"
    unfolding prod
  proof (simp add: weak_ring_morphism_range[OF assms] a b image_ring_def
image_group_def monoid.defs)
    obtain i j
      where i: "i ∈ I" "inv_into (carrier R) f (f a) = i ⊕ a"
        and j: "j ∈ I" "inv_into (carrier R) f (f b) = j ⊕ b"
      using weak_ring_morphism_inv_into[OF assms] a(1) b(1) by metis
```

```
    have "i ∈ carrier R" and "j ∈ carrier R"
      using I.Icarr i(1) j(1) by auto
    hence "(i ⊕ a) ⊗ (j ⊕ b) = (i ⊕ a) ⊗ j ⊕ (i ⊗ b) ⊕ (a ⊗ b)"
      using a(1) b(1) by algebra
    hence "(i ⊕ a) ⊗ (j ⊕ b) ∈ I +> (a ⊗ b)"
      using i(1) j(1) a(1) b(1) unfolding a_r_coset_def'
      by (simp add: I.I_l_closed I.I_r_closed)
    thus "f (a ⊗ b) = f (inv_into (carrier R) f (f a) ⊗ inv_into (carrier
R) f (f b))"
      unfolding i j using weak_ring_morphism_range[OF assms m_closed[OF
a(1) b(1)]]
      by (metis imageI singletonD)
  qed
qed

corollary (in ring) image_ring_zero':
  assumes "weak_ring_morphism f I R" shows "the_elem (f ` 0R Quot I)
= 0image_ring f R"
proof -
  interpret I: ideal I R
    using weak_ring_morphism.axioms(1)[OF assms] .

  have "0R Quot I = I +> 0"
    unfolding FactRing_def a_r_coset_def' by force
  thus ?thesis
    using weak_ring_morphism_range[OF assms zero_closed] unfolding image_ring_zero
by simp
qed

corollary (in ring) image_ring_is_ring:
  assumes "weak_ring_morphism f I R" shows "ring (image_ring f R)"
proof -
  interpret I: ideal I R
    using weak_ring_morphism.axioms(1)[OF assms] .

  have "ring ((image_ring f R) ⦇ zero := the_elem (f ` 0R Quot I) ⦈)"
    using ring.ring_iso_imp_img_ring[OF I.quotient_is_ring weak_ring_morphism_is_iso[OF
assms]] by simp
  thus ?thesis
    unfolding image_ring_zero'[OF assms] by simp
qed

corollary (in ring) image_ring_is_field:
  assumes "weak_ring_morphism f I R" and "field (R Quot I)" shows "field
(image_ring f R)"
  using field.ring_iso_imp_img_field[OF assms(2) weak_ring_morphism_is_iso[OF
assms(1)]]
  unfolding image_ring_zero'[OF assms(1)] by simp
```

```
corollary (in ring) weak_ring_morphism_is_hom:
  assumes "weak_ring_morphism f I R" shows "f ∈ ring_hom R (image_ring
f R)"
proof -
  interpret I: ideal I R
    using weak_ring_morphism.axioms(1)[OF assms] .

  have the_elem_hom: "(λx. the_elem (f ‘ x)) ∈ ring_hom (R Quot I) (image_ring
f R)"
    using weak_ring_morphism_is_iso[OF assms] by (simp add: ring_iso_def)
  have ring_hom: "(λx. the_elem (f ‘ x)) ∘ (+>) I ∈ ring_hom R (image_ring
f R)"
    using ring_hom_trans[OF I.rcos_ring_hom the_elem_hom] .
  have restrict: "⋀a. a ∈ carrier R ⟹ ((λx. the_elem (f ‘ x)) ∘ (+>)
I) a = f a"
    using weak_ring_morphism_range[OF assms] by auto
  show ?thesis
    using ring_hom_restrict[OF ring_hom restrict] by simp
qed

corollary (in ring) weak_ring_morphism_ring_hom:
  assumes "weak_ring_morphism f I R" shows "ring_hom_ring R (image_ring
f R) f"
  using ring_hom_ringI2[OF ring_axioms image_ring_is_ring[OF assms] weak_ring_morphism_is_h
assms]] .
```

## 29.4 Injective Functions

If the fuction is injective, we don't need to impose any algebraic restriction to the input scheme in order to state an isomorphism.

```
lemma inj_imp_image_group_iso:
  assumes "inj_on f (carrier G)" shows "f ∈ iso G (image_group f G)"
  using assms by (auto simp add: image_group_def iso_def bij_betw_def
hom_def)

lemma inj_imp_image_group_inv_iso:
  assumes "inj f" shows "Hilbert_Choice.inv f ∈ iso (image_group f G)
G"
  using assms by (auto simp add: image_group_def iso_def bij_betw_def
hom_def inj_on_def)

lemma inj_imp_image_ring_iso:
  assumes "inj_on f (carrier R)" shows "f ∈ ring_iso R (image_ring f
R)"
  using assms by (auto simp add: image_ring_def image_group_def ring_iso_def
                                 bij_betw_def ring_hom_def monoid.defs)

lemma inj_imp_image_ring_inv_iso:
  assumes "inj f" shows "Hilbert_Choice.inv f ∈ ring_iso (image_ring
```

```
f R) R"
  using assms by (auto simp add: image_ring_def image_group_def ring_iso_def
                                 bij_betw_def ring_hom_def inj_on_def
monoid.defs)

lemma (in group) inj_imp_image_group_is_group:
  assumes "inj_on f (carrier G)" shows "group (image_group f G)"
  using iso_imp_img_group[OF inj_imp_image_group_iso[OF assms]] by (simp
add: image_group_def)

lemma (in ring) inj_imp_image_ring_is_ring:
  assumes "inj_on f (carrier R)" shows "ring (image_ring f R)"
  using ring_iso_imp_img_ring[OF inj_imp_image_ring_iso[OF assms]]
  by (simp add: image_ring_def image_group_def monoid.defs)

lemma (in domain) inj_imp_image_ring_is_domain:
  assumes "inj_on f (carrier R)" shows "domain (image_ring f R)"
  using ring_iso_imp_img_domain[OF inj_imp_image_ring_iso[OF assms]]
  by (simp add: image_ring_def image_group_def monoid.defs)

lemma (in field) inj_imp_image_ring_is_field:
  assumes "inj_on f (carrier R)" shows "field (image_ring f R)"
  using ring_iso_imp_img_field[OF inj_imp_image_ring_iso[OF assms]]
  by (simp add: image_ring_def image_group_def monoid.defs)
```

# 30 Examples

In a lot of different contexts, the lack of dependent types make some definitions quite complicated. The tools developed in this theory give us a way to change the type of a scheme and preserve all of its algebraic properties. We show, in this section, how to make use of this feature in order to solve the problem mentioned above.

## 30.1 Direct Product

```
abbreviation nil_monoid :: "('a list) monoid"
  where "nil_monoid ≡ ⦇ carrier = { [] }, mult = (λa b. []), one = []
⦈"

definition DirProd_list :: "(('a, 'b) monoid_scheme) list ⇒ ('a list)
monoid"
  where "DirProd_list Gs = foldr (λG H. image_group (λ(x, xs). x # xs)
(G ×× H)) Gs nil_monoid"
```

### 30.1.1 Basic Properties

```
lemma DirProd_list_carrier:
```

```
  shows "carrier (DirProd_list (G # Gs)) = (λ(x, xs). x # xs) ‘ (carrier
G × carrier (DirProd_list Gs))"
  unfolding DirProd_list_def image_group_def by auto

lemma DirProd_list_one:
  shows "one (DirProd_list Gs) = foldr (λG tl. (one G) # tl) Gs []"
  unfolding DirProd_list_def DirProd_def image_group_def by (induct Gs)
(auto)

lemma DirProd_list_carrier_mem:
  assumes "gs ∈ carrier (DirProd_list Gs)"
  shows "length gs = length Gs" and "⋀i. i < length Gs ⟹ (gs ! i)
∈ carrier (Gs ! i)"
proof -
  let ?same_length = "λxs ys. length xs = length ys"
  let ?in_carrier = "λi gs Gs. (gs ! i) ∈ carrier (Gs ! i)"
  from assms have "?same_length gs Gs ∧ (∀i < length Gs. ?in_carrier
i gs Gs)"
  proof (induct Gs arbitrary: gs, simp add: DirProd_list_def)
    case (Cons G Gs)
    then obtain g’ gs’
      where g’: "g’ ∈ carrier G" and gs’: "gs’ ∈ carrier (DirProd_list
Gs)" and gs: "gs = g’ # gs’"
      unfolding DirProd_list_carrier by auto
    hence "?same_length gs (G # Gs)" and "⋀i. i ∈ {(Suc 0)..< length
(G # Gs)} ⟹ ?in_carrier i gs (G # Gs)"
      using Cons(1) by auto
    moreover have "?in_carrier 0 gs (G # Gs)"
      unfolding gs using g’ by simp
    ultimately show ?case
      by (metis atLeastLessThan_iff eq_imp_le less_Suc0 linorder_neqE_nat
nat_less_le)
  qed
  thus "?same_length gs Gs" and "⋀i. i < length Gs ⟹ ?in_carrier i
gs Gs"
    by simp+
qed

lemma DirProd_list_carrier_memI:
  assumes "length gs = length Gs" and "⋀i. i < length Gs ⟹ (gs ! i)
∈ carrier (Gs ! i)"
  shows "gs ∈ carrier (DirProd_list Gs)"
  using assms
proof (induct Gs arbitrary: gs, simp add: DirProd_list_def)
  case (Cons G Gs)
  then obtain g’ gs’ where gs: "gs = g’ # gs’"
    by (metis length_Suc_conv)
  have "g’ ∈ carrier G"
    using Cons(3)[of 0] unfolding gs by auto
```

```
  moreover have "gs' ∈ carrier (DirProd_list Gs)"
    using Cons unfolding gs by force
  ultimately show ?case
    unfolding DirProd_list_carrier gs by blast
qed

lemma inj_on_DirProd_carrier:
  shows "inj_on (λ(g, gs). g # gs) (carrier (G ×× (DirProd_list Gs)))"
  unfolding DirProd_def inj_on_def by auto

lemma DirProd_list_is_group:
  assumes "⋀i. i < length Gs ⟹ group (Gs ! i)" shows "group (DirProd_list
Gs)"
  using assms
proof (induct Gs)
  case Nil thus ?case
    unfolding DirProd_list_def by (unfold_locales, auto simp add: Units_def)
next
  case (Cons G Gs)
  hence is_group: "group (G ×× (DirProd_list Gs))"
    using DirProd_group[of G "DirProd_list Gs"] by force
  show ?case
    using group.inj_imp_image_group_is_group[OF is_group inj_on_DirProd_carrier]
    unfolding DirProd_list_def by auto
qed

lemma DirProd_list_iso:
  "(λ(g, gs). g # gs) ∈ iso (G ×× (DirProd_list Gs)) (DirProd_list (G
# Gs))"
  using inj_imp_image_group_iso[OF inj_on_DirProd_carrier] unfolding DirProd_list_def
by auto

end


theory Ring_Divisibility
imports Ideal Divisibility QuotRing Multiplicative_Group

begin


definition mult_of :: "('a, 'b) ring_scheme ⇒ 'a monoid" where
  "mult_of R ≡ ⦇ carrier = carrier R - {𝟬R}, mult = mult R, one = 𝟭R⦈"

lemma carrier_mult_of [simp]: "carrier (mult_of R) = carrier R - {𝟬R}"
  by (simp add: mult_of_def)

lemma mult_mult_of [simp]: "mult (mult_of R) = mult R"
 by (simp add: mult_of_def)
```

```
lemma nat_pow_mult_of: "([^]mult_of R) = (([^]R) :: _ ⇒ nat ⇒ _)"
  by (simp add: mult_of_def fun_eq_iff nat_pow_def)

lemma one_mult_of [simp]: "𝟭mult_of R = 𝟭R"
  by (simp add: mult_of_def)
```

# 31 The Arithmetic of Rings

In this section we study the links between the divisibility theory and that of rings

## 31.1 Definitions

```
locale factorial_domain = domain + factorial_monoid "mult_of R"

locale noetherian_ring = ring +
  assumes finetely_gen: "ideal I R ⟹ ∃A ⊆ carrier R. finite A ∧ I
= Idl A"

locale noetherian_domain = noetherian_ring + domain

locale principal_domain = domain +
  assumes exists_gen: "ideal I R ⟹ ∃a ∈ carrier R. I = PIdl a"

locale euclidean_domain =
  domain R for R (structure) + fixes φ :: "'a ⇒ nat"
  assumes euclidean_function:
  " ⟦ a ∈ carrier R - { 𝟬 }; b ∈ carrier R - { 𝟬 } ⟧ ⟹
   ∃q r. q ∈ carrier R ∧ r ∈ carrier R ∧ a = (b ⊗ q) ⊕ r ∧ ((r = 𝟬)
∨ (φ r < φ b))"

definition ring_irreducible :: "('a, 'b) ring_scheme ⇒ 'a ⇒ bool" (‹ring'_irreducibleı›)
  where "ring_irreducibleR a ⟷ (a ≠ 𝟬R) ∧ (irreducible R a)"

definition ring_prime :: "('a, 'b) ring_scheme ⇒ 'a ⇒ bool" (‹ring'_primeı›)
  where "ring_primeR a ⟷ (a ≠ 𝟬R) ∧ (prime R a)"
```

## 31.2 The cancellative monoid of a domain.

```
sublocale domain < mult_of: comm_monoid_cancel "mult_of R"
  rewrites "mult (mult_of R) = mult R"
       and "one  (mult_of R) =  one R"
  using m_comm m_assoc
  by (auto intro!: comm_monoid_cancelI comm_monoidI
         simp add: m_rcancel integral_iff)

sublocale factorial_domain < mult_of: factorial_monoid "mult_of R"
```

```
  rewrites "mult (mult_of R) = mult R"
       and "one  (mult_of R) =  one R"
  using factorial_monoid_axioms by auto

lemma (in ring) noetherian_ringI:
  assumes "⋀I. ideal I R ⟹ ∃A ⊆ carrier R. finite A ∧ I = Idl A"
  shows "noetherian_ring R"
  using assms by unfold_locales auto

lemma (in domain) euclidean_domainI:
  assumes "⋀a b. ⟦ a ∈ carrier R - { 𝟬 }; b ∈ carrier R - { 𝟬 } ⟧ ⟹
           ∃q r. q ∈ carrier R ∧ r ∈ carrier R ∧ a = (b ⊗ q) ⊕ r ∧
((r = 𝟬) ∨ (φ r < φ b))"
  shows "euclidean_domain R φ"
  using assms by unfold_locales auto
```

## 31.3 Passing from R to Ring_Divisibility.mult_of R and vice-versa.

```
lemma divides_mult_imp_divides [simp]: "a divides(mult_of R) b ⟹ a dividesR
b"
  unfolding factor_def by auto

lemma (in domain) divides_imp_divides_mult [simp]:
  "⟦ a ∈ carrier R; b ∈ carrier R - { 𝟬 } ⟧ ⟹ a dividesR b ⟹ a divides(mult_of R)
b"
  unfolding factor_def using integral_iff by auto

lemma (in cring) divides_one:
  assumes "a ∈ carrier R"
  shows "a divides 𝟭 ⟷ a ∈ Units R"
  using assms m_comm unfolding factor_def Units_def by force

lemma (in ring) one_divides:
  assumes "a ∈ carrier R" shows "𝟭 divides a"
  using assms unfolding factor_def by simp

lemma (in ring) divides_zero:
  assumes "a ∈ carrier R" shows "a divides 𝟬"
  using r_null[OF assms] unfolding factor_def by force

lemma (in ring) zero_divides:
  shows "𝟬 divides a ⟷ a = 𝟬"
  unfolding factor_def by auto

lemma (in domain) divides_mult_zero:
  assumes "a ∈ carrier R" shows "a divides(mult_of R) 𝟬 ⟹ a = 𝟬"
  using integral[OF _ assms] unfolding factor_def by auto
```

```
lemma (in ring) divides_mult:
  assumes "a ∈ carrier R" "c ∈ carrier R"
  shows "a divides b ⟹ (c ⊗ a) divides (c ⊗ b)"
  using m_assoc[OF assms(2,1)] unfolding factor_def by auto

lemma (in domain) mult_divides:
  assumes "a ∈ carrier R" "b ∈ carrier R" "c ∈ carrier R - { 0 }"
  shows "(c ⊗ a) divides (c ⊗ b) ⟹ a divides b"
  using assms m_assoc[of c] unfolding factor_def by (simp add: m_lcancel)

lemma (in domain) assoc_iff_assoc_mult:
  assumes "a ∈ carrier R" and "b ∈ carrier R"
  shows "a ∼ b ⟷ a ∼(mult_of R) b"
proof
  assume "a ∼(mult_of R) b" thus "a ∼ b"
    unfolding associated_def factor_def by auto
next
  assume A: "a ∼ b"
  then obtain c1 c2
    where c1: "c1 ∈ carrier R" "a = b ⊗ c1" and c2: "c2 ∈ carrier R"
"b = a ⊗ c2"
    unfolding associated_def factor_def by auto
  show "a ∼(mult_of R) b"
  proof (cases "a = 0")
    assume a: "a = 0" then have b: "b = 0"
      using c2 by auto
    show ?thesis
      unfolding associated_def factor_def a b by auto
  next
    assume a: "a ≠ 0"
    hence b: "b ≠ 0" and c1_not_zero: "c1 ≠ 0"
      using c1 assms by auto
    hence c2_not_zero: "c2 ≠ 0"
      using c2 assms by auto
    show ?thesis
      unfolding associated_def factor_def using c1 c2 c1_not_zero c2_not_zero
a b by auto
  qed
qed

lemma (in domain) Units_mult_eq_Units [simp]: "Units (mult_of R) = Units
R"
  unfolding Units_def using insert_Diff integral_iff by auto

lemma (in domain) ring_associated_iff:
  assumes "a ∈ carrier R" "b ∈ carrier R"
  shows "a ∼ b ⟷ (∃u ∈ Units R. a = u ⊗ b)"
proof (cases "a = 0")
  assume [simp]: "a = 0" show ?thesis
```

```
  proof
    assume "a ∼ b" thus "∃u ∈ Units R. a = u ⊗ b"
      using zero_divides unfolding associated_def by force
  next
    assume "∃u ∈ Units R. a = u ⊗ b" then have "b = 0"
      by (metis Units_closed Units_l_cancel ‹a = 0› assms r_null)
    thus "a ∼ b"
      using zero_divides[of 0] by auto
  qed
next
  assume a: "a ≠ 0" show ?thesis
  proof (cases "b = 0")
    assume "b = 0" thus ?thesis
      using assms a zero_divides[of a] r_null unfolding associated_def
by blast
  next
    assume b: "b ≠ 0"
    have "(∃u ∈ Units R. a = u ⊗ b) ⟷ (∃u ∈ Units R. a = b ⊗ u)"
      using m_comm[OF assms(2)] Units_closed by auto
    thus ?thesis
      using mult_of.associated_iff[of a b] a b assms
      unfolding assoc_iff_assoc_mult[OF assms] Units_mult_eq_Units
      by auto
  qed
qed

lemma (in domain) properfactor_mult_imp_properfactor:
  "⟦ a ∈ carrier R; b ∈ carrier R ⟧ ⟹ properfactor (mult_of R) b a ⟹
properfactor R b a"
proof -
  assume A: "a ∈ carrier R" "b ∈ carrier R" "properfactor (mult_of R)
b a"
  then obtain c where c: "c ∈ carrier (mult_of R)" "a = b ⊗ c"
    unfolding properfactor_def factor_def by auto
  have "a ≠ 0"
  proof (rule ccontr)
    assume a: "¬ a ≠ 0"
    hence "b = 0" using c A integral[of b c] by auto
    hence "b = a ⊗ 1" using a A by simp
    hence "a divides(mult_of R) b"
      unfolding factor_def by auto
    thus False using A unfolding properfactor_def by simp
  qed
  hence "b ≠ 0"
    using c A integral_iff by auto
  thus "properfactor R b a"
    using A divides_imp_divides_mult[of a b] unfolding properfactor_def
    by (meson DiffI divides_mult_imp_divides empty_iff insert_iff)
qed
```

```
lemma (in domain) properfactor_imp_properfactor_mult:
  "⟦ a ∈ carrier R - { 0 }; b ∈ carrier R ⟧ ⟹ properfactor R b a ⟹
properfactor (mult_of R) b a"
  unfolding properfactor_def factor_def by auto

lemma (in domain) properfactor_of_zero:
  assumes "b ∈ carrier R"
  shows "¬ properfactor (mult_of R) b 0" and "properfactor R b 0 ⟷
b ≠ 0"
  using divides_mult_zero[OF assms] divides_zero[OF assms] unfolding properfactor_def
by auto
```

## 31.4 Irreducible

The following lemmas justify the need for a definition of irreducible specific to rings: for `irreducible R`, we need to suppose we are not in a field (which is plausible, but `¬ field R` is an assumption we want to avoid; for `irreducible (Ring_Divisibility.mult_of R)`, zero is allowed.

```
lemma (in domain) zero_is_irreducible_mult:
  shows "irreducible (mult_of R) 0"
  unfolding irreducible_def Units_def properfactor_def factor_def
  using integral_iff by fastforce

lemma (in domain) zero_is_irreducible_iff_field:
  shows "irreducible R 0 ⟷ field R"
proof
  assume irr: "irreducible R 0"
  have "⋀a. ⟦ a ∈ carrier R; a ≠ 0 ⟧ ⟹ properfactor R a 0"
    unfolding properfactor_def factor_def by (auto, metis r_null zero_closed)
  hence "carrier R - { 0 } = Units R"
    using irr unfolding irreducible_def by auto
  thus "field R"
    using field.intro[OF domain_axioms] unfolding field_axioms_def by
simp
next
  assume field: "field R" show "irreducible R 0"
    using field.field_Units[OF field]
    unfolding irreducible_def properfactor_def factor_def by blast
qed

lemma (in domain) irreducible_imp_irreducible_mult:
  "⟦ a ∈ carrier R; irreducible R a ⟧ ⟹ irreducible (mult_of R) a"
  using zero_is_irreducible_mult Units_mult_eq_Units properfactor_mult_imp_properfactor
  by (cases "a = 0") (auto simp add: irreducible_def)

lemma (in domain) irreducible_mult_imp_irreducible:
  "⟦ a ∈ carrier R - { 0 }; irreducible (mult_of R) a ⟧ ⟹ irreducible
```

```
R a"
    unfolding irreducible_def using properfactor_imp_properfactor_mult
properfactor_divides by fastforce

lemma (in domain) ring_irreducibleE:
  assumes "r ∈ carrier R" and "ring_irreducible r"
  shows "r ≠ 0" "irreducible R r" "irreducible (mult_of R) r" "r ∉ Units
R"
    and "⋀a b. ⟦ a ∈ carrier R; b ∈ carrier R ⟧ ⟹ r = a ⊗ b ⟹ a
∈ Units R ∨ b ∈ Units R"
proof -
  show "irreducible (mult_of R) r" "irreducible R r"
    using assms irreducible_imp_irreducible_mult unfolding ring_irreducible_def
by auto
  show "r ≠ 0" "r ∉ Units R"
    using assms unfolding ring_irreducible_def irreducible_def by auto
next
  fix a b assume a: "a ∈ carrier R" and b: "b ∈ carrier R" and r: "r
= a ⊗ b"
  show "a ∈ Units R ∨ b ∈ Units R"
  proof (cases "properfactor R a r")
    assume "properfactor R a r" thus ?thesis
      using a assms(2) unfolding ring_irreducible_def irreducible_def
by auto
  next
    assume not_proper: "¬ properfactor R a r"
    hence "r divides a"
      using r b unfolding properfactor_def by auto
    then obtain c where c: "c ∈ carrier R" "a = r ⊗ c"
      unfolding factor_def by auto
    have "1 = c ⊗ b"
      using r c(1) b assms m_assoc m_lcancel[OF _ assms(1) one_closed
m_closed[OF c(1) b]]
      unfolding c(2) ring_irreducible_def by auto
    thus ?thesis
      using c(1) b m_comm unfolding Units_def by auto
  qed
qed

lemma (in domain) ring_irreducibleI:
  assumes "r ∈ carrier R - { 0 }" "r ∉ Units R"
    and "⋀a b. ⟦ a ∈ carrier R; b ∈ carrier R ⟧ ⟹ r = a ⊗ b ⟹ a
∈ Units R ∨ b ∈ Units R"
  shows "ring_irreducible r"
  unfolding ring_irreducible_def
proof
  show "r ≠ 0" using assms(1) by blast
next
  show "irreducible R r"
```

```
  proof (rule irreducibleI[OF assms(2)])
    fix a assume a: "a ∈ carrier R" "properfactor R a r"
    then obtain b where b: "b ∈ carrier R" "r = a ⊗ b"
      unfolding properfactor_def factor_def by blast
    hence "a ∈ Units R ∨ b ∈ Units R"
      using assms(3)[OF a(1) b(1)] by simp
    thus "a ∈ Units R"
    proof (auto)
      assume "b ∈ Units R"
      hence "r ⊗ inv b = a" and "inv b ∈ carrier R"
        using b a by (simp add: m_assoc)+
      thus ?thesis
        using a(2) unfolding properfactor_def factor_def by blast
    qed
  qed
qed

lemma (in domain) ring_irreducibleI':
  assumes "r ∈ carrier R - { 0 }" "irreducible (mult_of R) r"
  shows "ring_irreducible r"
  unfolding ring_irreducible_def
  using irreducible_mult_imp_irreducible[OF assms] assms(1) by auto
```

## 31.5 Primes

```
lemma (in domain) zero_is_prime: "prime R 0" "prime (mult_of R) 0"
  using integral unfolding prime_def factor_def Units_def by auto

lemma (in domain) prime_eq_prime_mult:
  assumes "p ∈ carrier R"
  shows "prime R p ⟷ prime (mult_of R) p"
proof (cases "p = 0", auto simp add: zero_is_prime)
  assume "p ≠ 0" "prime R p" thus "prime (mult_of R) p"
    unfolding prime_def
    using divides_mult_imp_divides Units_mult_eq_Units mult_mult_of
    by (metis DiffD1 assms carrier_mult_of divides_imp_divides_mult)
next
  assume p: "p ≠ 0" "prime (mult_of R) p" show "prime R p"
  proof (rule primeI)
    show "p ∉ Units R"
      using p(2) Units_mult_eq_Units unfolding prime_def by simp
  next
    fix a b assume a: "a ∈ carrier R" and b: "b ∈ carrier R" and dvd:
"p divides a ⊗ b"
    then obtain c where c: "c ∈ carrier R" "a ⊗ b = p ⊗ c"
      unfolding factor_def by auto
    show "p divides a ∨ p divides b"
    proof (cases "a ⊗ b = 0")
      case True thus ?thesis
```

```
        using integral[OF _ a b] c unfolding factor_def by force
    next
      case False
      hence "p divides(mult_of R) (a ⊗ b)"
        using divides_imp_divides_mult[OF assms _ dvd] m_closed[OF a b]
by simp
      moreover have "a ≠ 0" "b ≠ 0" "c ≠ 0"
        using False a b c p l_null integral_iff by (auto, simp add: assms)
      ultimately show ?thesis
        using a b p unfolding prime_def
        by (auto, metis Diff_iff divides_mult_imp_divides singletonD)
    qed
  qed
qed

lemma (in domain) ring_primeE:
  assumes "p ∈ carrier R" "ring_prime p"
  shows "p ≠ 0" "prime (mult_of R) p" "prime R p"
  using assms prime_eq_prime_mult unfolding ring_prime_def by auto

lemma (in ring) ring_primeI:
  assumes "p ≠ 0" "prime R p" shows "ring_prime p"
  using assms unfolding ring_prime_def by auto

lemma (in domain) ring_primeI':
  assumes "p ∈ carrier R - { 0 }" "prime (mult_of R) p"
  shows "ring_prime p"
  using assms prime_eq_prime_mult unfolding ring_prime_def by auto
```

## 31.6 Basic Properties

```
lemma (in cring) to_contain_is_to_divide:
  assumes "a ∈ carrier R" "b ∈ carrier R"
  shows "PIdl b ⊆ PIdl a ⟷ a divides b"
proof
  show "PIdl b ⊆ PIdl a ⟹ a divides b"
  proof -
    assume "PIdl b ⊆ PIdl a"
    hence "b ∈ PIdl a"
      by (meson assms(2) local.ring_axioms ring.cgenideal_self subsetCE)
    thus ?thesis
      unfolding factor_def cgenideal_def using m_comm assms(1) by blast
  qed
  show "a divides b ⟹ PIdl b ⊆ PIdl a"
  proof -
    assume "a divides b" then obtain c where c: "c ∈ carrier R" "b =
c ⊗ a"
      unfolding factor_def using m_comm[OF assms(1)] by blast
    show "PIdl b ⊆ PIdl a"
```

```
    proof
      fix x assume "x ∈ PIdl b"
      then obtain d where d: "d ∈ carrier R" "x = d ⊗ b"
        unfolding cgenideal_def by blast
      hence "x = (d ⊗ c) ⊗ a"
        using c d m_assoc assms by simp
      thus "x ∈ PIdl a"
        unfolding cgenideal_def using m_assoc assms c d by blast
    qed
  qed
qed

lemma (in cring) associated_iff_same_ideal:
  assumes "a ∈ carrier R" "b ∈ carrier R"
  shows "a ∼ b ⟷ PIdl a = PIdl b"
  unfolding associated_def
  using to_contain_is_to_divide[OF assms]
        to_contain_is_to_divide[OF assms(2,1)] by auto

lemma (in cring) ideal_eq_carrier_iff:
  assumes "a ∈ carrier R"
  shows "carrier R = PIdl a ⟷ a ∈ Units R"
proof
  assume "carrier R = PIdl a"
  hence "𝟭 ∈ PIdl a"
    by auto
  then obtain b where "b ∈ carrier R" "𝟭 = a ⊗ b" "𝟭 = b ⊗ a"
    unfolding cgenideal_def using m_comm[OF assms] by auto
  thus "a ∈ Units R"
    using assms unfolding Units_def by auto
next
  assume "a ∈ Units R"
  then have inv_a: "inv a ∈ carrier R" "inv a ⊗ a = 𝟭"
    by auto
  have "carrier R ⊆ PIdl a"
  proof
    fix b assume "b ∈ carrier R"
    hence "(b ⊗ inv a) ⊗ a = b" and "b ⊗ inv a ∈ carrier R"
      using m_assoc[OF _ inv_a(1) assms] inv_a by auto
    thus "b ∈ PIdl a"
      unfolding cgenideal_def by force
  qed
  thus "carrier R = PIdl a"
    using assms by (simp add: cgenideal_eq_rcos r_coset_subset_G subset_antisym)
qed

lemma (in domain) primeideal_iff_prime:
  assumes "p ∈ carrier R - { 𝟬 }"
  shows "primeideal (PIdl p) R ⟷ ring_prime p"
```

```
proof
  assume PIdl: "primeideal (PIdl p) R" show "ring_prime p"
  proof (rule ring_primeI)
    show "p ≠ 0" using assms by simp
  next
    show "prime R p"
    proof (rule primeI)
      show "p ∉ Units R"
        using ideal_eq_carrier_iff[of p] assms primeideal.I_notcarr[OF
PIdl] by auto
    next
      fix a b assume a: "a ∈ carrier R" and b: "b ∈ carrier R" and dvd:
"p divides a ⊗ b"
      hence "a ⊗ b ∈ PIdl p"
        by (meson assms DiffD1 cgenideal_self contra_subsetD m_closed
to_contain_is_to_divide)
      hence "a ∈ PIdl p ∨ b ∈ PIdl p"
        using primeideal.I_prime[OF PIdl a b] by simp
      thus "p divides a ∨ p divides b"
        using assms a b Idl_subset_ideal' cgenideal_eq_genideal to_contain_is_to_divide
by auto
    qed
  qed
next
  assume prime: "ring_prime p" show "primeideal (PIdl p) R"
  proof (rule primeidealI[OF cgenideal_ideal cring_axioms])
    show "p ∈ carrier R" and "carrier R ≠ PIdl p"
      using ideal_eq_carrier_iff[of p] assms prime
      unfolding ring_prime_def prime_def by auto
  next
    fix a b assume a: "a ∈ carrier R" and b: "b ∈ carrier R" "a ⊗ b
∈ PIdl p"
    hence "p divides a ⊗ b"
      using assms Idl_subset_ideal' cgenideal_eq_genideal to_contain_is_to_divide
by auto
    hence "p divides a ∨ p divides b"
      using a b assms primeE[OF ring_primeE(3)[OF _ prime]] by auto
    thus "a ∈ PIdl p ∨ b ∈ PIdl p"
      using a b assms Idl_subset_ideal' cgenideal_eq_genideal to_contain_is_to_divide
by auto
  qed
qed
```

## 31.7 Noetherian Rings

```
lemma (in ring) chain_Union_is_ideal:
  assumes "subset.chain { I. ideal I R } C"
  shows "ideal (if C = {} then { 0 } else (⋃C)) R"
proof (cases "C = {}")
```

```
  case True thus ?thesis by (simp add: zeroideal)
next
  case False have "ideal (⋃C) R"
  proof (rule idealI[OF ring_axioms])
    show "subgroup (⋃C) (add_monoid R)"
    proof
      show "⋃C ⊆ carrier (add_monoid R)"
        using assms subgroup.subset[OF additive_subgroup.a_subgroup]
        unfolding pred_on.chain_def ideal_def by auto

      obtain I where I: "I ∈ C" "ideal I R"
        using False assms unfolding pred_on.chain_def by auto
      thus "1_add_monoid R ∈ ⋃C"
        using additive_subgroup.zero_closed[OF ideal.axioms(1)[OF I(2)]]
by auto
    next
      fix x y assume "x ∈ ⋃C" "y ∈ ⋃C"
      then obtain I where I: "I ∈ C" "x ∈ I" "y ∈ I"
        using assms unfolding pred_on.chain_def by blast
      hence ideal: "ideal I R"
        using assms unfolding pred_on.chain_def by auto
      thus "x ⊗_add_monoid R y ∈ ⋃C"
        using UnionI I additive_subgroup.a_closed unfolding ideal_def
by fastforce

      show "inv_add_monoid R x ∈ ⋃C"
        using UnionI I additive_subgroup.a_inv_closed ideal unfolding
ideal_def a_inv_def by metis
    qed
  next
    fix a x assume a: "a ∈ ⋃C" and x: "x ∈ carrier R"
    then obtain I where I: "ideal I R" "a ∈ I" and "I ∈ C"
      using assms unfolding pred_on.chain_def by auto
    thus "x ⊗ a ∈ ⋃C" and "a ⊗ x ∈ ⋃C"
      using ideal.I_l_closed[OF I x] ideal.I_r_closed[OF I x] by auto
  qed
  thus ?thesis
    using False by simp
qed

lemma (in noetherian_ring) ideal_chain_is_trivial:
  assumes "C ≠ {}" "subset.chain { I. ideal I R } C"
  shows "⋃C ∈ C"
proof -
  have aux_lemma: "∃I. I ∈ C ∧ S ⊆ I" if "finite S" "S ⊆ ⋃C" for S
    using that
  proof (induct S)
    case empty
    thus ?case using assms(1) by blast
```

```
  next
    case (insert s S)
    then obtain I where I: "I ∈ C" "S ⊆ I"
      by blast
    moreover obtain I' where I': "I' ∈ C" "s ∈ I'"
      using insert(4) by blast
    ultimately have "I ⊆ I' ∨ I' ⊆ I"
      using assms unfolding pred_on.chain_def by blast
    thus ?case
      using I I' by blast
  qed

  obtain S where S: "finite S" "S ⊆ carrier R" "⋃C = Idl S"
    using finetely_gen[OF chain_Union_is_ideal[OF assms(2)]] assms(1)
by auto
  then obtain I where I: "I ∈ C" and "S ⊆ I"
    using aux_lemma[OF S(1)] genideal_self[OF S(2)] by blast
  hence "Idl S ⊆ I"
    using assms unfolding pred_on.chain_def
    by (metis genideal_minimal mem_Collect_eq rev_subsetD)
  hence "⋃C = I"
    using S(3) I by auto
  thus ?thesis
    using I by simp
qed

lemma (in ring) trivial_ideal_chain_imp_noetherian:
  assumes "⋀C. ⟦ C ≠ {}; subset.chain { I. ideal I R } C ⟧ ⟹ ⋃C ∈
C"
  shows "noetherian_ring R"
proof (rule noetherian_ringI)
  fix I assume I: "ideal I R"
  have in_carrier: "I ⊆ carrier R" and add_subgroup: "additive_subgroup
I R"
    using ideal.axioms(1)[OF I] additive_subgroup.a_subset by auto

  define S where "S = { Idl S' | S'. S' ⊆ I ∧ finite S' }"
  have "∃M ∈ S. ∀S' ∈ S. M ⊆ S' ⟶ S' = M"
  proof (rule subset_Zorn)
    fix C assume C: "subset.chain S C"
    show "∃U ∈ S. ∀S' ∈ C. S' ⊆ U"
    proof (cases "C = {}")
      case True
      have "{ 𝟬 } ∈ S"
        using additive_subgroup.zero_closed[OF add_subgroup] genideal_zero
        by (auto simp add: S_def)
      thus ?thesis
        using True by auto
    next
```

```
      case False
      have "S ⊆ { I. ideal I R }"
        using additive_subgroup.a_subset[OF add_subgroup] genideal_ideal
        by (auto simp add: S_def)
      hence "subset.chain { I. ideal I R } C"
        using C unfolding pred_on.chain_def by auto
      then have "⋃C ∈ C"
        using assms False by simp
      thus ?thesis
        by (meson C Union_upper pred_on.chain_def subsetCE)
    qed
  qed
  then obtain M where M: "M ∈ S" "⋀S'. ⟦S' ∈ S; M ⊆ S' ⟧ ⟹ S' = M"
    by auto
  then obtain S' where S': "S' ⊆ I" "finite S'" "M = Idl S'"
    by (auto simp add: S_def)
  hence "M ⊆ I"
    using I genideal_minimal by (auto simp add: S_def)
  moreover have "I ⊆ M"
  proof (rule ccontr)
    assume "¬ I ⊆ M"
    then obtain a where a: "a ∈ I" "a ∉ M"
      by auto
    have "M ⊆ Idl (insert a S')"
      using S' a(1) genideal_minimal[of "Idl (insert a S')" S']
            in_carrier genideal_ideal genideal_self
      by (meson insert_subset subset_trans)
    moreover have "Idl (insert a S') ∈ S"
      using a(1) S' by (auto simp add: S_def)
    ultimately have "M = Idl (insert a S')"
      using M(2) by auto
    hence "a ∈ M"
      using genideal_self S'(1) a (1) in_carrier by (meson insert_subset
subset_trans)
    from ‹a ∈ M› and ‹a ∉ M› show False by simp
  qed
  ultimately have "M = I" by simp
  thus "∃A ⊆ carrier R. finite A ∧ I = Idl A"
    using S' in_carrier by blast
qed

lemma (in noetherian_domain) factorization_property:
  assumes "a ∈ carrier R - { 𝟬 }" "a ∉ Units R"
  shows "∃fs. set fs ⊆ carrier (mult_of R) ∧ wfactors (mult_of R) fs
a" (is "?factorizable a")
proof (rule ccontr)
  assume a: "¬ ?factorizable a"
  define S where "S = { PIdl r | r. r ∈ carrier R - { 𝟬 } ∧ r ∉ Units
R ∧ ¬ ?factorizable r }"
```

```
  then obtain C where C: "subset.maxchain S C"
    using subset.Hausdorff by blast
  hence chain: "subset.chain S C"
    using pred_on.maxchain_def by blast
  moreover have "S ⊆ { I. ideal I R }"
    using cgenideal_ideal by (auto simp add: S_def)
  ultimately have "subset.chain { I. ideal I R } C"
    by (meson dual_order.trans pred_on.chain_def)
  moreover have "PIdl a ∈ S"
    using assms a by (auto simp add: S_def)
  hence "subset.chain S { PIdl a }"
    unfolding pred_on.chain_def by auto
  hence "C ≠ {}"
    using C unfolding pred_on.maxchain_def by auto
  ultimately have "⋃C ∈ C"
    using ideal_chain_is_trivial by simp
  hence "⋃C ∈ S"
    using chain unfolding pred_on.chain_def by auto
  then obtain r where r: "⋃C = PIdl r" "r ∈ carrier R - { 0 }" "r ∉
Units R" "¬ ?factorizable r"
    by (auto simp add: S_def)
  have "∃p. p ∈ carrier R - { 0 } ∧ p ∉ Units R ∧ ¬ ?factorizable p
∧ p divides r ∧ ¬ r divides p"
  proof -
    have "wfactors (mult_of R) [ r ] r" if "irreducible (mult_of R) r"
      using r(2) that unfolding wfactors_def by auto
    hence "¬ irreducible (mult_of R) r"
      using r(2,4) by auto
    hence "¬ ring_irreducible r"
      using ring_irreducibleE(3) r(2) by auto
    then obtain p1 p2
      where p1_p2: "p1 ∈ carrier R" "p2 ∈ carrier R" "r = p1 ⊗ p2" "p1
∉ Units R" "p2 ∉ Units R"
      using ring_irreducibleI[OF r(2-3)] by auto
    hence in_carrier: "p1 ∈ carrier (mult_of R)" "p2 ∈ carrier (mult_of
R)"
      using r(2) by auto

    have "⟦ ?factorizable p1; ?factorizable p2 ⟧ ⟹ ?factorizable r"
      using mult_of.wfactors_mult[OF _ _ in_carrier] p1_p2(3) by (metis
le_sup_iff set_append)
    hence "¬ ?factorizable p1 ∨ ¬ ?factorizable p2"
      using r(4) by auto

    moreover
    have "⋀p1 p2. ⟦ p1 ∈ carrier R; p2 ∈ carrier R; r = p1 ⊗ p2; r divides
p1 ⟧ ⟹ p2 ∈ Units R"
    proof -
      fix p1 p2 assume A: "p1 ∈ carrier R" "p2 ∈ carrier R" "r = p1 ⊗
```

```
p2" "r divides p1"
      then obtain c where c: "c ∈ carrier R" "p1 = r ⊗ c"
        unfolding factor_def by blast
      hence "1 = c ⊗ p2"
        using A m_lcancel[OF _ _ one_closed, of r "c ⊗ p2"] r(2) by (auto,
metis m_assoc m_closed)
      thus "p2 ∈ Units R"
        unfolding Units_def using c A(2) m_comm[OF c(1) A(2)] by auto
    qed
    hence "¬ r divides p1" and "¬ r divides p2"
      using p1_p2 m_comm[OF p1_p2(1-2)] by blast+

    ultimately show ?thesis
      using p1_p2 in_carrier by (metis carrier_mult_of dividesI' m_comm)
  qed
  then obtain p
    where p: "p ∈ carrier R - { 0 }" "p ∉ Units R" "¬ ?factorizable
p" "p divides r" "¬ r divides p"
    by blast
  hence "PIdl p ∈ S"
    unfolding S_def by auto
  moreover have "⋃C ⊂ PIdl p"
    using p r to_contain_is_to_divide unfolding r(1) by (metis Diff_iff
psubsetI)
  ultimately have "subset.chain S (insert (PIdl p) C)" and "C ⊂ (insert
(PIdl p) C)"
    unfolding pred_on.chain_def by (metis C psubsetE subset_maxchain_max,
blast)
  thus False
    using C unfolding pred_on.maxchain_def by blast
qed

lemma (in noetherian_domain) exists_irreducible_divisor:
  assumes "a ∈ carrier R - { 0 }" and "a ∉ Units R"
  obtains b where "b ∈ carrier R" and "ring_irreducible b" and "b divides
a"
proof -
  obtain fs where set_fs: "set fs ⊆ carrier (mult_of R)" and "wfactors
(mult_of R) fs a"
    using factorization_property[OF assms] by blast
  hence "a ∈ Units R" if "fs = []"
    using that assms(1) Units_cong assoc_iff_assoc_mult unfolding wfactors_def
by (simp, blast)
  hence "fs ≠ []"
    using assms(2) by auto
  then obtain f' fs' where fs: "fs = f' # fs'"
    using list.exhaust by blast
  from ‹wfactors (mult_of R) fs a› have "f' divides a"
    using mult_of.wfactors_dividesI[OF _ set_fs] assms(1) unfolding fs
```

```
by auto
  moreover from ‹wfactors (mult_of R) fs a› have "ring_irreducible f'"
and "f' ∈ carrier R"
    using set_fs ring_irreducibleI'[of f'] unfolding wfactors_def fs by
auto
  ultimately show thesis
    using that by blast
qed
```

## 31.8 Principal Domains

```
sublocale principal_domain ⊆ noetherian_domain
proof
  fix I assume "ideal I R"
  then obtain i where "i ∈ carrier R" "I = Idl { i }"
    using exists_gen cgenideal_eq_genideal by auto
  thus "∃A ⊆ carrier R. finite A ∧ I = Idl A"
    by blast
qed

lemma (in principal_domain) irreducible_imp_maximalideal:
  assumes "p ∈ carrier R"
    and "ring_irreducible p"
  shows "maximalideal (PIdl p) R"
proof (rule maximalidealI)
  show "ideal (PIdl p) R"
    using assms(1) by (simp add: cgenideal_ideal)
next
  show "carrier R ≠ PIdl p"
    using ideal_eq_carrier_iff[OF assms(1)] ring_irreducibleE(4)[OF assms]
by auto
next
  fix J assume J: "ideal J R" "PIdl p ⊆ J" "J ⊆ carrier R"
  then obtain q where q: "q ∈ carrier R" "J = PIdl q"
    using exists_gen[OF J(1)] cgenideal_eq_rcos by metis
  hence "q divides p"
    using to_contain_is_to_divide[of q p] using assms(1) J(1-2) by simp
  then obtain r where r: "r ∈ carrier R" "p = q ⊗ r"
    unfolding factor_def by auto
  hence "q ∈ Units R ∨ r ∈ Units R"
    using ring_irreducibleE(5)[OF assms q(1)] by auto
  thus "J = PIdl p ∨ J = carrier R"
  proof
    assume "q ∈ Units R" thus ?thesis
      using ideal_eq_carrier_iff[OF q(1)] q(2) by auto
  next
    assume "r ∈ Units R" hence "p ∼ q"
      using assms(1) r q(1) associatedI2' by blast
    thus ?thesis
```

```
      unfolding associated_iff_same_ideal[OF assms(1) q(1)] q(2) by auto
  qed
qed

corollary (in principal_domain) primeness_condition:
  assumes "p ∈ carrier R"
  shows "ring_irreducible p ⟷ ring_prime p"
proof
  show "ring_irreducible p ⟹ ring_prime p"
    using maximalideal_prime[OF irreducible_imp_maximalideal] ring_irreducibleE(1)
          primeideal_iff_prime assms by auto
next
  show "ring_prime p ⟹ ring_irreducible p"
    using mult_of.prime_irreducible ring_primeI[of p] ring_irreducibleI'
assms
    unfolding ring_prime_def prime_eq_prime_mult[OF assms] by auto
qed

lemma (in principal_domain) domain_iff_prime:
  assumes "a ∈ carrier R - { 0 }"
  shows "domain (R Quot (PIdl a)) ⟷ ring_prime a"
  using quot_domain_iff_primeideal[of "PIdl a"] primeideal_iff_prime[of
a]
        cgenideal_ideal[of a] assms by auto

lemma (in principal_domain) field_iff_prime:
  assumes "a ∈ carrier R - { 0 }"
  shows "field (R Quot (PIdl a)) ⟷ ring_prime a"
proof
  show "ring_prime a ⟹ field  (R Quot (PIdl a))"
    using  primeness_condition[of a] irreducible_imp_maximalideal[of a]
           maximalideal.quotient_is_field[of "PIdl a" R] is_cring assms
by auto
next
  show "field  (R Quot (PIdl a)) ⟹ ring_prime a"
    unfolding field_def using domain_iff_prime[of a] assms by auto
qed


sublocale principal_domain < mult_of: primeness_condition_monoid "mult_of
R"
  rewrites "mult (mult_of R) = mult R"
       and "one  (mult_of R) = one R"
    unfolding primeness_condition_monoid_def
              primeness_condition_monoid_axioms_def
proof (auto simp add: mult_of.is_comm_monoid_cancel)
  fix a assume a: "a ∈ carrier R" "a ≠ 0" "irreducible (mult_of R) a"
  show "prime (mult_of R) a"
    using primeness_condition[OF a(1)] irreducible_mult_imp_irreducible[OF
```

```
_ a(3)] a(1-2)
    unfolding ring_prime_def ring_irreducible_def prime_eq_prime_mult[OF
a(1)] by auto
qed

sublocale principal_domain < mult_of: factorial_monoid "mult_of R"
  rewrites "mult (mult_of R) = mult R"
       and "one  (mult_of R) = one R"
  using mult_of.wfactors_unique factorization_property mult_of.is_comm_monoid_cancel
  by (auto intro!: mult_of.factorial_monoidI)

sublocale principal_domain ⊆ factorial_domain
  unfolding factorial_domain_def using domain_axioms mult_of.factorial_monoid_axioms
by simp

lemma (in principal_domain) ideal_sum_iff_gcd:
  assumes "a ∈ carrier R" "b ∈ carrier R" "d ∈ carrier R"
  shows "PIdl d = PIdl a <+>R PIdl b ⟷ d gcdof a b"
proof -
  have aux_lemma: "d gcdof a b"
    if in_carrier: "a ∈ carrier R" "b ∈ carrier R" "d ∈ carrier R"
    and ideal_eq: "PIdl d = PIdl a <+>R PIdl b"
    for a b d
  proof (auto simp add: isgcd_def)
    have "a ∈ PIdl d" and "b ∈ PIdl d"
      using in_carrier(1-2)[THEN cgenideal_ideal] additive_subgroup.zero_closed[OF
ideal.axioms(1)]
            in_carrier(1-2)[THEN cgenideal_self] in_carrier(1-2)
      unfolding ideal_eq set_add_def' by force+
    thus "d divides a" and "d divides b"
      using in_carrier(1,2)[THEN to_contain_is_to_divide[OF in_carrier(3)]]
            cgenideal_minimal[OF cgenideal_ideal[OF in_carrier(3)]] by
simp+
  next
    fix c assume c: "c ∈ carrier R" "c divides a" "c divides b"
    hence "PIdl a ⊆ PIdl c" and "PIdl b ⊆ PIdl c"
      using to_contain_is_to_divide in_carrier by auto
    hence "PIdl a <+>R PIdl b ⊆ PIdl c"
      by (metis Un_subset_iff c(1) in_carrier(1-2) cgenideal_ideal genideal_minimal
union_genideal)
    thus "c divides d"
      using ideal_eq to_contain_is_to_divide[OF c(1) in_carrier(3)] by
simp
  qed

  have "PIdl d = PIdl a <+>R PIdl b ⟹ d gcdof a b"
    using aux_lemma assms by simp

  moreover
```

```
  have "d gcdof a b ⟹ PIdl d = PIdl a <+>R PIdl b"
  proof -
    assume d: "d gcdof a b"
    obtain c where c: "c ∈ carrier R" "PIdl c = PIdl a <+>R PIdl b"
      using exists_gen[OF add_ideals[OF assms(1-2)[THEN cgenideal_ideal]]]
by blast
    hence "c gcdof a b"
      using aux_lemma assms by simp
    from ‹d gcdof a b› and ‹c gcdof a b› have "d ∼ c"
      using assms c(1) by (simp add: associated_def isgcd_def)
    thus ?thesis
      using c(2) associated_iff_same_ideal[OF assms(3) c(1)] by simp
  qed

  ultimately show ?thesis by auto
qed

lemma (in principal_domain) bezout_identity:
  assumes "a ∈ carrier R" "b ∈ carrier R"
  shows "PIdl a <+>R PIdl b = PIdl (somegcd R a b)"
proof -
  have "∃d ∈ carrier R. d gcdof a b"
    using exists_gen[OF add_ideals[OF assms(1-2)[THEN cgenideal_ideal]]]
          ideal_sum_iff_gcd[OF assms(1-2)] by auto
  thus ?thesis
    using ideal_sum_iff_gcd[OF assms(1-2)] somegcd_def
    by (metis (no_types, lifting) tfl_some)
qed
```

## 31.9 Euclidean Domains

```
sublocale euclidean_domain ⊆ principal_domain
  unfolding principal_domain_def principal_domain_axioms_def
proof (auto)
  show "domain R" by (simp add: domain_axioms)
next
  fix I assume I: "ideal I R" have "principalideal I R"
  proof (cases "I = { 0 }")
    case True thus ?thesis by (simp add: zeropideal)
  next
    case False hence A: "I - { 0 } ≠ {}"
      using I additive_subgroup.zero_closed ideal.axioms(1) by auto
    define phi_img :: "nat set" where "phi_img = (φ ‘ (I - { 0 }))"
    hence "phi_img ≠ {}" using A by simp
    then obtain m where "m ∈ phi_img" "⋀k. k ∈ phi_img ⟹ m ≤ k"
      using exists_least_iff[of "λn. n ∈ phi_img"] not_less by force
    then obtain a where a: "a ∈ I - { 0 }" "⋀b. b ∈ I - { 0 } ⟹ φ
a ≤ φ b"
      using phi_img_def by blast
```

```
    have "I = PIdl a"
    proof (rule ccontr)
      assume "I ≠ PIdl a"
      then obtain b where b: "b ∈ I" "b ∉ PIdl a"
        using I ‹a ∈ I - {𝟬}› cgenideal_minimal by auto
      hence "b ≠ 𝟬"
        by (metis DiffD1 I a(1) additive_subgroup.zero_closed cgenideal_ideal
ideal.Icarr ideal.axioms(1))
      then obtain q r
        where eucl_div: "q ∈ carrier R" "r ∈ carrier R" "b = (a ⊗ q)
⊕ r" "r = 𝟬 ∨ φ r < φ a"
        using euclidean_function[of b a] a(1) b(1) ideal.Icarr[OF I] by
auto
      hence "r = 𝟬 ⟹ b ∈ PIdl a"
        unfolding cgenideal_def using m_comm[of a] ideal.Icarr[OF I] a(1)
by auto
      hence 1: "φ r < φ a ∧ r ≠ 𝟬"
        using eucl_div(4) b(2) by auto

      have "r = (⊖ (a ⊗ q)) ⊕ b"
        using eucl_div(1-3) a(1) b(1) ideal.Icarr[OF I] r_neg1 by auto
      moreover have "⊖ (a ⊗ q) ∈ I"
        using eucl_div(1) a(1) I
        by (meson DiffD1 additive_subgroup.a_inv_closed ideal.I_r_closed
ideal.axioms(1))
      ultimately have 2: "r ∈ I"
        using b(1) additive_subgroup.a_closed[OF ideal.axioms(1)[OF I]]
by auto

      from 1 and 2 show False
        using a(2) by fastforce
    qed
    thus ?thesis
      by (meson DiffD1 I cgenideal_is_principalideal ideal.Icarr local.a(1))
  qed
  thus "∃a ∈ carrier R. I = PIdl a"
    by (simp add: cgenideal_eq_genideal principalideal.generate)
qed


sublocale field ⊆ euclidean_domain R "λ_. 0"
proof (rule euclidean_domainI)
  fix a b
  let ?eucl_div = "λq r. q ∈ carrier R ∧ r ∈ carrier R ∧ a = b ⊗ q ⊕
r ∧ (r = 𝟬 ∨ 0 < 0)"

  assume a: "a ∈ carrier R - { 𝟬 }" and b: "b ∈ carrier R - { 𝟬 }"
  hence "a = b ⊗ ((inv b) ⊗ a) ⊕ 𝟬"
    by (metis DiffD1 Units_inv_closed Units_r_inv field_Units l_one m_assoc
```

```
r_zero)
  hence "?eucl_div _ ((inv b) ⊗ a) 0"
    using a b field_Units by auto
  thus "∃q r. ?eucl_div _ q r"
    by blast
qed

end


theory Subrings
  imports Ring RingHom QuotRing Multiplicative_Group
begin
```

# 32 Subrings

## 32.1 Definitions

```
locale subring =
  subgroup H "add_monoid R" + submonoid H R for H and R (structure)

locale subcring = subring +
  assumes sub_m_comm: "⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗ h2 = h2 ⊗ h1"

locale subdomain = subcring +
  assumes sub_one_not_zero [simp]: "1 ≠ 0"
  assumes subintegral: "⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗ h2 = 0 ⟹ h1 = 0
∨ h2 = 0"

locale subfield = subdomain K R for K and R (structure) +
  assumes subfield_Units: "Units (R ⦇ carrier := K ⦈) = K - { 0 }"
```

## 32.2 Basic Properties

### 32.2.1 Subrings

```
lemma (in ring) subringI:
  assumes "H ⊆ carrier R"
    and "1 ∈ H"
    and "⋀h. h ∈ H ⟹ ⊖ h ∈ H"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗ h2 ∈ H"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊕ h2 ∈ H"
  shows "subring H R"
  using add.subgroupI[OF assms(1) _ assms(3, 5)] assms(2)
        submonoid.intro[OF assms(1, 4, 2)]
  unfolding subring_def by auto

lemma subringE:
  assumes "subring H R"
  shows "H ⊆ carrier R"
```

```
    and "𝟬R ∈ H"
    and "𝟭R ∈ H"
    and "H ≠ {}"
    and "⋀h. h ∈ H ⟹ ⊖R h ∈ H"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗R h2 ∈ H"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊕R h2 ∈ H"
  using subring.axioms[OF assms]
  unfolding submonoid_def subgroup_def a_inv_def by auto

lemma (in ring) carrier_is_subring: "subring (carrier R) R"
  by (simp add: subringI)

lemma (in ring) subring_inter:
  assumes "subring I R" and "subring J R"
  shows "subring (I ∩ J) R"
  using subringE[OF assms(1)] subringE[OF assms(2)] subringI[of "I ∩ J"]
by auto

lemma (in ring) subring_Inter:
  assumes "⋀I. I ∈ S ⟹ subring I R" and "S ≠ {}"
  shows "subring (⋂S) R"
proof (rule subringI, auto simp add: assms subringE[of _ R])
  fix x assume "∀I ∈ S. x ∈ I" thus "x ∈ carrier R"
    using assms subringE(1)[of _ R] by blast
qed

lemma (in ring) subring_is_ring:
  assumes "subring H R" shows "ring (R ⦇ carrier := H ⦈)"
proof -
  interpret group "add_monoid (R ⦇ carrier := H ⦈)" + monoid "R ⦇ carrier
:= H ⦈"
    using subgroup.subgroup_is_group[OF subring.axioms(1) add.is_group]
assms
          submonoid.submonoid_is_monoid[OF subring.axioms(2) monoid_axioms]
by auto
  show ?thesis
    using subringE(1)[OF assms]
    by (unfold_locales, simp_all add: subringE(1)[OF assms] add.m_comm
subset_eq l_distr r_distr)
qed

lemma (in ring) ring_incl_imp_subring:
  assumes "H ⊆ carrier R"
    and "ring (R ⦇ carrier := H ⦈)"
  shows "subring H R"
  using group.group_incl_imp_subgroup[OF add.group_axioms, of H] assms(1)
        monoid.monoid_incl_imp_submonoid[OF monoid_axioms assms(1)]
        ring.axioms(1, 2)[OF assms(2)] abelian_group.a_group[of "R ⦇ carrier
:= H ⦈"]
```

```
  unfolding subring_def by auto

lemma (in ring) subring_iff:
  assumes "H ⊆ carrier R"
  shows "subring H R ⟷ ring (R ⦇ carrier := H ⦈)"
  using subring_is_ring ring_incl_imp_subring[OF assms] by auto
```

### 32.2.2 Subcrings

```
lemma (in ring) subcringI:
  assumes "subring H R"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗ h2 = h2 ⊗ h1"
  shows "subcring H R"
  unfolding subcring_def subcring_axioms_def using assms by simp+

lemma (in cring) subcringI':
  assumes "subring H R"
  shows "subcring H R"
  using subcringI[OF assms] subringE(1)[OF assms] m_comm by auto

lemma subcringE:
  assumes "subcring H R"
  shows "H ⊆ carrier R"
    and "𝟬R ∈ H"
    and "𝟭R ∈ H"
    and "H ≠ {}"
    and "⋀h. h ∈ H ⟹ ⊖R h ∈ H"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗R h2 ∈ H"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊕R h2 ∈ H"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗R h2 = h2 ⊗R h1"
  using subringE[OF subcring.axioms(1)[OF assms]] subcring.sub_m_comm[OF
assms] by simp+

lemma (in cring) carrier_is_subcring: "subcring (carrier R) R"
  by (simp add: subcringI' carrier_is_subring)

lemma (in ring) subcring_inter:
  assumes "subcring I R" and "subcring J R"
  shows "subcring (I ∩ J) R"
  using subcringE[OF assms(1)] subcringE[OF assms(2)]
        subcringI[of "I ∩ J"] subringI[of "I ∩ J"] by auto

lemma (in ring) subcring_Inter:
  assumes "⋀I. I ∈ S ⟹ subcring I R" and "S ≠ {}"
  shows "subcring (⋂S) R"
proof (rule subcringI)
  show "subring (⋂S) R"
    using subcring.axioms(1)[of _ R] subring_Inter[of S] assms by auto
next
```

```
  fix h1 h2 assume h1: "h1 ∈ ⋂S" and h2: "h2 ∈ ⋂S"
  obtain S' where S': "S' ∈ S"
    using assms(2) by blast
  hence "h1 ∈ S'" "h2 ∈ S'"
    using h1 h2 by blast+
  thus "h1 ⊗ h2 = h2 ⊗ h1"
    using subcring.sub_m_comm[OF assms(1)[OF S']] by simp
qed

lemma (in ring) subcring_iff:
  assumes "H ⊆ carrier R"
  shows "subcring H R ⟷ cring (R ⦇ carrier := H ⦈)"
proof
  assume A: "subcring H R"
  hence ring: "ring (R ⦇ carrier := H ⦈)"
    using subring_iff[OF assms] subcring.axioms(1)[OF A] by simp
  moreover have "comm_monoid (R ⦇ carrier := H ⦈)"
    using monoid.monoid_comm_monoidI[OF ring.is_monoid[OF ring]]
          subcring.sub_m_comm[OF A] by auto
  ultimately show "cring (R ⦇ carrier := H ⦈)"
    using cring_def by blast
next
  assume A: "cring (R ⦇ carrier := H ⦈)"
  hence "subring H R"
    using cring.axioms(1) subring_iff[OF assms] by simp
  moreover have "comm_monoid (R ⦇ carrier := H ⦈)"
    using A unfolding cring_def by simp
  hence"⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗ h2 = h2 ⊗ h1"
    using comm_monoid.m_comm[of "R ⦇ carrier := H ⦈"] by auto
  ultimately show "subcring H R"
    unfolding subcring_def subcring_axioms_def by auto
qed
```

### 32.2.3 Subdomains

```
lemma (in ring) subdomainI:
  assumes "subcring H R"
    and "1 ≠ 0"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗ h2 = 0 ⟹ h1 = 0 ∨ h2
= 0"
  shows "subdomain H R"
  unfolding subdomain_def subdomain_axioms_def using assms by simp+

lemma (in domain) subdomainI':
  assumes "subring H R"
  shows "subdomain H R"
proof (rule subdomainI[OF subcringI[OF assms]], simp_all)
  show "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗ h2 = h2 ⊗ h1"
    using m_comm subringE(1)[OF assms] by auto
```

```
  show "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H; h1 ⊗ h2 = 0 ⟧ ⟹ (h1 = 0) ∨ (h2 =
0)"
    using integral subringE(1)[OF assms] by auto
qed

lemma subdomainE:
  assumes "subdomain H R"
  shows "H ⊆ carrier R"
    and "0R ∈ H"
    and "1R ∈ H"
    and "H ≠ {}"
    and "⋀h. h ∈ H ⟹ ⊖R h ∈ H"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗R h2 ∈ H"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊕R h2 ∈ H"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗R h2 = h2 ⊗R h1"
    and "⋀h1 h2. ⟦ h1 ∈ H; h2 ∈ H ⟧ ⟹ h1 ⊗R h2 = 0R ⟹ h1 = 0R ∨
h2 = 0R"
    and "1R ≠ 0R"
  using subcringE[OF subdomain.axioms(1)[OF assms]] assms
  unfolding subdomain_def subdomain_axioms_def by auto

lemma (in ring) subdomain_iff:
  assumes "H ⊆ carrier R"
  shows "subdomain H R ⟷ domain (R ⦇ carrier := H ⦈)"
proof
  assume A: "subdomain H R"
  hence cring: "cring (R ⦇ carrier := H ⦈)"
    using subcring_iff[OF assms] subdomain.axioms(1)[OF A] by simp
  thus "domain (R ⦇ carrier := H ⦈)"
    using domain.intro[OF cring] subdomain.subintegral[OF A] subdomain.sub_one_not_zero[OF
A]
    unfolding domain_axioms_def by auto
next
  assume A: "domain (R ⦇ carrier := H ⦈)"
  hence subcring: "subcring H R"
    using subcring_iff[OF assms] unfolding domain_def by simp
  thus "subdomain H R"
    using subdomain.intro[OF subcring] domain.integral[OF A] domain.one_not_zero[OF
A]
    unfolding subdomain_axioms_def by auto
qed

lemma (in domain) subring_is_domain:
  assumes "subring H R" shows "domain (R ⦇ carrier := H ⦈)"
  using subdomainI'[OF assms] unfolding subdomain_iff[OF subringE(1)[OF
assms]] .


lemma (in ring) subdomain_is_domain:
```

```
  assumes "subdomain H R" shows "domain (R ⦇ carrier := H ⦈)"
  using assms unfolding subdomain_iff[OF subdomainE(1)[OF assms]] .
```

### 32.2.4 Subfields

```
lemma (in ring) subfieldI:
  assumes "subcring K R" and "Units (R ⦇ carrier := K ⦈) = K - { 𝟬 }"
  shows "subfield K R"
proof (rule subfield.intro)
  show "subfield_axioms K R"
    using assms(2) unfolding subfield_axioms_def .
  show "subdomain K R"
  proof (rule subdomainI[OF assms(1)], auto)
    have subM: "submonoid K R"
      using subring.axioms(2)[OF subcring.axioms(1)[OF assms(1)]] .

    show contr: "𝟭 = 𝟬 ⟹ False"
    proof -
      assume one_eq_zero: "𝟭 = 𝟬"
      have "𝟭 ∈ K" and "𝟭 ⊗ 𝟭 = 𝟭"
        using submonoid.one_closed[OF subM] by simp+
      hence "𝟭 ∈ Units (R ⦇ carrier := K ⦈)"
        unfolding Units_def by (simp, blast)
      hence "𝟭 ≠ 𝟬"
        using assms(2) by simp
      thus False
        using one_eq_zero by simp
    qed

    fix k1 k2 assume k1: "k1 ∈ K" and k2: "k2 ∈ K" "k2 ≠ 𝟬" and k12:
"k1 ⊗ k2 = 𝟬"
    obtain k2' where k2': "k2' ∈ K" "k2' ⊗ k2 = 𝟭" "k2 ⊗ k2' = 𝟭"
      using assms(2) k2 unfolding Units_def by auto
    have  "𝟬 = (k1 ⊗ k2) ⊗ k2'"
      using k12 k2'(1) submonoid.mem_carrier[OF subM] by fastforce
    also have  "... = k1"
      using k1 k2(1) k2'(1,3) submonoid.mem_carrier[OF subM] by (simp
add: m_assoc)
    finally have "𝟬 = k1" .
    thus "k1 = 𝟬" by simp
  qed
qed

lemma (in field) subfieldI':
  assumes "subring K R" and "⋀k. k ∈ K - { 𝟬 } ⟹ inv k ∈ K"
  shows "subfield K R"
proof (rule subfieldI)
  show "subcring K R"
    using subcringI[OF assms(1)] m_comm subringE(1)[OF assms(1)] by auto
```

```
  show "Units (R (| carrier := K |)) = K - { 0 }"
  proof
    show "K - { 0 } ⊆ Units (R (| carrier := K |))"
    proof
      fix k assume k: "k ∈ K - { 0 }"
      hence inv_k: "inv k ∈ K"
        using assms(2) by simp
      moreover have "k ∈ carrier R - { 0 }"
        using subringE(1)[OF assms(1)] k by auto
      ultimately have "k ⊗ inv k = 1" "inv k ⊗ k = 1"
        by (simp add: field_Units)+
      thus "k ∈ Units (R (| carrier := K |))"
        unfolding Units_def using k inv_k by auto
    qed
  next
    show "Units (R (| carrier := K |)) ⊆ K - { 0 }"
    proof
      fix k assume k: "k ∈ Units (R (| carrier := K |))"
      then obtain k' where k': "k' ∈ K" "k ⊗ k' = 1"
        unfolding Units_def by auto
      hence "k ∈ carrier R" and "k' ∈ carrier R"
        using k subringE(1)[OF assms(1)] unfolding Units_def by auto
      hence "0 = 1" if "k = 0"
        using that k'(2) by auto
      thus "k ∈ K - { 0 }"
        using k unfolding Units_def by auto
    qed
  qed
qed

lemma (in field) carrier_is_subfield: "subfield (carrier R) R"
  by (auto intro: subfieldI[OF carrier_is_subcring] simp add: field_Units)

lemma subfieldE:
  assumes "subfield K R"
  shows "subring K R" and "subcring K R"
    and "K ⊆ carrier R"
    and "⋀k1 k2. ⟦ k1 ∈ K; k2 ∈ K ⟧ ⟹ k1 ⊗R k2 = k2 ⊗R k1"
    and "⋀k1 k2. ⟦ k1 ∈ K; k2 ∈ K ⟧ ⟹ k1 ⊗R k2 = 0R ⟹ k1 = 0R ∨
k2 = 0R"
    and "1R ≠ 0R"
  using subdomain.axioms(1)[OF subfield.axioms(1)[OF assms]] subcring_def
        subdomainE(1, 8, 9, 10)[OF subfield.axioms(1)[OF assms]] by auto

lemma (in ring) subfield_m_inv:
  assumes "subfield K R" and "k ∈ K - { 0 }"
  shows "inv k ∈ K - { 0 }" and "k ⊗ inv k = 1" and "inv k ⊗ k = 1"
proof -
  have K: "subring K R" "submonoid K R"
```

```
    using subfieldE(1)[OF assms(1)] subring.axioms(2) by auto
  have monoid: "monoid (R ⦇ carrier := K ⦈)"
    using submonoid.submonoid_is_monoid[OF subring.axioms(2)[OF K(1)]
is_monoid] .

  have "monoid R"
    by (simp add: monoid_axioms)

  hence k: "k ∈ Units (R ⦇ carrier := K ⦈)"
    using subfield.subfield_Units[OF assms(1)] assms(2) by blast
  hence unit_of_R: "k ∈ Units R"
    using assms(2) subringE(1)[OF subfieldE(1)[OF assms(1)]] unfolding
Units_def by auto
  have "inv(R ⦇ carrier := K ⦈) k ∈ Units (R ⦇ carrier := K ⦈)"
    by (simp add: k monoid monoid.Units_inv_Units)
  hence "inv(R ⦇ carrier := K ⦈) k ∈ K - { 𝟬 }"
    using subfield.subfield_Units[OF assms(1)] by blast
  thus "inv k ∈ K - { 𝟬 }" and "k ⊗ inv k = 𝟭" and "inv k ⊗ k = 𝟭"
    using Units_l_inv[OF unit_of_R] Units_r_inv[OF unit_of_R]
    using monoid.m_inv_monoid_consistent[OF monoid_axioms k K(2)] by auto
qed

lemma (in ring) subfield_m_inv_simprule:
  assumes "subfield K R"
  shows "⟦ k ∈ K - { 𝟬 }; a ∈ carrier R ⟧ ⟹ k ⊗ a ∈ K ⟹ a ∈ K"
proof -
  note subring_props = subringE[OF subfieldE(1)[OF assms]]

  assume A: "k ∈ K - { 𝟬 }" "a ∈ carrier R" "k ⊗ a ∈ K"
  then obtain k' where k': "k' ∈ K" "k ⊗ a = k'" by blast
  have inv_k: "inv k ∈ K" "inv k ⊗ k = 𝟭"
    using subfield_m_inv[OF assms A(1)] by auto
  hence "inv k ⊗ (k ⊗ a) ∈ K"
    using k' A(3) subring_props(6) by auto
  thus "a ∈ K"
    using m_assoc[of "inv k" k a] A(2) inv_k subring_props(1)
    by (metis (no_types, opaque_lifting) A(1) Diff_iff l_one subsetCE)
qed

lemma (in ring) subfield_iff:
  shows "⟦ field (R ⦇ carrier := K ⦈); K ⊆ carrier R ⟧ ⟹ subfield K
R"
    and "subfield K R ⟹ field (R ⦇ carrier := K ⦈)"
proof-
  assume A: "field (R ⦇ carrier := K ⦈)" "K ⊆ carrier R"
  have "⋀k1 k2. ⟦ k1 ∈ K; k2 ∈ K ⟧ ⟹ k1 ⊗ k2 = k2 ⊗ k1"
    using comm_monoid.m_comm[OF cring.axioms(2)[OF fieldE(1)[OF A(1)]]]
by simp
  moreover have "subring K R"
```

```
    using ring_incl_imp_subring[OF A(2) cring.axioms(1)[OF fieldE(1)[OF
A(1)]]] .
  ultimately have "subcring K R"
    using subcringI by simp
  thus "subfield K R"
    using field.field_Units[OF A(1)] subfieldI by auto
next
  assume A: "subfield K R"
  have cring: "cring (R (| carrier := K |))"
    using subcring_iff[OF subringE(1)[OF subfieldE(1)[OF A]]] subfieldE(2)[OF
A] by simp
  thus "field (R (| carrier := K |))"
    using cring.cring_fieldI[OF cring] subfield.subfield_Units[OF A] by
simp
qed

lemma (in field) subgroup_mult_of :
  assumes "subfield K R"
  shows "subgroup (K - {0}) (mult_of R)"
proof (intro group.group_incl_imp_subgroup[OF field_mult_group])
  show "K - {0} ⊆ carrier (mult_of R)"
    by (simp add: Diff_mono assms carrier_mult_of subfieldE(3))
  show "group ((mult_of R) (| carrier := K - {0} |))"
    using field.field_mult_group[OF subfield_iff(2)[OF assms]]
    unfolding mult_of_def by simp
qed
```

## 32.3 Subring Homomorphisms

```
lemma (in ring) hom_imp_img_subring:
  assumes "h ∈ ring_hom R S" and "subring K R"
  shows "ring (S (| carrier := h ` K, one := h 1, zero := h 0 |))"
proof -
  have [simp]: "h 1 = 1_S"
    using assms ring_hom_one by blast
  have "ring (R (| carrier := K |))"
    by (simp add: assms(2) subring_is_ring)
  moreover have "h ∈ ring_hom (R (| carrier := K |)) S"
    using assms subringE(1)[OF assms (2)] unfolding ring_hom_def
    apply simp
    apply blast
    done
  ultimately show ?thesis
    using ring.ring_hom_imp_img_ring[of "R (| carrier := K |)" h S] by simp
qed

lemma (in ring_hom_ring) img_is_subring:
  assumes "subring K R" shows "subring (h ` K) S"
proof -
```

```
  have "ring (S ⦇ carrier := h ‘ K ⦈)"
    using R.hom_imp_img_subring[OF homh assms] hom_zero hom_one by simp
  moreover have "h ‘ K ⊆ carrier S"
    using ring_hom_memE(1)[OF homh] subringE(1)[OF assms] by auto
  ultimately show ?thesis
    using ring_incl_imp_subring by simp
qed

lemma (in ring_hom_ring) img_is_subfield:
  assumes "subfield K R" and "𝟭S ≠ 𝟬S"
  shows "inj_on h K" and "subfield (h ‘ K) S"
proof -
  have K: "K ⊆ carrier R" "subring K R" "subring (h ‘ K) S"
    using subfieldE(1)[OF assms(1)] subringE(1) img_is_subring by auto
  have field: "field (R ⦇ carrier := K ⦈)"
    using R.subfield_iff(2) ‹subfield K R› by blast
  moreover have ring: "ring (R ⦇ carrier := K ⦈)"
    using K R.ring_axioms R.subring_is_ring by blast
  moreover have ringS: "ring (S ⦇ carrier := h ‘ K ⦈)"
    using subring_is_ring K by simp
  ultimately have h: "h ∈ ring_hom (R ⦇ carrier := K ⦈) (S ⦇ carrier :=
h ‘ K ⦈)"
    unfolding ring_hom_def apply auto
    using ring_hom_memE[OF homh] K
    by (meson contra_subsetD)+
  hence ring_hom: "ring_hom_ring (R ⦇ carrier := K ⦈) (S ⦇ carrier :=
h ‘ K ⦈) h"
    using ring_axioms ring ringS ring_hom_ringI2 by blast
  have "h ‘ K ≠ { 𝟬S }"
    using subfieldE(1, 5)[OF assms(1)] subringE(3) assms(2)
    by (metis hom_one image_eqI singletonD)
  thus "inj_on h K"
    using ring_hom_ring.non_trivial_field_hom_imp_inj[OF ring_hom field]
by auto

  hence "h ∈ ring_iso (R ⦇ carrier := K ⦈) (S ⦇ carrier := h ‘ K ⦈)"
    using h unfolding ring_iso_def bij_betw_def by auto
  hence "field (S ⦇ carrier := h ‘ K ⦈)"
    using field.ring_iso_imp_img_field[OF field, of h "S ⦇ carrier :=
h ‘ K ⦈"] by auto
  thus "subfield (h ‘ K) S"
    using S.subfield_iff[of "h ‘ K"] K(1) ring_hom_memE(1)[OF homh] by
blast
qed


lemma (in ring_hom_ring) induced_ring_hom:
  assumes "subring K R" shows "ring_hom_ring (R ⦇ carrier := K ⦈) S h"
proof -
```

```
  have "h ∈ ring_hom (R ⦇ carrier := K ⦈) S"
    using homh subringE(1)[OF assms] unfolding ring_hom_def
    by (auto, meson hom_mult hom_add subsetCE)+
  thus ?thesis
    using R.subring_is_ring[OF assms] ring_axioms
    unfolding ring_hom_ring_def ring_hom_ring_axioms_def by auto
qed


lemma (in ring_hom_ring) inj_on_subgroup_iff_trivial_ker:
  assumes "subring K R"
  shows "inj_on h K ⟷ a_kernel (R ⦇ carrier := K ⦈) S h = { 𝟬 }"
  using ring_hom_ring.inj_iff_trivial_ker[OF induced_ring_hom[OF assms]]
by simp

lemma (in ring_hom_ring) inv_ring_hom:
  assumes "inj_on h K" and "subring K R"
  shows "ring_hom_ring (S ⦇ carrier := h ‘ K ⦈) R (inv_into K h)"
proof (intro ring_hom_ringI[OF _ R.ring_axioms], auto)
  show "ring (S ⦇ carrier := h ‘ K ⦈)"
    using subring_is_ring[OF img_is_subring[OF assms(2)]] .
next
  show "inv_into K h 𝟭_S = 𝟭_R"
    using assms(1) subringE(3)[OF assms(2)] hom_one by (simp add: inv_into_f_eq)
next
  fix k1 k2
  assume k1: "k1 ∈ K" and k2: "k2 ∈ K"
  with ‹k1 ∈ K› show "inv_into K h (h k1) ∈ carrier R"
    using assms(1) subringE(1)[OF assms(2)] by (simp add: subset_iff)

  from ‹k1 ∈ K› and ‹k2 ∈ K›
  have "h k1 ⊕_S h k2 = h (k1 ⊕_R k2)" and "k1 ⊕_R k2 ∈ K"
   and "h k1 ⊗_S h k2 = h (k1 ⊗_R k2)" and "k1 ⊗_R k2 ∈ K"
    using subringE(1,6,7)[OF assms(2)] by (simp add: subset_iff)+
  thus "inv_into K h (h k1 ⊕_S h k2) = inv_into K h (h k1) ⊕_R inv_into
K h (h k2)"
   and "inv_into K h (h k1 ⊗_S h k2) = inv_into K h (h k1) ⊗_R inv_into
K h (h k2)"
    using assms(1) k1 k2 by simp+
qed

end


theory Polynomials
  imports Ring Ring_Divisibility Subrings

begin
```

# 33 Polynomials

## 33.1 Definitions

```
abbreviation lead_coeff :: "'a list ⇒ 'a"
  where "lead_coeff ≡ hd"

abbreviation degree :: "'a list ⇒ nat"
  where "degree p ≡ length p - 1"

definition polynomial :: "_ ⇒ 'a set ⇒ 'a list ⇒ bool" (‹polynomialı ›)
  where "polynomial_R K p ⟷ p = [] ∨ (set p ⊆ K ∧ lead_coeff p ≠ 𝟬_R)"

definition (in ring) monom :: "'a ⇒ nat ⇒ 'a list"
  where "monom a n = a # (replicate n 𝟬_R)"

fun (in ring) eval :: "'a list ⇒ 'a ⇒ 'a"
  where
    "eval [] = (λ_. 𝟬)"
  | "eval p = (λx. ((lead_coeff p) ⊗ (x [^] (degree p))) ⊕ (eval (tl
p) x))"

fun (in ring) coeff :: "'a list ⇒ nat ⇒ 'a"
  where
    "coeff [] = (λ_. 𝟬)"
  | "coeff p = (λi. if i = degree p then lead_coeff p else (coeff (tl
p)) i)"

fun (in ring) normalize :: "'a list ⇒ 'a list"
  where
    "normalize [] = []"
  | "normalize p = (if lead_coeff p ≠ 𝟬 then p else normalize (tl p))"

fun (in ring) poly_add :: "'a list ⇒ 'a list ⇒ 'a list"
  where "poly_add p1 p2 =
          (if length p1 ≥ length p2
           then normalize (map2 (⊕) p1 ((replicate (length p1 - length
p2) 𝟬) @ p2))
           else poly_add p2 p1)"

fun (in ring) poly_mult :: "'a list ⇒ 'a list ⇒ 'a list"
  where
    "poly_mult [] p2 = []"
  | "poly_mult p1 p2 =
       poly_add ((map (λa. lead_coeff p1 ⊗ a) p2) @ (replicate (degree
p1) 𝟬)) (poly_mult (tl p1) p2)"

fun (in ring) dense_repr :: "'a list ⇒ ('a × nat) list"
  where
    "dense_repr [] = []"
```

```
  | "dense_repr p = (if lead_coeff p ≠ 0
                     then (lead_coeff p, degree p) # (dense_repr (tl p))
                     else (dense_repr (tl p)))"
```

**fun** (**in** ring) poly_of_dense :: "('a × nat) list ⇒ 'a list"
  **where** "poly_of_dense dl = foldr (λ(a, n) l. poly_add (monom a n) l)
dl []"

**definition** (**in** ring) poly_of_const :: "'a ⇒ 'a list"
  **where** "poly_of_const = (λk. normalize [ k ])"

## 33.2 Basic Properties

**context** ring
**begin**

**lemma** polynomialI [intro]: "⟦ set p ⊆ K; lead_coeff p ≠ **0** ⟧ ⟹ polynomial
K p"
  **unfolding** polynomial_def **by** auto

**lemma** polynomial_incl: "polynomial K p ⟹ set p ⊆ K"
  **unfolding** polynomial_def **by** auto

**lemma** monom_in_carrier [intro]: "a ∈ carrier R ⟹ set (monom a n) ⊆
carrier R"
  **unfolding** monom_def **by** auto

**lemma** lead_coeff_not_zero: "polynomial K (a # p) ⟹ a ∈ K - { **0** }"
  **unfolding** polynomial_def **by** simp

**lemma** zero_is_polynomial [intro]: "polynomial K []"
  **unfolding** polynomial_def **by** simp

**lemma** const_is_polynomial [intro]: "a ∈ K - { **0** } ⟹ polynomial K [
a ]"
  **unfolding** polynomial_def **by** auto

**lemma** normalize_gives_polynomial: "set p ⊆ K ⟹ polynomial K (normalize
p)"
  **by** (induction p) (auto simp add: polynomial_def)

**lemma** normalize_in_carrier: "set p ⊆ carrier R ⟹ set (normalize p)
⊆ carrier R"
  **by** (induction p) (auto)

**lemma** normalize_polynomial: "polynomial K p ⟹ normalize p = p"
  **unfolding** polynomial_def **by** (cases p) (auto)

**lemma** normalize_idem: "normalize ((normalize p) @ q) = normalize (p @

```
q)"
  by (induct p) (auto)

lemma normalize_length_le: "length (normalize p) ≤ length p"
  by (induction p) (auto)

lemma eval_in_carrier: "⟦ set p ⊆ carrier R; x ∈ carrier R ⟧ ⟹ (eval
p) x ∈ carrier R"
  by (induction p) (auto)

lemma coeff_in_carrier [simp]: "set p ⊆ carrier R ⟹ (coeff p) i ∈
carrier R"
  by (induction p) (auto)

lemma lead_coeff_simp [simp]: "p ≠ [] ⟹ (coeff p) (degree p) = lead_coeff
p"
  by (metis coeff.simps(2) list.exhaust_sel)

lemma coeff_list: "map (coeff p) (rev [0..< length p]) = p"
proof (induction p)
  case Nil thus ?case by simp
next
  case (Cons a p)
  have "map (coeff (a # p)) (rev [0..<length (a # p)]) =
         a # (map (coeff p) (rev [0..<length p]))"
    by auto
  also have " ... = a # p"
    using Cons by simp
  finally show ?case .
qed

lemma coeff_nth: "i < length p ⟹ (coeff p) i = p ! (length p - 1 -
i)"
proof -
  assume i_lt: "i < length p"
  hence "(coeff p) i = (map (coeff p) [0..< length p]) ! i"
    by simp
  also have " ... = (rev (map (coeff p) (rev [0..< length p]))) ! i"
    by (simp add: rev_map)
  also have " ... = (map (coeff p) (rev [0..< length p])) ! (length p
- 1 - i)"
    using coeff_list i_lt rev_nth by auto
  also have " ... = p ! (length p - 1 - i)"
    using coeff_list[of p] by simp
  finally show "(coeff p) i = p ! (length p - 1 - i)" .
qed

lemma coeff_iff_length_cond:
  assumes "length p1 = length p2"
```

```
  shows "p1 = p2 ⟷ coeff p1 = coeff p2"
proof
  show "p1 = p2 ⟹ coeff p1 = coeff p2"
    by simp
next
  assume A: "coeff p1 = coeff p2"
  have "p1 = map (coeff p1) (rev [0..< length p1])"
    using coeff_list[of p1] by simp
  also have " ... = map (coeff p2) (rev [0..< length p2])"
    using A assms by simp
  also have " ... = p2"
    using coeff_list[of p2] by simp
  finally show "p1 = p2" .
qed

lemma coeff_img_restrict: "(coeff p) ` {..< length p} = set p"
  using coeff_list[of p] by (metis atLeast_upt image_set set_rev)

lemma coeff_length: "⋀i. i ≥ length p ⟹ (coeff p) i = 𝟬"
  by (induction p) (auto)

lemma coeff_degree: "⋀i. i > degree p ⟹ (coeff p) i = 𝟬"
  using coeff_length by (simp)

lemma replicate_zero_coeff [simp]: "coeff (replicate n 𝟬) = (λ_. 𝟬)"
  by (induction n) (auto)

lemma scalar_coeff: "a ∈ carrier R ⟹ coeff (map (λb. a ⊗ b) p) = (λi.
a ⊗ (coeff p) i)"
  by (induction p) (auto)

lemma monom_coeff: "coeff (monom a n) = (λi. if i = n then a else 𝟬)"
  unfolding monom_def by (induction n) (auto)

lemma coeff_img:
  "(coeff p) ` {..< length p} = set p"
  "(coeff p) ` { length p ..} = { 𝟬 }"
  "(coeff p) ` UNIV = (set p) ∪ { 𝟬 }"
  using coeff_img_restrict
proof (simp)
  show coeff_img_up: "(coeff p) ` { length p ..} = { 𝟬 }"
    using coeff_length[of p] by force
  from coeff_img_up and coeff_img_restrict[of p]
  show "(coeff p) ` UNIV = (set p) ∪ { 𝟬 }"
    by force
qed

lemma degree_def':
  assumes "polynomial K p"
```

```
  shows "degree p = (LEAST n. ∀i. i > n ⟶ (coeff p) i = 0)"
proof (cases p)
  case Nil thus ?thesis by auto
next
  define P where "P = (λn. ∀i. i > n ⟶ (coeff p) i = 0)"

  case (Cons a ps)
  hence "(coeff p) (degree p) ≠ 0"
    using assms unfolding polynomial_def by auto
  hence "⋀n. n < degree p ⟹ ¬ P n"
    unfolding P_def by auto
  moreover have "P (degree p)"
    unfolding P_def using coeff_degree[of p] by simp
  ultimately have "degree p = (LEAST n. P n)"
    by (meson LeastI nat_neq_iff not_less_Least)
  thus ?thesis unfolding P_def .
qed

lemma coeff_iff_polynomial_cond:
  assumes "polynomial K p1" and "polynomial K p2"
  shows "p1 = p2 ⟷ coeff p1 = coeff p2"
proof
  show "p1 = p2 ⟹ coeff p1 = coeff p2"
    by simp
next
  assume coeff_eq: "coeff p1 = coeff p2"
  hence deg_eq: "degree p1 = degree p2"
    using degree_def'[OF assms(1)] degree_def'[OF assms(2)] by auto
  thus "p1 = p2"
  proof (cases "p1 ≠ [] ∧ p2 ≠ []")
    case True
    hence "length p1 = length p2"
      using deg_eq by (simp add: Nitpick.size_list_simp(2))
    thus ?thesis
      using coeff_iff_length_cond[of p1 p2] coeff_eq by simp
  next
    case False
    have aux_lemma: "p2 = []"
      if A: "p1 = []" "coeff p1 = coeff p2" "polynomial K p2"
      for p1 p2
    proof (rule ccontr)
      assume "p2 ≠ []"
      hence "(coeff p2) (degree p2) ≠ 0"
        using A(3) unfolding polynomial_def
        by (metis coeff.simps(2) list.collapse)
      moreover have "(coeff p1) ` UNIV = { 0 }"
        using A(1) by auto
      hence "(coeff p2) ` UNIV = { 0 }"
        using A(2) by simp
```

```
      ultimately show False
        by blast
    qed
    from False have "p1 = [] ∨ p2 = []" by simp
    thus ?thesis
      using assms coeff_eq aux_lemma[of p1 p2] aux_lemma[of p2 p1] by
auto
  qed
qed

lemma normalize_lead_coeff:
  assumes "length (normalize p) < length p"
  shows "lead_coeff p = 0"
proof (cases p)
  case Nil thus ?thesis
    using assms by simp
next
  case (Cons a ps) thus ?thesis
    using assms by (cases "a = 0") (auto)
qed

lemma normalize_length_lt:
  assumes "lead_coeff p = 0" and "length p > 0"
  shows "length (normalize p) < length p"
proof (cases p)
  case Nil thus ?thesis
    using assms by simp
next
  case (Cons a ps) thus ?thesis
    using normalize_length_le[of ps] assms by simp
qed

lemma normalize_length_eq:
  assumes "lead_coeff p ≠ 0"
  shows "length (normalize p) = length p"
  using normalize_length_le[of p] assms nat_less_le normalize_lead_coeff
by auto

lemma normalize_replicate_zero: "normalize ((replicate n 0) @ p) = normalize
p"
  by (induction n) (auto)

lemma normalize_def':
  shows   "p = (replicate (length p - length (normalize p)) 0) @
                   (drop (length p - length (normalize p)) p)" (is ?statement1)
  and "normalize p = drop (length p - length (normalize p)) p"  (is ?statement2)
proof -
  show ?statement1
  proof (induction p)
```

```
    case Nil thus ?case by simp
  next
    case (Cons a p) thus ?case
    proof (cases "a = 0")
      assume "a ≠ 0" thus ?case
        using Cons by simp
    next
      assume eq_zero: "a = 0"
      hence len_eq:
        "Suc (length p - length (normalize p)) = length (a # p) - length
(normalize (a # p))"
        by (simp add: Suc_diff_le normalize_length_le)
      have "a # p = 0 # (replicate (length p - length (normalize p)) 0
@
                             drop (length p - length (normalize p)) p)"
        using eq_zero Cons by simp
      also have " ... = (replicate (Suc (length p - length (normalize
p))) 0 @
                             drop (Suc (length p - length (normalize
p))) (a # p))"
        by simp
      also have " ... = (replicate (length (a # p) - length (normalize
(a # p))) 0 @
                             drop (length (a # p) - length (normalize
(a # p))) (a # p))"
        using len_eq by simp
      finally show ?case .
    qed
  qed
next
  show ?statement2
  proof -
    have "∃m. normalize p = drop m p"
    proof (induction p)
      case Nil thus ?case by simp
    next
      case (Cons a p) thus ?case
        apply (cases "a = 0")
        apply (auto)
        apply (metis drop_Suc_Cons)
        apply (metis drop0)
        done
    qed
    then obtain m where m: "normalize p = drop m p" by auto
    hence "length (normalize p) = length p - m" by simp
    thus ?thesis
      using m by (metis rev_drop rev_rev_ident take_rev)
  qed
qed
```

```
corollary normalize_trick:
  shows "p = (replicate (length p - length (normalize p)) 0) @ (normalize
p)"
  using normalize_def'(1)[of p] unfolding sym[OF normalize_def'(2)] .

lemma normalize_coeff: "coeff p = coeff (normalize p)"
proof (induction p)
  case Nil thus ?case by simp
next
  case (Cons a p)
  have "coeff (normalize p) (length p) = 0"
    using normalize_length_le[of p] coeff_degree[of "normalize p"] coeff_length
by blast
  then show ?case
    using Cons by (cases "a = 0") (auto)
qed

lemma append_coeff:
  "coeff (p @ q) = (λi. if i < length q then (coeff q) i else (coeff p)
(i - length q))"
proof (induction p)
  case Nil thus ?case
    using coeff_length[of q] by auto
next
  case (Cons a p)
  have "coeff ((a # p) @ q) = (λi. if i = length p + length q then a else
(coeff (p @ q)) i)"
    by auto
  also have " ... = (λi. if i = length p + length q then a
                         else if i < length q then (coeff q) i
                         else (coeff p) (i - length q))"
    using Cons by auto
  also have " ... = (λi. if i < length q then (coeff q) i
                         else if i = length p + length q then a else (coeff
p) (i - length q))"
    by auto
  also have " ... = (λi. if i < length q then (coeff q) i
                         else if i - length q = length p then a else (coeff
p) (i - length q))"
    by fastforce
  also have " ... = (λi. if i < length q then (coeff q) i else (coeff
(a # p)) (i - length q))"
    by auto
  finally show ?case .
qed

lemma prefix_replicate_zero_coeff: "coeff p = coeff ((replicate n 0)
@ p)"
```

```
  using append_coeff[of "replicate n 0" p] replicate_zero_coeff[of n]
coeff_length[of p] by auto


context
  fixes K :: "'a set" assumes K: "subring K R"
begin

lemma polynomial_in_carrier [intro]: "polynomial K p ⟹ set p ⊆ carrier
R"
  unfolding polynomial_def using subringE(1)[OF K] by auto

lemma carrier_polynomial [intro]: "polynomial K p ⟹ polynomial (carrier
R) p"
  unfolding polynomial_def using subringE(1)[OF K] by auto

lemma append_is_polynomial: "⟦ polynomial K p; p ≠ [] ⟧ ⟹ polynomial
K (p @ (replicate n 0))"
  unfolding polynomial_def using subringE(2)[OF K] by auto

lemma lead_coeff_in_carrier: "polynomial K (a # p) ⟹ a ∈ carrier R
- { 0 }"
  unfolding polynomial_def using subringE(1)[OF K] by auto

lemma monom_is_polynomial [intro]: "a ∈ K - { 0 } ⟹ polynomial K (monom
a n)"
  unfolding polynomial_def monom_def using subringE(2)[OF K] by auto

lemma eval_poly_in_carrier: "⟦ polynomial K p; x ∈ carrier R ⟧ ⟹ (eval
p) x ∈ carrier R"
  using eval_in_carrier[OF polynomial_in_carrier] .

lemma poly_coeff_in_carrier [simp]: "polynomial K p ⟹ coeff p i ∈
carrier R"
  using coeff_in_carrier[OF polynomial_in_carrier] .

end
```

## 33.3 Polynomial Addition

```
context
  fixes K :: "'a set" assumes K: "subring K R"
begin

lemma poly_add_is_polynomial:
  assumes "set p1 ⊆ K" and "set p2 ⊆ K"
  shows "polynomial K (poly_add p1 p2)"
proof -
  have "polynomial K (poly_add p1 p2)"
```

```
    if A: "set p1 ⊆ K" "set p2 ⊆ K" "length p1 ≥ length p2" for p1 p2
  proof -
    define p2' where "p2' = (replicate (length p1 - length p2) 0) @ p2"
    hence "set p2' ⊆ K" and "length p1 = length p2'"
      using A(2-3) subringE(2)[OF K] by auto
    hence "set (map2 (⊕) p1 p2') ⊆ K"
      using A(1) subringE(7)[OF K]
      by (induct p1) (auto, metis set_ConsD subsetD set_zip_leftD set_zip_rightD)
    thus ?thesis
      unfolding p2'_def using normalize_gives_polynomial A(3) by simp
  qed
  thus ?thesis using assms by auto
qed

lemma poly_add_closed: "⟦ polynomial K p1; polynomial K p2 ⟧ ⟹ polynomial
K (poly_add p1 p2)"
  using poly_add_is_polynomial polynomial_incl by simp

lemma poly_add_length_eq:
  assumes "polynomial K p1" "polynomial K p2" and "length p1 ≠ length
p2"
  shows "length (poly_add p1 p2) = max (length p1) (length p2)"
proof -
  have "length (poly_add p1 p2) = max (length p1) (length p2)"
    if A: "polynomial K p1" "polynomial K p2" "length p1 > length p2" for
p1 p2
  proof -
    let ?p2 = "(replicate (length p1 - length p2) 0) @ p2"
    have p1: "p1 ≠ []" and p2: "?p2 ≠ []"
      using A(3) by auto
    then have "zip p1 (replicate (length p1 - length p2) 0 @ p2) =
        zip (lead_coeff p1 # tl p1) (lead_coeff (replicate (length p1
- length p2) 0 @ p2) # tl (replicate (length p1 - length p2) 0 @ p2))"
      by auto
    hence "lead_coeff (map2 (⊕) p1 ?p2) = lead_coeff p1 ⊕ lead_coeff
?p2"
      by simp
    moreover have "lead_coeff p1 ∈ carrier R"
      using p1 A(1) lead_coeff_in_carrier[OF K, of "hd p1" "tl p1"] by
auto
    ultimately have "lead_coeff (map2 (⊕) p1 ?p2) = lead_coeff p1"
      using A(3) by auto
    moreover have "lead_coeff p1 ≠ 0"
      using p1 A(1) unfolding polynomial_def by simp
    ultimately have "length (normalize (map2 (⊕) p1 ?p2)) = length p1"
      using normalize_length_eq by auto
    thus ?thesis using A(3) by auto
  qed
  thus ?thesis using assms by auto
```

```
qed

lemma poly_add_degree_eq:
  assumes "polynomial K p1" "polynomial K p2" and "degree p1 ≠ degree
p2"
  shows "degree (poly_add p1 p2) = max (degree p1) (degree p2)"
  using poly_add_length_eq[OF assms(1-2)] assms(3) by simp

end


lemma poly_add_in_carrier:
  "⟦ set p1 ⊆ carrier R; set p2 ⊆ carrier R ⟧ ⟹ set (poly_add p1 p2)
⊆ carrier R"
  using polynomial_incl[OF poly_add_is_polynomial[OF carrier_is_subring]]
by simp

lemma poly_add_length_le: "length (poly_add p1 p2) ≤ max (length p1)
(length p2)"
proof -
  have "length (poly_add p1 p2) ≤ max (length p1) (length p2)"
    if "length p1 ≥ length p2" for p1 p2 :: "'a list"
    using normalize_length_le[of "map2 (⊕) p1 ((replicate (length p1 -
length p2) 0) @ p2)"] that
    by auto
  thus ?thesis
    by (metis le_cases max.commute poly_add.simps)
qed

lemma poly_add_degree: "degree (poly_add p1 p2) ≤ max (degree p1) (degree
p2)"
  using poly_add_length_le by (meson diff_le_mono le_max_iff_disj)

lemma poly_add_coeff_aux:
  assumes "length p1 ≥ length p2"
  shows "coeff (poly_add p1 p2) = (λi. ((coeff p1) i) ⊕ ((coeff p2) i))"
proof
  fix i
  have "i < length p1 ⟹ (coeff (poly_add p1 p2)) i = ((coeff p1) i)
⊕ ((coeff p2) i)"
  proof -
    let ?p2 = "(replicate (length p1 - length p2) 0) @ p2"
    have len_eqs: "length p1 = length ?p2" "length (map2 (⊕) p1 ?p2)
= length p1"
      using assms by auto
    assume i_lt: "i < length p1"
    have "(coeff (poly_add p1 p2)) i = (coeff (map2 (⊕) p1 ?p2)) i"
      using normalize_coeff[of "map2 (⊕) p1 ?p2"] assms by auto
    also have " ... = (map2 (⊕) p1 ?p2) ! (length p1 - 1 - i)"
```

```
      using coeff_nth[of i "map2 (⊕) p1 ?p2"] len_eqs(2) i_lt by auto
    also have " ... = (p1 ! (length p1 - 1 - i)) ⊕ (?p2 ! (length ?p2
- 1 - i))"
      using len_eqs i_lt by auto
    also have " ... = ((coeff p1) i) ⊕ ((coeff ?p2) i)"
      using coeff_nth[of i p1] coeff_nth[of i ?p2] i_lt len_eqs(1) by
auto
    also have " ... = ((coeff p1) i) ⊕ ((coeff p2) i)"
      using prefix_replicate_zero_coeff by simp
    finally show "(coeff (poly_add p1 p2)) i = ((coeff p1) i) ⊕ ((coeff
p2) i)" .
  qed
  moreover
  have "i ≥ length p1 ⟹ (coeff (poly_add p1 p2)) i = ((coeff p1) i)
⊕ ((coeff p2) i)"
    using coeff_length[of "poly_add p1 p2"] coeff_length[of p1] coeff_length[of
p2]
          poly_add_length_le[of p1 p2] assms by auto
  ultimately show "(coeff (poly_add p1 p2)) i = ((coeff p1) i) ⊕ ((coeff
p2) i)"
    using not_le by blast
qed

lemma poly_add_coeff:
  assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
  shows "coeff (poly_add p1 p2) = (λi. ((coeff p1) i) ⊕ ((coeff p2) i))"
proof -
  have "length p1 ≥ length p2 ∨ length p2 > length p1"
    by auto
  thus ?thesis
  proof
    assume "length p1 ≥ length p2" thus ?thesis
      using poly_add_coeff_aux by simp
  next
    assume "length p2 > length p1"
    hence "coeff (poly_add p1 p2) = (λi. ((coeff p2) i) ⊕ ((coeff p1)
i))"
      using poly_add_coeff_aux by simp
    thus ?thesis
      using assms by (simp add: add.m_comm)
  qed
qed

lemma poly_add_comm:
  assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
  shows "poly_add p1 p2 = poly_add p2 p1"
proof -
  have "coeff (poly_add p1 p2) = coeff (poly_add p2 p1)"
    using poly_add_coeff[OF assms] poly_add_coeff[OF assms(2) assms(1)]
```

```
          coeff_in_carrier[OF assms(1)] coeff_in_carrier[OF assms(2)]
add.m_comm by auto
  thus ?thesis
    using coeff_iff_polynomial_cond[OF
          poly_add_is_polynomial[OF carrier_is_subring assms]
          poly_add_is_polynomial[OF carrier_is_subring assms(2,1)]] by
simp
qed

lemma poly_add_monom:
  assumes "set p ⊆ carrier R" and "a ∈ carrier R - { 0 }"
  shows "poly_add (monom a (length p)) p = a # p"
  unfolding monom_def using assms by (induction p) (auto)

lemma poly_add_append_replicate:
  assumes "set p ⊆ carrier R" "set q ⊆ carrier R"
  shows "poly_add (p @ (replicate (length q) 0)) q = normalize (p @ q)"
proof -
  have "map2 (⊕) (p @ (replicate (length q) 0)) ((replicate (length p)
0) @ q) = p @ q"
    using assms by (induct p) (induct q, auto)
  thus ?thesis by simp
qed

lemma poly_add_append_zero:
  assumes "set p ⊆ carrier R" "set q ⊆ carrier R"
  shows "poly_add (p @ [ 0 ]) (q @ [ 0 ]) = normalize ((poly_add p q)
@ [ 0 ])"
proof -
  have in_carrier: "set (p @ [ 0 ]) ⊆ carrier R" "set (q @ [ 0 ]) ⊆ carrier
R"
    using assms by auto
  have "coeff (poly_add (p @ [ 0 ]) (q @ [ 0 ])) = coeff ((poly_add p
q) @ [ 0 ])"
    using append_coeff[of p "[ 0 ]"] poly_add_coeff[OF in_carrier]
          append_coeff[of q "[ 0 ]"] append_coeff[of "poly_add p q" "[
0 ]"]
          poly_add_coeff[OF assms] assms[THEN coeff_in_carrier] by auto
  hence "coeff (poly_add (p @ [ 0 ]) (q @ [ 0 ])) = coeff (normalize
((poly_add p q) @ [ 0 ]))"
    using normalize_coeff by simp
  moreover have "set ((poly_add p q) @ [ 0 ]) ⊆ carrier R"
    using poly_add_in_carrier[OF assms] by simp
  ultimately show ?thesis
    using coeff_iff_polynomial_cond[OF poly_add_is_polynomial[OF carrier_is_subring
in_carrier]
          normalize_gives_polynomial] by simp
qed
```

```
lemma poly_add_normalize_aux:
  assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
  shows "poly_add p1 p2 = poly_add (normalize p1) p2"
proof -
  have aux_lemma: "poly_add p1 p2 = poly_add ((replicate n 0) @ p1) p2"
    if "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
    for n p1 p2
    using that
  proof (induction n)
    case 0
    thus ?case by simp
  next
    case (Suc n)
    have aux_lemma: "poly_add p1 p2 = poly_add (0 # p1) p2"
      if in_carrier: "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
      for p1 p2
    proof -
      have "length p1 ≥ length p2 ⟹ ?thesis"
      proof -
        assume A: "length p1 ≥ length p2"
        let ?p2 = "λn. (replicate n 0) @ p2"
        have "poly_add p1 p2 = normalize (map2 (⊕) (0 # p1) (0 # ?p2
(length p1 - length p2)))"
          using A by simp
        also have " ... = normalize (map2 (⊕) (0 # p1) (?p2 (length (0
# p1) - length p2)))"
          by (simp add: A Suc_diff_le)
        also have " ... = poly_add (0 # p1) p2"
          using A by simp
        finally show ?thesis .
      qed
      moreover have "length p2 > length p1 ⟹ ?thesis"
      proof -
        assume A: "length p2 > length p1"
        let ?f = "λn p. (replicate n 0) @ p"
        have "poly_add p1 p2 = poly_add p2 p1"
          using A by simp
        also have " ... = normalize (map2 (⊕) p2 (?f (length p2 - length
p1) p1))"
          using A by simp
        also have " ... = normalize (map2 (⊕) p2 (?f (length p2 - Suc
(length p1)) (0 # p1)))"
          by (metis A Suc_diff_Suc append_Cons replicate_Suc replicate_app_Cons_same)
        also have " ... = poly_add p2 (0 # p1)"
          using A by simp
        also have " ... = poly_add (0 # p1) p2"
          using poly_add_comm[of p2 "0 # p1"] in_carrier by auto
        finally show ?thesis .
      qed
```

```
      ultimately show ?thesis by auto
    qed

    from Suc have in_carrier: "set (replicate n 0 @ p1) ⊆ carrier R"
      by auto
    have "poly_add p1 p2 = poly_add (replicate n 0 @ p1) p2"
      using Suc by simp
    also have " ... = poly_add (replicate (Suc n) 0 @ p1) p2"
      using aux_lemma[OF in_carrier Suc(3)] by simp
    finally show ?case .
  qed

  have "poly_add p1 p2 =
        poly_add ((replicate (length p1 - length (normalize p1)) 0) @
normalize p1) p2"
    using normalize_def'[of p1] by simp
  also have " ... = poly_add (normalize p1) p2"
    using aux_lemma[OF normalize_in_carrier[OF assms(1)] assms(2)] by
simp
  finally show ?thesis .
qed

lemma poly_add_normalize:
  assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
  shows "poly_add p1 p2 = poly_add (normalize p1) p2"
    and "poly_add p1 p2 = poly_add p1 (normalize p2)"
    and "poly_add p1 p2 = poly_add (normalize p1) (normalize p2)"
proof -
  show "poly_add p1 p2 = poly_add p1 (normalize p2)"
    unfolding poly_add_comm[OF assms] poly_add_normalize_aux[OF assms(2)
assms(1)]
              poly_add_comm[OF normalize_in_carrier[OF assms(2)] assms(1)]
by simp
next
  show "poly_add p1 p2 = poly_add (normalize p1) p2"
    using poly_add_normalize_aux[OF assms] .
  also have " ... = poly_add (normalize p2) (normalize p1)"
    unfolding  poly_add_comm[OF normalize_in_carrier[OF assms(1)] assms(2)]
               poly_add_normalize_aux[OF assms(2) normalize_in_carrier[OF
assms(1)]] by simp
  finally show "poly_add p1 p2 = poly_add (normalize p1) (normalize p2)"
    unfolding  poly_add_comm[OF assms[THEN normalize_in_carrier]] .
qed

lemma poly_add_zero':
  assumes "set p ⊆ carrier R"
  shows "poly_add p [] = normalize p" and "poly_add [] p = normalize
p"
proof -
```

```
  have "map2 (⊕) p (replicate (length p) 0) = p"
    using assms by (induct p) (auto)
  thus "poly_add p [] = normalize p" and "poly_add [] p = normalize p"
    using poly_add_comm[OF assms, of "[]"] by simp+
qed

lemma poly_add_zero:
  assumes "subring K R" "polynomial K p"
  shows "poly_add p [] = p" and "poly_add [] p = p"
  using poly_add_zero' normalize_polynomial polynomial_in_carrier assms
by auto

lemma poly_add_replicate_zero':
  assumes "set p ⊆ carrier R"
  shows "poly_add p (replicate n 0) = normalize p" and "poly_add (replicate
n 0) p = normalize p"
proof -
  have "poly_add p (replicate n 0) = poly_add p []"
    using poly_add_normalize(2)[OF assms, of "replicate n 0"]
          normalize_replicate_zero[of n "[]"] by force
  also have " ... = normalize p"
    using poly_add_zero'[OF assms] by simp
  finally show "poly_add p (replicate n 0) = normalize p" .
  thus "poly_add (replicate n 0) p = normalize p"
    using poly_add_comm[OF assms, of "replicate n 0"] by force
qed

lemma poly_add_replicate_zero:
  assumes "subring K R" "polynomial K p"
  shows "poly_add p (replicate n 0) = p" and "poly_add (replicate n 0)
p = p"
  using poly_add_replicate_zero' normalize_polynomial polynomial_in_carrier
assms by auto
```

## 33.4 Dense Representation

```
lemma dense_repr_replicate_zero: "dense_repr ((replicate n 0) @ p) =
dense_repr p"
  by (induction n) (auto)

lemma dense_repr_normalize: "dense_repr (normalize p) = dense_repr p"
  by (induct p) (auto)

lemma polynomial_dense_repr:
  assumes "polynomial K p" and "p ≠ []"
  shows "dense_repr p = (lead_coeff p, degree p) # dense_repr (normalize
(tl p))"
proof -
  let ?len = length and ?norm = normalize
```

```
  obtain a p' where p: "p = a # p'"
    using assms(2) list.exhaust_sel by blast
  hence a: "a ∈ K - { 0 }" and p': "set p' ⊆ K"
    using assms(1) unfolding p by (auto simp add: polynomial_def)
  hence "dense_repr p = (lead_coeff p, degree p) # dense_repr p'"
    unfolding p by simp
  also have " ... =
    (lead_coeff p, degree p) # dense_repr ((replicate (?len p' - ?len
(?norm p')) 0) @ ?norm p')"
    using normalize_def' dense_repr_replicate_zero by simp
  also have " ... = (lead_coeff p, degree p) # dense_repr (?norm p')"
    using dense_repr_replicate_zero by simp
  finally show ?thesis
    unfolding p by simp
qed

lemma monom_decomp:
  assumes "subring K R" "polynomial K p"
  shows "p = poly_of_dense (dense_repr p)"
  using assms(2)
proof (induct "length p" arbitrary: p rule: less_induct)
  case less thus ?case
  proof (cases p)
    case Nil thus ?thesis by simp
  next
    case (Cons a l)
    hence a: "a ∈ carrier R - { 0 }" and l: "set l ⊆ carrier R"  "set
l ⊆ K"
      using less(2) subringE(1)[OF assms(1)] by (auto simp add: polynomial_def)
    hence "a # l = poly_add (monom a (degree (a # l))) l"
      using poly_add_monom[of l a] by simp
    also have " ... = poly_add (monom a (degree (a # l))) (normalize l)"
      using poly_add_normalize(2)[of "monom a (degree (a # l))", OF _
l(1)] a
      unfolding monom_def by force
    also have " ... = poly_add (monom a (degree (a # l))) (poly_of_dense
(dense_repr (normalize l)))"
      using less(1)[OF _ normalize_gives_polynomial[OF l(2)]] normalize_length_le[of
l]
      unfolding Cons by simp
    also have " ... = poly_of_dense ((a, degree (a # l)) # dense_repr
(normalize l))"
      by simp
    also have " ... = poly_of_dense (dense_repr (a # l))"
      using polynomial_dense_repr[OF less(2)] unfolding Cons by simp
    finally show ?thesis
      unfolding Cons by simp
  qed
qed
```

## 33.5 Polynomial Multiplication

```
lemma poly_mult_is_polynomial:
  assumes "subring K R" "set p1 ⊆ K" and "set p2 ⊆ K"
  shows "polynomial K (poly_mult p1 p2)"
  using assms(2-3)
proof (induction p1)
  case Nil thus ?case
    by (simp add: polynomial_def)
next
  case (Cons a p1)
  let ?a_p2 = "(map (λb. a ⊗ b) p2) @ (replicate (degree (a # p1)) 0)"

  have "set (poly_mult p1 p2) ⊆ K"
    using Cons unfolding polynomial_def by auto
  moreover have "set ?a_p2 ⊆ K"
      using assms(3) Cons(2) subringE(1-2,6)[OF assms(1)] by(induct p2)
(auto)
  ultimately have "polynomial K (poly_add ?a_p2 (poly_mult p1 p2))"
    using poly_add_is_polynomial[OF assms(1)] by blast
  thus ?case by simp
qed

lemma poly_mult_closed:
  assumes "subring K R"
  shows "⟦ polynomial K p1; polynomial K p2 ⟧ ⟹ polynomial K (poly_mult
p1 p2)"
  using poly_mult_is_polynomial polynomial_incl assms by simp

lemma poly_mult_in_carrier:
  "⟦ set p1 ⊆ carrier R; set p2 ⊆ carrier R ⟧ ⟹ set (poly_mult p1 p2)
⊆ carrier R"
  using poly_mult_is_polynomial polynomial_in_carrier carrier_is_subring
by simp

lemma poly_mult_coeff:
  assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
  shows "coeff (poly_mult p1 p2) = (λi. ⨁ k ∈ {..i}. (coeff p1) k ⊗
(coeff p2) (i - k))"
  using assms(1)
proof (induction p1)
  case Nil thus ?case using assms(2) by auto
next
  case (Cons a p1)
  hence in_carrier:
    "a ∈ carrier R" "⋀i. (coeff p1) i ∈ carrier R" "⋀i. (coeff p2) i
∈ carrier R"
    using coeff_in_carrier assms(2) by auto

  let ?a_p2 = "(map (λb. a ⊗ b) p2) @ (replicate (degree (a # p1)) 0)"
```

```
  have "coeff  (replicate (degree (a # p1)) 0) = (λ_. 0)"
   and "length (replicate (degree (a # p1)) 0) = length p1"
    using prefix_replicate_zero_coeff[of "[]" "length p1"] by auto
  hence "coeff ?a_p2 = (λi. if i < length p1 then 0 else (coeff (map
(λb. a ⊗ b) p2)) (i - length p1))"
    using append_coeff[of "map (λb. a ⊗ b) p2" "replicate (length p1)
0"] by auto
  also have " ... = (λi. if i < length p1 then 0 else a ⊗ ((coeff p2)
(i - length p1)))"
  proof -
    have "⋀i. i < length p2 ⟹ (coeff (map (λb. a ⊗ b) p2)) i = a ⊗
((coeff p2) i)"
    proof -
      fix i assume i_lt: "i < length p2"
      hence "(coeff (map (λb. a ⊗ b) p2)) i = (map (λb. a ⊗ b) p2) !
(length p2 - 1 - i)"
        using coeff_nth[of i "map (λb. a ⊗ b) p2"] by auto
      also have " ... = a ⊗ (p2 ! (length p2 - 1 - i))"
        using i_lt by auto
      also have " ... = a ⊗ ((coeff p2) i)"
        using coeff_nth[OF i_lt] by simp
      finally show "(coeff (map (λb. a ⊗ b) p2)) i = a ⊗ ((coeff p2)
i)" .
    qed
    moreover have "⋀i. i ≥ length p2 ⟹ (coeff (map (λb. a ⊗ b) p2))
i = a ⊗ ((coeff p2) i)"
      using coeff_length[of p2] coeff_length[of "map (λb. a ⊗ b) p2"]
in_carrier by auto
    ultimately show ?thesis by (meson not_le)
  qed
  also have " ... = (λi. ⨁ k ∈ {..i}. (if k = length p1 then a else
0) ⊗ (coeff p2) (i - k))"
  (is "?f1 = (λi. (⨁ k ∈ {..i}. ?f2 k ⊗ ?f3 (i - k)))")
  proof
    fix i
    have "⋀k. k ∈ {..i} ⟹ ?f2 k ⊗ ?f3 (i - k) = 0" if "i < length p1"
      using in_carrier that by auto
    hence "(⨁ k ∈ {..i}. ?f2 k ⊗ ?f3 (i - k)) = 0" if "i < length p1"
      using that in_carrier
            add.finprod_cong'[of "{..i}" "{..i}" "λk. ?f2 k ⊗ ?f3 (i
- k)" "λi. 0"]
      by auto
    hence eq_lt: "?f1 i = (λi. (⨁ k ∈ {..i}. ?f2 k ⊗ ?f3 (i - k)))
i" if "i < length p1"
      using that by auto

    have "⋀k. k ∈ {..i} ⟹
              ?f2 k ⊗R ?f3 (i - k) = (if length p1 = k then a ⊗ coeff
p2 (i - k) else 0)"
```

```
      using in_carrier by auto
    hence "(⨁ k ∈ {..i}. ?f2 k ⊗ ?f3 (i - k)) =
           (⨁ k ∈ {..i}. (if length p1 = k then a ⊗ coeff p2 (i - k)
else 0))"
      using in_carrier
            add.finprod_cong'[of "{..i}" "{..i}" "λk. ?f2 k ⊗ ?f3 (i
- k)"
                                "λk. (if length p1 = k then a ⊗ coeff p2
(i - k) else 0)"]
      by fastforce
    also have " ... = a ⊗ (coeff p2) (i - length p1)" if "i ≥ length p1"
      using add.finprod_singleton[of "length p1" "{..i}" "λj. a ⊗ (coeff
p2) (i - j)"]
            in_carrier that by auto
    finally
    have "(⨁ k ∈ {..i}. ?f2 k ⊗ ?f3 (i - k)) =  a ⊗ (coeff p2) (i -
length p1)" if "i ≥ length p1"
      using that by simp
    hence eq_ge: "?f1 i = (λi. (⨁ k ∈ {..i}. ?f2 k ⊗ ?f3 (i - k)))
i" if "i ≥ length p1"
      using that by auto

    from eq_lt eq_ge show "?f1 i = (λi. (⨁ k ∈ {..i}. ?f2 k ⊗ ?f3 (i
- k))) i" by auto
  qed

  finally have coeff_a_p2:
    "coeff ?a_p2 = (λi. ⨁ k ∈ {..i}. (if k = length p1 then a else 0)
⊗ (coeff p2) (i - k))" .

  have "set ?a_p2 ⊆ carrier R"
    using in_carrier(1) assms(2) by auto

  moreover have "set (poly_mult p1 p2) ⊆ carrier R"
    using poly_mult_in_carrier[OF _ assms(2)] Cons(2) by simp

  ultimately
  have "coeff (poly_mult (a # p1) p2) = (λi. ((coeff ?a_p2) i) ⊕ ((coeff
(poly_mult p1 p2)) i))"
    using poly_add_coeff[of ?a_p2 "poly_mult p1 p2"] by simp
  also have " ... = (λi. (⨁ k ∈ {..i}. (if k = length p1 then a else
0) ⊗ (coeff p2) (i - k)) ⊕
                         (⨁ k ∈ {..i}. (coeff p1) k ⊗ (coeff p2) (i
- k)))"
    using Cons  coeff_a_p2 by simp
  also have " ... = (λi. (⨁ k ∈ {..i}. ((if k = length p1 then a else
0) ⊗ (coeff p2) (i - k)) ⊕
                                                         ((coeff p1)
k ⊗ (coeff p2) (i - k))))"
```

```
    using add.finprod_multf in_carrier by auto
  also have " ... = (λi. (⨁ k ∈ {..i}. (coeff (a # p1) k) ⊗ (coeff p2)
(i - k)))"
   (is "(λi. (⨁ k ∈ {..i}. ?f i k)) = (λi. (⨁ k ∈ {..i}. ?g i k))")
  proof
    fix i
    have "⋀k. ?f i k = ?g i k"
      using in_carrier coeff_length[of p1] by auto
    thus "(⨁ k ∈ {..i}. ?f i k) = (⨁ k ∈ {..i}. ?g i k)" by simp
  qed
  finally show ?case .
qed

lemma poly_mult_zero:
  assumes "set p ⊆ carrier R"
  shows "poly_mult [] p = []" and "poly_mult p [] = []"
proof (simp)
  have "coeff (poly_mult p []) = (λ_. 0)"
    using poly_mult_coeff[OF assms, of "[]"] coeff_in_carrier[OF assms]
by auto
  thus "poly_mult p [] = []"
    using coeff_iff_polynomial_cond[OF
          poly_mult_is_polynomial[OF carrier_is_subring assms] zero_is_polynomial]
by simp
qed

lemma poly_mult_l_distr':
  assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R" "set p3 ⊆ carrier
R"
  shows "poly_mult (poly_add p1 p2) p3 = poly_add (poly_mult p1 p3) (poly_mult
p2 p3)"
proof -
  let ?c1 = "coeff p1" and ?c2 = "coeff p2" and ?c3 = "coeff p3"
  have in_carrier:
    "⋀i. ?c1 i ∈ carrier R" "⋀i. ?c2 i ∈ carrier R" "⋀i. ?c3 i ∈ carrier
R"
    using assms coeff_in_carrier by auto

  have "coeff (poly_mult (poly_add p1 p2) p3) = (λn. ⨁i ∈ {..n}. (?c1
i ⊕ ?c2 i) ⊗ ?c3 (n - i))"
    using poly_mult_coeff[of "poly_add p1 p2" p3]  poly_add_coeff[OF assms(1-2)]
          poly_add_in_carrier[OF assms(1-2)] assms by auto
  also have " ... = (λn. ⨁i ∈ {..n}. (?c1 i ⊗ ?c3 (n - i)) ⊕ (?c2 i
⊗ ?c3 (n - i)))"
    using in_carrier l_distr by auto
  also
  have " ... = (λn. (⨁i ∈ {..n}. (?c1 i ⊗ ?c3 (n - i))) ⊕ (⨁i ∈ {..n}.
(?c2 i ⊗ ?c3 (n - i))))"
    using add.finprod_multf in_carrier by auto
```

```
  also have " ... = coeff (poly_add (poly_mult p1 p3) (poly_mult p2 p3))"
    using poly_mult_coeff[OF assms(1) assms(3)] poly_mult_coeff[OF assms(2-3)]
          poly_add_coeff[OF poly_mult_in_carrier[OF assms(1) assms(3)]]
                            poly_mult_in_carrier[OF assms(2-3)] by simp
  finally have "coeff (poly_mult (poly_add p1 p2) p3) =
                coeff (poly_add (poly_mult p1 p3) (poly_mult p2 p3))"
.
  moreover have "polynomial (carrier R) (poly_mult (poly_add p1 p2) p3)"
           and "polynomial (carrier R) (poly_add (poly_mult p1 p3) (poly_mult
p2 p3))"
    using assms poly_add_is_polynomial poly_mult_is_polynomial polynomial_in_carrier
          carrier_is_subring by auto
  ultimately show ?thesis
    using coeff_iff_polynomial_cond by auto
qed

lemma poly_mult_l_distr:
  assumes "subring K R" "polynomial K p1" "polynomial K p2" "polynomial
K p3"
  shows "poly_mult (poly_add p1 p2) p3 = poly_add (poly_mult p1 p3) (poly_mult
p2 p3)"
  using poly_mult_l_distr' polynomial_in_carrier assms by auto

lemma poly_mult_prepend_replicate_zero:
  assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
  shows "poly_mult p1 p2 = poly_mult ((replicate n 0) @ p1) p2"
proof -
  have aux_lemma: "poly_mult p1 p2 = poly_mult (0 # p1) p2"
    if A: "set p1 ⊆ carrier R" "set p2 ⊆ carrier R" for p1 p2
  proof -
    let ?a_p2 = "(map ((⊗) 0) p2) @ (replicate (length p1) 0)"
    have "?a_p2 = replicate (length p2 + length p1) 0"
      using A(2) by (induction p2) (auto)
    hence "poly_mult (0 # p1) p2 = poly_add (replicate (length p2 + length
p1) 0) (poly_mult p1 p2)"
      by simp
    also have " ... = poly_add (normalize (replicate (length p2 + length
p1) 0)) (poly_mult p1 p2)"
      using poly_add_normalize(1)[of "replicate (length p2 + length p1)
0" "poly_mult p1 p2"]
            poly_mult_in_carrier[OF A] by force
    also have " ... = poly_mult p1 p2"
      using poly_add_zero(2)[OF _ poly_mult_is_polynomial[OF _ A]] carrier_is_subring
            normalize_replicate_zero[of "length p2 + length p1" "[]"]
by simp
    finally show ?thesis by auto
  qed
  from assms show ?thesis
  proof (induction n)
```

```
    case 0
    thus ?case by simp
  next
    case (Suc n)
    thus ?case using aux_lemma[of "replicate n 0 @ p1" p2] by force
  qed
qed

lemma poly_mult_normalize:
  assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
  shows "poly_mult p1 p2 = poly_mult (normalize p1) p2"
proof -
  let ?replicate = "replicate (length p1 - length (normalize p1)) 0"
  have "poly_mult p1 p2 = poly_mult (?replicate @ (normalize p1)) p2"
    using normalize_def'[of p1] by simp
  thus ?thesis
    using poly_mult_prepend_replicate_zero normalize_in_carrier assms
by auto
qed

lemma poly_mult_append_zero:
  assumes "set p ⊆ carrier R" "set q ⊆ carrier R"
  shows "poly_mult (p @ [ 0 ]) q = normalize ((poly_mult p q) @ [ 0 ])"
  using assms(1)
proof (induct p)
  case Nil thus ?case
    using poly_mult_normalize[OF _ assms(2), of "[] @ [ 0 ]"]
          poly_mult_zero(1) poly_mult_zero(1)[of "q @ [ 0 ]"] assms(2)
by auto
next
  case (Cons a p)
  let ?q_a = "λn. (map ((⊗) a) q) @ (replicate n 0)"
  have set_q_a: "⋀n. set (?q_a n) ⊆ carrier R"
    using Cons(2) assms(2) by (induct q) (auto)
  have set_poly_mult: "set ((poly_mult p q) @ [ 0 ]) ⊆ carrier R"
    using poly_mult_in_carrier[OF _ assms(2)] Cons(2) by auto
  have "poly_mult ((a # p) @ [0]) q = poly_add (?q_a (Suc (length p)))
(poly_mult (p @ [0]) q)"
    by auto
  also have " ... = poly_add (?q_a (Suc (length p))) (normalize ((poly_mult
p q) @ [ 0 ]))"
    using Cons by simp
  also have " ... = poly_add ((?q_a (length p)) @ [ 0 ]) ((poly_mult p
q) @ [ 0 ])"
    using poly_add_normalize(2)[OF set_q_a[of "Suc (length p)"] set_poly_mult]
    by (simp add: replicate_append_same)
  also have " ... = normalize ((poly_add (?q_a (length p)) (poly_mult
p q)) @ [ 0 ])"
    using poly_add_append_zero[OF set_q_a[of "length p"] poly_mult_in_carrier[OF
```

```
_ assms(2)]] Cons(2) by auto
  also have " ... = normalize ((poly_mult (a # p) q) @ [ 0 ])"
    by auto
  finally show ?case .
qed

end
```

## 33.6 Properties Within a Domain

```
context domain
begin

lemma one_is_polynomial [intro]: "subring K R ⟹ polynomial K [ 1 ]"
  unfolding polynomial_def using subringE(3) by auto

lemma poly_mult_comm:
  assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R"
  shows "poly_mult p1 p2 = poly_mult p2 p1"
proof -
  let ?c1 = "coeff p1" and ?c2 = "coeff p2"
  have "⋀i. (⨁k ∈ {..i}. ?c1 k ⊗ ?c2 (i - k)) = (⨁k ∈ {..i}. ?c2
k ⊗ ?c1 (i - k))"
  proof -
    fix i :: nat
    let ?f = "λk. ?c1 k ⊗ ?c2 (i - k)"
    have in_carrier: "⋀i. ?c1 i ∈ carrier R" "⋀i. ?c2 i ∈ carrier R"
      using coeff_in_carrier[OF assms(1)] coeff_in_carrier[OF assms(2)]
by auto

    have reindex_inj: "inj_on (λk. i - k) {..i}"
      using inj_on_def by force
    moreover have "(λk. i - k) ` {..i} ⊆ {..i}" by auto
    hence "(λk. i - k) ` {..i} = {..i}"
      using reindex_inj endo_inj_surj[of "{..i}" "λk. i - k"] by simp

    ultimately have "(⨁k ∈ {..i}. ?f k) = (⨁k ∈ {..i}. ?f (i - k))"
      using add.finprod_reindex[of ?f "λk. i - k" "{..i}"] in_carrier
by auto

    moreover have "⋀k. k ∈ {..i} ⟹ ?f (i - k) = ?c2 k ⊗ ?c1 (i -
k)"
      using in_carrier m_comm by auto
    hence "(⨁k ∈ {..i}. ?f (i - k)) = (⨁k ∈ {..i}. ?c2 k ⊗ ?c1 (i
- k))"
      using add.finprod_cong'[of "{..i}" "{..i}"] in_carrier by auto
    ultimately show "(⨁k ∈ {..i}. ?f k) = (⨁k ∈ {..i}. ?c2 k ⊗ ?c1
(i - k))"
      by simp
```

```
  qed
  hence "coeff (poly_mult p1 p2) = coeff (poly_mult p2 p1)"
    using poly_mult_coeff[OF assms] poly_mult_coeff[OF assms(2,1)] by
simp
  thus ?thesis
    using coeff_iff_polynomial_cond[OF poly_mult_is_polynomial[OF _ assms]
                                       poly_mult_is_polynomial[OF _ assms(2,1)]]
          carrier_is_subring by simp
qed

lemma poly_mult_r_distr':
  assumes "set p1 ⊆ carrier R" "set p2 ⊆ carrier R" "set p3 ⊆ carrier
R"
  shows "poly_mult p1 (poly_add p2 p3) = poly_add (poly_mult p1 p2) (poly_mult
p1 p3)"
  unfolding poly_mult_comm[OF assms(1) poly_add_in_carrier[OF assms(2-3)]]
            poly_mult_l_distr'[OF assms(2-3,1)] assms(2-3)[THEN poly_mult_comm[OF
_ assms(1)]] ..

lemma poly_mult_r_distr:
  assumes "subring K R" "polynomial K p1" "polynomial K p2" "polynomial
K p3"
  shows "poly_mult p1 (poly_add p2 p3) = poly_add (poly_mult p1 p2) (poly_mult
p1 p3)"
  using poly_mult_r_distr' polynomial_in_carrier assms by auto

lemma poly_mult_replicate_zero:
  assumes "set p ⊆ carrier R"
  shows "poly_mult (replicate n 0) p = []"
    and "poly_mult p (replicate n 0) = []"
proof -
  have in_carrier: "⋀n. set (replicate n 0) ⊆ carrier R" by auto
  show "poly_mult (replicate n 0) p = []" using assms
  proof (induction n)
    case 0 thus ?case by simp
  next
    case (Suc n)
    hence "poly_mult (replicate (Suc n) 0) p = poly_mult (0 # (replicate
n 0)) p"
      by simp
    also have " ... = poly_add ((map (λa. 0 ⊗ a) p) @ (replicate n 0))
[]"
      using Suc by simp
    also have " ... = poly_add ((map (λa. 0) p) @ (replicate n 0)) []"
    proof -
      have "map ((⊗) 0) p = map (λa. 0) p"
        using Suc.prems by auto
      then show ?thesis
        by presburger
```

```
    qed
    also have " ... = poly_add (replicate (length p + n) 0) []"
      by (simp add: map_replicate_const replicate_add)
    also have " ... = poly_add [] []"
      using poly_add_normalize(1)[of "replicate (length p + n) 0" "[]"]
            normalize_replicate_zero[of "length p + n" "[]"] by auto
    also have " ... = []" by simp
    finally show ?case .
  qed
  thus "poly_mult p (replicate n 0) = []"
    using poly_mult_comm[OF assms in_carrier] by simp
qed

lemma poly_mult_const':
  assumes "set p ⊆ carrier R" "a ∈ carrier R"
  shows "poly_mult [ a ] p = normalize (map (λb. a ⊗ b) p)"
    and "poly_mult p [ a ] = normalize (map (λb. a ⊗ b) p)"
proof -
  have "map2 (⊕) (map ((⊗) a) p) (replicate (length p) 0) = map ((⊗)
a) p"
    using assms by (induction p) (auto)
  thus "poly_mult [ a ] p = normalize (map (λb. a ⊗ b) p)" by simp
  thus "poly_mult p [ a ] = normalize (map (λb. a ⊗ b) p)"
    using poly_mult_comm[OF assms(1), of "[ a ]"] assms(2) by auto
qed

lemma poly_mult_const:
  assumes "subring K R" "polynomial K p" "a ∈ K - { 0 }"
  shows "poly_mult [ a ] p = map (λb. a ⊗ b) p"
    and "poly_mult p [ a ] = map (λb. a ⊗ b) p"
proof -
  have in_carrier: "set p ⊆ carrier R" "a ∈ carrier R"
    using polynomial_in_carrier[OF assms(1-2)] assms(3) subringE(1)[OF
assms(1)] by auto

  show "poly_mult [ a ] p = map (λb. a ⊗ b) p"
  proof (cases p)
    case Nil thus ?thesis
      using poly_mult_const'(1) in_carrier by auto
  next
    case (Cons b q)
    have "lead_coeff (map (λb. a ⊗ b) p) ≠ 0"
      using assms subringE(1)[OF assms(1)] integral[of a b] Cons lead_coeff_in_carrier
by auto
    hence "normalize (map (λb. a ⊗ b) p) = (map (λb. a ⊗ b) p)"
      unfolding Cons by simp
    thus ?thesis
      using poly_mult_const'(1) in_carrier by auto
  qed
```

```
  thus "poly_mult p [ a ] = map (λb. a ⊗ b) p"
    using poly_mult_comm[OF in_carrier(1)] in_carrier(2) by auto
qed

lemma poly_mult_semiassoc:
  assumes "set p ⊆ carrier R" "set q ⊆ carrier R" and "a ∈ carrier R"
  shows "poly_mult (poly_mult [ a ] p) q = poly_mult [ a ] (poly_mult
p q)"
proof -
  let ?cp = "coeff p" and ?cq = "coeff q"
  have "coeff (poly_mult [ a ] p) = (λi. (a ⊗ ?cp i))"
    using poly_mult_const'(1)[OF assms(1,3)] normalize_coeff scalar_coeff[OF
assms(3)] by simp

  hence "coeff (poly_mult (poly_mult [ a ] p) q) = (λi. (⨁j ∈ {..i}.
(a ⊗ ?cp j) ⊗ ?cq (i - j)))"
    using poly_mult_coeff[OF poly_mult_in_carrier[OF _ assms(1)] assms(2),
of "[ a ]"] assms(3) by auto
  also have " ... = (λi. a ⊗ (⨁j ∈ {..i}. ?cp j ⊗ ?cq (i - j)))"
  proof
    fix i show "(⨁j ∈ {..i}. (a ⊗ ?cp j) ⊗ ?cq (i - j)) = a ⊗ (⨁j
∈ {..i}. ?cp j ⊗ ?cq (i - j))"
      using finsum_rdistr[OF _ assms(3), of _ "λj. ?cp j ⊗ ?cq (i - j)"]
            assms(1-2)[THEN coeff_in_carrier] by (simp add: assms(3) m_assoc)
  qed
  also have " ... = coeff (poly_mult [ a ] (poly_mult p q))"
    unfolding poly_mult_const'(1)[OF poly_mult_in_carrier[OF assms(1-2)]
assms(3)]
    using scalar_coeff[OF assms(3), of "poly_mult p q"]
          poly_mult_coeff[OF assms(1-2)] normalize_coeff by simp
  finally have "coeff (poly_mult (poly_mult [ a ] p) q) = coeff (poly_mult
[ a ] (poly_mult p q))" .
  moreover have "polynomial (carrier R) (poly_mult (poly_mult [ a ] p)
q)"
            and "polynomial (carrier R) (poly_mult [ a ] (poly_mult p
q))"
    using poly_mult_is_polynomial[OF _ poly_mult_in_carrier[OF _ assms(1)]
assms(2)]
          poly_mult_is_polynomial[OF _ _ poly_mult_in_carrier[OF assms(1-2)]]
          carrier_is_subring assms(3) by (auto simp del: poly_mult.simps)
  ultimately show ?thesis
    using coeff_iff_polynomial_cond by simp
qed
```

Note that "polynomial (carrier R) p" and "subring K p; polynomial K p" are "equivalent" assumptions for any lemma in ring which the result doesn't depend on K, because carrier is a subring and a polynomial for a subset of the carrier is a carrier polynomial. The decision between one of them should be based on how the lemma is going to be used and proved. These are some tips: (a) Lemmas about the algebraic structure of polynomials should use the latter option. (b) Also, if the lemma deals with lots of polynomials, then the latter option is preferred. (c) If the proof is going to be much easier with the first option, do not hesitate.

```
lemma poly_mult_monom':
  assumes "set p ⊆ carrier R" "a ∈ carrier R"
  shows "poly_mult (monom a n) p = normalize ((map ((⊗) a) p) @ (replicate
n 0))"
proof -
  have set_map: "set ((map ((⊗) a) p) @ (replicate n 0)) ⊆ carrier R"
    using assms by (induct p) (auto)
  show ?thesis
  using poly_mult_replicate_zero(1)[OF assms(1), of n]
        poly_add_zero'(1)[OF set_map]
  unfolding monom_def by simp
qed

lemma poly_mult_monom:
  assumes "polynomial (carrier R) p" "a ∈ carrier R - { 0 }"
  shows "poly_mult (monom a n) p =
          (if p = [] then [] else (poly_mult [ a ] p) @ (replicate n
0))"
proof (cases p)
  case Nil thus ?thesis
    using poly_mult_zero(2)[of "monom a n"] assms(2) monom_def by fastforce
next
  case (Cons b ps)
  hence "lead_coeff ((map (λb. a ⊗ b) p) @ (replicate n 0)) ≠ 0"
    using Cons assms integral[of a b] unfolding polynomial_def by auto
  thus ?thesis
    using poly_mult_monom'[OF polynomial_incl[OF assms(1)], of a n] assms(2)
Cons
    unfolding poly_mult_const(1)[OF carrier_is_subring assms] by simp
qed

lemma poly_mult_one':
  assumes "set p ⊆ carrier R"
  shows "poly_mult [ 1 ] p = normalize p" and "poly_mult p [ 1 ] = normalize
p"
proof -
  have "map2 (⊕) (map ((⊗) 1) p) (replicate (length p) 0) = p"
    using assms by (induct p) (auto)
  thus "poly_mult [ 1 ] p = normalize p" and "poly_mult p [ 1 ] = normalize
p"
    using poly_mult_comm[OF assms, of "[ 1 ]"] by auto
qed

lemma poly_mult_one:
  assumes "subring K R" "polynomial K p"
```

```
  shows "poly_mult [ 1 ] p = p" and "poly_mult p [ 1 ] = p"
  using poly_mult_one'[OF polynomial_in_carrier[OF assms]] normalize_polynomial[OF
assms(2)] by auto

lemma poly_mult_lead_coeff_aux:
  assumes "subring K R" "polynomial K p1" "polynomial K p2" and "p1 ≠
[]" and "p2 ≠ []"
  shows "(coeff (poly_mult p1 p2)) (degree p1 + degree p2) = (lead_coeff
p1) ⊗ (lead_coeff p2)"
proof -
  have p1: "lead_coeff p1 ∈ carrier R - { 0 }" and p2: "lead_coeff p2
∈ carrier R - { 0 }"
    using assms(2-5) lead_coeff_in_carrier[OF assms(1)] by (metis list.collapse)+

  have "(coeff (poly_mult p1 p2)) (degree p1 + degree p2) =
        (⨁ k ∈ {..((degree p1) + (degree p2))}.
          (coeff p1) k ⊗ (coeff p2) ((degree p1) + (degree p2) - k))"
    using poly_mult_coeff[OF assms(2-3)[THEN polynomial_in_carrier[OF
assms(1)]]] by simp
  also have " ... = (lead_coeff p1) ⊗ (lead_coeff p2)"
  proof -
    let ?f = "λi. (coeff p1) i ⊗ (coeff p2) ((degree p1) + (degree p2)
- i)"
    have in_carrier: "⋀i. (coeff p1) i ∈ carrier R" "⋀i. (coeff p2)
i ∈ carrier R"
      using coeff_in_carrier assms by auto
    have "⋀i. i < degree p1 ⟹ ?f i = 0"
      using coeff_degree[of p2] in_carrier by auto
    moreover have "⋀i. i > degree p1 ⟹ ?f i = 0"
      using coeff_degree[of p1] in_carrier by auto
    moreover have "?f (degree p1) = (lead_coeff p1) ⊗ (lead_coeff p2)"
      using assms(4-5) lead_coeff_simp by simp
    ultimately have "?f = (λi. if degree p1 = i then (lead_coeff p1) ⊗
(lead_coeff p2) else 0)"
      using nat_neq_iff by auto
    thus ?thesis
      using add.finprod_singleton[of "degree p1" "{..((degree p1) + (degree
p2))}"
                                      "λi. (lead_coeff p1) ⊗ (lead_coeff
p2)"] p1 p2 by auto
  qed
  finally show ?thesis .
qed

lemma poly_mult_degree_eq:
  assumes "subring K R" "polynomial K p1" "polynomial K p2"
  shows "degree (poly_mult p1 p2) = (if p1 = [] ∨ p2 = [] then 0 else
(degree p1) + (degree p2))"
proof (cases p1)
```

```
  case Nil thus ?thesis by simp
next
  case (Cons a p1') note p1 = Cons
  show ?thesis
  proof (cases p2)
    case Nil thus ?thesis
      using poly_mult_zero(2)[OF polynomial_in_carrier[OF assms(1-2)]]
by simp
  next
    case (Cons b p2') note p2 = Cons
    have a: "a ∈ carrier R" and b: "b ∈ carrier R"
      using p1 p2 polynomial_in_carrier[OF assms(1-2)] polynomial_in_carrier[OF
assms(1,3)] by auto
    have "(coeff (poly_mult p1 p2)) ((degree p1) + (degree p2)) = a ⊗
b"
      using poly_mult_lead_coeff_aux[OF assms] p1 p2 by simp
    hence neq0: "(coeff (poly_mult p1 p2)) ((degree p1) + (degree p2))
≠ 0"
      using assms(2-3) integral[of a b] lead_coeff_in_carrier[OF assms(1)]
p1 p2 by auto
    moreover have eq0: "⋀i. i > (degree p1) + (degree p2) ⟹ (coeff
(poly_mult p1 p2)) i = 0"
    proof -
      have aux_lemma: "degree (poly_mult p1 p2) ≤ (degree p1) + (degree
p2)"
      proof (induct p1)
        case Nil
        then show ?case by simp
      next
        case (Cons a p1)
        let ?a_p2 = "(map (λb. a ⊗ b) p2) @ (replicate (degree (a # p1))
0)"
        have "poly_mult (a # p1) p2 = poly_add ?a_p2 (poly_mult p1 p2)"
by simp
        hence "degree (poly_mult (a # p1) p2) ≤ max (degree ?a_p2) (degree
(poly_mult p1 p2))"
          using poly_add_degree[of ?a_p2 "poly_mult p1 p2"] by simp
        also have " ... ≤ max ((degree (a # p1)) + (degree p2)) (degree
(poly_mult p1 p2))"
          by auto
        also have " ... ≤ max ((degree (a # p1)) + (degree p2)) ((degree
p1) + (degree p2))"
          using Cons by simp
        also have " ... ≤ (degree (a # p1)) + (degree p2)"
          by auto
        finally show ?case .
      qed
      fix i show "i > (degree p1) + (degree p2) ⟹ (coeff (poly_mult
p1 p2)) i = 0"
```

```
        using coeff_degree aux_lemma by simp
    qed
    moreover have "polynomial K (poly_mult p1 p2)"
        by (simp add: assms poly_mult_closed)
    ultimately have "degree (poly_mult p1 p2) = degree p1 + degree p2"
      by (metis (no_types) assms(1) coeff.simps(1) coeff_degree domain.poly_mult_one(1)
domain_axioms eq0 lead_coeff_simp length_greater_0_conv neq0 normalize_length_lt
not_less_iff_gr_or_eq poly_mult_one'(1) polynomial_in_carrier)
    thus ?thesis
      using p1 p2 by auto
  qed
qed

lemma poly_mult_integral:
  assumes "subring K R" "polynomial K p1" "polynomial K p2"
  shows "poly_mult p1 p2 = [] ⟹ p1 = [] ∨ p2 = []"
proof (rule ccontr)
  assume A: "poly_mult p1 p2 = []" "¬ (p1 = [] ∨ p2 = [])"
  hence "degree (poly_mult p1 p2) = degree p1 + degree p2"
    using poly_mult_degree_eq[OF assms] by simp
  hence "length p1 = 1 ∧ length p2 = 1"
    using A Suc_diff_Suc by fastforce
  then obtain a b where p1: "p1 = [ a ]" and p2: "p2 = [ b ]"
    by (metis One_nat_def length_0_conv length_Suc_conv)
  hence "a ∈ carrier R - { 𝟬 }" and "b ∈ carrier R - { 𝟬 }"
    using assms lead_coeff_in_carrier by auto
  hence "poly_mult [ a ] [ b ] = [ a ⊗ b ]"
    using integral by auto
  thus False using A(1) p1 p2 by simp
qed

lemma poly_mult_lead_coeff:
  assumes "subring K R" "polynomial K p1" "polynomial K p2" and "p1 ≠
[]" and "p2 ≠ []"
  shows "lead_coeff (poly_mult p1 p2) = (lead_coeff p1) ⊗ (lead_coeff
p2)"
proof -
  have "poly_mult p1 p2 ≠ []"
    using poly_mult_integral[OF assms(1-3)] assms(4-5) by auto
  hence "lead_coeff (poly_mult p1 p2) = (coeff (poly_mult p1 p2)) (degree
p1 + degree p2)"
    using poly_mult_degree_eq[OF assms(1-3)] assms(4-5) by (metis coeff.simps(2)
list.collapse)
  thus ?thesis
    using poly_mult_lead_coeff_aux[OF assms] by simp
qed

lemma poly_mult_append_zero_lcancel:
  assumes "subring K R" and "polynomial K p" "polynomial K q"
```

```
  shows "poly_mult (p @ [ 0 ]) q = r @ [ 0 ] ⟹ poly_mult p q = r"
proof -
  note in_carrier = assms(2-3)[THEN polynomial_in_carrier[OF assms(1)]]

  assume pmult: "poly_mult (p @ [ 0 ]) q = r @ [ 0 ]"
  have "poly_mult (p @ [ 0 ]) q = []" if "q = []"
    using poly_mult_zero(2)[of "p @ [ 0 ]"] that in_carrier(1) by auto
  moreover have "poly_mult (p @ [ 0 ]) q = []" if "p = []"
    using poly_mult_normalize[OF _ in_carrier(2), of "p @ [ 0 ]"] poly_mult_zero[OF
in_carrier(2)]
    unfolding that by auto
  ultimately have "p ≠ []" and "q ≠ []"
    using pmult by auto
  hence "poly_mult p q ≠ []"
    using poly_mult_integral[OF assms] by auto
  hence "normalize ((poly_mult p q) @ [ 0 ]) = (poly_mult p q) @ [ 0
]"
    using normalize_polynomial[OF append_is_polynomial[OF assms(1) poly_mult_closed[OF
assms], of "Suc 0"]] by auto
  thus "poly_mult p q = r"
    using poly_mult_append_zero[OF assms(2-3)[THEN polynomial_in_carrier[OF
assms(1)]]] pmult by simp
qed

lemma poly_mult_append_zero_rcancel:
  assumes "subring K R" and "polynomial K p" "polynomial K q"
  shows "poly_mult p (q @ [ 0 ]) = r @ [ 0 ] ⟹ poly_mult p q = r"
  using poly_mult_append_zero_lcancel[OF assms(1,3,2)]
        poly_mult_comm[of p "q @ [ 0 ]"] poly_mult_comm[of p q]
        assms(2-3)[THEN polynomial_in_carrier[OF assms(1)]]
  by auto

end
```

## 33.7 Algebraic Structure of Polynomials

```
definition univ_poly :: "('a, 'b) ring_scheme ⇒'a set ⇒ ('a list) ring"
   (‹(‹open_block notation=‹postfix X››_ [X]ı)› 80)
  where "univ_poly R K =
         ⦇ carrier = { p. polynomial_R K p },
              mult = ring.poly_mult R,
               one = [ 1_R ],
              zero = [],
               add = ring.poly_add R ⦈"
```

These lemmas allow you to unfold one field of the record at a time.

```
lemma univ_poly_carrier: "polynomial_R K p ⟷ p ∈ carrier (K[X]_R)"
  unfolding univ_poly_def by simp
```

```
lemma univ_poly_mult: "mult (K[X]R) = ring.poly_mult R"
  unfolding univ_poly_def by simp

lemma univ_poly_one: "one (K[X]R) = [ 1R ]"
  unfolding univ_poly_def by simp

lemma univ_poly_zero: "zero (K[X]R) = []"
  unfolding univ_poly_def by simp

lemma univ_poly_add: "add (K[X]R) = ring.poly_add R"
  unfolding univ_poly_def by simp



lemma univ_poly_zero_closed [intro]: "[] ∈ carrier (K[X]R)"
  unfolding sym[OF univ_poly_carrier] polynomial_def by simp


context domain
begin

lemma poly_mult_monom_assoc:
  assumes "set p ⊆ carrier R" "set q ⊆ carrier R" and "a ∈ carrier R"
    shows "poly_mult (poly_mult (monom a n) p) q =
           poly_mult (monom a n) (poly_mult p q)"
proof (induct n)
  case 0 thus ?case
    unfolding monom_def using poly_mult_semiassoc[OF assms] by (auto simp
del: poly_mult.simps)
next
  case (Suc n)
  have "poly_mult (poly_mult (monom a (Suc n)) p) q =
        poly_mult (normalize ((poly_mult (monom a n) p) @ [ 0 ])) q"
    using poly_mult_append_zero[OF monom_in_carrier[OF assms(3), of n]
assms(1)]
    unfolding monom_def by (auto simp del: poly_mult.simps simp add: replicate_append_same)
  also have " ... = normalize ((poly_mult (poly_mult (monom a n) p) q)
@ [ 0 ])"
    using poly_mult_normalize[OF _ assms(2)] poly_mult_append_zero[OF
_ assms(2)]
          poly_mult_in_carrier[OF monom_in_carrier[OF assms(3), of n]
assms(1)] by auto
  also have " ... = normalize ((poly_mult (monom a n) (poly_mult p q))
@ [ 0 ])"
    using Suc by simp
  also have " ... = poly_mult (monom a (Suc n)) (poly_mult p q)"
    using poly_mult_append_zero[OF monom_in_carrier[OF assms(3), of n]
                                   poly_mult_in_carrier[OF assms(1-2)]]
    unfolding monom_def by (simp add: replicate_append_same)
```

```
  finally show ?case .
qed


context
  fixes K :: "'a set" assumes K: "subring K R"
begin

lemma univ_poly_is_monoid: "monoid (K[X])"
  unfolding univ_poly_def using poly_mult_one[OF K]
proof (auto simp add: K poly_add_closed poly_mult_closed one_is_polynomial
monoid_def)
  fix p1 p2 p3
  let ?P = "poly_mult (poly_mult p1 p2) p3 = poly_mult p1 (poly_mult p2
p3)"

  assume A: "polynomial K p1" "polynomial K p2" "polynomial K p3"
  show ?P using polynomial_in_carrier[OF K A(1)]
  proof (induction p1)
    case Nil thus ?case by simp
  next
next
    case (Cons a p1) thus ?case
    proof (cases "a = 0")
      assume eq_zero: "a = 0"
      have p1: "set p1 ⊆ carrier R"
        using Cons(2) by simp
      have "poly_mult (poly_mult (a # p1) p2) p3 = poly_mult (poly_mult
p1 p2) p3"
        using poly_mult_prepend_replicate_zero[OF p1 polynomial_in_carrier[OF
K A(2)], of "Suc 0"]
              eq_zero by simp
      also have " ... = poly_mult p1 (poly_mult p2 p3)"
        using p1[THEN Cons(1)] by simp
      also have " ... = poly_mult (a # p1) (poly_mult p2 p3)"
        using poly_mult_prepend_replicate_zero[OF p1
              poly_mult_in_carrier[OF A(2-3)[THEN polynomial_in_carrier[OF
K]]], of "Suc 0"] eq_zero
        by simp
      finally show ?thesis .
    next
      assume "a ≠ 0" hence in_carrier:
        "set p1 ⊆ carrier R" "set p2 ⊆ carrier R" "set p3 ⊆ carrier R"
"a ∈ carrier R - { 0 }"
        using A(2-3) polynomial_in_carrier[OF K] Cons by auto

      let ?a_p2 = "(map (λb. a ⊗ b) p2) @ (replicate (length p1) 0)"
      have a_p2_in_carrier: "set ?a_p2 ⊆ carrier R"
        using in_carrier by auto
```

```
      have "poly_mult (poly_mult (a # p1) p2) p3 = poly_mult (poly_add
?a_p2 (poly_mult p1 p2)) p3"
        by simp
      also have " ... = poly_add (poly_mult ?a_p2 p3) (poly_mult (poly_mult
p1 p2) p3)"
        using poly_mult_l_distr'[OF a_p2_in_carrier poly_mult_in_carrier[OF
in_carrier(1-2)] in_carrier(3)] .
      also have " ... = poly_add (poly_mult ?a_p2 p3) (poly_mult p1 (poly_mult
p2 p3))"
        using Cons(1)[OF in_carrier(1)] by simp
      also have " ... = poly_add (poly_mult (normalize ?a_p2) p3) (poly_mult
p1 (poly_mult p2 p3))"
        using poly_mult_normalize[OF a_p2_in_carrier in_carrier(3)] by
simp
      also have " ... = poly_add (poly_mult (poly_mult (monom a (length
p1)) p2) p3)
                                 (poly_mult p1 (poly_mult p2 p3))"
        using poly_mult_monom'[OF in_carrier(2), of a "length p1"] in_carrier(4)
by simp
      also have " ... = poly_add (poly_mult (a # (replicate (length p1)
0)) (poly_mult p2 p3))
                                 (poly_mult p1 (poly_mult p2 p3))"
        using poly_mult_monom_assoc[of p2 p3 a "length p1"] in_carrier
unfolding monom_def by simp
      also have " ... = poly_mult (poly_add (a # (replicate (length p1)
0)) p1) (poly_mult p2 p3)"
        using poly_mult_l_distr'[of "a # (replicate (length p1) 0)" p1
"poly_mult p2 p3"]
              poly_mult_in_carrier[OF in_carrier(2-3)] in_carrier by force
      also have " ... = poly_mult (a # p1) (poly_mult p2 p3)"
        using poly_add_monom[OF in_carrier(1) in_carrier(4)] unfolding
monom_def by simp
      finally show ?thesis .
    qed
  qed
qed

declare poly_add.simps[simp del]

lemma univ_poly_is_abelian_monoid: "abelian_monoid (K[X])"
  unfolding univ_poly_def
  using poly_add_closed poly_add_zero zero_is_polynomial K
proof (auto simp add: abelian_monoid_def comm_monoid_def monoid_def comm_monoid_axioms_def)
  fix p1 p2 p3
  let ?c = "λp. coeff p"
  assume A: "polynomial K p1" "polynomial K p2" "polynomial K p3"
  hence
    p1: "⋀i. (?c p1) i ∈ carrier R" "set p1 ⊆ carrier R" and
```

```
    p2: "⋀i. (?c p2) i ∈ carrier R" "set p2 ⊆ carrier R" and
    p3: "⋀i. (?c p3) i ∈ carrier R" "set p3 ⊆ carrier R"
    using A[THEN polynomial_in_carrier[OF K]] coeff_in_carrier by auto
  have "?c (poly_add (poly_add p1 p2) p3) = (λi. (?c p1 i ⊕ ?c p2 i)
⊕ (?c p3 i))"
    using poly_add_coeff[OF poly_add_in_carrier[OF p1(2) p2(2)] p3(2)]
          poly_add_coeff[OF p1(2) p2(2)] by simp
  also have " ... = (λi. (?c p1 i) ⊕ ((?c p2 i) ⊕ (?c p3 i)))"
    using p1 p2 p3 add.m_assoc by simp
  also have " ... = ?c (poly_add p1 (poly_add p2 p3))"
    using poly_add_coeff[OF p1(2) poly_add_in_carrier[OF p2(2) p3(2)]]
          poly_add_coeff[OF p2(2) p3(2)] by simp
  finally have "?c (poly_add (poly_add p1 p2) p3) = ?c (poly_add p1 (poly_add
p2 p3))" .
  thus "poly_add (poly_add p1 p2) p3 = poly_add p1 (poly_add p2 p3)"
    using coeff_iff_polynomial_cond poly_add_closed[OF K] A by meson
  show "poly_add p1 p2 = poly_add p2 p1"
    using poly_add_comm[OF p1(2) p2(2)] .
qed

lemma univ_poly_is_abelian_group: "abelian_group (K[X])"
proof -
  interpret abelian_monoid "K[X]"
    using univ_poly_is_abelian_monoid .
  show ?thesis
  proof (unfold_locales)
    show "carrier (add_monoid (K[X])) ⊆ Units (add_monoid (K[X]))"
      unfolding univ_poly_def Units_def
    proof (auto)
      fix p assume p: "polynomial K p"
      have "polynomial K [ ⊖ 1 ]"
        unfolding polynomial_def using r_neg subringE(3,5)[OF K] by force
      hence cond0: "polynomial K (poly_mult [ ⊖ 1 ] p)"
        using poly_mult_closed[OF K, of "[ ⊖ 1 ]" p] p by simp

      have "poly_add p (poly_mult [ ⊖ 1 ] p) = poly_add (poly_mult [
1 ] p) (poly_mult [ ⊖ 1 ] p)"
        using poly_mult_one[OF K p] by simp
      also have " ... = poly_mult (poly_add [ 1 ] [ ⊖ 1 ]) p"
        using poly_mult_l_distr' polynomial_in_carrier[OF K p] by auto
      also have " ... = poly_mult [] p"
        using poly_add.simps[of "[ 1 ]" "[ ⊖ 1 ]"]
        by (simp add: case_prod_unfold r_neg)
      also have " ... = []" by simp
      finally have cond1: "poly_add p (poly_mult [ ⊖ 1 ] p) = []" .

      have "poly_add (poly_mult [ ⊖ 1 ] p) p = poly_add (poly_mult [
⊖ 1 ] p) (poly_mult [ 1 ] p)"
        using poly_mult_one[OF K p] by simp
```

```
      also have " ... = poly_mult (poly_add [ ⊖  1 ] [ 1 ]) p"
        using poly_mult_l_distr' polynomial_in_carrier[OF K p] by auto
      also have " ... = poly_mult [] p"
        using ‹poly_mult (poly_add [1] [⊖ 1]) p = poly_mult [] p› poly_add_comm
by auto
      also have " ... = []" by simp
      finally have cond2: "poly_add (poly_mult [ ⊖ 1 ] p) p = []" .

      from cond0 cond1 cond2 show "∃q. polynomial K q ∧ poly_add q p
= [] ∧ poly_add p q = []"
        by auto
    qed
  qed
qed

lemma univ_poly_is_ring: "ring (K[X])"
proof -
  interpret UP: abelian_group "K[X]" + monoid "K[X]"
    using univ_poly_is_abelian_group univ_poly_is_monoid .
  show ?thesis
    by (unfold_locales)
       (auto simp add: univ_poly_def poly_mult_r_distr[OF K] poly_mult_l_distr[OF
K])
qed

lemma univ_poly_is_cring: "cring (K[X])"
proof -
  interpret UP: ring "K[X]"
    using univ_poly_is_ring .
  have "⋀p q. ⟦ p ∈ carrier (K[X]); q ∈ carrier (K[X]) ⟧ ⟹ p ⊗K[X]
q = q ⊗K[X] p"
    unfolding univ_poly_def using poly_mult_comm polynomial_in_carrier[OF
K] by auto
  thus ?thesis
    by unfold_locales auto
qed

lemma univ_poly_is_domain: "domain (K[X])"
proof -
  interpret UP: cring "K[X]"
    using univ_poly_is_cring .
  show ?thesis
    by (unfold_locales, auto simp add: univ_poly_def poly_mult_integral[OF
K])
qed

declare poly_add.simps[simp]

lemma univ_poly_a_inv_def':
```

```
  assumes "p ∈ carrier (K[X])" shows "⊖K[X] p = map (λa. ⊖ a) p"
proof -
  have aux_lemma:
    "⋀p. p ∈ carrier (K[X]) ⟹ p ⊕K[X] (map (λa. ⊖ a) p) = []"
    "⋀p. p ∈ carrier (K[X]) ⟹ (map (λa. ⊖ a) p) ∈ carrier (K[X])"
  proof -
    fix p assume p: "p ∈ carrier (K[X])"
    hence set_p: "set p ⊆ K"
      unfolding univ_poly_def using polynomial_incl by auto
    show "(map (λa. ⊖ a) p) ∈ carrier (K[X])"
    proof (cases "p = []")
      assume "p = []" thus ?thesis
        unfolding univ_poly_def polynomial_def by auto
    next
      assume not_nil: "p ≠ []"
      hence "lead_coeff p ≠ 0"
        using p unfolding univ_poly_def polynomial_def by auto
      moreover have "lead_coeff (map (λa. ⊖ a) p) = ⊖ (lead_coeff p)"
        using not_nil by (simp add: hd_map)
      ultimately have "lead_coeff (map (λa. ⊖ a) p) ≠ 0"
        using hd_in_set local.minus_zero not_nil set_p subringE(1)[OF
K] by force
      moreover have "set (map (λa. ⊖ a) p) ⊆ K"
        using set_p subringE(5)[OF K] by (induct p) (auto)
      ultimately show ?thesis
        unfolding univ_poly_def polynomial_def by simp
    qed

    have "map2 (⊕) p (map (λa. ⊖ a) p) = replicate (length p) 0"
      using set_p subringE(1)[OF K] by (induct p) (auto simp add: r_neg)
    thus "p ⊕K[X] (map (λa. ⊖ a) p) = []"
      unfolding univ_poly_def using normalize_replicate_zero[of "length
p" "[]"] by auto
  qed

  interpret UP: ring "K[X]"
    using univ_poly_is_ring .

  from aux_lemma
  have "⋀p. p ∈ carrier (K[X]) ⟹ ⊖K[X] p = map (λa. ⊖ a) p"
    by (metis Nil_is_map_conv UP.add.inv_closed UP.l_zero UP.r_neg1 UP.r_zero
UP.zero_closed)
  thus ?thesis
    using assms by simp
qed


corollary univ_poly_a_inv_length:
  assumes "p ∈ carrier (K[X])" shows "length (⊖K[X] p) = length p"
```

```
  unfolding univ_poly_a_inv_def'[OF assms] by simp


corollary univ_poly_a_inv_degree:
  assumes "p ∈ carrier (K[X])" shows "degree (⊖K[X] p) = degree p"
  using univ_poly_a_inv_length[OF assms] by simp
```

## 33.8 Long Division Theorem

```
lemma long_division_theorem:
  assumes "polynomial K p" and "polynomial K b" "b ≠ []"
     and "lead_coeff b ∈ Units (R (| carrier := K |))"
  shows "∃q r. polynomial K q ∧ polynomial K r ∧
             p = (b ⊗K[X] q) ⊕K[X] r ∧ (r = [] ∨ degree r < degree
b)"
    (is "∃q r. ?long_division p q r")
  using assms(1)
proof (induct "length p" arbitrary: p rule: less_induct)
  case less thus ?case
  proof (cases p)
    case Nil
    hence "?long_division p [] []"
      using zero_is_polynomial poly_mult_zero[OF polynomial_in_carrier[OF
K assms(2)]]
      by (simp add: univ_poly_def)
    thus ?thesis by blast
  next
    case (Cons a p') thus ?thesis
    proof (cases "length b > length p")
      assume "length b > length p"
      hence "p = [] ∨ degree p < degree b"
        by (meson diff_less_mono length_0_conv less_one not_le)
      hence "?long_division p [] p"
        using poly_mult_zero(2)[OF polynomial_in_carrier[OF K assms(2)]]
              poly_add_zero(2)[OF K less(2)] zero_is_polynomial less(2)
        by (simp add: univ_poly_def)
      thus ?thesis by blast
    next
      interpret UP: cring "K[X]"
        using univ_poly_is_cring .

      assume "¬ length b > length p"
      hence len_ge: "length p ≥ length b" by simp
      obtain c b' where b: "b = c # b'"
        using assms(3) list.exhaust_sel by blast
      then obtain c' where c': "c' ∈ carrier R" "c' ∈ K" "c' ⊗ c = 1"
"c ⊗ c' = 1"
        using assms(4) subringE(1)[OF K] unfolding Units_def by auto
      have c: "c ∈ carrier R" "c ∈ K" "c ≠ 0" and a: "a ∈ carrier R"
```

```
"a ∈ K" "a ≠ 0"
        using less(2) assms(2) lead_coeff_not_zero subringE(1)[OF K] b
Cons by auto
      hence lc: "c' ⊗ (⊖ a) ∈ K - { 0 }"
        using subringE(5-6)[OF K] c' add.inv_solve_right integral_iff
by fastforce

      let ?len = "length"
      define s where "s = monom (c' ⊗ (⊖ a)) (?len p - ?len b)"
      hence s: "polynomial K s" "s ≠ []" "degree s = ?len p - ?len b"
"length s ≥ 1"
        using monom_is_polynomial[OF K lc] unfolding monom_def by auto
      hence is_polynomial: "polynomial K (p ⊕K[X] (b ⊗K[X] s))"
        using poly_add_closed[OF K less(2) poly_mult_closed[OF K assms(2),
of s]]
        by (simp add: univ_poly_def)

      have "lead_coeff (b ⊗K[X] s) = ⊖ a"
        using poly_mult_lead_coeff[OF K assms(2) s(1) assms(3) s(2)] c
c' a
        unfolding b s_def monom_def univ_poly_def by (auto simp del: poly_mult.simps,
algebra)
      then obtain s' where s': "b ⊗K[X] s = (⊖ a) # s'"
        using poly_mult_integral[OF K assms(2) s(1)] assms(2-3) s(2)
        by (simp add: univ_poly_def, metis hd_Cons_tl)
      moreover have "degree p = degree (b ⊗K[X] s)"
        using poly_mult_degree_eq[OF K assms(2) s(1)] assms(3) s(2-4)
len_ge b Cons
        by (auto simp add: univ_poly_def)
      hence "?len p = ?len (b ⊗K[X] s)"
        unfolding Cons s' by simp
      hence "?len (p ⊕K[X] (b ⊗K[X] s)) < ?len p"
        unfolding Cons s' using a normalize_length_le[of "map2 (⊕) p'
s'"]
        by (auto simp add: univ_poly_def r_neg)
      then obtain q' r' where l_div: "?long_division (p ⊕K[X] (b ⊗K[X]
s)) q' r'"
        using less(1)[OF _ is_polynomial] by blast

      have in_carrier:
         "p ∈ carrier (K[X])"  "b ∈ carrier (K[X])" "s ∈ carrier (K[X])"
        "q' ∈ carrier (K[X])" "r' ∈ carrier (K[X])"
        using l_div assms less(2) s unfolding univ_poly_def by auto
      have "(p ⊕K[X] (b ⊗K[X] s)) ⊖K[X] (b ⊗K[X] s) =
          ((b ⊗K[X] q') ⊕K[X] r') ⊖K[X] (b ⊗K[X] s)"
        using l_div by simp
      hence "p = (b ⊗K[X] (q' ⊖K[X] s)) ⊕K[X] r'"
        using in_carrier by algebra
      moreover have "q' ⊖K[X] s ∈ carrier (K[X])"
```

```
        using in_carrier by algebra
      hence "polynomial K (q' ⊖K[X] s)"
        unfolding univ_poly_def by simp
      ultimately have "?long_division p (q' ⊖K[X] s) r'"
        using l_div by auto
      thus ?thesis by blast
    qed
  qed
qed

end

end
```

```
lemma (in domain) field_long_division_theorem:
  assumes "subfield K R" "polynomial K p" and "polynomial K b" "b ≠
[]"
  shows "∃q r. polynomial K q ∧ polynomial K r ∧
              p = (b ⊗K[X] q) ⊕K[X] r ∧ (r = [] ∨ degree r < degree
b)"
  using long_division_theorem[OF subfieldE(1)[OF assms(1)] assms(2-4)]
assms(3-4)
        subfield.subfield_Units[OF assms(1)] lead_coeff_not_zero[of K
"hd b" "tl b"]
  by simp
```

The same theorem as above, but now, everything is in a shell.

```
lemma (in domain) field_long_division_theorem_shell:
  assumes "subfield K R" "p ∈ carrier (K[X])" and "b ∈ carrier (K[X])"
"b ≠ 0K[X]"
  shows "∃q r. q ∈ carrier (K[X]) ∧ r ∈ carrier (K[X]) ∧
              p = (b ⊗K[X] q) ⊕K[X] r ∧ (r = 0K[X] ∨ degree r < degree
b)"
  using field_long_division_theorem assms by (auto simp add: univ_poly_def)
```

## 33.9 Consistency Rules

```
lemma polynomial_consistent [simp]:
  shows "polynomial(R ⦇ carrier := K ⦈) K p ⟹ polynomialR K p"
  unfolding polynomial_def by auto

lemma (in ring) eval_consistent [simp]:
  assumes "subring K R" shows "ring.eval (R ⦇ carrier := K ⦈) = eval"
proof
  fix p show "ring.eval (R ⦇ carrier := K ⦈) p = eval p"
    using nat_pow_consistent ring.eval.simps[OF subring_is_ring[OF assms]]
by (induct p) (auto)
qed
```

```
lemma (in ring) coeff_consistent [simp]:
  assumes "subring K R" shows "ring.coeff (R ⦇ carrier := K ⦈) = coeff"
proof
  fix p show "ring.coeff (R ⦇ carrier := K ⦈) p = coeff p"
    using ring.coeff.simps[OF subring_is_ring[OF assms]] by (induct p)
(auto)
qed

lemma (in ring) normalize_consistent [simp]:
  assumes "subring K R" shows "ring.normalize (R ⦇ carrier := K ⦈) =
normalize"
proof
  fix p show "ring.normalize (R ⦇ carrier := K ⦈) p = normalize p"
    using ring.normalize.simps[OF subring_is_ring[OF assms]] by (induct
p) (auto)
qed

lemma (in ring) poly_add_consistent [simp]:
  assumes "subring K R" shows "ring.poly_add (R ⦇ carrier := K ⦈) = poly_add"

proof -
  have "⋀p q. ring.poly_add (R ⦇ carrier := K ⦈) p q = poly_add p q"
  proof -
    fix p q show "ring.poly_add (R ⦇ carrier := K ⦈) p q = poly_add p
q"
    using ring.poly_add.simps[OF subring_is_ring[OF assms]] normalize_consistent[OF
assms] by auto
  qed
  thus ?thesis by (auto simp del: poly_add.simps)
qed

lemma (in ring) poly_mult_consistent [simp]:
  assumes "subring K R" shows "ring.poly_mult (R ⦇ carrier := K ⦈) =
poly_mult"
proof -
  have "⋀p q. ring.poly_mult (R ⦇ carrier := K ⦈) p q = poly_mult p q"
  proof -
    fix p q show "ring.poly_mult (R ⦇ carrier := K ⦈) p q = poly_mult
p q"
      using ring.poly_mult.simps[OF subring_is_ring[OF assms]] poly_add_consistent[OF
assms]
      by (induct p) (auto)
  qed
  thus ?thesis by auto
qed

lemma (in domain) univ_poly_a_inv_consistent:
  assumes "subring K R" "p ∈ carrier (K[X])"
```

```
  shows "⊖K[X] p = ⊖(carrier R)[X] p"
proof -
  have in_carrier: "p ∈ carrier ((carrier R)[X])"
    using assms carrier_polynomial by (auto simp add: univ_poly_def)
  show ?thesis
    using univ_poly_a_inv_def'[OF assms]
          univ_poly_a_inv_def'[OF carrier_is_subring in_carrier] by simp
qed

lemma (in domain) univ_poly_a_minus_consistent:
  assumes "subring K R" "q ∈ carrier (K[X])"
  shows "p ⊖K[X] q = p ⊖(carrier R)[X] q"
  using univ_poly_a_inv_consistent[OF assms]
  unfolding a_minus_def univ_poly_def by auto

lemma (in ring) univ_poly_consistent:
  assumes "subring K R"
  shows "univ_poly (R ⦇ carrier := K ⦈) = univ_poly R"
  unfolding univ_poly_def polynomial_def
  using poly_add_consistent[OF assms]
        poly_mult_consistent[OF assms]
        subringE(1)[OF assms]
  by auto
```

### 33.9.1 Corollaries

```
corollary (in ring) subfield_long_division_theorem_shell:
  assumes "subfield K R" "p ∈ carrier (K[X])" and "b ∈ carrier (K[X])"
"b ≠ 𝟬K[X]"
  shows "∃q r. q ∈ carrier (K[X]) ∧ r ∈ carrier (K[X]) ∧
               p = (b ⊗K[X] q) ⊕K[X] r ∧ (r = 𝟬K[X] ∨ degree r < degree
b)"
  using domain.field_long_division_theorem_shell[OF subdomain_is_domain[OF
subfield.axioms(1)]
        field.carrier_is_subfield[OF subfield_iff(2)[OF assms(1)]]] assms(1-4)
  unfolding univ_poly_consistent[OF subfieldE(1)[OF assms(1)]]
  by auto

corollary (in domain) univ_poly_is_euclidean:
  assumes "subfield K R" shows "euclidean_domain (K[X]) degree"
proof -
  interpret UP: domain "K[X]"
    using univ_poly_is_domain[OF subfieldE(1)[OF assms]] field_def by
blast
  show ?thesis
    using subfield_long_division_theorem_shell[OF assms]
    by (auto intro!: UP.euclidean_domainI)
qed
```

```
corollary (in domain) univ_poly_is_principal:
  assumes "subfield K R" shows "principal_domain (K[X])"
proof -
  interpret UP: euclidean_domain "K[X]" degree
    using univ_poly_is_euclidean[OF assms] .
  show ?thesis ..
qed
```

## 33.10 The Evaluation Homomorphism

```
lemma (in ring) eval_replicate:
  assumes "set p ⊆ carrier R" "a ∈ carrier R"
  shows "eval ((replicate n 0) @ p) a = eval p a"
  using assms eval_in_carrier by (induct n) (auto)

lemma (in ring) eval_normalize:
  assumes "set p ⊆ carrier R" "a ∈ carrier R"
  shows "eval (normalize p) a = eval p a"
  using eval_replicate[OF normalize_in_carrier] normalize_def'[of p] assms
by metis

lemma (in ring) eval_poly_add_aux:
  assumes "set p ⊆ carrier R" "set q ⊆ carrier R" and "length p = length
q" and "a ∈ carrier R"
  shows "eval (poly_add p q) a = (eval p a) ⊕ (eval q a)"
proof -
  have "eval (map2 (⊕) p q) a = (eval p a) ⊕ (eval q a)"
    using assms
  proof (induct p arbitrary: q)
    case Nil thus ?case by simp
  next
    case (Cons b1 p')
    then obtain b2 q' where q: "q = b2 # q'"
      by (metis length_Cons list.exhaust list.size(3) nat.simps(3))
    show ?case
      using eval_in_carrier[OF _ Cons(5), of q']
            eval_in_carrier[OF _ Cons(5), of p'] Cons unfolding q
      by (auto simp add: ring_simprules(7,13,22))
  qed
  moreover have "set (map2 (⊕) p q) ⊆ carrier R"
    using assms(1-2)
    by (induct p arbitrary: q) (auto, metis add.m_closed in_set_zipE set_ConsD
subsetCE)
  ultimately show ?thesis
    using assms(3) eval_normalize[OF _ assms(4), of "map2 (⊕) p q"] by
auto
qed

lemma (in ring) eval_poly_add:
```

```
  assumes "set p ⊆ carrier R" "set q ⊆ carrier R" and "a ∈ carrier R"
  shows "eval (poly_add p q) a = (eval p a) ⊕ (eval q a)"
proof -
  have aux_lemma: "eval (poly_add p q) a = (eval p a) ⊕ (eval q a)"
    if A: "set p ⊆ carrier R" "set q ⊆ carrier R" "length p ≥ length
q" for p q
  proof -
    from that have "eval (poly_add p ((replicate (length p - length q)
0) @ q)) a =
         (eval p a) ⊕ (eval ((replicate (length p - length q) 0) @ q)
a)"
      using eval_poly_add_aux[OF A(1) _ _ assms(3), of "(replicate (length
p - length q) 0) @ q"] by force
    then show "eval (poly_add p q) a = (eval p a) ⊕ (eval q a)"
      using eval_replicate[OF A(2) assms(3)] A(3) by auto
  qed
  have ?thesis if "length q ≥ length p"
    using assms(1-2)[THEN eval_in_carrier[OF _ assms(3)]] poly_add_comm[OF
assms(1-2)]
          aux_lemma[OF assms(2,1) that]
    by (auto simp del: poly_add.simps simp add: add.m_comm)
  moreover have ?thesis if "length p ≥ length q"
    using aux_lemma[OF assms(1-2) that] .
  ultimately show ?thesis by auto
qed

lemma (in ring) eval_append_aux:
  assumes "set p ⊆ carrier R" and "b ∈ carrier R" and "a ∈ carrier
R"
  shows "eval (p @ [ b ]) a = ((eval p a) ⊗ a) ⊕ b"
  using assms(1)
proof (induct p)
  case Nil thus ?case by (auto simp add: assms(2-3))
next
  case (Cons l q)
  have "a [^] length q ∈ carrier R" "eval q a ∈ carrier R"
    using eval_in_carrier Cons(2) assms(2-3) by auto
  thus ?case
    using Cons assms(2-3) by (auto, algebra)
qed

lemma (in ring) eval_append:
  assumes "set p ⊆ carrier R" "set q ⊆ carrier R" and "a ∈ carrier R"
  shows "eval (p @ q) a = ((eval p a) ⊗ (a [^] (length q))) ⊕ (eval q
a)"
  using assms(2)
proof (induct "length q" arbitrary: q)
  case 0 thus ?case
    using eval_in_carrier[OF assms(1,3)] by auto
```

```
next
  case (Suc n)
  then obtain b q' where q: "q = q' @ [ b ]"
    by (metis length_Suc_conv list.simps(3) rev_exhaust)
  hence in_carrier: "eval p a ∈ carrier R" "eval q' a ∈ carrier R"
                    "a [^] (length q') ∈ carrier R" "b ∈ carrier R"
    using assms(1,3) Suc(3) eval_in_carrier[OF _ assms(3)] by auto

  have "eval (p @ q) a = ((eval (p @ q') a) ⊗ a) ⊕ b"
    using eval_append_aux[OF _ _ assms(3), of "p @ q'" b] assms(1) Suc(3)
unfolding q by auto
  also have " ... = ((((eval p a) ⊗ (a [^] (length q'))) ⊕ (eval q' a))
⊗ a) ⊕ b"
    using Suc unfolding q by auto
  also have " ... = (((eval p a) ⊗ ((a [^] (length q')) ⊗ a))) ⊕ (((eval
q' a) ⊗ a) ⊕ b)"
    using assms(3) in_carrier by algebra
  also have " ... = (eval p a) ⊗ (a [^] (length q)) ⊕ (eval q a)"
    using eval_append_aux[OF _ in_carrier(4) assms(3), of q'] Suc(3) un-
folding q by auto
  finally show ?case .
qed

lemma (in ring) eval_monom:
  assumes "b ∈ carrier R" and "a ∈ carrier R"
  shows "eval (monom b n) a = b ⊗ (a [^] n)"
proof (induct n)
  case 0 thus ?case
    using assms unfolding monom_def by auto
next
  case (Suc n)
  have "monom b (Suc n) = (monom b n) @ [ 0 ]"
    unfolding monom_def by (simp add: replicate_append_same)
  hence "eval (monom b (Suc n)) a = ((eval (monom b n) a) ⊗ a) ⊕ 0"
    using eval_append_aux[OF monom_in_carrier[OF assms(1)] zero_closed
assms(2), of n] by simp
  also have " ... =  b ⊗ (a [^] (Suc n))"
    using Suc assms m_assoc by auto
  finally show ?case .
qed

lemma (in cring) eval_poly_mult:
  assumes "set p ⊆ carrier R" "set q ⊆ carrier R" and "a ∈ carrier R"
  shows "eval (poly_mult p q) a = (eval p a) ⊗ (eval q a)"
  using assms(1)
proof (induct p)
  case Nil thus ?case
    using eval_in_carrier[OF assms(2-3)] by simp
next
```

```
  case (Cons b p)

  have aux_lemma: "eval ((map ((⊗) b) q) @ (replicate n 0)) a = (eval
(monom b n) a) ⊗ (eval q a)"
    if b: "b ∈ carrier R" for n b
  proof -
    from that have "set (map ((⊗) b) q) ⊆ carrier R" and "set (replicate
n 0) ⊆ carrier R"
      using assms(2) by (induct q) (auto)
    hence "eval ((map ((⊗) b) q) @ (replicate n 0)) a = (eval ((map ((⊗)
b) q)) a) ⊗ (a [^] n) ⊕ 0"
      using eval_append[OF _ _ assms(3), of "map ((⊗) b) q" "replicate
n 0"]
            eval_replicate[OF _ assms(3), of "[]"] by auto
    moreover have "eval (map ((⊗) b) q) a = b ⊗ eval q a"
      using assms(2-3) eval_in_carrier b by(induct q) (auto simp add:
m_assoc r_distr)
    ultimately have "eval ((map ((⊗) b) q) @ (replicate n 0)) a = (b
⊗ eval q a) ⊗ (a [^] n) ⊕ 0"
      by simp
    also have " ... = (b ⊗ (a [^] n)) ⊗ (eval q a)"
      using eval_in_carrier[OF assms(2-3)] b assms(3) m_assoc m_comm by
auto
    finally show ?thesis
      using eval_monom[OF b assms(3)] by simp
  qed

  from Cons have in_carrier:
    "eval (monom b (length p)) a ∈ carrier R" "eval p a ∈ carrier R" "eval
q a ∈ carrier R" "b ∈ carrier R"
    using eval_in_carrier monom_in_carrier assms by auto
  have set_map: "set ((map ((⊗) b) q) @ (replicate (length p) 0)) ⊆ carrier
R"
    using in_carrier(4) assms(2) by (induct q) (auto)
  have set_poly: "set (poly_mult p q) ⊆ carrier R"
    using poly_mult_in_carrier[OF _ assms(2), of p] Cons(2) by auto
  have "eval (poly_mult (b # p) q) a =
      ((eval (monom b (length p)) a) ⊗ (eval q a)) ⊕ ((eval p a) ⊗ (eval
q a))"
    using eval_poly_add[OF set_map set_poly assms(3)] aux_lemma[OF in_carrier(4),
of "length p"] Cons
    by (auto simp del: poly_add.simps)
  also have " ... = ((eval (monom b (length p)) a) ⊕ (eval p a)) ⊗ (eval
q a)"
    using l_distr[OF in_carrier(1-3)] by simp
  also have " ... = (eval (b # p) a) ⊗ (eval q a)"
    unfolding eval_monom[OF in_carrier(4) assms(3), of "length p"] by
auto
  finally show ?case .
```

```
qed

proposition (in cring) eval_is_hom:
  assumes "subring K R" and "a ∈ carrier R"
  shows "(λp. (eval p) a) ∈ ring_hom (K[X]) R"
  unfolding univ_poly_def
  using polynomial_in_carrier[OF assms(1)] eval_in_carrier
        eval_poly_add eval_poly_mult assms(2)
  by (auto intro!: ring_hom_memI
         simp add: univ_poly_carrier
         simp del: poly_add.simps poly_mult.simps)

theorem (in domain) eval_cring_hom:
  assumes "subring K R" and "a ∈ carrier R"
  shows "ring_hom_cring (K[X]) R (λp. (eval p) a)"
  unfolding ring_hom_cring_def ring_hom_cring_axioms_def
  using domain.axioms(1)[OF univ_poly_is_domain[OF assms(1)]]
        eval_is_hom[OF assms] cring_axioms by auto

corollary (in domain) eval_ring_hom:
  assumes "subring K R" and "a ∈ carrier R"
  shows "ring_hom_ring (K[X]) R (λp. (eval p) a)"
  using eval_cring_hom[OF assms] ring_hom_ringI2
  unfolding ring_hom_cring_def ring_hom_cring_axioms_def cring_def by
auto
```

## 33.11 Homomorphisms

```
lemma (in ring_hom_ring) eval_hom':
  assumes "a ∈ carrier R" and "set p ⊆ carrier R"
  shows "h (R.eval p a) = eval (map h p) (h a)"
  using assms by (induct p, auto simp add: R.eval_in_carrier hom_nat_pow)

lemma (in ring_hom_ring) eval_hom:
  assumes "subring K R" and "a ∈ carrier R" and "p ∈ carrier (K[X])"
  shows "h (R.eval p a) = eval (map h p) (h a)"
proof -
  have "set p ⊆ carrier R"
    using subringE(1)[OF assms(1)] R.polynomial_incl assms(3)
    unfolding sym[OF univ_poly_carrier[of R]] by auto
  thus ?thesis
    using eval_hom'[OF assms(2)] by simp
qed

lemma (in ring_hom_ring) coeff_hom':
  assumes "set p ⊆ carrier R" shows "h (R.coeff p i) = coeff (map h
p) i"
  using assms by (induct p) (auto)
```

```
lemma (in ring_hom_ring) poly_add_hom':
  assumes "set p ⊆ carrier R" and "set q ⊆ carrier R"
  shows "normalize (map h (R.poly_add p q)) = poly_add (map h p) (map
h q)"
proof -
  have set_map: "set (map h s) ⊆ carrier S" if "set s ⊆ carrier R" for
s
    using that by auto
  have "coeff (normalize (map h (R.poly_add p q))) = coeff (map h (R.poly_add
p q))"
    using S.normalize_coeff by auto
  also have " ... = (λi. h ((R.coeff p i) ⊕ (R.coeff q i)))"
    using coeff_hom'[OF R.poly_add_in_carrier[OF assms]] R.poly_add_coeff[OF
assms] by simp
  also have " ... = (λi. (coeff (map h p) i) ⊕S (coeff (map h q) i))"
    using assms[THEN R.coeff_in_carrier] assms[THEN coeff_hom'] by simp
  also have " ... = (λi. coeff (poly_add (map h p) (map h q)) i)"
    using S.poly_add_coeff[OF assms[THEN set_map]] by simp
  finally have "coeff (normalize (map h (R.poly_add p q))) = (λi. coeff
(poly_add (map h p) (map h q)) i)" .
  thus ?thesis
    unfolding coeff_iff_polynomial_cond[OF
              normalize_gives_polynomial[OF set_map[OF R.poly_add_in_carrier[OF
assms]]]
              poly_add_is_polynomial[OF carrier_is_subring assms[THEN
set_map]]] .
qed

lemma (in ring_hom_ring) poly_mult_hom':
  assumes "set p ⊆ carrier R" and "set q ⊆ carrier R"
  shows "normalize (map h (R.poly_mult p q)) = poly_mult (map h p) (map
h q)"
  using assms(1)
proof (induct p, simp)
  case (Cons a p)
  have set_map: "set (map h s) ⊆ carrier S" if "set s ⊆ carrier R" for
s
    using that by auto

  let ?q_a = "(map ((⊗) a) q) @ (replicate (length p) 0)"
  have set_q_a: "set ?q_a ⊆ carrier R"
    using assms(2) Cons(2) by (induct q) (auto)
  have q_a_simp: "map h ?q_a = (map ((⊗S) (h a)) (map h q)) @ (replicate
(length (map h p)) 0S)"
    using assms(2) Cons(2) by (induct q) (auto)

  have "S.normalize (map h (R.poly_mult (a # p) q)) =
       S.normalize (map h (R.poly_add ?q_a (R.poly_mult p q)))"
    by simp
```

```
  also have " ... = S.poly_add (map h ?q_a) (map h (R.poly_mult p q))"
    using poly_add_hom'[OF set_q_a R.poly_mult_in_carrier[OF _ assms(2)]]
Cons by simp
  also have " ... = S.poly_add (map h ?q_a) (S.normalize (map h (R.poly_mult
p q)))"
    using poly_add_normalize(2)[OF set_map[OF set_q_a] set_map[OF R.poly_mult_in_carrier[OF
_ assms(2)]]] Cons by simp
  also have " ... = S.poly_add (map h ?q_a) (S.poly_mult (map h p) (map
h q))"
    using Cons by simp
  also have " ... = S.poly_mult (map h (a # p)) (map h q)"
    unfolding q_a_simp by simp
  finally show ?case .
qed
```

## 33.12 The X Variable

```
definition var :: "_ ⇒ 'a list" (‹X𝔦›)
  where "X_R = [ 1_R, 0_R ]"

lemma (in ring) eval_var:
  assumes "x ∈ carrier R" shows "eval X x = x"
  using assms unfolding var_def by auto

lemma (in domain) var_closed:
  assumes "subring K R" shows "X ∈ carrier (K[X])" and "polynomial K
X"
  using subringE(2-3)[OF assms]
  by (auto simp add: var_def univ_poly_def polynomial_def)

lemma (in domain) poly_mult_var':
  assumes "set p ⊆ carrier R"
  shows "poly_mult X p = normalize (p @ [ 0 ])"
    and "poly_mult p X = normalize (p @ [ 0 ])"
proof -
  from ‹set p ⊆ carrier R› have "poly_mult [ 1 ] p = normalize p"
    using poly_mult_one' by simp
  thus "poly_mult X p = normalize (p @ [ 0 ])"
    using poly_mult_append_zero[OF _ assms, of "[ 1 ]"] normalize_idem
    unfolding var_def by (auto simp del: poly_mult.simps)
  thus "poly_mult p X = normalize (p @ [ 0 ])"
    using poly_mult_comm[OF assms] unfolding var_def by simp
qed

lemma (in domain) poly_mult_var:
  assumes "subring K R" "p ∈ carrier (K[X])"
  shows "p ⊗_K[X] X = (if p = [] then [] else p @ [ 0 ])"
proof -
  have is_poly: "polynomial K p"
```

```
    using assms(2) unfolding univ_poly_def by simp
  hence "polynomial K (p @ [ 0 ])" if "p ≠ []"
    using that subringE(2)[OF assms(1)] unfolding polynomial_def by auto
  thus ?thesis
    using poly_mult_var'(2)[OF polynomial_in_carrier[OF assms(1) is_poly]]
          normalize_polynomial[of K "p @ [ 0 ]"]
    by (auto simp add: univ_poly_mult[of R K])
qed

lemma (in domain) var_pow_closed:
  assumes "subring K R" shows "X [^]K[X] (n :: nat) ∈ carrier (K[X])"
  using monoid.nat_pow_closed[OF univ_poly_is_monoid[OF assms] var_closed(1)[OF
assms]] .

lemma (in domain) unitary_monom_eq_var_pow:
  assumes "subring K R" shows "monom 1 n = X [^]K[X] n"
  using poly_mult_var[OF assms var_pow_closed[OF assms]] unfolding nat_pow_def
monom_def
  by (induct n) (auto simp add: univ_poly_one, metis append_Cons replicate_append_same)

lemma (in domain) monom_eq_var_pow:
  assumes "subring K R" "a ∈ carrier R - { 0 }"
  shows "monom a n = [ a ] ⊗K[X] (X [^]K[X] n)"
proof -
  have "monom a n = map ((⊗) a) (monom 1 n)"
    unfolding monom_def using assms(2) by (induct n) (auto)
  also have " ... = poly_mult [ a ] (monom 1 n)"
    using poly_mult_const(1)[OF _ monom_is_polynomial assms(2)] carrier_is_subring
by simp
  also have " ... = [ a ] ⊗K[X] (X [^]K[X] n)"
    unfolding unitary_monom_eq_var_pow[OF assms(1)] univ_poly_mult[of
R K] by simp
  finally show ?thesis .
qed

lemma (in domain) eval_rewrite:
  assumes "subring K R" and "p ∈ carrier (K[X])"
  shows "p = (ring.eval (K[X])) (map poly_of_const p) X"
proof -
  let ?map_norm = "λp. map poly_of_const p"

  interpret UP: domain "K[X]"
    using univ_poly_is_domain[OF assms(1)] .

  have aux_lemma1: "set (?map_norm l) ⊆ carrier (K[X])" if "set l ⊆ K"
for l
  proof -
    from that have "poly_of_const a ∈ carrier (K[X])" if "a ∈ set l"
for a
```

```
      using that normalize_gives_polynomial[of "[ a ]" K]
      unfolding univ_poly_carrier poly_of_const_def by auto
    then show ?thesis by auto
  qed

  have aux_lemma2: "UP.eval (?map_norm l) q = UP.eval (?map_norm ((replicate
n 0) @ l)) q"
    if set_l: "set l ⊆ K" and q: "q ∈ carrier (K[X])"
    for n q l
    using set_l
  proof (induct n)
    case 0
    then show ?case by simp
  next
    case (Suc n)
    from ‹set l ⊆ K› have set_replicate: "set ((replicate n 0) @ l)
⊆ K"
      using subringE(2)[OF assms(1)] by (induct n) (auto)
    have step: "UP.eval (?map_norm l') q = UP.eval (?map_norm (0 # l'))
q" if "set l' ⊆ K" for l'
      using UP.eval_in_carrier[OF aux_lemma1[OF that]] q unfolding poly_of_const_def
      by (simp, simp add: sym[OF univ_poly_zero[of R K]])
    have "UP.eval (?map_norm l) q = UP.eval (?map_norm ((replicate n 0)
@ l)) q"
      using Suc by simp
    also have " ... = UP.eval (map poly_of_const ((replicate (Suc n) 0)
@ l)) q"
      using step[OF set_replicate] by simp
    finally show ?case .
  qed

  have aux_lemma3: "UP.eval (?map_norm l) q = UP.eval (?map_norm (normalize
l)) q"
    if "set l ⊆ K" and q: "q ∈ carrier (K[X])" for q l
  proof -
    from ‹set l ⊆ K› have set_norm: "set (normalize l) ⊆ K"
      by (induct l) (auto)
    show ?thesis
      using aux_lemma2[OF set_norm q, of "length l - length (local.normalize
l)"]
      unfolding sym[OF normalize_trick[of l]] ..
  qed

  from ‹p ∈ carrier (K[X])› show ?thesis
  proof (induct "length p" arbitrary: p rule: less_induct)
    case less thus ?case
    proof (cases p)
      case Nil
      then show ?thesis by (simp add: univ_poly_zero)
```

```
    next
      case (Cons a l)
      hence a: "a ∈ carrier R - { 0 }" and set_l: "set l ⊆ carrier R"
"set l ⊆ K"
        using less(2) subringE(1)[OF assms(1)] unfolding sym[OF univ_poly_carrier]
polynomial_def by auto

      have "a # l = poly_add (monom a (length l)) l"
        using poly_add_monom[OF set_l(1) a] ..
      also have " ... = poly_add (monom a (length l)) (normalize l)"
        using poly_add_normalize(2)[OF monom_in_carrier[of a] set_l(1)]
a by simp
      also have " ... = poly_add (monom a (length l)) (UP.eval (?map_norm
(normalize l)) X)"
        using less(1)[of "normalize l"] normalize_gives_polynomial[OF
set_l(2)] normalize_length_le[of l]
        by (auto simp add: univ_poly_carrier Cons(1))
      also have " ... = poly_add ([ a ] ⊗K[X] (X [^]K[X] (length l)))
(UP.eval (?map_norm l) X)"
        unfolding monom_eq_var_pow[OF assms(1) a] aux_lemma3[OF set_l(2)
var_closed(1)[OF assms(1)]] ..
      also have " ... = UP.eval (?map_norm (a # l)) X"
        using a unfolding sym[OF univ_poly_add[of R K]] unfolding poly_of_const_def
by auto
      finally show ?thesis
        unfolding Cons(1) .
    qed
  qed
qed

lemma (in ring) dense_repr_set_fst:
  assumes "set p ⊆ K" shows "fst ‘ (set (dense_repr p)) ⊆ K - { 0 }"
  using assms by (induct p) (auto)

lemma (in ring) dense_repr_set_snd:
  shows "snd ‘ (set (dense_repr p)) ⊆ {..< length p}"
  by (induct p) (auto)

lemma (in domain) dense_repr_monom_closed:
  assumes "subring K R" "set p ⊆ K"
  shows "t ∈ set (dense_repr p) ⟹ monom (fst t) (snd t) ∈ carrier (K[X])"
  using dense_repr_set_fst[OF assms(2)] monom_is_polynomial[OF assms(1)]
  by (auto simp add: univ_poly_carrier)

lemma (in domain) monom_finsum_decomp:
  assumes "subring K R" "p ∈ carrier (K[X])"
  shows "p = (⨁K[X] t ∈ set (dense_repr p). monom (fst t) (snd t))"
proof -
  interpret UP: domain "K[X]"
```

```
    using univ_poly_is_domain[OF assms(1)] .

  from ‹p ∈ carrier (K[X])› show ?thesis
  proof (induct "length p" arbitrary: p rule: less_induct)
    case less thus ?case
    proof (cases p)
      case Nil thus ?thesis
        using UP.finsum_empty univ_poly_zero[of R K] by simp
    next
      case (Cons a l)
      hence in_carrier:
        "normalize l ∈ carrier (K[X])" "polynomial K (normalize l)" "polynomial
K (a # l)"
        using normalize_gives_polynomial polynomial_incl[of K p] less(2)
        unfolding univ_poly_carrier by auto
      have len_lt: "length (local.normalize l) < length p"
        using normalize_length_le by (simp add: Cons le_imp_less_Suc)


      have a: "a ∈ K - { 𝟬 }"
        using less(2) subringE(1)[OF assms(1)] unfolding Cons univ_poly_def
polynomial_def by auto
      hence "p = (monom a (length l)) ⊕K[X] (poly_of_dense (dense_repr
(normalize l)))"
        using monom_decomp[OF assms(1), of p] less(2) dense_repr_normalize
        unfolding univ_poly_add univ_poly_carrier Cons by (auto simp del:
poly_add.simps)
      also have " ... = (monom a (length l)) ⊕K[X] (normalize l)"
        using monom_decomp[OF assms(1) in_carrier(2)] by simp
      finally have "p = monom a (length l) ⊕K[X]
                     (⨁K[X] t ∈ set (dense_repr l). monom (fst t) (snd
t))"
        using less(1)[OF len_lt in_carrier(1)] dense_repr_normalize by
simp

      moreover have "(a, (length l)) ∉ set (dense_repr l)"
        using dense_repr_set_snd[of l] by auto
      moreover have "monom a (length l) ∈ carrier (K[X])"
        using monom_is_polynomial[OF assms(1) a] unfolding univ_poly_carrier
by simp
      moreover have "⋀t. t ∈ set (dense_repr l) ⟹ monom (fst t) (snd
t) ∈ carrier (K[X])"
        using dense_repr_monom_closed[OF assms(1)] polynomial_incl[OF
in_carrier(3)] by auto
      ultimately have "p = (⨁K[X] t ∈ set (dense_repr (a # l)). monom
(fst t) (snd t))"
        using UP.add.finprod_insert a by auto
      thus ?thesis unfolding Cons .
    qed
```

```
  qed
qed

lemma (in domain) var_pow_finsum_decomp:
  assumes "subring K R" "p ∈ carrier (K[X])"
  shows "p = (⨁K[X] t ∈ set (dense_repr p). [ fst t ] ⊗K[X] (X [^]K[X]
(snd t)))"
proof -
  let ?f = "λt. monom (fst t) (snd t)"
  let ?g = "λt. [ fst t ] ⊗K[X] (X [^]K[X] (snd t))"

  interpret UP: domain "K[X]"
    using univ_poly_is_domain[OF assms(1)] .

  have set_p: "set p ⊆ K"
    using polynomial_incl assms(2) by (simp add: univ_poly_carrier)
  hence f: "?f ∈ set (dense_repr p) → carrier (K[X])"
    using dense_repr_monom_closed[OF assms(1)] by auto

  moreover
  have "⋀t. t ∈ set (dense_repr p) ⟹ fst t ∈ carrier R - { 𝟬 }"
    using dense_repr_set_fst[OF set_p] subringE(1)[OF assms(1)] by auto
  hence "⋀t. t ∈ set (dense_repr p) ⟹ monom (fst t) (snd t) = [ fst
t ] ⊗K[X] (X [^]K[X] (snd t))"
    using monom_eq_var_pow[OF assms(1)] by auto

  ultimately show ?thesis
    using UP.add.finprod_cong[of _ _ ?f ?g] monom_finsum_decomp[OF assms]
by auto
qed

corollary (in domain) hom_var_pow_finsum:
  assumes "subring K R" and "p ∈ carrier (K[X])" "ring_hom_ring (K[X])
A h"
  shows "h p = (⨁A t ∈ set (dense_repr p). h [ fst t ] ⊗A (h X [^]A
(snd t)))"
proof -
  let ?f = "λt. [ fst t ] ⊗K[X] (X [^]K[X] (snd t))"
  let ?g = "λt. h [ fst t ] ⊗A (h X [^]A (snd t))"

  interpret UP: domain "K[X]" + A: ring A
    using univ_poly_is_domain[OF assms(1)] ring_hom_ring.axioms(2)[OF
assms(3)] by simp+

  have const_in_carrier:
    "⋀t. t ∈ set (dense_repr p) ⟹ [ fst t ] ∈ carrier (K[X])"
    using dense_repr_set_fst[OF polynomial_incl, of K p] assms(2) const_is_polynomial[of
_ K]
    by (auto simp add: univ_poly_carrier)
```

```
  hence f: "?f: set (dense_repr p) → carrier (K[X])"
    using UP.m_closed[OF _ var_pow_closed[OF assms(1)]] by auto
  hence h: "h ∘ ?f: set (dense_repr p) → carrier A"
    using ring_hom_memE(1)[OF ring_hom_ring.homh[OF assms(3)]] by (auto
simp add: Pi_def)

  have hp: "h p = (⨁A t ∈ set (dense_repr p). (h ∘ ?f) t)"
    using ring_hom_ring.hom_finsum[OF assms(3) f] var_pow_finsum_decomp[OF
assms(1-2)]
    by (auto, meson o_apply)
  have eq: "⋀t. t ∈ set (dense_repr p) ⟹ h [ fst t ] ⊗A (h X [^]A
(snd t)) = (h ∘ ?f) t"
    using ring_hom_memE(2)[OF ring_hom_ring.homh[OF assms(3)]
          const_in_carrier var_pow_closed[OF assms(1)]]
          ring_hom_ring.hom_nat_pow[OF assms(3) var_closed(1)[OF assms(1)]]
by auto
  show ?thesis
    using A.add.finprod_cong'[OF _ h eq] hp by simp
qed

corollary (in domain) determination_of_hom:
  assumes "subring K R"
    and "ring_hom_ring (K[X]) A h" "ring_hom_ring (K[X]) A g"
    and "⋀k. k ∈ K ⟹ h [ k ] = g [ k ]" and "h X = g X"
  shows "⋀p. p ∈ carrier (K[X]) ⟹ h p = g p"
proof -
  interpret A: ring A
    using ring_hom_ring.axioms(2)[OF assms(2)] by simp

  fix p assume p: "p ∈ carrier (K[X])"
  hence
    "⋀t. t ∈ set (dense_repr p) ⟹ [ fst t ] ∈ carrier (K[X])"
    using dense_repr_set_fst[OF polynomial_incl, of K p] const_is_polynomial[of
_ K]
    by (auto simp add: univ_poly_carrier)
  hence f: "(λt. h [ fst t ] ⊗A (h X [^]A (snd t))): set (dense_repr
p) → carrier A"
    using ring_hom_memE(1)[OF ring_hom_ring.homh[OF assms(2)]] var_closed(1)[OF
assms(1)]
          A.m_closed[OF _ A.nat_pow_closed]
    by auto

  have eq: "⋀t. t ∈ set (dense_repr p) ⟹
    g [ fst t ] ⊗A (g X [^]A (snd t)) = h [ fst t ] ⊗A (h X [^]A (snd
t))"
    using dense_repr_set_fst[OF polynomial_incl, of K p] p assms(4-5)
    by (auto simp add: univ_poly_carrier)
  show "h p = g p"
    unfolding assms(2-3)[THEN hom_var_pow_finsum[OF assms(1) p]]
```

```
    using A.add.finprod_cong'[OF _ f eq] by simp
qed

corollary (in domain) eval_as_unique_hom:
  assumes "subring K R" "x ∈ carrier R"
    and "ring_hom_ring (K[X]) R h"
    and "⋀k. k ∈ K ⟹ h [ k ] = k" and "h X = x"
  shows "⋀p. p ∈ carrier (K[X]) ⟹ h p = eval p x"
  using determination_of_hom[OF assms(1,3) eval_ring_hom[OF assms(1-2)]]
        eval_var[OF assms(2)] assms(4-5) subringE(1)[OF assms(1)]
  by fastforce
```

## 33.13 The Constant Term

```
definition (in ring) const_term :: "'a list ⇒ 'a"
  where "const_term p = eval p 0"

lemma (in ring) const_term_eq_last:
  assumes "set p ⊆ carrier R" and "a ∈ carrier R"
  shows "const_term (p @ [ a ]) = a"
  using assms by (induct p) (auto simp add: const_term_def)

lemma (in ring) const_term_not_zero:
  assumes "const_term p ≠ 0" shows "p ≠ []"
  using assms by (auto simp add: const_term_def)

lemma (in ring) const_term_explicit:
  assumes "set p ⊆ carrier R" "p ≠ []" and "const_term p = a"
  obtains p' where "set p' ⊆ carrier R" and "p = p' @ [ a ]"
proof -
  obtain a' p' where p: "p = p' @ [ a' ]"
    using assms(2) rev_exhaust by blast
  have p': "set p' ⊆ carrier R" and a: "a = a'"
    using assms const_term_eq_last[of p' a'] unfolding p by auto
  show thesis
    using p p' that unfolding a by blast
qed

lemma (in ring) const_term_zero:
  assumes "subring K R" "polynomial K p" "p ≠ []" and "const_term p
= 0"
  obtains p' where "polynomial K p'" "p' ≠ []" and "p = p' @ [ 0 ]"
proof -
  obtain p' where p': "p = p' @ [ 0 ]"
    using const_term_explicit[OF polynomial_in_carrier[OF assms(1-2)]
assms(3-4)] by auto
  have "polynomial K p'" "p' ≠ []"
    using assms(2) unfolding p' polynomial_def by auto
  thus thesis using p' ..
```

```
qed

lemma (in cring) const_term_simprules:
  shows "⋀p. set p ⊆ carrier R ⟹ const_term p ∈ carrier R"
    and "⋀p q. ⟦ set p ⊆ carrier R; set q ⊆ carrier R ⟧ ⟹
                 const_term (poly_mult p q) = const_term p ⊗ const_term
q"
    and "⋀p q. ⟦ set p ⊆ carrier R; set q ⊆ carrier R ⟧ ⟹
                 const_term (poly_add  p q) = const_term p ⊕ const_term
q"
  using eval_poly_mult eval_poly_add eval_in_carrier zero_closed
  unfolding const_term_def by auto

lemma (in domain) const_term_simprules_shell:
  assumes "subring K R"
  shows "⋀p. p ∈ carrier (K[X]) ⟹ const_term p ∈ K"
    and "⋀p q. ⟦ p ∈ carrier (K[X]); q ∈ carrier (K[X]) ⟧ ⟹
                 const_term (p ⊗K[X] q) = const_term p ⊗ const_term q"
    and "⋀p q. ⟦ p ∈ carrier (K[X]); q ∈ carrier (K[X]) ⟧ ⟹
                 const_term (p ⊕K[X] q) = const_term p ⊕ const_term q"
    and "⋀p. p ∈ carrier (K[X]) ⟹ const_term (⊖K[X] p) = ⊖ (const_term
p)"
  using eval_is_hom[OF assms(1) zero_closed]
  unfolding ring_hom_def const_term_def
proof (auto)
  fix p assume p: "p ∈ carrier (K[X])"
  hence "set p ⊆ carrier R"
    using polynomial_in_carrier[OF assms(1)] by (auto simp add: univ_poly_def)
  thus "eval (⊖K [X] p) 𝟬 = ⊖ local.eval p 𝟬"
    unfolding univ_poly_a_inv_def'[OF assms(1) p]
    by (induct p) (auto simp add: eval_in_carrier l_minus local.minus_add)

  have "set p ⊆ K"
    using p by (auto simp add: univ_poly_def polynomial_def)
  thus "eval p 𝟬 ∈ K"
    using subringE(1-2,6-7)[OF assms]
    by (induct p) (auto, metis assms nat_pow_0 nat_pow_zero subringE(3))
qed
```

## 33.14 The Canonical Embedding of K in K[X]

```
lemma (in ring) poly_of_const_consistent:
  assumes "subring K R" shows "ring.poly_of_const (R ⦇ carrier := K ⦈)
= poly_of_const"
  unfolding ring.poly_of_const_def[OF subring_is_ring[OF assms]]
            normalize_consistent[OF assms] poly_of_const_def ..

lemma (in domain) canonical_embedding_is_hom:
  assumes "subring K R" shows "poly_of_const ∈ ring_hom (R ⦇ carrier
```

```
:= K ⦈) (K[X])"
  using subringE(1)[OF assms] unfolding subset_iff poly_of_const_def
  by (auto intro!: ring_hom_memI simp add: univ_poly_def)

lemma (in domain) canonical_embedding_ring_hom:
  assumes "subring K R" shows "ring_hom_ring (R ⦇ carrier := K ⦈) (K[X])
poly_of_const"
  using canonical_embedding_is_hom[OF assms] unfolding symmetric[OF ring_hom_ring_axioms_de
  by (rule ring_hom_ring.intro[OF subring_is_ring[OF assms] univ_poly_is_ring[OF
assms]])

lemma (in field) poly_of_const_over_carrier:
  shows "poly_of_const ‘ (carrier R) = { p ∈ carrier ((carrier R)[X]).
degree p = 0 }"
proof -
  have "poly_of_const ‘ (carrier R) = insert [] { [ k ] | k. k ∈ carrier
R - { 0 } }"
    unfolding poly_of_const_def by auto
  also have " ... = { p ∈ carrier ((carrier R)[X]). degree p = 0 }"
    unfolding univ_poly_def polynomial_def
    by (auto, metis le_Suc_eq le_zero_eq length_0_conv length_Suc_conv
list.sel(1) list.set_sel(1) subsetCE)
  finally show ?thesis .
qed

lemma (in ring) poly_of_const_over_subfield:
  assumes "subfield K R" shows "poly_of_const ‘ K = { p ∈ carrier (K[X]).
degree p = 0 }"
  using field.poly_of_const_over_carrier[OF subfield_iff(2)[OF assms]]
        poly_of_const_consistent[OF subfieldE(1)[OF assms]]
        univ_poly_consistent[OF subfieldE(1)[OF assms]] by simp

lemma (in field) univ_poly_carrier_subfield_of_consts:
  "subfield (poly_of_const ‘ (carrier R)) ((carrier R)[X])"
proof -
  have ring_hom: "ring_hom_ring R ((carrier R)[X]) poly_of_const"
    using canonical_embedding_ring_hom[OF carrier_is_subring] by simp
  thus ?thesis
    using ring_hom_ring.img_is_subfield(2)[OF ring_hom carrier_is_subfield]
    unfolding univ_poly_def by auto
qed

proposition (in ring) univ_poly_subfield_of_consts:
  assumes "subfield K R" shows "subfield (poly_of_const ‘ K) (K[X])"
  using field.univ_poly_carrier_subfield_of_consts[OF subfield_iff(2)[OF
assms]]
  unfolding poly_of_const_consistent[OF subfieldE(1)[OF assms]]
            univ_poly_consistent[OF subfieldE(1)[OF assms]] by simp
```

```
end
```

```
theory Embedded_Algebras
  imports Subrings Generated_Groups
begin
```

# 34 Definitions

```
locale embedded_algebra =
  K?: subfield K R + R?: ring R for K :: "'a set" and R :: "('a, 'b) ring_scheme"
(structure)

definition (in ring) line_extension :: "'a set ⇒ 'a ⇒ 'a set ⇒ 'a set"
  where "line_extension K a E = (K #> a) <+>R E"

fun (in ring) Span :: "'a set ⇒ 'a list ⇒ 'a set"
  where "Span K Us = foldr (line_extension K) Us { 0 }"

fun (in ring) combine :: "'a list ⇒ 'a list ⇒ 'a"
  where
    "combine (k # Ks) (u # Us) = (k ⊗ u) ⊕ (combine Ks Us)"
  | "combine Ks Us = 0"

inductive (in ring) independent :: "'a set ⇒ 'a list ⇒ bool"
  where
    li_Nil [simp, intro]: "independent K []"
  | li_Cons: "⟦ u ∈ carrier R; u ∉ Span K Us; independent K Us ⟧ ⟹ independent
K (u # Us)"

inductive (in ring) dimension :: "nat ⇒ 'a set ⇒ 'a set ⇒ bool"
  where
    zero_dim [simp, intro]: "dimension 0 K { 0 }"
   | Suc_dim: "⟦ v ∈ carrier R; v ∉ E; dimension n K E ⟧ ⟹ dimension
(Suc n) K (line_extension K v E)"
```

### 34.0.1 Syntactic Definitions

```
abbreviation (in ring) dependent ::  "'a set ⇒ 'a list ⇒ bool"
  where "dependent K U ≡ ¬ independent K U"

definition over :: "('a ⇒ 'b) ⇒ 'a ⇒ 'b" (infixl ‹over› 65)
  where "f over a = f a"
```

```
context ring
begin
```

## 34.1 Basic Properties - First Part

```
lemma line_extension_consistent:
  assumes "subring K R" shows "ring.line_extension (R ⦇ carrier := K
⦈) = line_extension"
  unfolding ring.line_extension_def[OF subring_is_ring[OF assms]] line_extension_def
  by (simp add: set_add_def set_mult_def)

lemma Span_consistent:
  assumes "subring K R" shows "ring.Span (R ⦇ carrier := K ⦈) = Span"
  unfolding ring.Span.simps[OF subring_is_ring[OF assms]] Span.simps
            line_extension_consistent[OF assms] by simp

lemma combine_in_carrier [simp, intro]:
  "⟦ set Ks ⊆ carrier R; set Us ⊆ carrier R ⟧ ⟹ combine Ks Us ∈ carrier
R"
  by (induct Ks Us rule: combine.induct) (auto)

lemma combine_r_distr:
  "⟦ set Ks ⊆ carrier R; set Us ⊆ carrier R ⟧ ⟹
     k ∈ carrier R ⟹ k ⊗ (combine Ks Us) = combine (map ((⊗) k) Ks)
Us"
  by (induct Ks Us rule: combine.induct) (auto simp add: m_assoc r_distr)

lemma combine_l_distr:
  "⟦ set Ks ⊆ carrier R; set Us ⊆ carrier R ⟧ ⟹
     u ∈ carrier R ⟹ (combine Ks Us) ⊗ u = combine Ks (map (λu'. u'
⊗ u) Us)"
  by (induct Ks Us rule: combine.induct) (auto simp add: m_assoc l_distr)

lemma combine_eq_foldr:
  "combine Ks Us = foldr (λ(k, u). λl. (k ⊗ u) ⊕ l) (zip Ks Us) 𝟬"
  by (induct Ks Us rule: combine.induct) (auto)

lemma combine_replicate:
  "set Us ⊆ carrier R ⟹ combine (replicate (length Us) 𝟬) Us = 𝟬"
  by (induct Us) (auto)

lemma combine_take:
  "combine (take (length Us) Ks) Us = combine Ks Us"
  by (induct Us arbitrary: Ks)
     (auto, metis combine.simps(1) list.exhaust take.simps(1) take_Suc_Cons)

lemma combine_append_zero:
  "set Us ⊆ carrier R ⟹ combine (Ks @ [ 𝟬 ]) Us = combine Ks Us"
proof (induct Ks arbitrary: Us)
  case Nil thus ?case by (induct Us) (auto)
next
  case Cons thus ?case by (cases Us) (auto)
qed
```

```
lemma combine_prepend_replicate:
  "⟦ set Ks ⊆ carrier R; set Us ⊆ carrier R ⟧ ⟹
     combine ((replicate n 𝟬) @ Ks) Us = combine Ks (drop n Us)"
proof (induct n arbitrary: Us, simp)
  case (Suc n) thus ?case
    by (cases Us) (auto, meson combine_in_carrier ring_simprules(8) set_drop_subset
subset_trans)
qed

lemma combine_append_replicate:
  "set Us ⊆ carrier R ⟹ combine (Ks @ (replicate n 𝟬)) Us = combine
Ks Us"
  by (induct n) (auto, metis append.assoc combine_append_zero replicate_append_same)

lemma combine_append:
  assumes "length Ks = length Us"
    and "set Ks  ⊆ carrier R" "set Us ⊆ carrier R"
    and "set Ks' ⊆ carrier R" "set Vs ⊆ carrier R"
  shows "(combine Ks Us) ⊕ (combine Ks' Vs) = combine (Ks @ Ks') (Us
@ Vs)"
  using assms
proof (induct Ks arbitrary: Us)
  case Nil thus ?case by auto
next
  case (Cons k Ks)
  then obtain u Us' where Us: "Us = u # Us'"
    by (metis length_Suc_conv)
  hence u: "u ∈ carrier R" and Us': "set Us' ⊆ carrier R"
    using Cons(4) by auto
  then show ?case
    using combine_in_carrier[OF _ Us', of Ks] Cons
          combine_in_carrier[OF Cons(5-6)] unfolding Us
    by (auto, simp add: add.m_assoc)
qed

lemma combine_add:
  assumes "length Ks = length Us" and "length Ks' = length Us"
    and "set Ks  ⊆ carrier R" "set Ks'  ⊆ carrier R" "set Us ⊆ carrier
R"
  shows "(combine Ks Us) ⊕ (combine Ks' Us) = combine (map2 (⊕) Ks Ks')
Us"
  using assms
proof (induct Us arbitrary: Ks Ks')
  case Nil thus ?case by simp
next
  case (Cons u Us)
  then obtain c c' Cs Cs' where Ks: "Ks = c # Cs" and Ks': "Ks' = c'
# Cs'"
```

```
    by (metis length_Suc_conv)
  hence in_carrier:
    "c  ∈ carrier R" "set Cs  ⊆ carrier R"
    "c' ∈ carrier R" "set Cs' ⊆ carrier R"
    "u  ∈ carrier R" "set Us  ⊆ carrier R"
    using Cons(4-6) by auto
  hence lc_in_carrier: "combine Cs Us ∈ carrier R" "combine Cs' Us ∈
carrier R"
    using combine_in_carrier by auto
  have "combine Ks (u # Us) ⊕ combine Ks' (u # Us) =
        ((c ⊗ u) ⊕ combine Cs Us) ⊕ ((c' ⊗ u) ⊕ combine Cs' Us)"
    unfolding Ks Ks' by auto
  also have " ... = ((c ⊕ c') ⊗ u ⊕ (combine Cs Us ⊕ combine Cs' Us))"
    using lc_in_carrier in_carrier(1,3,5) by (simp add: l_distr ring_simprules(7,22))
  also have " ... = combine (map2 (⊕) Ks Ks') (u # Us)"
    using Cons unfolding Ks Ks' by auto
  finally show ?case .
qed

lemma combine_normalize:
  assumes "set Ks ⊆ carrier R" "set Us ⊆ carrier R" "combine Ks Us =
a"
  obtains Ks'
  where "set (take (length Us) Ks) ⊆ set Ks'" "set Ks' ⊆ set (take (length
Us) Ks) ∪ { 0 }"
    and "length Ks' = length Us" "combine Ks' Us = a"
proof -
  define Ks'
    where "Ks' = (if length Ks ≤ length Us
                  then Ks @ (replicate (length Us - length Ks) 0) else
take (length Us) Ks)"
  hence "set (take (length Us) Ks) ⊆ set Ks'" "set Ks' ⊆ set (take (length
Us) Ks) ∪ { 0 }"
        "length Ks' = length Us" "a = combine Ks' Us"
    using combine_append_replicate[OF assms(2)] combine_take assms(3)
by auto
  thus thesis
    using that by blast
qed

lemma line_extension_mem_iff: "u ∈ line_extension K a E ⟷ (∃k ∈ K.
∃v ∈ E. u = k ⊗ a ⊕ v)"
  unfolding line_extension_def set_add_def'[of R "K #> a" E] unfolding
r_coset_def by blast

lemma line_extension_in_carrier:
  assumes "K ⊆ carrier R" "a ∈ carrier R" "E ⊆ carrier R"
  shows "line_extension K a E ⊆ carrier R"
  using set_add_closed[OF r_coset_subset_G[OF assms(1-2)] assms(3)]
```

```
  by (simp add: line_extension_def)

lemma Span_in_carrier:
  assumes "K ⊆ carrier R" "set Us ⊆ carrier R"
  shows "Span K Us ⊆ carrier R"
  using assms by (induct Us) (auto simp add: line_extension_in_carrier)
```

## 34.2 Some Basic Properties of Linear Independence

```
lemma independent_in_carrier: "independent K Us ⟹ set Us ⊆ carrier
R"
  by (induct Us rule: independent.induct) (simp_all)

lemma independent_backwards:
  "independent K (u # Us) ⟹ u ∉ Span K Us"
  "independent K (u # Us) ⟹ independent K Us"
  "independent K (u # Us) ⟹ u ∈ carrier R"
  by (cases rule: independent.cases, auto)+

lemma dimension_independent [intro]: "independent K Us ⟹ dimension
(length Us) K (Span K Us)"
proof (induct Us)
  case Nil thus ?case by simp
next
  case Cons thus ?case
    using Suc_dim independent_backwards[OF Cons(2)] by auto
qed
```

Now, we fix K, a subfield of the ring. Many lemmas would also be true for weaker structures, but our interest is to work with subfields, so generalization could be the subject of a future work.

```
context
  fixes K :: "'a set" assumes K: "subfield K R"
begin
```

## 34.3 Basic Properties - Second Part

```
lemmas subring_props [simp] =
  subringE[OF subfieldE(1)[OF K]]

lemma line_extension_is_subgroup:
  assumes "subgroup E (add_monoid R)" "a ∈ carrier R"
  shows "subgroup (line_extension K a E) (add_monoid R)"
proof (rule add.subgroupI)
  show "line_extension K a E ⊆ carrier R"
    by (simp add: assms add.subgroupE(1) line_extension_def r_coset_subset_G
set_add_closed)
next
  have "0 = 0 ⊗ a ⊕ 0"
```

```
    using assms(2) by simp
  hence "0 ∈ line_extension K a E"
    using line_extension_mem_iff subgroup.one_closed[OF assms(1)] by auto
  thus "line_extension K a E ≠ {}" by auto
next
  fix u1 u2
  assume "u1 ∈ line_extension K a E" and "u2 ∈ line_extension K a E"
  then obtain k1 k2 v1 v2
    where u1: "k1 ∈ K" "v1 ∈ E" "u1 = (k1 ⊗ a) ⊕ v1"
      and u2: "k2 ∈ K" "v2 ∈ E" "u2 = (k2 ⊗ a) ⊕ v2"
      and in_carr: "k1 ∈ carrier R" "v1 ∈ carrier R" "k2 ∈ carrier R"
"v2 ∈ carrier R"
    using line_extension_mem_iff by (meson add.subgroupE(1)[OF assms(1)]
subring_props(1) subsetCE)

  hence "u1 ⊕ u2 = ((k1 ⊕ k2) ⊗ a) ⊕ (v1 ⊕ v2)"
    using assms(2) by algebra
  moreover have "k1 ⊕ k2 ∈ K" and "v1 ⊕ v2 ∈ E"
    using add.subgroupE(4)[OF assms(1)] u1 u2 by auto
  ultimately show "u1 ⊕ u2 ∈ line_extension K a E"
    using line_extension_mem_iff by auto

  have "⊖ u1 = ((⊖ k1) ⊗ a) ⊕ (⊖ v1)"
    using in_carr(1-2) u1(3) assms(2) by algebra
  moreover have "⊖ k1 ∈ K" and "⊖ v1 ∈ E"
    using add.subgroupE(3)[OF assms(1)] u1 by auto
  ultimately show "(⊖ u1) ∈ line_extension K a E"
    using line_extension_mem_iff by auto
qed

corollary Span_is_add_subgroup:
  "set Us ⊆ carrier R ⟹ subgroup (Span K Us) (add_monoid R)"
  using line_extension_is_subgroup normal_imp_subgroup[OF add.one_is_normal]
by (induct Us) (auto)

lemma line_extension_smult_closed:
  assumes "⋀k v. ⟦ k ∈ K; v ∈ E ⟧ ⟹ k ⊗ v ∈ E" and "E ⊆ carrier R"
"a ∈ carrier R"
  shows "⋀k u. ⟦ k ∈ K; u ∈ line_extension K a E ⟧ ⟹ k ⊗ u ∈ line_extension
K a E"
proof -
  fix k u assume A: "k ∈ K" "u ∈ line_extension K a E"
  then obtain k' v'
    where u: "k' ∈ K" "v' ∈ E" "u = k' ⊗ a ⊕ v'"
      and in_carr: "k ∈ carrier R" "k' ∈ carrier R" "v' ∈ carrier R"
    using line_extension_mem_iff assms(2) by (meson subring_props(1) subsetCE)
  hence "k ⊗ u = (k ⊗ k') ⊗ a ⊕ (k ⊗ v')"
    using assms(3) by algebra
  thus "k ⊗ u ∈ line_extension K a E"
```

```
    using assms(1)[OF A(1) u(2)] line_extension_mem_iff u(1) A(1) by auto
qed

lemma Span_subgroup_props [simp]:
  assumes "set Us ⊆ carrier R"
  shows "Span K Us ⊆ carrier R"
    and "0 ∈ Span K Us"
    and "⋀v1 v2. ⟦ v1 ∈ Span K Us; v2 ∈ Span K Us ⟧ ⟹ (v1 ⊕ v2) ∈
Span K Us"
    and "⋀v. v ∈ Span K Us ⟹ (⊖ v) ∈ Span K Us"
  using add.subgroupE subgroup.one_closed[of _ "add_monoid R"]
        Span_is_add_subgroup[OF assms(1)] by auto

lemma Span_smult_closed [simp]:
  assumes "set Us ⊆ carrier R"
  shows "⋀k v. ⟦ k ∈ K; v ∈ Span K Us ⟧ ⟹ k ⊗ v ∈ Span K Us"
  using assms
proof (induct Us)
  case Nil thus ?case
    using r_null subring_props(1) by (auto, blast)
next
  case Cons thus ?case
    using Span_subgroup_props(1) line_extension_smult_closed by auto
qed

lemma Span_m_inv_simprule [simp]:
  assumes "set Us ⊆ carrier R"
  shows "⟦ k ∈ K - { 0 }; a ∈ carrier R ⟧ ⟹ k ⊗ a ∈ Span K Us ⟹ a
∈ Span K Us"
proof -
  assume k: "k ∈ K - { 0 }" and a: "a ∈ carrier R" and ka: "k ⊗ a ∈
Span K Us"
  have inv_k: "inv k ∈ K" "inv k ⊗ k = 1"
    using subfield_m_inv[OF K k] by simp+
  hence "inv k ⊗ (k ⊗ a) ∈ Span K Us"
    using Span_smult_closed[OF assms _ ka] by simp
  thus ?thesis
    using inv_k subring_props(1)a k
    by (metis (no_types, lifting) DiffE l_one m_assoc subset_iff)
qed
```

## 34.4 Span as Linear Combinations

We show that Span is the set of linear combinations

```
lemma line_extension_of_combine_set:
  assumes "u ∈ carrier R"
  shows "line_extension K u { combine Ks Us | Ks. set Ks ⊆ K } =
               { combine Ks (u # Us) | Ks. set Ks ⊆ K }"
  (is "?line_extension = ?combinations")
```

```
proof
  show "?line_extension ⊆ ?combinations"
  proof
    fix v assume "v ∈ ?line_extension"
    then obtain k Ks
      where "k ∈ K" "set Ks ⊆ K" and "v = combine (k # Ks) (u # Us)"
      using line_extension_mem_iff by auto
    thus "v ∈ ?combinations"
      by (metis (mono_tags, lifting) insert_subset list.simps(15) mem_Collect_eq)
  qed
next
  show "?combinations ⊆ ?line_extension"
  proof
    fix v assume "v ∈ ?combinations"
    then obtain Ks where v: "set Ks ⊆ K" "v = combine Ks (u # Us)"
      by auto
    thus "v ∈ ?line_extension"
    proof (cases Ks)
      case Cons thus ?thesis
        using v line_extension_mem_iff by auto
    next
      case Nil
      hence "v = 0"
        using v by simp
      moreover have "combine [] Us = 0" by simp
      hence "0 ∈ { combine Ks Us | Ks. set Ks ⊆ K }"
        by (metis (mono_tags, lifting) local.Nil mem_Collect_eq v(1))
      hence "(0 ⊗ u) ⊕ 0 ∈ ?line_extension"
        using line_extension_mem_iff subring_props(2) by blast
      hence "0 ∈ ?line_extension"
        using assms by auto
      ultimately show ?thesis by auto
    qed
  qed
qed

lemma Span_eq_combine_set:
  assumes "set Us ⊆ carrier R" shows "Span K Us = { combine Ks Us |
Ks. set Ks ⊆ K }"
  using assms line_extension_of_combine_set
  by (induct Us) (auto, metis empty_set empty_subsetI)

lemma line_extension_of_combine_set_length_version:
  assumes "u ∈ carrier R"
  shows "line_extension K u { combine Ks Us | Ks. length Ks = length Us
∧ set Ks ⊆ K } =
                      { combine Ks (u # Us) | Ks. length Ks = length (u
# Us) ∧ set Ks ⊆ K }"
  (is "?line_extension = ?combinations")
```

```
proof
  show "?line_extension ⊆ ?combinations"
  proof
    fix v assume "v ∈ ?line_extension"
    then obtain k Ks
      where "v = combine (k # Ks) (u # Us)" "length (k # Ks) = length
(u # Us)" "set (k # Ks) ⊆ K"
      using line_extension_mem_iff by auto
    thus "v ∈ ?combinations" by blast
  qed
next
  show "?combinations ⊆ ?line_extension"
  proof
    fix c assume "c ∈ ?combinations"
    then obtain Ks where c: "c = combine Ks (u # Us)" "length Ks = length
(u # Us)" "set Ks ⊆ K"
      by blast
    then obtain k Ks' where k: "Ks = k # Ks'"
      by (metis length_Suc_conv)
    thus "c ∈ ?line_extension"
      using c line_extension_mem_iff unfolding k by auto
  qed
qed

lemma Span_eq_combine_set_length_version:
  assumes "set Us ⊆ carrier R"
  shows "Span K Us = { combine Ks Us | Ks. length Ks = length Us ∧ set
Ks ⊆ K }"
  using assms line_extension_of_combine_set_length_version by (induct
Us) (auto)
```

### 34.4.1 Corollaries

```
corollary Span_mem_iff_length_version:
  assumes "set Us ⊆ carrier R"
  shows "a ∈ Span K Us ⟷ (∃Ks. set Ks ⊆ K ∧ length Ks = length Us
∧ a = combine Ks Us)"
  using Span_eq_combine_set_length_version[OF assms] by blast

corollary Span_mem_imp_non_trivial_combine:
  assumes "set Us ⊆ carrier R" and "a ∈ Span K Us"
  obtains k Ks
  where "k ∈ K - { 0 }" "set Ks ⊆ K" "length Ks = length Us" "combine
(k # Ks) (a # Us) = 0"
proof -
  obtain Ks where Ks: "set Ks ⊆ K" "length Ks = length Us" "a = combine
Ks Us"
    using Span_mem_iff_length_version[OF assms(1)] assms(2) by auto
  hence "((⊖ 1) ⊗ a) ⊕ a = combine ((⊖ 1) # Ks) (a # Us)"
```

```
    by auto
  moreover have "((⊖ 1) ⊗ a) ⊕ a = 0"
    using assms(2) Span_subgroup_props(1)[OF assms(1)] l_minus l_neg by
auto
  moreover have "⊖ 1 ≠ 0"
    using subfieldE(6)[OF K] l_neg by force
  ultimately show ?thesis
    using that subring_props(3,5) Ks(1-2) by (force simp del: combine.simps)
qed

corollary Span_mem_iff:
  assumes "set Us ⊆ carrier R" and "a ∈ carrier R"
  shows "a ∈ Span K Us ⟷ (∃k ∈ K - { 0 }. ∃Ks. set Ks ⊆ K ∧ combine
(k # Ks) (a # Us) = 0)"
         (is "?in_Span ⟷ ?exists_combine")
proof
  assume "?in_Span"
  then obtain Ks where Ks: "set Ks ⊆ K" "a = combine Ks Us"
    using Span_eq_combine_set[OF assms(1)] by auto
  hence "((⊖ 1) ⊗ a) ⊕ a = combine ((⊖ 1) # Ks) (a # Us)"
    by auto
  moreover have "((⊖ 1) ⊗ a) ⊕ a = 0"
    using assms(2) l_minus l_neg by auto
  moreover have "⊖ 1 ≠ 0"
    using subfieldE(6)[OF K] l_neg by force
  ultimately show "?exists_combine"
    using subring_props(3,5) Ks(1) by (force simp del: combine.simps)
next
  assume "?exists_combine"
  then obtain k Ks
    where k: "k ∈ K" "k ≠ 0" and Ks: "set Ks ⊆ K" and a: "(k ⊗ a) ⊕
combine Ks Us = 0"
    by auto
  hence "combine Ks Us ∈ Span K Us"
    using Span_eq_combine_set[OF assms(1)] by auto
  hence "k ⊗ a ∈ Span K Us"
    using Span_subgroup_props[OF assms(1)] k Ks a
    by (metis (no_types, lifting) assms(2) contra_subsetD m_closed minus_equality
subring_props(1))
  thus "?in_Span"
    using Span_m_inv_simprule[OF assms(1) _ assms(2), of k] k by auto
qed
```

## 34.5 Span as the minimal subgroup that contains `K <#> set Us`

Now we show the link between Span and Group.generate

```
lemma mono_Span:
  assumes "set Us ⊆ carrier R" and "u ∈ carrier R"
```

```
  shows "Span K Us ⊆ Span K (u # Us)"
proof
  fix v assume v: "v ∈ Span K Us"
  hence "(0 ⊗ u) ⊕ v ∈ Span K (u # Us)"
    using line_extension_mem_iff by auto
  thus "v ∈ Span K (u # Us)"
    using Span_subgroup_props(1)[OF assms(1)] assms(2) v
    by (auto simp del: Span.simps)
qed

lemma Span_min:
  assumes "set Us ⊆ carrier R" and "subgroup E (add_monoid R)"
  shows "K <#> (set Us) ⊆ E ⟹ Span K Us ⊆ E"
proof -
  assume "K <#> (set Us) ⊆ E" show "Span K Us ⊆ E"
  proof
    fix v assume "v ∈ Span K Us"
    then obtain Ks where v: "set Ks ⊆ K" "v = combine Ks Us"
      using Span_eq_combine_set[OF assms(1)] by auto
    from ‹set Ks ⊆ K› ‹set Us ⊆ carrier R› and ‹K <#> (set Us) ⊆ E›
    show "v ∈ E" unfolding v(2)
    proof (induct Ks Us rule: combine.induct)
      case (1 k Ks u Us)
      hence "k ∈ K" and "u ∈ set (u # Us)" by auto
      hence "k ⊗ u ∈ E"
        using 1(4) unfolding set_mult_def by auto
      moreover have "K <#> set Us ⊆ E"
        using 1(4) unfolding set_mult_def by auto
      hence "combine Ks Us ∈ E"
        using 1 by auto
      ultimately show ?case
        using add.subgroupE(4)[OF assms(2)] by auto
    next
      case "2_1" thus ?case
        using subgroup.one_closed[OF assms(2)] by auto
    next
      case  "2_2" thus ?case
        using subgroup.one_closed[OF assms(2)] by auto
    qed
  qed
qed

lemma Span_eq_generate:
  assumes "set Us ⊆ carrier R" shows "Span K Us = generate (add_monoid
R) (K <#> (set Us))"
proof (rule add.generateI)
  show "subgroup (Span K Us) (add_monoid R)"
    using Span_is_add_subgroup[OF assms] .
next
```

```
  show "⋀E. ⟦ subgroup E (add_monoid R); K <#> set Us ⊆ E ⟧ ⟹ Span
K Us ⊆ E"
    using Span_min assms by blast
next
  show "K <#> set Us ⊆ Span K Us"
  using assms
  proof (induct Us)
    case Nil thus ?case
      unfolding set_mult_def by auto
  next
    case (Cons u Us)
    have "K <#> set (u # Us) = (K <#> { u }) ∪ (K <#> set Us)"
      unfolding set_mult_def by auto
    moreover have "⋀k. k ∈ K ⟹ k ⊗ u ∈ Span K (u # Us)"
    proof -
      fix k assume k: "k ∈ K"
      hence "combine [ k ] (u # Us) ∈ Span K (u # Us)"
        using Span_eq_combine_set[OF Cons(2)] by (auto simp del: combine.simps)
      moreover have "k ∈ carrier R" and "u ∈ carrier R"
        using Cons(2) k subring_props(1) by (blast, auto)
      ultimately show "k ⊗ u ∈ Span K (u # Us)"
        by (auto simp del: Span.simps)
    qed
    hence "K <#> { u } ⊆ Span K (u # Us)"
      unfolding set_mult_def by auto
    moreover have "K <#> set Us ⊆ Span K (u # Us)"
      using mono_Span[of Us u] Cons by (auto simp del: Span.simps)
    ultimately show ?case
      using Cons by (auto simp del: Span.simps)
  qed
qed
```

### 34.5.1 Corollaries

```
corollary Span_same_set:
  assumes "set Us ⊆ carrier R"
  shows "set Us = set Vs ⟹ Span K Us = Span K Vs"
  using Span_eq_generate assms by auto

corollary Span_incl: "set Us ⊆ carrier R ⟹ K <#> (set Us) ⊆ Span K Us"
  using Span_eq_generate generate.incl[of _ _ "add_monoid R"] by auto

corollary Span_base_incl: "set Us ⊆ carrier R ⟹ set Us ⊆ Span K Us"
proof -
  assume A: "set Us ⊆ carrier R"
  hence "{ 1 } <#> set Us = set Us"
    unfolding set_mult_def by force
  moreover have "{ 1 } <#> set Us ⊆ K <#> set Us"
    using subring_props(3) unfolding set_mult_def by blast
```

```
  ultimately show ?thesis
    using Span_incl[OF A] by auto
qed

corollary mono_Span_sublist:
  assumes "set Us ⊆ set Vs" "set Vs ⊆ carrier R"
  shows "Span K Us ⊆ Span K Vs"
  using add.mono_generate[OF mono_set_mult[OF _ assms(1), of K K R]]
        Span_eq_generate[OF assms(2)] Span_eq_generate[of Us] assms by
auto

corollary mono_Span_append:
  assumes "set Us ⊆ carrier R" "set Vs ⊆ carrier R"
  shows "Span K Us ⊆ Span K (Us @ Vs)"
    and "Span K Us ⊆ Span K (Vs @ Us)"
  using mono_Span_sublist[of Us "Us @ Vs"] assms
        Span_same_set[of "Us @ Vs" "Vs @ Us"] by auto

corollary mono_Span_subset:
  assumes "set Us ⊆ Span K Vs" "set Vs ⊆ carrier R"
  shows "Span K Us ⊆ Span K Vs"
proof (rule Span_min[OF _ Span_is_add_subgroup[OF assms(2)]])
  show "set Us ⊆ carrier R"
    using Span_subgroup_props(1)[OF assms(2)] assms by auto
  show "K <#> set Us ⊆ Span K Vs"
    using Span_smult_closed[OF assms(2)] assms(1) unfolding set_mult_def
by blast
qed

lemma Span_strict_incl:
  assumes "set Us ⊆ carrier R" "set Vs ⊆ carrier R"
  shows "Span K Us ⊂ Span K Vs ⟹ (∃v ∈ set Vs. v ∉ Span K Us)"
proof -
  assume "Span K Us ⊂ Span K Vs" show "∃v ∈ set Vs. v ∉ Span K Us"
  proof (rule ccontr)
    assume "¬ (∃v ∈ set Vs. v ∉ Span K Us)"
    hence "Span K Vs ⊆ Span K Us"
      using mono_Span_subset[OF _ assms(1), of Vs] by auto
    from ‹Span K Us ⊂ Span K Vs› and ‹Span K Vs ⊆ Span K Us›
    show False by simp
  qed
qed

lemma Span_append_eq_set_add:
  assumes "set Us ⊆ carrier R" and "set Vs ⊆ carrier R"
  shows "Span K (Us @ Vs) = (Span K Us <+>_R Span K Vs)"
  using assms
proof (induct Us)
  case Nil thus ?case
```

```
    using Span_subgroup_props(1)[OF Nil(2)] unfolding set_add_def' by
force
next
  case (Cons u Us)
  hence in_carrier:
    "u ∈ carrier R" "set Us ⊆ carrier R" "set Vs ⊆ carrier R"
    by auto

  have "line_extension K u (Span K Us <+>R Span K Vs) = (Span K (u # Us)
<+>R Span K Vs)"
  proof
    show "line_extension K u (Span K Us <+>R Span K Vs) ⊆ (Span K (u
# Us) <+>R Span K Vs)"
    proof
      fix v assume "v ∈ line_extension K u (Span K Us <+>R Span K Vs)"
      then obtain k u' v'
        where v: "k ∈ K" "u' ∈ Span K Us" "v' ∈ Span K Vs" "v = k ⊗
u ⊕ (u' ⊕ v')"
        using line_extension_mem_iff[of v _ u "Span K Us <+>R Span K Vs"]
        unfolding set_add_def' by blast
      hence "v = (k ⊗ u ⊕ u') ⊕ v'"
        using in_carrier(2-3)[THEN Span_subgroup_props(1)] in_carrier(1)
subring_props(1)
        by (metis (no_types, lifting) rev_subsetD ring_simprules(7) semiring_simprules(3))
      moreover have "k ⊗ u ⊕ u' ∈ Span K (u # Us)"
        using line_extension_mem_iff v(1-2) by auto
      ultimately show "v ∈ Span K (u # Us) <+>R Span K Vs"
        unfolding set_add_def' using v(3) by auto
    qed
  next
    show "Span K (u # Us) <+>R Span K Vs ⊆ line_extension K u (Span K
Us <+>R Span K Vs)"
    proof
      fix v assume "v ∈ Span K (u # Us) <+>R Span K Vs"
      then obtain k u' v'
        where v: "k ∈ K" "u' ∈ Span K Us" "v' ∈ Span K Vs" "v = (k ⊗
u ⊕ u') ⊕ v'"
        using line_extension_mem_iff[of _ _ u "Span K Us"] unfolding set_add_def'
by auto
      hence "v = (k ⊗ u) ⊕ (u' ⊕ v')"
        using in_carrier(2-3)[THEN Span_subgroup_props(1)] in_carrier(1)
subring_props(1)
        by (metis (no_types, lifting) rev_subsetD ring_simprules(5,7))
      thus "v ∈ line_extension K u (Span K Us <+>R Span K Vs)"
        using line_extension_mem_iff[of "(k ⊗ u) ⊕ (u' ⊕ v')" K u "Span
K Us <+>R Span K Vs"]
        unfolding set_add_def' using v by auto
    qed
  qed
```

```
  thus ?case
    using Cons by auto
qed
```

## 34.6 Characterisation of Linearly Independent "Sets"

```
declare independent_backwards [intro]
declare independent_in_carrier [intro]

lemma independent_distinct: "independent K Us ⟹ distinct Us"
proof (induct Us rule: list.induct)
  case Nil thus ?case by simp
next
  case Cons thus ?case
    using independent_backwards[OF Cons(2)]
          independent_in_carrier[OF Cons(2)]
          Span_base_incl
    by auto
qed

lemma independent_strict_incl:
  assumes "independent K (u # Us)" shows "Span K Us ⊂ Span K (u # Us)"
proof -
  have "u ∈ Span K (u # Us)"
    using Span_base_incl[OF independent_in_carrier[OF assms]] by auto
  moreover have "Span K Us ⊆ Span K (u # Us)"
    using mono_Span independent_in_carrier[OF assms] by auto
  ultimately show ?thesis
    using independent_backwards(1)[OF assms] by auto
qed

corollary independent_replacement:
  assumes "independent K (u # Us)" and "independent K Vs"
  shows "Span K (u # Us) ⊆ Span K Vs ⟹ (∃v ∈ set Vs. independent K
(v # Us))"
proof -
  assume "Span K (u # Us) ⊆ Span K Vs"
  hence "Span K Us ⊂ Span K Vs"
    using independent_strict_incl[OF assms(1)] by auto
  then obtain v where v: "v ∈ set Vs" "v ∉ Span K Us"
    using Span_strict_incl[of Us Vs] assms[THEN independent_in_carrier]
by auto
  thus ?thesis
    using li_Cons[of v K Us] assms independent_in_carrier[OF assms(2)]
by auto
qed

lemma independent_split:
  assumes "independent K (Us @ Vs)"
```

```
  shows "independent K Vs"
    and "independent K Us"
    and "Span K Us ∩ Span K Vs = { 𝟬 }"
proof -
  from assms show "independent K Vs"
    by (induct Us) (auto)
next
  from assms show "independent K Us"
  proof (induct Us)
    case Nil thus ?case by simp
  next
    case (Cons u Us')
    hence u: "u ∈ carrier R" and "set Us' ⊆ carrier R" "set Vs ⊆ carrier
R"
      using independent_in_carrier[of K "(u # Us') @ Vs"] by auto
    hence "Span K Us' ⊆ Span K (Us' @ Vs)"
      using mono_Span_append(1) by simp
    thus ?case
      using independent_backwards[of K u "Us' @ Vs"] Cons li_Cons[OF u]
by auto
  qed
next
  from assms show "Span K Us ∩ Span K Vs = { 𝟬 }"
  proof (induct Us rule: list.induct)
    case Nil thus ?case
      using Span_subgroup_props(2)[OF independent_in_carrier[of K Vs]]
by simp
  next
    case (Cons u Us)
    hence IH: "Span K Us ∩ Span K Vs = {𝟬}" by auto
    have in_carrier:
      "u ∈ carrier R" "set Us ⊆ carrier R" "set Vs ⊆ carrier R" "set
(u # Us) ⊆ carrier R"
      using Cons(2)[THEN independent_in_carrier] by auto
    hence "{ 𝟬 } ⊆ Span K (u # Us) ∩ Span K Vs"
      using in_carrier(3-4)[THEN Span_subgroup_props(2)] by auto

    moreover have "Span K (u # Us) ∩ Span K Vs ⊆ { 𝟬 }"
    proof (rule ccontr)
      assume "¬ Span K (u # Us) ∩ Span K Vs ⊆ {𝟬}"
      hence "∃a. a ≠ 𝟬 ∧ a ∈ Span K (u # Us) ∧ a ∈ Span K Vs" by auto
      then obtain k u' v'
        where u': "u' ∈ Span K Us" "u' ∈ carrier R"
          and v': "v' ∈ Span K Vs" "v' ∈ carrier R" "v' ≠ 𝟬"
          and k: "k ∈ K" "(k ⊗ u ⊕ u') = v'"
        using line_extension_mem_iff[of _ _ u "Span K Us"] in_carrier(2-3)[THEN
Span_subgroup_props(1)]
              subring_props(1) by force
      hence "v' = 𝟬" if "k = 𝟬"
```

```
        using in_carrier(1) that IH by auto
      hence diff_zero: "k ≠ 0" using v'(3) by auto

      have "k ∈ carrier R"
        using subring_props(1) k(1) by blast
      hence "k ⊗ u = (⊖ u') ⊕ v'"
        using in_carrier(1) k(2) u'(2) v'(2) add.m_comm r_neg1 by auto
      hence "k ⊗ u ∈ Span K (Us @ Vs)"
        using Span_subgroup_props(4)[OF in_carrier(2) u'(1)] v'(1)
              Span_append_eq_set_add[OF in_carrier(2-3)] unfolding set_add_def'
by blast
      hence "u ∈ Span K (Us @ Vs)"
        using Cons(2) Span_m_inv_simprule[OF _ _ in_carrier(1), of "Us
@ Vs" k]
              diff_zero k(1) in_carrier(2-3) by auto
      moreover have "u ∉ Span K (Us @ Vs)"
        using independent_backwards(1)[of K u "Us @ Vs"] Cons(2) by auto
      ultimately show False by simp
    qed

    ultimately show ?case by auto
  qed
qed

lemma independent_append:
  assumes "independent K Us" and "independent K Vs" and "Span K Us ∩
Span K Vs = { 0 }"
  shows "independent K (Us @ Vs)"
  using assms
proof (induct Us rule: list.induct)
  case Nil thus ?case by simp
next
  case (Cons u Us)
  hence in_carrier:
    "u ∈ carrier R" "set Us ⊆ carrier R" "set Vs ⊆ carrier R" "set (u
# Us) ⊆ carrier R"
    using Cons(2-3)[THEN independent_in_carrier] by auto
  hence "Span K Us ⊆ Span K (u # Us)"
    using mono_Span by auto
  hence "Span K Us ∩ Span K Vs = { 0 }"
    using Cons(4) Span_subgroup_props(2)[OF in_carrier(2)] by auto
  hence "independent K (Us @ Vs)"
    using Cons by auto
  moreover have "u ∉ Span K (Us @ Vs)"
  proof (rule ccontr)
    assume "¬ u ∉ Span K (Us @ Vs)"
    then obtain u' v'
      where u': "u' ∈ Span K Us" "u' ∈ carrier R"
        and v': "v' ∈ Span K Vs" "v' ∈ carrier R" and u:"u = u' ⊕ v'"
```

```
      using Span_append_eq_set_add[OF in_carrier(2-3)] in_carrier(2-3)[THEN
Span_subgroup_props(1)]
      unfolding set_add_def' by blast
    hence "u ⊕ (⊖ u') = v'"
      using in_carrier(1) by algebra
    moreover have "u ∈ Span K (u # Us)" and "u' ∈ Span K (u # Us)"
      using Span_base_incl[OF in_carrier(4)] mono_Span[OF in_carrier(2,1)]
u'(1)
      by (auto simp del: Span.simps)
    hence "u ⊕ (⊖ u') ∈ Span K (u # Us)"
      using Span_subgroup_props(3-4)[OF in_carrier(4)] by (auto simp del:
Span.simps)
    ultimately have "u ⊕ (⊖ u') = 0"
      using Cons(4) v'(1) by auto
    hence "u = u'"
      using Cons(4) v'(1) in_carrier(1) u'(2) ‹u ⊕ ⊖ u' = v'› u by auto
    thus False
      using u'(1) independent_backwards(1)[OF Cons(2)] by simp
  qed
  ultimately show ?case
    using in_carrier(1) li_Cons by simp
qed

lemma independent_imp_trivial_combine:
  assumes "independent K Us"
  shows "⋀Ks. ⟦ set Ks ⊆ K; combine Ks Us = 0 ⟧ ⟹ set (take (length
Us) Ks) ⊆ { 0 }"
  using assms
proof (induct Us rule: list.induct)
  case Nil thus ?case by simp
next
  case (Cons u Us) thus ?case
  proof (cases "Ks = []")
    assume "Ks = []" thus ?thesis by auto
  next
    assume "Ks ≠ []"
    then obtain k Ks' where k: "k ∈ K" and Ks': "set Ks' ⊆ K" and Ks:
"Ks = k # Ks'"
      using Cons(2) by (metis insert_subset list.exhaust_sel list.simps(15))
    hence Us: "set Us ⊆ carrier R" and u: "u ∈ carrier R"
      using independent_in_carrier[OF Cons(4)] by auto
    have "u ∈ Span K Us" if "k ≠ 0"
      using that Span_mem_iff[OF Us u] Cons(3-4) Ks' k unfolding Ks by
blast
    hence k_zero: "k = 0"
      using independent_backwards[OF Cons(4)] by blast
    hence "combine Ks' Us = 0"
      using combine_in_carrier[OF _ Us, of Ks'] Ks' u Cons(3) subring_props(1)
unfolding Ks by auto
```

```
    hence "set (take (length Us) Ks') ⊆ { 0 }"
      using Cons(1)[OF Ks' _ independent_backwards(2)[OF Cons(4)]] by
simp
    thus ?thesis
      using k_zero unfolding Ks by auto
  qed
qed

lemma non_trivial_combine_imp_dependent:
  assumes "set Ks ⊆ K" and "combine Ks Us = 0" and "¬ set (take (length
Us) Ks) ⊆ { 0 }"
  shows "dependent K Us"
  using independent_imp_trivial_combine[OF _ assms(1-2)] assms(3) by blast


lemma trivial_combine_imp_independent:
  assumes "set Us ⊆ carrier R"
    and "⋀Ks. ⟦ set Ks ⊆ K; combine Ks Us = 0 ⟧ ⟹ set (take (length
Us) Ks) ⊆ { 0 }"
  shows "independent K Us"
  using assms
proof (induct Us)
  case Nil thus ?case by simp
next
  case (Cons u Us)
  hence Us: "set Us ⊆ carrier R" and u: "u ∈ carrier R" by auto

  have "⋀Ks. ⟦ set Ks ⊆ K; combine Ks Us = 0 ⟧ ⟹ set (take (length
Us) Ks) ⊆ { 0 }"
  proof -
    fix Ks assume Ks: "set Ks ⊆ K" and lin_c: "combine Ks Us = 0"
    hence "combine (0 # Ks) (u # Us) = 0"
      using u subring_props(1) combine_in_carrier[OF _ Us] by auto
    hence "set (take (length (u # Us)) (0 # Ks)) ⊆ { 0 }"
      using Cons(3)[of "0 # Ks"] subring_props(2) Ks by auto
    thus "set (take (length Us) Ks) ⊆ { 0 }" by auto
  qed
  hence "independent K Us"
    using Cons(1)[OF Us] by simp

  moreover have "u ∉ Span K Us"
  proof (rule ccontr)
    assume "¬ u ∉ Span K Us"
    then obtain k Ks where k: "k ∈ K" "k ≠ 0" and Ks: "set Ks ⊆ K"
and u: "combine (k # Ks) (u # Us) = 0"
      using Span_mem_iff[OF Us u] by auto
    have "set (take (length (u # Us)) (k # Ks)) ⊆ { 0 }"
      using Cons(3)[OF _ u] k(1) Ks by auto
    hence "k = 0" by auto
```

```
    from ‹k = 0› and ‹k ≠ 0› show False by simp
  qed

  ultimately show ?case
    using li_Cons[OF u] by simp
qed

corollary dependent_imp_non_trivial_combine:
  assumes "set Us ⊆ carrier R" and "dependent K Us"
  obtains Ks where "length Ks = length Us" "combine Ks Us = 0" "set Ks
⊆ K" "set Ks ≠ { 0 }"
proof -
  obtain Ks
    where Ks: "set Ks ⊆ carrier R" "set Ks ⊆ K" "combine Ks Us = 0"
"¬ set (take (length Us) Ks) ⊆ { 0 }"
    using trivial_combine_imp_independent[OF assms(1)] assms(2) subring_props(1)
by blast
  obtain Ks'
    where Ks': "set (take (length Us) Ks) ⊆ set Ks'" "set Ks' ⊆ set
(take (length Us) Ks) ∪ { 0 }"
               "length Ks' = length Us" "combine Ks' Us = 0"
    using combine_normalize[OF Ks(1) assms(1) Ks(3)] by metis
  have "set (take (length Us) Ks) ⊆ set Ks"
    by (simp add: set_take_subset)
  hence "set Ks' ⊆ K"
    using Ks(2) Ks'(2) subring_props(2) Un_commute by blast
  moreover have "set Ks' ≠ { 0 }"
    using Ks'(1) Ks(4) by auto
  ultimately show thesis
    using that Ks' by blast
qed

corollary unique_decomposition:
  assumes "independent K Us"
  shows "a ∈ Span K Us ⟹ ∃!Ks. set Ks ⊆ K ∧ length Ks = length Us
∧ a = combine Ks Us"
proof -
  note in_carrier = independent_in_carrier[OF assms]

  assume "a ∈ Span K Us"
  then obtain Ks where Ks: "set Ks ⊆ K" "length Ks = length Us" "a =
combine Ks Us"
    using Span_mem_iff_length_version[OF in_carrier] by blast

  moreover
  have "⋀Ks'. ⟦ set Ks' ⊆ K; length Ks' = length Us; a = combine Ks'
Us ⟧ ⟹ Ks = Ks'"
  proof -
    fix Ks' assume Ks': "set Ks' ⊆ K" "length Ks' = length Us" "a = combine
```

```
Ks' Us"
    hence set_Ks: "set Ks ⊆ carrier R" and set_Ks': "set Ks' ⊆ carrier
R"
      using subring_props(1) Ks(1) by blast+
    have same_length: "length Ks = length Ks'"
      using Ks Ks' by simp

    have "(combine Ks Us) ⊕ ((⊖ 1) ⊗ (combine Ks' Us)) = 0"
      using combine_in_carrier[OF set_Ks  in_carrier]
            combine_in_carrier[OF set_Ks' in_carrier] Ks(3) Ks'(3) by
algebra
    hence "(combine Ks Us) ⊕ (combine (map ((⊗) (⊖ 1)) Ks') Us) = 0"
      using combine_r_distr[OF set_Ks' in_carrier, of "⊖ 1"] subring_props
by auto
    moreover have set_map: "set (map ((⊗) (⊖ 1)) Ks') ⊆ K"
      using Ks'(1) subring_props by (induct Ks') (auto)
    hence "set (map ((⊗) (⊖ 1)) Ks') ⊆ carrier R"
      using subring_props(1) by blast
    ultimately have "combine (map2 (⊕) Ks (map ((⊗) (⊖ 1)) Ks')) Us
= 0"
      using combine_add[OF Ks(2) _ set_Ks _ in_carrier, of "map ((⊗) (⊖
1)) Ks'"] Ks'(2) by auto
    moreover have "set (map2 (⊕) Ks (map ((⊗) (⊖ 1)) Ks')) ⊆ K"
      using Ks(1) set_map subring_props(7)
      by (induct Ks) (auto, metis contra_subsetD in_set_zipE local.set_map
set_ConsD subring_props(7))
    ultimately have "set (take (length Us) (map2 (⊕) Ks (map ((⊗) (⊖
1)) Ks'))) ⊆ { 0 }"
      using independent_imp_trivial_combine[OF assms] by auto
    hence "set (map2 (⊕) Ks (map ((⊗) (⊖ 1)) Ks')) ⊆ { 0 }"
      using Ks(2) Ks'(2) by auto
    thus "Ks = Ks'"
      using set_Ks set_Ks' same_length
    proof (induct Ks arbitrary: Ks')
      case Nil thus?case by simp
    next
      case (Cons k Ks)
      then obtain k' Ks'' where k': "Ks' = k' # Ks''"
        by (metis Suc_length_conv)
      have "Ks = Ks''"
        using Cons unfolding k' by auto
      moreover have "k = k'"
        using Cons(2-4) l_minus minus_equality unfolding k' by (auto,
fastforce)
      ultimately show ?case
        unfolding k' by simp
    qed
  qed
```

```
  ultimately show ?thesis by blast
qed
```

## 34.7 Replacement Theorem

```
lemma independent_rotate1_aux:
  "independent K (u # Us @ Vs) ⟹ independent K ((Us @ [u]) @ Vs)"
proof -
  assume "independent K (u # Us @ Vs)"
  hence li: "independent K [u]" "independent K Us" "independent K Vs"
    and inter: "Span K [u] ∩ Span K Us = { 0 }"
               "Span K (u # Us) ∩ Span K Vs = { 0 }"
    using independent_split[of "u # Us" Vs] independent_split[of "[u]"
Us] by auto
  hence "independent K (Us @ [u])"
    using independent_append[OF li(2,1)] by auto
  moreover have "Span K (Us @ [u]) ∩ Span K Vs = { 0 }"
    using Span_same_set[of "u # Us" "Us @ [u]"] li(1-2)[THEN independent_in_carrier]
inter(2) by auto
  ultimately show "independent K ((Us @ [u]) @ Vs)"
    using independent_append[OF _ li(3), of "Us @ [u]"] by simp
qed

corollary independent_rotate1:
  "independent K (Us @ Vs) ⟹ independent K ((rotate1 Us) @ Vs)"
  using independent_rotate1_aux by (cases Us) (auto)



corollary independent_same_set:
  assumes "set Us = set Vs" and "length Us = length Vs"
  shows "independent K Us ⟹ independent K Vs"
proof -
  assume "independent K Us" thus ?thesis
    using assms
  proof (induct Us arbitrary: Vs rule: list.induct)
    case Nil thus ?case by simp
  next
    case (Cons u Us)
    then obtain Vs' Vs'' where Vs: "Vs = Vs' @ (u # Vs'')"
      by (metis list.set_intros(1) split_list)

    have in_carrier: "u ∈ carrier R" "set Us ⊆ carrier R"
      using independent_in_carrier[OF Cons(2)] by auto

    have "distinct Vs"
      using Cons(3-4) independent_distinct[OF Cons(2)]
      by (metis card_distinct distinct_card)
    hence "u ∉ set (Vs' @ Vs'')" and "u ∉ set Us"
```

```
      using independent_distinct[OF Cons(2)] unfolding Vs by auto
    hence set_eq: "set Us = set (Vs' @ Vs'')" and "length (Vs' @ Vs'')
= length Us"
      using Cons(3-4) unfolding Vs by auto
    hence "independent K (Vs' @ Vs'')"
      using Cons(1)[OF independent_backwards(2)[OF Cons(2)]] unfolding
Vs by simp
    hence "independent K (u # (Vs' @ Vs''))"
      using li_Cons Span_same_set[OF _ set_eq] independent_backwards(1)[OF
Cons(2)] in_carrier by auto
    hence "independent K (Vs' @ (u # Vs''))"
      using independent_rotate1[of "u # Vs'" Vs''] by auto
    thus ?case unfolding Vs .
  qed
qed

lemma replacement_theorem:
  assumes "independent K (Us' @ Us)" and "independent K Vs"
    and "Span K (Us' @ Us) ⊆ Span K Vs"
  shows "∃Vs'. set Vs' ⊆ set Vs ∧ length Vs' = length Us' ∧ independent
K (Vs' @ Us)"
  using assms
proof (induct "length Us'" arbitrary: Us' Us)
  case 0 thus ?case by auto
next
  case (Suc n)
  then obtain u Us'' where Us'': "Us' = Us'' @ [u]"
    by (metis list.size(3) nat.simps(3) rev_exhaust)
  then obtain Vs' where Vs': "set Vs' ⊆ set Vs" "length Vs' = n" "independent
K (Vs' @ (u # Us))"
    using Suc(1)[of Us'' "u # Us"] Suc(2-5) by auto
  hence li: "independent K ((u # Vs') @ Us)"
    using independent_same_set[OF _ _ Vs'(3), of "(u # Vs') @ Us"] by
auto
  moreover have in_carrier:
    "u ∈ carrier R" "set Us ⊆ carrier R" "set Us' ⊆ carrier R" "set Vs
⊆ carrier R"
    using Suc(3-4)[THEN independent_in_carrier] Us'' by auto
  moreover have "Span K ((u # Vs') @ Us) ⊆ Span K Vs"
  proof -
    have "set Us ⊆ Span K Vs" "u ∈ Span K Vs"
      using Suc(5) Span_base_incl[of "Us' @ Us"] Us'' in_carrier(2-3)
by auto
    moreover have "set Vs' ⊆ Span K Vs"
      using Span_base_incl[OF in_carrier(4)] Vs'(1) by auto
    ultimately have "set ((u # Vs') @ Us) ⊆ Span K Vs" by auto
    thus ?thesis
      using mono_Span_subset[OF _ in_carrier(4)] by (simp del: Span.simps)
  qed
```

```
  ultimately obtain v where "v ∈ set Vs" "independent K ((v # Vs') @
Us)"
    using independent_replacement[OF _ Suc(4), of u "Vs' @ Us"] by auto
  thus ?case
    using Vs'(1-2) Suc(2)
    by (metis (mono_tags, lifting) insert_subset length_Cons list.simps(15))
qed

corollary independent_length_le:
  assumes "independent K Us" and "independent K Vs"
  shows "set Us ⊆ Span K Vs ⟹ length Us ≤ length Vs"
proof -
  assume "set Us ⊆ Span K Vs"
  hence "Span K Us ⊆ Span K Vs"
    using mono_Span_subset[OF _ independent_in_carrier[OF assms(2)]] by
simp
  then obtain Vs' where Vs': "set Vs' ⊆ set Vs" "length Vs' = length
Us" "independent K Vs'"
    using replacement_theorem[OF _ assms(2), of Us "[]"] assms(1) by auto
  hence "card (set Vs') ≤ card (set Vs)"
    by (simp add: card_mono)
  thus "length Us ≤ length Vs"
    using independent_distinct assms(2) Vs'(2-3) by (simp add: distinct_card)
qed
```

## 34.8 Dimension

```
lemma exists_base:
  assumes "dimension n K E"
  shows "∃Vs. set Vs ⊆ carrier R ∧ independent K Vs ∧ length Vs = n
∧ Span K Vs = E"
    (is "∃Vs. ?base K Vs E n")
  using assms
proof (induct E rule: dimension.induct)
  case zero_dim thus ?case by auto
next
  case (Suc_dim v E n K)
  then obtain Vs where Vs: "set Vs ⊆ carrier R" "independent K Vs" "length
Vs = n" "Span K Vs = E"
    by auto
  hence "?base K (v # Vs) (line_extension K v E) (Suc n)"
    using Suc_dim li_Cons by auto
  thus ?case by blast
qed

lemma dimension_zero: "dimension 0 K E ⟹ E = { 0 }"
proof -
  assume "dimension 0 K E"
  then obtain Vs where "length Vs = 0" "Span K Vs = E"
```

```
    using exists_base by blast
  thus ?thesis
    by auto
qed

lemma dimension_one [iff]: "dimension 1 K K"
proof -
  have "K = Span K [ 1 ]"
    using line_extension_mem_iff[of _ K 1 "{ 0 }"] subfieldE(3)[OF K]
by (auto simp add: rev_subsetD)
  thus ?thesis
    using dimension.Suc_dim[OF one_closed _ dimension.zero_dim, of K]
subfieldE(6)[OF K] by auto
qed

lemma dimensionI:
  assumes "independent K Us" "Span K Us = E"
  shows "dimension (length Us) K E"
  using dimension_independent[OF assms(1)] assms(2) by simp

lemma space_subgroup_props:
  assumes "dimension n K E"
  shows "E ⊆ carrier R"
    and "0 ∈ E"
    and "⋀v1 v2. ⟦ v1 ∈ E; v2 ∈ E ⟧ ⟹ (v1 ⊕ v2) ∈ E"
    and "⋀v. v ∈ E ⟹ (⊖ v) ∈ E"
    and "⋀k v. ⟦ k ∈ K; v ∈ E ⟧ ⟹ k ⊗ v ∈ E"
    and "⟦ k ∈ K - { 0 }; a ∈ carrier R ⟧ ⟹ k ⊗ a ∈ E ⟹ a ∈ E"
  using exists_base[OF assms] Span_subgroup_props Span_smult_closed Span_m_inv_simprule
by auto

lemma independent_length_le_dimension:
  assumes "dimension n K E" and "independent K Us" "set Us ⊆ E"
  shows "length Us ≤ n"
proof -
  obtain Vs where Vs: "set Vs ⊆ carrier R" "independent K Vs" "length
Vs = n" "Span K Vs = E"
    using exists_base[OF assms(1)] by auto
  thus ?thesis
    using independent_length_le assms(2-3) by auto
qed

lemma dimension_is_inj:
  assumes "dimension n K E" and "dimension m K E"
  shows "n = m"
proof -
  have aux_lemma: "n ≤ m" if n: "dimension n K E" and m: "dimension m
K E" for n m
  proof -
```

```
    from that obtain Vs
      where Vs: "set Vs ⊆ carrier R" "independent K Vs" "length Vs =
n" "Span K Vs = E"
      using exists_base by meson
    then show ?thesis
      using independent_length_le_dimension[OF m Vs(2)] Span_base_incl[OF
Vs(1)] by auto
  qed
  show ?thesis
    using aux_lemma[OF assms] aux_lemma[OF assms(2,1)] by simp
qed

corollary independent_length_eq_dimension:
  assumes "dimension n K E" and "independent K Us" "set Us ⊆ E"
  shows "length Us = n ⟷ Span K Us = E"
proof
  assume len: "length Us = n" show "Span K Us = E"
  proof (rule ccontr)
    assume "Span K Us ≠ E"
    hence "Span K Us ⊂ E"
      using mono_Span_subset[of Us] exists_base[OF assms(1)] assms(3)
by blast
    then obtain v where v: "v ∈ E" "v ∉ Span K Us"
      using Span_strict_incl exists_base[OF assms(1)] space_subgroup_props(1)[OF
assms(1)] assms by blast
    hence "independent K (v # Us)"
      using li_Cons[OF _ _ assms(2)] space_subgroup_props(1)[OF assms(1)]
by auto
    hence "length (v # Us) ≤ n"
      using independent_length_le_dimension[OF assms(1)] v(1) assms(2-3)
by fastforce
    moreover have "length (v # Us) = Suc n"
      using len by simp
    ultimately show False by simp
  qed
next
  assume "Span K Us = E"
  hence "dimension (length Us) K E"
    using dimensionI assms by auto
  thus "length Us = n"
    using dimension_is_inj[OF assms(1)] by auto
qed

lemma complete_base:
  assumes "dimension n K E" and "independent K Us" "set Us ⊆ E"
  shows "∃Vs. length (Vs @ Us) = n ∧ independent K (Vs @ Us) ∧ Span K
(Vs @ Us) = E"
proof -
  have aux_lemma: "∃Vs. length (Vs @ Us) = n ∧ independent K (Vs @ Us)
```

```
∧ Span K (Vs @ Us) = E"
    if "k ≤ n" "independent K Us" "set Us ⊆ E" "length Us = k" for Us
k
    using that
  proof (induct arbitrary: Us rule: inc_induct)
    case base
    thus ?case using independent_length_eq_dimension[OF assms(1) base(1-2)]
by auto
  next
    case (step m)
    have "Span K Us ⊆ E"
      using mono_Span_subset step(4-6) exists_base[OF assms(1)] by blast
    hence "Span K Us ⊂ E"
      using independent_length_eq_dimension[OF assms(1) step(4-5)] step(2,6)
assms(1) by blast
    then obtain v where v: "v ∈ E" "v ∉ Span K Us"
      using Span_strict_incl exists_base[OF assms(1)] by blast
    hence "independent K (v # Us)"
      using space_subgroup_props(1)[OF assms(1)] li_Cons[OF _ v(2) step(4)]
by auto
    then obtain Vs
      where "length (Vs @ (v # Us)) = n" "independent K (Vs @ (v # Us))"
"Span K (Vs @ (v # Us)) = E"
      using step(3)[of "v # Us"] step(1-2,4-6) v by auto
    thus ?case
      by (metis append.assoc append_Cons append_Nil)
  qed
  have "length Us ≤ n"
    using independent_length_le_dimension[OF assms] .
  thus ?thesis
    using aux_lemma[OF _ assms(2-3)] by auto
qed

lemma filter_base:
  assumes "set Us ⊆ carrier R"
  obtains Vs where "set Vs ⊆ carrier R" and "independent K Vs" and "Span
K Vs = Span K Us"
proof -
  from ‹set Us ⊆ carrier R› have "∃Vs. independent K Vs ∧ Span K Vs
= Span K Us"
  proof (induction Us)
    case Nil thus ?case by auto
  next
    case (Cons u Us)
    then obtain Vs where Vs: "independent K Vs" "Span K Vs = Span K Us"
      by auto
    show ?case
    proof (cases "u ∈ Span K Us")
      case True
```

```
      hence "Span K (u # Us) = Span K Us"
        using Span_base_incl mono_Span_subset
        by (metis Cons.prems insert_subset list.simps(15) subset_antisym)
      thus ?thesis
        using Vs by blast
    next
      case False
      hence "Span K (u # Vs) = Span K (u # Us)" and "independent K (u
# Vs)"
        using li_Cons[of u K Vs] Cons(2) Vs by auto
      thus ?thesis
        by blast
    qed
  qed
  thus ?thesis
    using independent_in_carrier that by auto
qed

lemma dimension_backwards:
  "dimension (Suc n) K E ⟹ ∃v ∈ carrier R. ∃E'. dimension n K E' ∧
v ∉ E' ∧ E = line_extension K v E'"
  by (cases rule: dimension.cases) (auto)

lemma dimension_direct_sum_space:
  assumes "dimension n K E" and "dimension m K F" and "E ∩ F = { 0 }"
  shows "dimension (n + m) K (E <+>R F)"
proof -
  obtain Us Vs
    where Vs: "set Vs ⊆ carrier R" "independent K Vs" "length Vs = n"
"Span K Vs = E"
      and Us: "set Us ⊆ carrier R" "independent K Us" "length Us = m"
"Span K Us = F"
    using assms(1-2)[THEN exists_base] by auto
  hence "Span K (Vs @ Us) = E <+>R F"
    using Span_append_eq_set_add by auto
  moreover have "independent K (Vs @ Us)"
    using assms(3) independent_append[OF Vs(2) Us(2)] unfolding Vs(4)
Us(4) by simp
  ultimately show "dimension (n + m) K (E <+>R F)"
    using dimensionI[of "Vs @ Us"] Vs(3) Us(3) by auto
qed

lemma dimension_sum_space:
  assumes "dimension n K E" and "dimension m K F" and "dimension k K
(E ∩ F)"
  shows "dimension (n + m - k) K (E <+>R F)"
proof -
  obtain Bs
    where Bs: "set Bs ⊆ carrier R" "length Bs = k" "independent K Bs"
```

```
"Span K Bs = E ∩ F"
    using exists_base[OF assms(3)] by blast
  then obtain Us Vs
    where Us: "length (Us @ Bs) = n" "independent K (Us @ Bs)" "Span
K (Us @ Bs) = E"
      and Vs: "length (Vs @ Bs) = m" "independent K (Vs @ Bs)" "Span K
(Vs @ Bs) = F"
    using Span_base_incl[OF Bs(1)] assms(1-2)[THEN complete_base] by (metis
le_infE)
  hence in_carrier: "set Us ⊆ carrier R" "set (Vs @ Bs) ⊆ carrier R"
    using independent_in_carrier[OF Us(2)] independent_in_carrier[OF Vs(2)]
by auto
  hence "Span K Us ∩ (Span K (Vs @ Bs)) ⊆ Span K Bs"
    using Bs(4) Us(3) Vs(3) mono_Span_append(1)[OF _ Bs(1), of Us] by
auto
  hence "Span K Us ∩ (Span K (Vs @ Bs)) ⊆ { 0 }"
    using independent_split(3)[OF Us(2)] by blast
  hence "Span K Us ∩ (Span K (Vs @ Bs)) = { 0 }"
    using in_carrier[THEN Span_subgroup_props(2)] by auto

  hence dim: "dimension (n + m - k) K (Span K (Us @ (Vs @ Bs)))"
    using independent_append[OF independent_split(2)[OF Us(2)] Vs(2)]
Us(1) Vs(1) Bs(2)
          dimension_independent[of K "Us @ (Vs @ Bs)"] by auto

  have "(Span K Us) <+>_R F ⊆ E <+>_R F"
    using mono_Span_append(1)[OF in_carrier(1) Bs(1)] Us(3) unfolding
set_add_def' by auto
  moreover have "E <+>_R F ⊆ (Span K Us) <+>_R F"
  proof
    fix v assume "v ∈ E <+>_R F"
    then obtain u' v' where v: "u' ∈ E" "v' ∈ F" "v = u' ⊕ v'"
      unfolding set_add_def' by auto
    then obtain u1' u2' where u1': "u1' ∈ Span K Us" and u2': "u2' ∈
Span K Bs" and u': "u' = u1' ⊕ u2'"
      using Span_append_eq_set_add[OF in_carrier(1) Bs(1)] Us(3) unfold-
ing set_add_def' by blast

    have "v = u1' ⊕ (u2' ⊕ v')"
      using Span_subgroup_props(1)[OF Bs(1)] Span_subgroup_props(1)[OF
in_carrier(1)]
            space_subgroup_props(1)[OF assms(2)] u' v u1' u2' a_assoc[of
u1' u2' v'] by auto
    moreover have "u2' ⊕ v' ∈ F"
      using space_subgroup_props(3)[OF assms(2) _ v(2)] u2' Bs(4) by auto
    ultimately show "v ∈ (Span K Us) <+>_R F"
      using u1' unfolding set_add_def' by auto
  qed
  ultimately have "Span K (Us @ (Vs @ Bs)) = E <+>_R F"
```

```
    using Span_append_eq_set_add[OF in_carrier] Vs(3) by auto

  thus ?thesis using dim by simp
qed

end

end


lemma (in ring) telescopic_base_aux:
  assumes "subfield K R" "subfield F R"
    and "dimension n K F" and "dimension 1 F E"
  shows "dimension n K E"
proof -
  obtain Us u
    where Us: "set Us ⊆ carrier R" "length Us = n" "independent K Us"
"Span K Us = F"
      and u: "u ∈ carrier R" "independent F [u]" "Span F [u] = E"
    using exists_base[OF assms(2,4)] exists_base[OF assms(1,3)] independent_backwards(3)
assms(2)
    by (metis One_nat_def length_0_conv length_Suc_conv)
  have in_carrier: "set (map (λu'. u' ⊗ u) Us) ⊆ carrier R"
    using Us(1) u(1) by (induct Us) (auto)

  have li: "independent K (map (λu'. u' ⊗ u) Us)"
  proof (rule trivial_combine_imp_independent[OF assms(1) in_carrier])
    fix Ks assume Ks: "set Ks ⊆ K" and "combine Ks (map (λu'. u' ⊗ u)
Us) = 0"
    hence "(combine Ks Us) ⊗ u = 0"
      using combine_l_distr[OF _ Us(1) u(1)] subring_props(1)[OF assms(1)]
by auto
    hence "combine [ combine Ks Us ] [ u ] = 0"
      by simp
    moreover have "combine Ks Us ∈ F"
      using Us(4) Ks(1) Span_eq_combine_set[OF assms(1) Us(1)] by auto
    ultimately have "combine Ks Us = 0"
      using independent_imp_trivial_combine[OF assms(2) u(2), of "[ combine
Ks Us ]"] by auto
    hence "set (take (length Us) Ks) ⊆ { 0 }"
      using independent_imp_trivial_combine[OF assms(1) Us(3) Ks(1)] by
simp
    thus "set (take (length (map (λu'. u' ⊗ u) Us)) Ks) ⊆ { 0 }" by simp
  qed

  have "E ⊆ Span K (map (λu'. u' ⊗ u) Us)"
  proof
    fix v assume "v ∈ E"
    then obtain f where f: "f ∈ F" "v = f ⊗ u ⊕ 0"
```

```
      using u(1,3) line_extension_mem_iff by auto
    then obtain Ks where Ks: "set Ks ⊆ K" "f = combine Ks Us"
      using Span_eq_combine_set[OF assms(1) Us(1)] Us(4) by auto
    have "v = f ⊗ u"
      using subring_props(1)[OF assms(2)] f u(1) by auto
    hence "v = combine Ks (map (λu'. u' ⊗ u) Us)"
      using combine_l_distr[OF _ Us(1) u(1), of Ks] Ks(1-2)
            subring_props(1)[OF assms(1)] by blast
    thus "v ∈ Span K (map (λu'. u' ⊗ u) Us)"
      unfolding Span_eq_combine_set[OF assms(1) in_carrier] using Ks(1)
by blast
  qed
  moreover have "Span K (map (λu'. u' ⊗ u) Us) ⊆ E"
  proof
    fix v assume "v ∈ Span K (map (λu'. u' ⊗ u) Us)"
    then obtain Ks where Ks: "set Ks ⊆ K" "v = combine Ks (map (λu'.
u' ⊗ u) Us)"
      unfolding Span_eq_combine_set[OF assms(1) in_carrier] by blast
    hence "v = (combine Ks Us) ⊗ u"
      using combine_l_distr[OF _ Us(1) u(1), of Ks] subring_props(1)[OF
assms(1)] by auto
    moreover have "combine Ks Us ∈ F"
      using Us(4) Span_eq_combine_set[OF assms(1) Us(1)] Ks(1) by blast
    ultimately have "v = (combine Ks Us) ⊗ u ⊕ 0" and "combine Ks Us
∈ F"
      using subring_props(1)[OF assms(2)] u(1) by auto
    thus "v ∈ E"
      using u(3) line_extension_mem_iff by auto
  qed
  ultimately have "Span K (map (λu'. u' ⊗ u) Us) = E" by auto
  thus ?thesis
    using dimensionI[OF assms(1) li] Us(2) by simp
qed

lemma (in ring) telescopic_base:
  assumes "subfield K R" "subfield F R"
    and "dimension n K F" and "dimension m F E"
  shows "dimension (n * m) K E"
  using assms(4)
proof (induct m arbitrary: E)
  case 0 thus ?case
    using dimension_zero[OF assms(2)] zero_dim by auto
next
  case (Suc m)
  obtain Vs
    where Vs: "set Vs ⊆ carrier R" "length Vs = Suc m" "independent F
Vs" "Span F Vs = E"
    using exists_base[OF assms(2) Suc(2)] by blast
  then obtain v Vs' where v: "Vs = v # Vs'"
```

```
    by (meson length_Suc_conv)
  hence li: "independent F [ v ]" "independent F Vs'" and inter: "Span
F [ v ] ∩ Span F Vs' = { 0 }"
    using Vs(3) independent_split[OF assms(2), of "[ v ]" Vs'] by auto
  have "dimension n K (Span F [ v ])"
    using dimension_independent[OF li(1)] telescopic_base_aux[OF assms(1-3)]
by simp
  moreover have "dimension (n * m) K (Span F Vs')"
    using Suc(1) dimension_independent[OF li(2)] Vs(2) unfolding v by
auto
  ultimately have "dimension (n * Suc m) K (Span F [ v ] <+>R Span F Vs')"
    using dimension_direct_sum_space[OF assms(1) _ _ inter] by auto
  thus "dimension (n * Suc m) K E"
    using Span_append_eq_set_add[OF assms(2) li[THEN independent_in_carrier]]
Vs(4) v by auto
qed


context ring_hom_ring
begin

lemma combine_hom:
  "⟦ set Ks ⊆ carrier R; set Us ⊆ carrier R ⟧ ⟹ combine (map h Ks) (map
h Us) = h (R.combine Ks Us)"
  by (induct Ks Us rule: R.combine.induct) (auto)

lemma line_extension_hom:
  assumes "K ⊆ carrier R" "a ∈ carrier R" "E ⊆ carrier R"
  shows "line_extension (h ` K) (h a) (h ` E) = h ` R.line_extension K
a E"
  using set_add_hom[OF homh R.r_coset_subset_G[OF assms(1-2)] assms(3)]
        coset_hom(2)[OF ring_hom_in_hom(1)[OF homh] assms(1-2)]
  unfolding R.line_extension_def S.line_extension_def
  by simp

lemma Span_hom:
  assumes "K ⊆ carrier R" "set Us ⊆ carrier R"
  shows "Span (h ` K) (map h Us) = h ` R.Span K Us"
  using assms line_extension_hom R.Span_in_carrier by (induct Us) (auto)

lemma inj_on_subgroup_iff_trivial_ker:
  assumes "subgroup H (add_monoid R)"
  shows "inj_on h H ⟷ a_kernel (R ⦇ carrier := H ⦈) S h = { 0 }"
  using group_hom.inj_on_subgroup_iff_trivial_ker[OF a_group_hom assms]
  unfolding a_kernel_def[of "R ⦇ carrier := H ⦈" S h] by simp

corollary inj_on_Span_iff_trivial_ker:
  assumes "subfield K R" "set Us ⊆ carrier R"
  shows "inj_on h (R.Span K Us) ⟷ a_kernel (R ⦇ carrier := R.Span K
```

```
Us |)) S h = { 0 }"
  using inj_on_subgroup_iff_trivial_ker[OF R.Span_is_add_subgroup[OF assms]]
.


context
  fixes K :: "'a set" assumes K: "subfield K R" and one_zero: "1S ≠ 0S"
begin

lemma inj_hom_preserves_independent:
  assumes "inj_on h (R.Span K Us)"
  and "R.independent K Us" shows "independent (h ` K) (map h Us)"
proof (rule ccontr)
  have in_carrier: "set Us ⊆ carrier R" "set (map h Us) ⊆ carrier S"
    using R.independent_in_carrier[OF assms(2)] by auto

  assume ld: "dependent (h ` K) (map h Us)"
  obtain Ks :: "'c list"
    where Ks: "length Ks = length Us" "combine Ks (map h Us) = 0S" "set
Ks ⊆ h ` K" "set Ks ≠ { 0S }"
    using dependent_imp_non_trivial_combine[OF img_is_subfield(2)[OF K
one_zero] in_carrier(2) ld]
    by (metis length_map)
  obtain Ks' where Ks': "set Ks' ⊆ K" "Ks = map h Ks'"
    using Ks(3) by (induct Ks) (auto, metis insert_subset list.simps(15,9))
  hence "h (R.combine Ks' Us) = 0S"
    using combine_hom[OF _ in_carrier(1)] Ks(2) subfieldE(3)[OF K] by
(metis subset_trans)
  moreover have "R.combine Ks' Us ∈ R.Span K Us"
    using R.Span_eq_combine_set[OF K in_carrier(1)] Ks'(1) by auto
  ultimately have "R.combine Ks' Us = 0"
    using assms hom_zero R.Span_subgroup_props(2)[OF K in_carrier(1)]
by (auto simp add: inj_on_def)
  hence "set Ks' ⊆ { 0 }"
    using R.independent_imp_trivial_combine[OF K assms(2)] Ks' Ks(1)
    by (metis length_map order_refl take_all)
  hence "set Ks ⊆ { 0S }"
    unfolding Ks' using hom_zero by (induct Ks') (auto)
  hence "Ks = []"
    using Ks(4) by (metis set_empty2 subset_singletonD)
  hence "independent (h ` K) (map h Us)"
    using independent.li_Nil Ks(1) by simp
  from ‹dependent (h ` K) (map h Us)› and this show False by simp
qed

corollary inj_hom_dimension:
  assumes "inj_on h E"
  and "R.dimension n K E" shows "dimension n (h ` K) (h ` E)"
proof -
```

```
  obtain Us
    where Us: "set Us ⊆ carrier R" "R.independent K Us" "length Us =
n" "R.Span K Us = E"
    using R.exists_base[OF K assms(2)] by blast
  hence "dimension n (h ‘ K) (Span (h ‘ K) (map h Us))"
    using dimension_independent[OF inj_hom_preserves_independent[OF _
Us(2)]] assms(1) by auto
  thus ?thesis
    using Span_hom[OF subfieldE(3)[OF K] Us(1)] Us(4) by simp
qed

corollary rank_nullity_theorem:
  assumes "R.dimension n K E" and "R.dimension m K (a_kernel (R ⦇ carrier
:= E ⦈) S h)"
  shows "dimension (n - m) (h ‘ K) (h ‘ E)"
proof -
  obtain Us
    where Us: "set Us ⊆ carrier R" "R.independent K Us" "length Us =
m"
              "R.Span K Us = a_kernel (R ⦇ carrier := E ⦈) S h"
    using R.exists_base[OF K assms(2)] by blast
  obtain Vs
    where Vs: "R.independent K (Vs @ Us)" "length (Vs @ Us) = n" "R.Span
K (Vs @ Us) = E"
    using R.complete_base[OF K assms(1) Us(2)] R.Span_base_incl[OF K Us(1)]
Us(4)
    unfolding a_kernel_def' by auto
  have set_Vs: "set Vs ⊆ carrier R"
    using R.independent_in_carrier[OF Vs(1)] by auto
  have "R.Span K Vs ∩ a_kernel (R ⦇ carrier := E ⦈) S h = { 0 }"
    using R.independent_split[OF K Vs(1)] Us(4) by simp
  moreover have "R.Span K Vs ⊆ E"
    using R.mono_Span_append(1)[OF K set_Vs Us(1)] Vs(3) by auto
  ultimately have "a_kernel (R ⦇ carrier := R.Span K Vs ⦈) S h ⊆ { 0 }"
    unfolding a_kernel_def' by (simp del: R.Span.simps, blast)
  hence "a_kernel (R ⦇ carrier := R.Span K Vs ⦈) S h = { 0 }"
    using R.Span_subgroup_props(2)[OF K set_Vs]
    unfolding a_kernel_def' by (auto simp del: R.Span.simps)
  hence "inj_on h (R.Span K Vs)"
    using inj_on_Span_iff_trivial_ker[OF K set_Vs] by simp
  moreover have "R.dimension (n - m) K (R.Span K Vs)"
    using R.dimension_independent[OF R.independent_split(2)[OF K Vs(1)]]
Vs(2) Us(3) by auto
  ultimately have "dimension (n - m) (h ‘ K) (h ‘ (R.Span K Vs))"
    using assms(1) inj_hom_dimension by simp

  have "h ‘ E = h ‘ (R.Span K Vs <+>R R.Span K Us)"
    using R.Span_append_eq_set_add[OF K set_Vs Us(1)] Vs(3) by simp
  hence "h ‘ E = h ‘ (R.Span K Vs) <+>S h ‘ (R.Span K Us)"
```

```
    using R.Span_subgroup_props(1)[OF K] set_Vs Us(1) set_add_hom[OF homh]
by auto
  moreover have "h ‘ (R.Span K Us) = { 0S }"
    using R.space_subgroup_props(2)[OF K assms(1)] unfolding Us(4) a_kernel_def'
by force
  ultimately have "h ‘ E = h ‘ (R.Span K Vs) <+>S { 0S }"
    by simp
  hence "h ‘ E = h ‘ (R.Span K Vs)"
    using R.Span_subgroup_props(1-2)[OF K set_Vs] unfolding set_add_def'
by force

  from ‹dimension (n - m) (h ‘ K) (h ‘ (R.Span K Vs))› and this show
?thesis by simp
qed

end

end

lemma (in ring_hom_ring)
  assumes "subfield K R" and "set Us ⊆ carrier R" and "1S ≠ 0S"
    and "independent (h ‘ K) (map h Us)" shows "R.independent K Us"
proof (rule ccontr)
  assume "R.dependent K Us"
  then obtain Ks
    where "length Ks = length Us" and "R.combine Ks Us = 0" and "set
Ks ⊆ K" and "set Ks ≠ { 0 }"
    using R.dependent_imp_non_trivial_combine[OF assms(1-2)] by metis
  hence "combine (map h Ks) (map h Us) = 0S"
    using combine_hom[OF _ assms(2), of Ks] subfieldE(3)[OF assms(1)]
by simp
  moreover from ‹set Ks ⊆ K› have "set (map h Ks) ⊆ h ‘ K"
    by (induction Ks) (auto)
  moreover have "¬ set (map h Ks) ⊆ { h 0 }"
  proof (rule ccontr)
    assume "¬ ¬ set (map h Ks) ⊆ { h 0 }" then have "set (map h Ks)
⊆ { h 0 }"
      by simp
    moreover from ‹R.dependent K Us› and ‹length Ks = length Us› have
"Ks ≠ []"
      by auto
    ultimately have "set (map h Ks) = { h 0 }"
      using subset_singletonD by fastforce
    with ‹set Ks ⊆ K› have "set Ks = { 0 }"
      using inj_onD[OF _ _ _ subringE(2)[OF subfieldE(1)[OF assms(1)]],
of h]
            img_is_subfield(1)[OF assms(1,3)] subset_singletonD
      by (induction Ks) (auto simp add: subset_singletonD, fastforce)
    with ‹set Ks ≠ { 0 }› show False
```

```
    by simp
  qed
  with ‹length Ks = length Us› have "¬ set (take (length (map h Us))
(map h Ks)) ⊆ { h 𝟬 }"
    by auto
  ultimately have "dependent (h ‘ K) (map h Us)"
    using non_trivial_combine_imp_dependent[OF img_is_subfield(2)[OF assms(1,3)],
of "map h Ks"] by simp
  with ‹independent (h ‘ K) (map h Us)› show False
    by simp
qed
```

## 34.9 Finite Dimension

```
definition (in ring) finite_dimension :: "'a set ⇒ 'a set ⇒ bool"
  where "finite_dimension K E ⟷ (∃n. dimension n K E)"

abbreviation (in ring) infinite_dimension :: "'a set ⇒ 'a set ⇒ bool"
  where "infinite_dimension K E ≡ ¬ finite_dimension K E"

definition (in ring) dim :: "'a set ⇒ 'a set ⇒ nat"
  where "dim K E = (THE n. dimension n K E)"

locale subalgebra = subgroup V "add_monoid R" for K and V and R (structure)
+
  assumes smult_closed: "⟦ k ∈ K; v ∈ V ⟧ ⟹ k ⊗ v ∈ V"
```

### 34.9.1 Basic Properties

```
lemma (in ring) unique_dimension:
  assumes "subfield K R" and "finite_dimension K E" shows "∃!n. dimension
n K E"
  using assms(2) dimension_is_inj[OF assms(1)] unfolding finite_dimension_def
by auto

lemma (in ring) finite_dimensionI:
  assumes "dimension n K E" shows "finite_dimension K E"
  using assms unfolding finite_dimension_def by auto

lemma (in ring) finite_dimensionE:
  assumes "subfield K R" and "finite_dimension K E" shows "dimension
((dim over K) E) K E"
  using theI'[OF unique_dimension[OF assms]] unfolding over_def dim_def
by simp

lemma (in ring) dimI:
  assumes "subfield K R" and "dimension n K E" shows "(dim over K) E
= n"
  using finite_dimensionE[OF assms(1) finite_dimensionI] dimension_is_inj[OF
assms(1)] assms(2)
```

```
  unfolding over_def dim_def by auto

lemma (in ring) finite_dimensionE' [elim]:
  assumes "finite_dimension K E" and "⋀n. dimension n K E ⟹ P" shows
P
  using assms unfolding finite_dimension_def by auto

lemma (in ring) Span_finite_dimension:
  assumes "subfield K R" and "set Us ⊆ carrier R"
  shows "finite_dimension K (Span K Us)"
  using filter_base[OF assms] finite_dimensionI[OF dimension_independent[of
K]] by metis

lemma (in ring) carrier_is_subalgebra:
  assumes "K ⊆ carrier R" shows "subalgebra K (carrier R) R"
  using assms subalgebra.intro[OF add.group_incl_imp_subgroup[of "carrier
R"], of K] add.group_axioms
  unfolding subalgebra_axioms_def by auto

lemma (in ring) subalgebra_in_carrier:
  assumes "subalgebra K V R" shows "V ⊆ carrier R"
  using subgroup.subset[OF subalgebra.axioms(1)[OF assms]] by simp

lemma (in ring) subalgebra_inter:
  assumes "subalgebra K V R" and "subalgebra K V' R" shows "subalgebra
K (V ∩ V') R"
  using add.subgroups_Inter_pair assms unfolding subalgebra_def subalgebra_axioms_def
by auto

lemma (in ring_hom_ring) img_is_subalgebra:
  assumes "K ⊆ carrier R" and "subalgebra K V R" shows "subalgebra (h
` K) (h ` V) S"
proof (intro subalgebra.intro)
  have "group_hom (add_monoid R) (add_monoid S) h"
    using ring_hom_in_hom(2)[OF homh] R.add.group_axioms add.group_axioms
    unfolding group_hom_def group_hom_axioms_def by auto
  thus "subgroup (h ` V) (add_monoid S)"
    using group_hom.subgroup_img_is_subgroup[OF _ subalgebra.axioms(1)[OF
assms(2)]] by force
next
  show "subalgebra_axioms (h ` K) (h ` V) S"
    using R.subalgebra_in_carrier[OF assms(2)] subalgebra.axioms(2)[OF
assms(2)] assms(1)
    unfolding subalgebra_axioms_def
    by (auto, metis hom_mult image_eqI subset_iff)
qed

lemma (in ring) ideal_is_subalgebra:
  assumes "K ⊆ carrier R" "ideal I R" shows "subalgebra K I R"
```

```
  using ideal.axioms(1)[OF assms(2)] ideal.I_l_closed[OF assms(2)] assms(1)
  unfolding subalgebra_def subalgebra_axioms_def additive_subgroup_def
by auto

lemma (in ring) Span_is_subalgebra:
  assumes "subfield K R" "set Us ⊆ carrier R" shows "subalgebra K (Span
K Us) R"
  using Span_smult_closed[OF assms] Span_is_add_subgroup[OF assms]
  unfolding subalgebra_def subalgebra_axioms_def by auto

lemma (in ring) finite_dimension_imp_subalgebra:
  assumes "subfield K R" "finite_dimension K E" shows "subalgebra K E
R"
  using exists_base[OF assms(1) finite_dimensionE[OF assms]] Span_is_subalgebra[OF
assms(1)] by auto

lemma (in ring) subalgebra_Span_incl:
  assumes "subfield K R" and "subalgebra K V R" "set Us ⊆ V" shows "Span
K Us ⊆ V"
proof -
  have "K <#> (set Us) ⊆ V"
    using subalgebra.smult_closed[OF assms(2)] assms(3) unfolding set_mult_def
by blast
  moreover have "set Us ⊆ carrier R"
    using subalgebra_in_carrier[OF assms(2)] assms(3) by auto
  ultimately show ?thesis
    using subalgebra.axioms(1)[OF assms(2)] Span_min[OF assms(1)] by blast
qed

lemma (in ring) Span_subalgebra_minimal:
  assumes "subfield K R" "set Us ⊆ carrier R"
  shows "Span K Us = ⋂ { V. subalgebra K V R ∧ set Us ⊆ V }"
  using Span_is_subalgebra[OF assms] Span_base_incl[OF assms] subalgebra_Span_incl[OF
assms(1)]
  by blast

lemma (in ring) Span_subalgebraI:
  assumes "subfield K R"
    and "subalgebra K E R" "set Us ⊆ E"
    and "⋀V. ⟦ subalgebra K V R; set Us ⊆ V ⟧ ⟹ E ⊆ V"
  shows "E = Span K Us"
proof -
  have "⋂ { V. subalgebra K V R ∧ set Us ⊆ V } = E"
    using assms(2-4) by auto
  thus "E = Span K Us"
    using Span_subalgebra_minimal subalgebra_in_carrier[of K E] assms
by auto
qed
```

```
lemma (in ring) subalbegra_incl_imp_finite_dimension:
  assumes "subfield K R" and "finite_dimension K E"
  and "subalgebra K V R" "V ⊆ E" shows "finite_dimension K V"
proof -
  obtain n where n: "dimension n K E"
    using assms(2) by auto

  define S where "S = { Us. set Us ⊆ V ∧ independent K Us }"
  have "length ‘ S ⊆ {..n}"
    unfolding S_def using independent_length_le_dimension[OF assms(1)
n] assms(4) by auto
  moreover have "[] ∈ S"
    unfolding S_def by simp
  hence "length ‘ S ≠ {}" by blast
  ultimately obtain m where m: "m ∈ length ‘ S" and greatest: "⋀k. k
∈ length ‘ S ⟹ k ≤ m"
    by (meson Max_ge Max_in finite_atMost rev_finite_subset)
  then obtain Us where Us: "set Us ⊆ V" "independent K Us" "m = length
Us"
      unfolding S_def by auto
  have "Span K Us = V"
  proof (rule ccontr)
    assume "¬ Span K Us = V" then have "Span K Us ⊂ V"
      using subalgebra_Span_incl[OF assms(1,3) Us(1)] by blast
    then obtain v where v:"v ∈ V" "v ∉ Span K Us"
      by blast
    hence "independent K (v # Us)"
      using independent.li_Cons[OF _ _ Us(2)] subalgebra_in_carrier[OF
assms(3)] by auto
    hence "(v # Us) ∈ S"
      unfolding S_def using Us(1) v(1) by auto
    hence "length (v # Us) ≤ m"
      using greatest by blast
    moreover have "length (v # Us) = Suc m"
      using Us(3) by auto
    ultimately show False by simp
  qed
  thus ?thesis
    using finite_dimensionI[OF dimension_independent[OF Us(2)]] by simp
qed

lemma (in ring_hom_ring) infinite_dimension_hom:
  assumes "subfield K R" and "1⇘S⇙ ≠ 0⇘S⇙" and "inj_on h E" and "subalgebra
K E R"
  shows "R.infinite_dimension K E ⟹ infinite_dimension (h ‘ K) (h ‘
E)"
proof -
  note subfield = img_is_subfield(2)[OF assms(1-2)]
```

```
  assume "R.infinite_dimension K E"
  show "infinite_dimension (h ‘ K) (h ‘ E)"
  proof (rule ccontr)
    assume "¬ infinite_dimension (h ‘ K) (h ‘ E)"
    then obtain Vs where "set Vs ⊆ carrier S" and "Span (h ‘ K) Vs =
h ‘ E"
      using exists_base[OF subfield] by blast
    hence "set Vs ⊆ h ‘ E"
      using Span_base_incl[OF subfield] by blast
    hence "∃Us. set Us ⊆ E ∧ Vs = map h Us"
      by (induct Vs) (auto, metis insert_subset list.simps(9,15))
    then obtain Us where "set Us ⊆ E" and "Vs = map h Us"
      by blast
    with ‹Span (h ‘ K) Vs = h ‘ E› have "h ‘ (R.Span K Us) = h ‘ E"
      using R.subalgebra_in_carrier[OF assms(4)] Span_hom assms(1) by
auto
    moreover from ‹set Us ⊆ E› have "R.Span K Us ⊆ E"
      using R.subalgebra_Span_incl assms(1-4) by blast
    ultimately have "R.Span K Us = E"
    proof (auto simp del: R.Span.simps)
      fix a assume "a ∈ E"
      with ‹h ‘ (R.Span K Us) = h ‘ E› obtain b where "b ∈ R.Span K
Us" and "h a = h b"
        by auto
      with ‹R.Span K Us ⊆ E› and ‹a ∈ E› have "a = b"
        using inj_onD[OF assms(3)] by auto
      with ‹b ∈ R.Span K Us› show "a ∈ R.Span K Us"
        by simp
    qed
    with ‹set Us ⊆ E› have "R.finite_dimension K E"
      using R.Span_finite_dimension[OF assms(1)] R.subalgebra_in_carrier[OF
assms(4)] by auto
    with ‹R.infinite_dimension K E› show False
      by simp
  qed
qed
```

### 34.9.2 Reformulation of some lemmas in this new language.

```
lemma (in ring) sum_space_dim:
  assumes "subfield K R" "finite_dimension K E" "finite_dimension K F"
  shows "finite_dimension K (E <+>R F)"
    and "((dim over K) (E <+>R F)) = ((dim over K) E) + ((dim over K)
F) - ((dim over K) (E ∩ F))"
proof -
  obtain n m k where n: "dimension n K E" and m: "dimension m K F" and
k: "dimension k K (E ∩ F)"
    using assms(2-3) subalbegra_incl_imp_finite_dimension[OF assms(1-2)
          subalgebra_inter[OF assms(2-3)[THEN finite_dimension_imp_subalgebra[OF
```

```
assms(1)]]]]
    by (meson inf_le1 finite_dimension_def)
  hence "dimension (n + m - k) K (E <+>R F)"
    using dimension_sum_space[OF assms(1)] by simp
  thus "finite_dimension K (E <+>R F)"
   and "((dim over K) (E <+>R F)) = ((dim over K) E) + ((dim over K) F)
- ((dim over K) (E ∩ F))"
    using finite_dimensionI dimI[OF assms(1)] n m k by auto
qed

lemma (in ring) telescopic_base_dim:
  assumes "subfield K R" "subfield F R" and "finite_dimension K F" and
"finite_dimension F E"
  shows "finite_dimension K E" and "(dim over K) E = ((dim over K) F)
* ((dim over F) E)"
  using telescopic_base[OF assms(1-2)
        finite_dimensionE[OF assms(1,3)]
        finite_dimensionE[OF assms(2,4)]]
        dimI[OF assms(1)] finite_dimensionI
  by auto

end


theory Polynomial_Divisibility
  imports Polynomials Embedded_Algebras "HOL-Library.Multiset"

begin
```

# 35 Divisibility of Polynomials

## 35.1 Definitions

```
abbreviation poly_ring :: "_ ⇒ ('a  list) ring"
  where "poly_ring R ≡ univ_poly R (carrier R)"

abbreviation pirreducible :: "_ ⇒ 'a set ⇒ 'a list ⇒ bool" (‹pirreducibleı›)
  where "pirreducibleR K p ≡ ring_irreducible(univ_poly R K) p"

abbreviation pprime :: "_ ⇒ 'a set ⇒ 'a list ⇒ bool" (‹pprimeı›)
  where "ppprimeR K p ≡ ring_prime(univ_poly R K) p"

definition pdivides :: "_ ⇒ 'a list ⇒ 'a list ⇒ bool" (infix ‹pdividesı›
65)
  where "p pdividesR q = p divides(univ_poly R (carrier R)) q"

definition rupture :: "_ ⇒ 'a set ⇒ 'a list ⇒ (('a list) set) ring"
(‹Ruptı›)
  where "RuptR K p = (K[X]R) Quot (PIdlK[X]R p)"
```

```
abbreviation (in ring) rupture_surj :: "'a set ⇒ 'a list ⇒ 'a list ⇒
('a list) set"
  where "rupture_surj K p ≡ (λq. (PIdl_K[X] p) +>_K[X] q)"
```

## 35.2 Basic Properties

```
lemma (in ring) carrier_polynomial_shell [intro]:
  assumes "subring K R" and "p ∈ carrier (K[X])" shows "p ∈ carrier
(poly_ring R)"
  using carrier_polynomial[OF assms(1), of p] assms(2) unfolding sym[OF
univ_poly_carrier] by simp

lemma (in domain) pdivides_zero:
  assumes "subring K R" and "p ∈ carrier (K[X])" shows "p pdivides []"
  using ring.divides_zero[OF univ_poly_is_ring[OF carrier_is_subring]
        carrier_polynomial_shell[OF assms]]
  unfolding univ_poly_zero pdivides_def .

lemma (in domain) zero_pdivides_zero: "[] pdivides []"
  using pdivides_zero[OF carrier_is_subring] univ_poly_carrier by blast

lemma (in domain) zero_pdivides:
  shows "[] pdivides p ⟷ p = []"
  using ring.zero_divides[OF univ_poly_is_ring[OF carrier_is_subring]]
  unfolding univ_poly_zero pdivides_def .

lemma (in domain) pprime_iff_pirreducible:
  assumes "subfield K R" and "p ∈ carrier (K[X])"
  shows "pprime K p ⟷ pirreducible K p"
  using principal_domain.primeness_condition[OF univ_poly_is_principal]
assms by simp

lemma (in domain) pirreducibleE:
  assumes "subring K R" "p ∈ carrier (K[X])" "pirreducible K p"
  shows "p ≠ []" "p ∉ Units (K[X])"
    and "⋀q r. ⟦ q ∈ carrier (K[X]); r ∈ carrier (K[X])⟧ ⟹
                 p = q ⊗_K[X] r ⟹ q ∈ Units (K[X]) ∨ r ∈ Units (K[X])"
  using domain.ring_irreducibleE[OF univ_poly_is_domain[OF assms(1)] _
assms(3)] assms(2)
  by (auto simp add: univ_poly_zero)

lemma (in domain) pirreducibleI:
  assumes "subring K R" "p ∈ carrier (K[X])" "p ≠ []" "p ∉ Units (K[X])"
    and "⋀q r. ⟦ q ∈ carrier (K[X]); r ∈ carrier (K[X])⟧ ⟹
                 p = q ⊗_K[X] r ⟹ q ∈ Units (K[X]) ∨ r ∈ Units (K[X])"
  shows "pirreducible K p"
  using domain.ring_irreducibleI[OF univ_poly_is_domain[OF assms(1)] _
assms(4)] assms(2-3,5)
```

```
  by (auto simp add: univ_poly_zero)

lemma (in domain) univ_poly_carrier_units_incl:
  shows "Units ((carrier R) [X]) ⊆ { [ k ] | k. k ∈ carrier R - { 0 }
}"
proof
  fix p assume "p ∈ Units ((carrier R) [X])"
  then obtain q
    where p: "polynomial (carrier R) p" and q: "polynomial (carrier R)
q" and pq: "poly_mult p q = [ 1 ]"
    unfolding Units_def univ_poly_def by auto
  hence not_nil: "p ≠ []" and "q ≠ []"
    using poly_mult_integral[OF carrier_is_subring p q] poly_mult_zero[OF
polynomial_incl[OF p]] by auto
  hence "degree p = 0"
    using poly_mult_degree_eq[OF carrier_is_subring p q] unfolding pq
by simp
  hence "length p = 1"
    using not_nil by (metis One_nat_def Suc_pred length_greater_0_conv)
  then obtain k where k: "p = [ k ]"
    by (metis One_nat_def length_0_conv length_Suc_conv)
  hence "k ∈ carrier R - { 0 }"
    using p unfolding polynomial_def by auto
  thus "p ∈ { [ k ] | k. k ∈ carrier R - { 0 } }"
    unfolding k by blast
qed

lemma (in field) univ_poly_carrier_units:
  "Units ((carrier R) [X]) = { [ k ] | k. k ∈ carrier R - { 0 } }"
proof
  show "Units ((carrier R) [X]) ⊆ { [ k ] | k. k ∈ carrier R - { 0 }
}"
    using univ_poly_carrier_units_incl by simp
next
  show "{ [ k ] | k. k ∈ carrier R - { 0 } } ⊆ Units ((carrier R) [X])"
  proof (auto)
    fix k assume k: "k ∈ carrier R" "k ≠ 0"
    hence inv_k: "inv k ∈ carrier R" "inv k ≠ 0" and "k ⊗ inv k = 1"
"inv k ⊗ k = 1"
      using subfield_m_inv[OF carrier_is_subfield, of k] by auto
    hence "poly_mult [ k ] [ inv k ] = [ 1 ]" and "poly_mult [ inv k
] [ k ] = [ 1 ]"
      by (auto simp add: k)
    moreover have "polynomial (carrier R) [ k ]" and "polynomial (carrier
R) [ inv k ]"
      using const_is_polynomial k inv_k by auto
    ultimately show "[ k ] ∈ Units ((carrier R) [X])"
      unfolding Units_def univ_poly_def by (auto simp del: poly_mult.simps)
  qed
```

```
qed

lemma (in domain) univ_poly_units_incl:
  assumes "subring K R" shows "Units (K[X]) ⊆ { [ k ] | k. k ∈ K - {
0 } }"
  using domain.univ_poly_carrier_units_incl[OF subring_is_domain[OF assms]]
        univ_poly_consistent[OF assms] by auto

lemma (in ring) univ_poly_units:
  assumes "subfield K R" shows "Units (K[X]) = { [ k ] | k. k ∈ K - {
0 } }"
  using field.univ_poly_carrier_units[OF subfield_iff(2)[OF assms]]
        univ_poly_consistent[OF subfieldE(1)[OF assms]] by auto

lemma (in domain) univ_poly_units':
  assumes "subfield K R" shows "p ∈ Units (K[X]) ⟷ p ∈ carrier (K[X])
∧ p ≠ [] ∧ degree p = 0"
  unfolding univ_poly_units[OF assms] sym[OF univ_poly_carrier] polynomial_def
  by (auto, metis hd_in_set le_0_eq le_Suc_eq length_0_conv length_Suc_conv
list.sel(1) subsetD)

corollary (in domain) rupture_one_not_zero:
  assumes "subfield K R" and "p ∈ carrier (K[X])" and "degree p > 0"
  shows "1_{Rupt K p} ≠ 0_{Rupt K p}"
proof (rule ccontr)
  interpret UP: principal_domain "K[X]"
    using univ_poly_is_principal[OF assms(1)] .

  assume "¬ 1_{Rupt K p} ≠ 0_{Rupt K p}"
  then have "PIdl_{K[X]} p +>_{K[X]} 1_{K[X]} = PIdl_{K[X]} p"
    unfolding rupture_def FactRing_def by simp
  hence "1_{K[X]} ∈ PIdl_{K[X]} p"
    using ideal.rcos_const_imp_mem[OF UP.cgenideal_ideal[OF assms(2)]]
by auto
  then obtain q where "q ∈ carrier (K[X])" and "1_{K[X]} = q ⊗_{K[X]} p"
    using assms(2) unfolding cgenideal_def by auto
  hence "p ∈ Units (K[X])"
    unfolding Units_def using assms(2) UP.m_comm by auto
  hence "degree p = 0"
    unfolding univ_poly_units[OF assms(1)] by auto
  with ‹degree p > 0› show False
    by simp
qed

corollary (in ring) pirreducible_degree:
  assumes "subfield K R" "p ∈ carrier (K[X])" "pirreducible K p"
  shows "degree p ≥ 1"
proof (rule ccontr)
  assume "¬ degree p ≥ 1" then have "length p ≤ 1"
```

```
    by simp
  moreover have "p ≠ []" and "p ∉ Units (K[X])"
    using assms(3) by (auto simp add: ring_irreducible_def irreducible_def
univ_poly_zero)
  ultimately obtain k where k: "p = [ k ]"
    by (metis append_butlast_last_id butlast_take diff_is_0_eq le_refl
self_append_conv2 take0 take_all)
  hence "k ∈ K" and "k ≠ 0"
    using assms(2) by (auto simp add: polynomial_def univ_poly_def)
  hence "p ∈ Units (K[X])"
    using univ_poly_units[OF assms(1)] unfolding k by auto
  from ‹p ∈ Units (K[X])› and ‹p ∉ Units (K[X])› show False by simp
qed

corollary (in domain) univ_poly_not_field:
  assumes "subring K R" shows "¬ field (K[X])"
proof -
  have "X ∈ carrier (K[X]) - { 0(K[X]) }" and "X ∉ { [ k ] | k. k ∈ K
- { 0 } }"
    using var_closed(1)[OF assms] unfolding univ_poly_zero var_def by
auto
  thus ?thesis
    using field.field_Units[of "K[X]"] univ_poly_units_incl[OF assms]
by blast
qed

lemma (in domain) rupture_is_field_iff_pirreducible:
  assumes "subfield K R" and "p ∈ carrier (K[X])"
  shows "field (Rupt K p) ⟷ pirreducible K p"
proof
  assume "pirreducible K p" thus "field (Rupt K p)"
    using principal_domain.field_iff_prime[OF univ_poly_is_principal[OF
assms(1)]] assms(2)
          pprime_iff_pirreducible[OF assms] pirreducibleE(1)[OF subfieldE(1)[OF
assms(1)]]
    by (simp add: univ_poly_zero rupture_def)
next
  interpret UP: principal_domain "K[X]"
    using univ_poly_is_principal[OF assms(1)] .

  assume field: "field (Rupt K p)"
  have "p ≠ []"
  proof (rule ccontr)
    assume "¬ p ≠ []" then have p: "p = []"
      by simp
    hence "Rupt K p ≃ (K[X])"
      using UP.FactRing_zeroideal(1) UP.genideal_zero
            UP.cgenideal_eq_genideal[OF UP.zero_closed]
      by (simp add: rupture_def univ_poly_zero)
```

```
    then obtain h where h: "h ∈ ring_iso (Rupt K p) (K[X])"
      unfolding is_ring_iso_def by blast
    moreover have "ring (Rupt K p)"
      using field by (simp add: cring_def domain_def field_def)
    ultimately interpret R: ring_hom_ring "Rupt K p" "K[X]" h
      unfolding ring_hom_ring_def ring_hom_ring_axioms_def ring_iso_def
      using UP.ring_axioms by simp
    have "field (K[X])"
      using field.ring_iso_imp_img_field[OF field h] by simp
    thus False
      using univ_poly_not_field[OF subfieldE(1)[OF assms(1)]] by simp
  qed
  thus "pirreducible K p"
    using UP.field_iff_prime pprime_iff_pirreducible[OF assms] assms(2)
field
    by (simp add: univ_poly_zero rupture_def)
qed

lemma (in domain) rupture_surj_hom:
  assumes "subring K R" and "p ∈ carrier (K[X])"
  shows "(rupture_surj K p) ∈ ring_hom (K[X]) (Rupt K p)"
    and "ring_hom_ring (K[X]) (Rupt K p) (rupture_surj K p)"
proof -
  interpret UP: domain "K[X]"
    using univ_poly_is_domain[OF assms(1)] .
  interpret I: ideal "PIdl_K[X] p" "K[X]"
    using UP.cgenideal_ideal[OF assms(2)] .
  show "(rupture_surj K p) ∈ ring_hom (K[X]) (Rupt K p)"
   and "ring_hom_ring (K[X]) (Rupt K p) (rupture_surj K p)"
    using ring_hom_ring.intro[OF UP.ring_axioms I.quotient_is_ring] I.rcos_ring_hom
    unfolding symmetric[OF ring_hom_ring_axioms_def] rupture_def by auto
qed

corollary (in domain) rupture_surj_norm_is_hom:
  assumes "subring K R" and "p ∈ carrier (K[X])"
  shows "((rupture_surj K p) ∘ poly_of_const) ∈ ring_hom (R ⦇ carrier
:= K ⦈) (Rupt K p)"
  using ring_hom_trans[OF canonical_embedding_is_hom[OF assms(1)] rupture_surj_hom(1)[OF
assms]] .

lemma (in domain) norm_map_in_poly_ring_carrier:
  assumes "p ∈ carrier (poly_ring R)" and "⋀a. a ∈ carrier R ⟹ f a
∈ carrier (poly_ring R)"
  shows "ring.normalize (poly_ring R) (map f p) ∈ carrier (poly_ring
(poly_ring R))"
proof -
  have "set p ⊆ carrier R"
    using assms(1) unfolding sym[OF univ_poly_carrier] polynomial_def
by auto
```

```
  hence "set (map f p) ⊆ carrier (poly_ring R)"
    using assms(2) by auto
  thus ?thesis
    using ring.normalize_gives_polynomial[OF univ_poly_is_ring[OF carrier_is_subring]]
    unfolding univ_poly_carrier by simp
qed

lemma (in domain) map_in_poly_ring_carrier:
  assumes "p ∈ carrier (poly_ring R)" and "⋀a. a ∈ carrier R ⟹ f a
∈ carrier (poly_ring R)"
    and "⋀a. a ≠ 𝟬 ⟹ f a ≠ []"
  shows "map f p ∈ carrier (poly_ring (poly_ring R))"
proof -
  interpret UP: ring "poly_ring R"
    using univ_poly_is_ring[OF carrier_is_subring] .
  have "lead_coeff p ≠ 𝟬" if "p ≠ []"
    using that assms(1) unfolding sym[OF univ_poly_carrier] polynomial_def
by auto
  hence "ring.normalize (poly_ring R) (map f p) = map f p"
    by (cases p) (simp_all add: assms(3) univ_poly_zero)
  thus ?thesis
    using norm_map_in_poly_ring_carrier[of p f] assms(1-2) by simp
qed

lemma (in domain) map_norm_in_poly_ring_carrier:
  assumes "subring K R" and "p ∈ carrier (K[X])"
  shows "map poly_of_const p ∈ carrier (poly_ring (K[X]))"
  using domain.map_in_poly_ring_carrier[OF subring_is_domain[OF assms(1)]]
proof -
  have "⋀a. a ∈ K ⟹ poly_of_const a ∈ carrier (K[X])"
   and "⋀a. a ≠ 𝟬 ⟹ poly_of_const a ≠ []"
    using ring_hom_memE(1)[OF canonical_embedding_is_hom[OF assms(1)]]
    by (auto simp: poly_of_const_def)
  thus ?thesis
    using domain.map_in_poly_ring_carrier[OF subring_is_domain[OF assms(1)]]
assms(2)
    unfolding univ_poly_consistent[OF assms(1)] by simp
qed

lemma (in domain) polynomial_rupture:
  assumes "subring K R" and "p ∈ carrier (K[X])"
  shows "(ring.eval (Rupt K p)) (map ((rupture_surj K p) ∘ poly_of_const)
p) (rupture_surj K p X) = 𝟬Rupt K p"
proof -
  let ?surj = "rupture_surj K p"

  interpret UP: domain "K[X]"
    using univ_poly_is_domain[OF assms(1)] .
  interpret Hom: ring_hom_ring "K[X]" "Rupt K p" ?surj
```

```
    using rupture_surj_hom(2)[OF assms] .

  have "(Hom.S.eval) (map (?surj ∘ poly_of_const) p) (?surj X) = ?surj
((UP.eval) (map poly_of_const p) X)"
    using Hom.eval_hom[OF UP.carrier_is_subring var_closed(1)[OF assms(1)]
          map_norm_in_poly_ring_carrier[OF assms]] by simp
  also have " ... = ?surj p"
    unfolding sym[OF eval_rewrite[OF assms]] ..
  also have " ... = 0Rupt K p"
    using UP.a_rcos_zero[OF UP.cgenideal_ideal[OF assms(2)] UP.cgenideal_self[OF
assms(2)]]
    unfolding rupture_def FactRing_def by simp
  finally show ?thesis .
qed
```

### 35.3 Division

```
definition (in ring) long_divides :: "'a list ⇒ 'a list ⇒ ('a list ×
'a list) ⇒ bool"
  where "long_divides p q t ⟷
          — i   (t ∈ carrier (poly_ring R) × carrier (poly_ring R))
∧
          — ii  (p = (q ⊗poly_ring R (fst t)) ⊕poly_ring R (snd t)) ∧
          — iii (snd t = [] ∨ degree (snd t) < degree q)"

definition (in ring) long_division :: "'a list ⇒ 'a list ⇒ ('a list ×
'a list)"
  where "long_division p q = (THE t. long_divides p q t)"

definition (in ring) pdiv :: "'a list ⇒ 'a list ⇒ 'a list" (infixl ‹pdiv›
65)
  where "p pdiv q = (if q = [] then [] else fst (long_division p q))"

definition (in ring) pmod :: "'a list ⇒ 'a list ⇒ 'a list" (infixl ‹pmod›
65)
  where "p pmod q = (if q = [] then p else snd (long_division p q))"


lemma (in ring) long_dividesI:
  assumes "b ∈ carrier (poly_ring R)" and "r ∈ carrier (poly_ring R)"
      and "p = (q ⊗poly_ring R b) ⊕poly_ring R r" and "r = [] ∨ degree
r < degree q"
    shows "long_divides p q (b, r)"
  using assms unfolding long_divides_def by auto

lemma (in domain) exists_long_division:
  assumes "subfield K R" and "p ∈ carrier (K[X])" and "q ∈ carrier (K[X])"
"q ≠ []"
  obtains b r where "b ∈ carrier (K[X])" and "r ∈ carrier (K[X])" and
```

```
"long_divides p q (b, r)"
  using subfield_long_division_theorem_shell[OF assms(1-3)] assms(4)
        carrier_polynomial_shell[OF subfieldE(1)[OF assms(1)]]
  unfolding long_divides_def univ_poly_zero univ_poly_add univ_poly_mult
by auto

lemma (in domain) exists_unique_long_division:
  assumes "subfield K R" and "p ∈ carrier (K[X])" and "q ∈ carrier (K[X])"
"q ≠ []"
  shows "∃!t. long_divides p q t"
proof -
  let ?padd   = "λa b. a ⊕poly_ring R b"
  let ?pmult  = "λa b. a ⊗poly_ring R b"
  let ?pminus = "λa b. a ⊖poly_ring R b"

  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .

  obtain b r where ldiv: "long_divides p q (b, r)"
    using exists_long_division[OF assms] by metis

  moreover have "(b, r) = (b', r')" if "long_divides p q (b', r')" for
b' r'
  proof -
    have q: "q ∈ carrier (poly_ring R)" "q ≠ []"
      using assms(3-4) carrier_polynomial[OF subfieldE(1)[OF assms(1)]]
      unfolding univ_poly_carrier by auto
    hence in_carrier: "q ∈ carrier (poly_ring R)"
      "b  ∈ carrier (poly_ring R)" "r  ∈ carrier (poly_ring R)"
      "b' ∈ carrier (poly_ring R)" "r' ∈ carrier (poly_ring R)"
      using assms(3) that ldiv unfolding long_divides_def by auto
    have "?pminus (?padd (?pmult q b) r) r' = ?pminus (?padd (?pmult q
b') r') r'"
      using ldiv and that unfolding long_divides_def by auto
    hence eq: "?padd (?pmult q (?pminus b b')) (?pminus r r') = 𝟬poly_ring R"
      using in_carrier by algebra
    have "b = b'"
    proof (rule ccontr)
      assume "b ≠ b'"
      hence pminus: "?pminus b b' ≠ 𝟬poly_ring R" "?pminus b b' ∈ carrier
(poly_ring R)"
        using in_carrier(2,4) by (metis UP.add.inv_closed UP.l_neg UP.minus_eq
UP.minus_unique, algebra)
      hence degree_ge: "degree (?pmult q (?pminus b b')) ≥ degree q"
        using poly_mult_degree_eq[OF carrier_is_subring, of q "?pminus
b b'"] q
        unfolding univ_poly_zero univ_poly_carrier univ_poly_mult by simp

      have "?pminus b b' = 𝟬poly_ring R" if "?pminus r r' = 𝟬poly_ring R"
```

```
        using eq pminus(2) q UP.integral univ_poly_zero unfolding that
by auto
      hence "?pminus r r' ≠ []"
        using pminus(1) unfolding univ_poly_zero by blast
      moreover have "?pminus r r' = []" if "r = []" and "r' = []"
        using univ_poly_a_inv_def'[OF carrier_is_subring UP.zero_closed]
that
        unfolding a_minus_def univ_poly_add univ_poly_zero by auto
      ultimately have "r ≠ [] ∨ r' ≠ []"
        by blast
      hence "max (degree r) (degree r') < degree q"
        using ldiv and that unfolding long_divides_def by auto
      moreover have "degree (?pminus r r') ≤ max (degree r) (degree
r')"
        using poly_add_degree[of r "map (a_inv R) r'"]
        unfolding a_minus_def univ_poly_add univ_poly_a_inv_def'[OF carrier_is_subring
in_carrier(5)]
        by auto
      ultimately have degree_lt: "degree (?pminus r r') < degree q"
        by linarith
      have is_poly: "polynomial (carrier R) (?pmult q (?pminus b b'))"
"polynomial (carrier R) (?pminus r r')"
        using in_carrier pminus(2) unfolding univ_poly_carrier by algebra+

      have "degree (?padd (?pmult q (?pminus b b')) (?pminus r r')) =
degree (?pmult q (?pminus b b'))"
        using poly_add_degree_eq[OF carrier_is_subring is_poly] degree_ge
degree_lt
        unfolding univ_poly_carrier sym[OF univ_poly_add[of R "carrier
R"]] max_def by simp
      hence "degree (?padd (?pmult q (?pminus b b')) (?pminus r r')) >
0"
        using degree_ge degree_lt by simp
      moreover have "degree (?padd (?pmult q (?pminus b b')) (?pminus
r r')) = 0"
        using eq unfolding univ_poly_zero by simp
      ultimately show False by simp
    qed
    hence "?pminus r r' = 0_poly_ring R"
      using in_carrier eq by algebra
    hence "r = r'"
      using in_carrier by (metis UP.add.inv_closed UP.add.right_cancel
UP.minus_eq UP.r_neg)
    with ‹b = b'› show ?thesis
      by simp
  qed

  ultimately show ?thesis
    by auto
```

```
qed

lemma (in domain) long_divisionE:
  assumes "subfield K R" and "p ∈ carrier (K[X])" and "q ∈ carrier (K[X])"
"q ≠ []"
  shows "long_divides p q (p pdiv q, p pmod q)"
  using theI'[OF exists_unique_long_division[OF assms]] assms(4)
  unfolding pmod_def pdiv_def long_division_def by auto

lemma (in domain) long_divisionI:
  assumes "subfield K R" and "p ∈ carrier (K[X])" and "q ∈ carrier (K[X])"
"q ≠ []"
  shows "long_divides p q (b, r) ⟹ (b, r) = (p pdiv q, p pmod q)"
  using exists_unique_long_division[OF assms] long_divisionE[OF assms]
by metis

lemma (in domain) long_division_closed:
  assumes "subfield K R" and "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
  shows "p pdiv q ∈ carrier (K[X])" and "p pmod q ∈ carrier (K[X])"
proof -
  have "p pdiv q ∈ carrier (K[X]) ∧ p pmod q ∈ carrier (K[X])"
    using assms univ_poly_zero_closed[of R] long_divisionI[of K] exists_long_division[OF
assms]
    by (cases "q = []") (simp add: pdiv_def pmod_def, metis Pair_inject)+
  thus "p pdiv q ∈ carrier (K[X])" and "p pmod q ∈ carrier (K[X])"
    by auto
qed

lemma (in domain) pdiv_pmod:
  assumes "subfield K R" and "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
  shows "p = (q ⊗K[X] (p pdiv q)) ⊕K[X] (p pmod q)"
proof (cases)
  interpret UP: ring "K[X]"
    using univ_poly_is_ring[OF subfieldE(1)[OF assms(1)]] .
  assume "q = []" thus ?thesis
    using assms(2) unfolding pdiv_def pmod_def sym[OF univ_poly_zero[of
R K]] by simp
next
  assume "q ≠ []" thus ?thesis
    using long_divisionE[OF assms] unfolding long_divides_def univ_poly_mult
univ_poly_add by simp
qed

lemma (in domain) pmod_degree:
  assumes "subfield K R" and "p ∈ carrier (K[X])" and "q ∈ carrier (K[X])"
"q ≠ []"
  shows "p pmod q = [] ∨ degree (p pmod q) < degree q"
  using long_divisionE[OF assms] unfolding long_divides_def by auto
```

```
lemma (in domain) pmod_const:
  assumes "subfield K R" and "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
and "degree q > degree p"
  shows "p pdiv q = []" and "p pmod q = p"
proof -
  have "p pdiv q = [] ∧ p pmod q = p"
  proof (cases)
    interpret UP: ring "K[X]"
      using univ_poly_is_ring[OF subfieldE(1)[OF assms(1)]] .

    assume "q ≠ []"
    have "p = (q ⊗K[X] []) ⊕K[X] p"
      using assms(2-3) unfolding sym[OF univ_poly_zero[of R K]] by simp
    moreover have "([], p) ∈ carrier (poly_ring R) × carrier (poly_ring
R)"
      using carrier_polynomial_shell[OF subfieldE(1)[OF assms(1)] assms(2)]
by auto
    ultimately have "long_divides p q ([], p)"
      using assms(4) unfolding long_divides_def univ_poly_mult univ_poly_add
by auto
    with ‹q ≠ []› show ?thesis
      using long_divisionI[OF assms(1-3)] by auto
  qed (simp add: pmod_def pdiv_def)
  thus "p pdiv q = []" and "p pmod q = p"
    by auto
qed

lemma (in domain) long_division_zero:
  assumes "subfield K R" and "q ∈ carrier (K[X])" shows "[] pdiv q =
[]" and "[] pmod q = []"
proof -
  interpret UP: ring "poly_ring R"
    using univ_poly_is_ring[OF carrier_is_subring] .

  have "[] pdiv q = [] ∧ [] pmod q = []"
  proof (cases)
    assume "q ≠ []"
    have "q ∈ carrier (poly_ring R)"
      using carrier_polynomial_shell[OF subfieldE(1)[OF assms(1)] assms(2)]
.
    hence "long_divides [] q ([], [])"
      unfolding long_divides_def sym[OF univ_poly_zero[of R "carrier R"]]
by auto
    with ‹q ≠ []› show ?thesis
      using long_divisionI[OF assms(1) univ_poly_zero_closed assms(2)]
by simp
  qed (simp add: pmod_def pdiv_def)
  thus "[] pdiv q = []" and "[] pmod q = []"
    by auto
```

```
qed

lemma (in domain) long_division_a_inv:
  assumes "subfield K R" and "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
  shows "((⊖K[X] p) pdiv q) = ⊖K[X] (p pdiv q)" (is "?pdiv")
    and "((⊖K[X] p) pmod q) = ⊖K[X] (p pmod q)" (is "?pmod")
proof -
  interpret UP: ring "K[X]"
    using univ_poly_is_ring[OF subfieldE(1)[OF assms(1)]] .

  have "?pdiv ∧ ?pmod"
  proof (cases)
    assume "q = []" thus ?thesis
      unfolding pmod_def pdiv_def sym[OF univ_poly_zero[of R K]] by simp
  next
    assume not_nil: "q ≠ []"
    have "⊖K[X] p = ⊖K[X] ((q ⊗K[X] (p pdiv q)) ⊕K[X] (p pmod q))"
      using pdiv_pmod[OF assms] by simp
    hence "⊖K[X] p = (q ⊗K[X] (⊖K[X] (p pdiv q))) ⊕K[X] (⊖K[X] (p pmod
q))"
      using assms(2-3) long_division_closed[OF assms] by algebra
    moreover have "⊖K[X] (p pdiv q) ∈ carrier (K[X])" "⊖K[X] (p pmod
q) ∈ carrier (K[X])"
      using long_division_closed[OF assms] by algebra+
    hence "(⊖K[X] (p pdiv q), ⊖K[X] (p pmod q)) ∈ carrier (poly_ring
R) × carrier (poly_ring R)"
      using carrier_polynomial_shell[OF subfieldE(1)[OF assms(1)]] by
auto
    moreover have "⊖K[X] (p pmod q) = [] ∨ degree (⊖K[X] (p pmod q))
< degree q"
      using univ_poly_a_inv_length[OF subfieldE(1)[OF assms(1)]
            long_division_closed(2)[OF assms]] pmod_degree[OF assms not_nil]
      by auto
    ultimately have "long_divides (⊖K[X] p) q (⊖K[X] (p pdiv q), ⊖K[X]
(p pmod q))"
      unfolding long_divides_def univ_poly_mult univ_poly_add by simp
    thus ?thesis
      using long_divisionI[OF assms(1) UP.a_inv_closed[OF assms(2)] assms(3)
not_nil] by simp
  qed
  thus ?pdiv and ?pmod
    by auto
qed

lemma (in domain) long_division_add:
  assumes "subfield K R" and "a ∈ carrier (K[X])" "b ∈ carrier (K[X])"
"q ∈ carrier (K[X])"
  shows "(a ⊕K[X] b) pdiv q = (a pdiv q) ⊕K[X] (b pdiv q)" (is "?pdiv")
    and "(a ⊕K[X] b) pmod q = (a pmod q) ⊕K[X] (b pmod q)" (is "?pmod")
```

```
proof -
  let ?pdiv_add = "(a pdiv q) ⊕K[X] (b pdiv q)"
  let ?pmod_add = "(a pmod q) ⊕K[X] (b pmod q)"

  interpret UP: ring "K[X]"
    using univ_poly_is_ring[OF subfieldE(1)[OF assms(1)]] .

  have "?pdiv ∧ ?pmod"
  proof (cases)
    assume "q = []" thus ?thesis
      using assms(2-3) unfolding pmod_def pdiv_def sym[OF univ_poly_zero[of
R K]] by simp
  next
    note in_carrier = long_division_closed[OF assms(1,2,4)]
                      long_division_closed[OF assms(1,3,4)]

    assume "q ≠ []"
    have "a ⊕K[X] b = ((q ⊗K[X] (a pdiv q)) ⊕K[X] (a pmod q)) ⊕K[X]
                       ((q ⊗K[X] (b pdiv q)) ⊕K[X] (b pmod q))"
      using assms(2-3)[THEN pdiv_pmod[OF assms(1) _ assms(4)]] by simp
    hence "a ⊕K[X] b = (q ⊗K[X] ?pdiv_add) ⊕K[X] ?pmod_add"
      using assms(4) in_carrier by algebra
    moreover have "(?pdiv_add, ?pmod_add) ∈ carrier (poly_ring R) ×
carrier (poly_ring R)"
      using in_carrier carrier_polynomial_shell[OF subfieldE(1)[OF assms(1)]]
by auto
    moreover have "?pmod_add = [] ∨ degree ?pmod_add < degree q"
    proof (cases)
      assume "?pmod_add ≠ []"
      hence "a pmod q ≠ [] ∨ b pmod q ≠ []"
        using in_carrier(2,4) unfolding sym[OF univ_poly_zero[of R K]]
by auto
      moreover from ‹q ≠ []›
      have "a pmod q = [] ∨ degree (a pmod q) < degree q" and "b pmod
q = [] ∨ degree (b pmod q) < degree q"
        using assms(2-3)[THEN pmod_degree[OF assms(1) _ assms(4)]] by
auto
      ultimately have "max (degree (a pmod q)) (degree (b pmod q)) < degree
q"
        by auto
      thus ?thesis
        using poly_add_degree le_less_trans unfolding univ_poly_add by
blast
    qed simp
    ultimately have "long_divides (a ⊕K[X] b) q (?pdiv_add, ?pmod_add)"
      unfolding long_divides_def univ_poly_mult univ_poly_add by simp
    with ‹q ≠ []› show ?thesis
      using long_divisionI[OF assms(1) UP.a_closed[OF assms(2-3)] assms(4)]
by simp
```

```
  qed
  thus ?pdiv and ?pmod
    by auto
qed

lemma (in domain) long_division_add_iff:
  assumes "subfield K R"
    and "a ∈ carrier (K[X])" "b ∈ carrier (K[X])" "c ∈ carrier (K[X])"
"q ∈ carrier (K[X])"
  shows "a pmod q = b pmod q ⟷ (a ⊕K[X] c) pmod q = (b ⊕K[X] c) pmod
q"
proof -
  interpret UP: ring "K[X]"
    using univ_poly_is_ring[OF subfieldE(1)[OF assms(1)]] .
  show ?thesis
    using assms(2-4)[THEN long_division_closed(2)[OF assms(1) _ assms(5)]]
    unfolding assms(2-3)[THEN long_division_add(2)[OF assms(1) _ assms(4-5)]]
by auto
qed

lemma (in domain) pdivides_iff:
  assumes "subfield K R" and "polynomial K p" "polynomial K q"
  shows "p pdivides q ⟷ p dividesK[X] q"
proof
  show "p dividesK [X] q ⟹ p pdivides q"
    using carrier_polynomial[OF subfieldE(1)[OF assms(1)]]
    unfolding pdivides_def factor_def univ_poly_mult univ_poly_carrier
by auto
next
  interpret UP: ring "poly_ring R"
    using univ_poly_is_ring[OF carrier_is_subring] .

  have in_carrier: "p ∈ carrier (poly_ring R)" "q ∈ carrier (poly_ring
R)"
    using carrier_polynomial[OF subfieldE(1)[OF assms(1)]] assms
    unfolding univ_poly_carrier by auto

  assume "p pdivides q"
  then obtain b where "b ∈ carrier (poly_ring R)" and "q = p ⊗poly_ring R
b"
      unfolding pdivides_def factor_def by blast
  show "p dividesK[X] q"
  proof (cases)
    assume "p = []"
    with ‹b ∈ carrier (poly_ring R)› and ‹q = p ⊗poly_ring R b› have
"q = []"
      unfolding univ_poly_mult sym[OF univ_poly_carrier]
      using poly_mult_zero(1)[OF polynomial_incl] by simp
    with ‹p = []› show ?thesis
```

```
      using poly_mult_zero(2)[of "[]"]
      unfolding factor_def univ_poly_mult by auto
  next
    interpret UP: ring "poly_ring R"
      using univ_poly_is_ring[OF carrier_is_subring] .

    assume "p ≠ []"
    from ‹p pdivides q› obtain b where "b ∈ carrier (poly_ring R)" and
"q = p ⊗poly_ring R b"
      unfolding pdivides_def factor_def by blast
    moreover have "p ∈ carrier (poly_ring R)" and "q ∈ carrier (poly_ring
R)"
      using assms carrier_polynomial[OF subfieldE(1)[OF assms(1)]] un-
folding univ_poly_carrier by auto
    ultimately have "q = (p ⊗poly_ring R b) ⊕poly_ring R 0poly_ring R"
      by algebra
    with ‹b ∈ carrier (poly_ring R)› have "long_divides q p (b, [])"
      unfolding long_divides_def univ_poly_zero by auto
    with ‹p ≠ []› have "b ∈ carrier (K[X])"
      using long_divisionI[of K q p b] long_division_closed[of K q p]
assms
      unfolding univ_poly_carrier by auto
    with ‹q = p ⊗poly_ring R b› show ?thesis
      unfolding factor_def univ_poly_mult by blast
  qed
qed

lemma (in domain) pdivides_iff_shell:
  assumes "subfield K R" and "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
  shows "p pdivides q ⟷ p dividesK[X] q"
  using pdivides_iff assms by (simp add: univ_poly_carrier)

lemma (in domain) pmod_zero_iff_pdivides:
  assumes "subfield K R" and "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
  shows "p pmod q = [] ⟷ q pdivides p"
proof -
  interpret UP: domain "K[X]"
    using univ_poly_is_domain[OF subfieldE(1)[OF assms(1)]] .

  show ?thesis
  proof
    assume pmod: "p pmod q = []"
    have "p pdiv q ∈ carrier (K[X])" and "p pmod q ∈ carrier (K[X])"
      using long_division_closed[OF assms] by auto
    hence "p = q ⊗K[X] (p pdiv q)"
      using pdiv_pmod[OF assms] assms(3) unfolding pmod sym[OF univ_poly_zero[of
R K]] by algebra
    with ‹p pdiv q ∈ carrier (K[X])› show "q pdivides p"
      unfolding pdivides_iff_shell[OF assms(1,3,2)] factor_def by blast
```

```
  next
    assume "q pdivides p" show "p pmod q = []"
    proof (cases)
      assume "q = []" with ‹q pdivides p› show ?thesis
        using zero_pdivides unfolding pmod_def by simp
    next
      assume "q ≠ []"
      from ‹q pdivides p› obtain r where "r ∈ carrier (K[X])" and "p
= q ⊗K[X] r"
        unfolding pdivides_iff_shell[OF assms(1,3,2)] factor_def by blast
      hence "p = (q ⊗K[X] r) ⊕K[X] []"
        using assms(2) unfolding sym[OF univ_poly_zero[of R K]] by simp
      moreover from ‹r ∈ carrier (K[X])› have "r ∈ carrier (poly_ring
R)"
        using carrier_polynomial_shell[OF subfieldE(1)[OF assms(1)]] by
auto
      ultimately have "long_divides p q (r, [])"
        unfolding long_divides_def univ_poly_mult univ_poly_add by auto
      with ‹q ≠ []› show ?thesis
        using long_divisionI[OF assms] by simp
    qed
  qed
qed

lemma (in domain) same_pmod_iff_pdivides:
  assumes "subfield K R" and "a ∈ carrier (K[X])" "b ∈ carrier (K[X])"
"q ∈ carrier (K[X])"
  shows "a pmod q = b pmod q ⟷ q pdivides (a ⊖K[X] b)"
proof -
  interpret UP: domain "K[X]"
    using univ_poly_is_domain[OF subfieldE(1)[OF assms(1)]] .

  have "a pmod q = b pmod q ⟷ (a ⊕K[X] (⊖K[X] b)) pmod q = (b ⊕K[X]
(⊖K[X] b)) pmod q"
    using long_division_add_iff[OF assms(1-3) UP.a_inv_closed[OF assms(3)]
assms(4)] .
  also have " ... ⟷ (a ⊖K[X] b) pmod q = 0K[X] pmod q"
    using assms(2-3) by algebra
  also have " ... ⟷ q pdivides (a ⊖K[X] b)"
    using pmod_zero_iff_pdivides[OF assms(1) UP.minus_closed[OF assms(2-3)]
assms(4)]
    unfolding univ_poly_zero long_division_zero(2)[OF assms(1,4)] .
  finally show ?thesis .
qed

lemma (in domain) pdivides_imp_degree_le:
  assumes "subring K R" and "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
"q ≠ []"
  shows "p pdivides q ⟹ degree p ≤ degree q"
```

```
proof -
  assume "p pdivides q"
  then obtain r where r: "polynomial (carrier R) r" "q = poly_mult p
r"
    unfolding pdivides_def factor_def univ_poly_mult univ_poly_carrier
by blast
  moreover have p: "polynomial (carrier R) p"
    using assms(2) carrier_polynomial[OF assms(1)] unfolding univ_poly_carrier
by auto
  moreover have "p ≠ []" and "r ≠ []"
    using poly_mult_zero(2)[OF polynomial_incl[OF p]] r(2) assms(4) by
auto
  ultimately show "degree p ≤ degree q"
    using poly_mult_degree_eq[OF carrier_is_subring, of p r] by auto
qed

lemma (in domain) pprimeE:
  assumes "subfield K R" "p ∈ carrier (K[X])" "pprime K p"
  shows "p ≠ []" "p ∉ Units (K[X])"
    and "⋀q r. ⟦ q ∈ carrier (K[X]); r ∈ carrier (K[X])⟧ ⟹
                 p pdivides (q ⊗K[X] r) ⟹ p pdivides q ∨ p pdivides
r"
  using assms(2-3) poly_mult_closed[OF subfieldE(1)[OF assms(1)]] pdivides_iff[OF
assms(1)]
  unfolding ring_prime_def prime_def
  by (auto simp add: univ_poly_mult univ_poly_carrier univ_poly_zero)

lemma (in domain) pprimeI:
  assumes "subfield K R" "p ∈ carrier (K[X])" "p ≠ []" "p ∉ Units (K[X])"
    and "⋀q r. ⟦ q ∈ carrier (K[X]); r ∈ carrier (K[X])⟧ ⟹
                 p pdivides (q ⊗K[X] r) ⟹ p pdivides q ∨ p pdivides
r"
  shows "pprime K p"
  using assms(2-5) poly_mult_closed[OF subfieldE(1)[OF assms(1)]] pdivides_iff[OF
assms(1)]
  unfolding ring_prime_def prime_def
  by (auto simp add: univ_poly_mult univ_poly_carrier univ_poly_zero)

lemma (in domain) associated_polynomials_iff:
  assumes "subfield K R" and "p ∈ carrier (K[X])" "q ∈ carrier (K[X])"
  shows "p ∼K[X] q ⟷ (∃k ∈ K - { 0 }. p = [ k ] ⊗K[X] q)"
  using domain.ring_associated_iff[OF univ_poly_is_domain[OF subfieldE(1)[OF
assms(1)]] assms(2-3)]
  unfolding univ_poly_units[OF assms(1)] by auto

corollary (in domain) associated_polynomials_imp_same_length:
  assumes "subring K R" and "p ∈ carrier (K[X])" and "q ∈ carrier (K[X])"
  shows "p ∼K[X] q ⟹ length p = length q"
proof -
```

```
  have aux_lemma: "length p ≤ length q"
    if p: "p ∈ carrier (K[X])" and q: "q ∈ carrier (K[X])" and "p ∼K[X]
q" for p q
  proof (cases "q = []")
    case True with ‹p ∼K[X] q› have "p = []"
      unfolding associated_def True factor_def univ_poly_def by auto
    thus ?thesis
      using True by simp
  next
    case False
    from ‹p ∼K[X] q› have "p dividesK [X] q"
      unfolding associated_def by simp
    hence "p dividespoly_ring R q"
      using carrier_polynomial[OF assms(1)]
      unfolding factor_def univ_poly_carrier univ_poly_mult by auto
    with ‹q ≠ []› have "degree p ≤ degree q"
      using pdivides_imp_degree_le[OF assms(1) p q] unfolding pdivides_def
by simp
    with ‹q ≠ []› show ?thesis
      by (cases "p = []", auto simp add: Suc_leI le_diff_iff)
  qed

  interpret UP: domain "K[X]"
    using univ_poly_is_domain[OF assms(1)] .

  assume "p ∼K[X] q" thus ?thesis
    using aux_lemma[OF assms(2-3)] aux_lemma[OF assms(3,2) UP.associated_sym]
by simp
qed

lemma (in ring) divides_pirreducible_condition:
  assumes "pirreducible K q" and "p ∈ carrier (K[X])"
  shows "p dividesK[X] q ⟹ p ∈ Units (K[X]) ∨ p ∼K[X] q"
  using divides_irreducible_condition[of "K[X]" q p] assms
  unfolding ring_irreducible_def by auto
```

## 35.4 Polynomial Power

```
lemma (in domain) polynomial_pow_not_zero:
  assumes "p ∈ carrier (poly_ring R)" and "p ≠ []"
  shows "p [^]poly_ring R (n::nat) ≠ []"
proof -
  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .

  from assms UP.integral show ?thesis
    unfolding sym[OF univ_poly_zero[of R "carrier R"]]
    by (induction n, auto)
qed
```

```
lemma (in domain) subring_polynomial_pow_not_zero:
  assumes "subring K R" and "p ∈ carrier (K[X])" and "p ≠ []"
  shows "p [^]K[X] (n::nat) ≠ []"
  using domain.polynomial_pow_not_zero[OF subring_is_domain, of K p n]
assms
  unfolding univ_poly_consistent[OF assms(1)] by simp

lemma (in domain) polynomial_pow_degree:
  assumes "p ∈ carrier (poly_ring R)"
  shows "degree (p [^]poly_ring R n) = n * degree p"
proof -
  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .

  show ?thesis
  proof (induction n)
    case 0 thus ?case
      using UP.nat_pow_0 unfolding univ_poly_one by auto
  next
    let ?ppow = "λn. p [^]poly_ring R n"
    case (Suc n) thus ?case
    proof (cases "p = []")
      case True thus ?thesis
        using univ_poly_zero[of R "carrier R"] UP.r_null assms by auto
    next
      case False
      hence "?ppow n ∈ carrier (poly_ring R)" and "?ppow n ≠ []" and
"p ≠ []"
        using polynomial_pow_not_zero[of p n] assms by (auto simp add:
univ_poly_one)
      thus ?thesis
        using poly_mult_degree_eq[OF carrier_is_subring, of "?ppow n"
p] Suc assms
        unfolding univ_poly_carrier univ_poly_zero
        by (auto simp add: add.commute univ_poly_mult)
    qed
  qed
qed

lemma (in domain) subring_polynomial_pow_degree:
  assumes "subring K R" and "p ∈ carrier (K[X])"
  shows "degree (p [^]K[X] n) = n * degree p"
  using domain.polynomial_pow_degree[OF subring_is_domain, of K p n] assms
  unfolding univ_poly_consistent[OF assms(1)] by simp

lemma (in domain) polynomial_pow_division:
  assumes "p ∈ carrier (poly_ring R)" and "(n::nat) ≤ m"
  shows "(p [^]poly_ring R n) pdivides (p [^]poly_ring R m)"
```

```
proof -
  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .

  let ?ppow = "λn. p [^]poly_ring R n"

  have "?ppow n ⊗poly_ring R ?ppow k = ?ppow (n + k)" for k
    using assms(1) by (simp add: UP.nat_pow_mult)
  thus ?thesis
    using dividesI[of "?ppow (m - n)" "poly_ring R" "?ppow m" "?ppow n"]
assms
    unfolding pdivides_def by auto
qed

lemma (in domain) subring_polynomial_pow_division:
  assumes "subring K R" and "p ∈ carrier (K[X])" and "(n::nat) ≤ m"
  shows "(p [^]K[X] n) dividesK[X] (p [^]K[X] m)"
  using domain.polynomial_pow_division[OF subring_is_domain, of K p n
m] assms
  unfolding univ_poly_consistent[OF assms(1)] pdivides_def by simp

lemma (in domain) pirreducible_pow_pdivides_iff:
  assumes "subfield K R" "p ∈ carrier (K[X])" "q ∈ carrier (K[X])" "r
∈ carrier (K[X])"
    and "pirreducible K p" and "¬ (p pdivides q)"
  shows "(p [^]K[X] (n :: nat)) pdivides (q ⊗K[X] r) ⟷ (p [^]K[X] n)
pdivides r"
proof -
  interpret UP: principal_domain "K[X]"
    using univ_poly_is_principal[OF assms(1)] .
  show ?thesis
  proof (cases "r = []")
    case True with ‹q ∈ carrier (K[X])› have "q ⊗K[X] r = []" and "r
= []"
      unfolding  sym[OF univ_poly_zero[of R K]] by auto
    thus ?thesis
      using pdivides_zero[OF subfieldE(1),of K] assms by auto
  next
    case False then have not_zero: "p ≠ []" "q ≠ []" "r ≠ []" "q ⊗K[X]
r ≠ []"
      using subfieldE(1) pdivides_zero[OF _ assms(2)] assms(1-2,5-6) pirreducibleE(1)
            UP.integral_iff[OF assms(3-4)] univ_poly_zero[of R K] by auto
    from ‹p ≠ []›
    have ppow: "p [^]K[X] (n :: nat) ≠ []" "p [^]K[X] (n :: nat) ∈ carrier
(K[X])"
      using subring_polynomial_pow_not_zero[OF subfieldE(1)] assms(1-2)
by auto
    have not_pdiv: "¬ (p dividesmult_of (K[X]) q)"
      using assms(6) pdivides_iff_shell[OF assms(1-3)] unfolding pdivides_def
```

```
by auto
    have prime: "prime (mult_of (K[X])) p"
      using assms(5) pprime_iff_pirreducible[OF assms(1-2)]
      unfolding sym[OF UP.prime_eq_prime_mult[OF assms(2)]] ring_prime_def
by simp
    have "a pdivides b ⟷ a divides_mult_of (K[X]) b"
      if "a ∈ carrier (K[X])" "a ≠ 0_K[X]" "b ∈ carrier (K[X])" "b ≠ 0_K[X]"
for a b
      using that UP.divides_imp_divides_mult[of a b] divides_mult_imp_divides[of
"K[X]" a b]
      unfolding pdivides_iff_shell[OF assms(1) that(1,3)] by blast
    thus ?thesis
      using UP.mult_of.prime_pow_divides_iff[OF _ _ _ prime not_pdiv,
of r] ppow not_zero assms(2-4)
      unfolding nat_pow_mult_of carrier_mult_of mult_mult_of sym[OF univ_poly_zero[of
R K]]
      by (metis DiffI UP.m_closed singletonD)
  qed
qed

lemma (in domain) subring_degree_one_imp_pirreducible:
  assumes "subring K R" and "a ∈ Units (R ⦇ carrier := K ⦈)" and "b
∈ K"
  shows "pirreducible K [ a, b ]"
proof (rule pirreducibleI[OF assms(1)])
  have "a ∈ K" and "a ≠ 0"
    using assms(2) subringE(1)[OF assms(1)] unfolding Units_def by auto
  thus "[ a, b ] ∈ carrier (K[X])" and "[ a, b ] ≠ []" and "[ a, b ]
∉ Units (K [X])"
    using univ_poly_units_incl[OF assms(1)] assms(2-3)
    unfolding sym[OF univ_poly_carrier] polynomial_def by auto
next
  interpret UP: domain "K[X]"
    using univ_poly_is_domain[OF assms(1)] .

  have aux_lemma1: "degree q + degree r = 1" "q ≠ []" "r ≠ []"
    if q: "q ∈ carrier (K[X])" and r: "r ∈ carrier (K[X])" and "[ a,
b ] = q ⊗_K[X] r" for q r
  proof -
    from that have not_zero: "q ≠ []" "r ≠ []"
      by (metis UP.integral_iff list.distinct(1) univ_poly_zero)+
    have "degree (q ⊗_K[X] r) = degree q + degree r"
      using not_zero poly_mult_degree_eq[OF assms(1)] q r
      by (simp add: univ_poly_carrier univ_poly_mult)
    with sym[OF ‹[ a, b ] = q ⊗_K[X] r›] show "degree q + degree r =
1" and "q ≠ []" "r ≠ []"
      using not_zero by auto
  qed
```

```
  have aux_lemma2: "r ∈ Units (K[X])"
    if q: "q ∈ carrier (K[X])" "q ≠ []" and r: "r ∈ carrier (K[X])" "r
≠ []"
    and "[ a, b ] = q ⊗K[X] r" and "degree q = 1" and "degree r = 0"
for q r
  proof -
    from that have "length q = Suc (Suc 0)" and "length r = Suc 0"
      by (linarith, metis add.right_neutral add_eq_if length_0_conv)
    from ‹length q = Suc (Suc 0)› obtain c d where q_def: "q = [ c,
d ]"
      by (metis length_0_conv length_Cons list.exhaust nat.inject)
    from ‹length r = Suc 0› obtain e where r_def: "r = [ e ]"
      by (metis length_0_conv length_Suc_conv)
    from ‹r = [ e ]› and ‹q = [ c, d ]›
    have c: "c ∈ K" "c ≠ 0" and d: "d ∈ K" and e: "e ∈ K" "e ≠ 0"
      using r q subringE(1)[OF assms(1)] unfolding sym[OF univ_poly_carrier]
polynomial_def by auto
    with sym[OF ‹[ a, b ] = q ⊗K[X] r›] have "a = c ⊗ e"
      using poly_mult_lead_coeff[OF assms(1), of q r]
      unfolding polynomial_def sym[OF univ_poly_mult[of R K]] r_def q_def
by auto
    obtain inv_a where a: "a ∈ K" and inv_a: "inv_a ∈ K" "a ⊗ inv_a
= 1" "inv_a ⊗ a = 1"
      using assms(2) unfolding Units_def by auto
    hence "a ≠ 0" and "inv_a ≠ 0"
      using subringE(1)[OF assms(1)] integral_iff by auto
    with ‹c ∈ K› and ‹c ≠ 0› have in_carrier: "[ c ⊗ inv_a ] ∈ carrier
(K[X])"
      using subringE(1,6)[OF assms(1)] inv_a integral
      unfolding sym[OF univ_poly_carrier] polynomial_def
      by (auto, meson subsetD)
    moreover have "[ c ⊗ inv_a ] ⊗K[X] r = [ 1 ]"
      using ‹a = c ⊗ e› a inv_a c e subsetD[OF subringE(1)[OF assms(1)]]
      unfolding r_def univ_poly_mult by (auto) (simp add: m_assoc m_lcomm
integral_iff)+
    ultimately show ?thesis
      using r(1) UP.m_comm[OF in_carrier r(1)] unfolding sym[OF univ_poly_one[of
R K]] Units_def by auto
  qed

  fix q r
  assume q: "q ∈ carrier (K[X])" and r: "r ∈ carrier (K[X])" and qr:
"[ a, b ] = q ⊗K[X] r"
  thus "q ∈ Units (K[X]) ∨ r ∈ Units (K[X])"
    using aux_lemma1[OF q r qr] aux_lemma2[of q r] aux_lemma2[of r q]
UP.m_comm add_is_1 by auto
qed

lemma (in domain) degree_one_imp_pirreducible:
```

```
  assumes "subfield K R" and "p ∈ carrier (K[X])" and "degree p = 1"
  shows "pirreducible K p"
proof -
  from ‹degree p = 1› have "length p = Suc (Suc 0)"
    by simp
  then obtain a b where p: "p = [ a, b ]"
    by (metis length_0_conv length_Cons nat.inject neq_Nil_conv)
  with ‹p ∈ carrier (K[X])› show ?thesis
    using subring_degree_one_imp_pirreducible[OF subfieldE(1)[OF assms(1)],
of a b]
          subfield.subfield_Units[OF assms(1)]
    unfolding sym[OF univ_poly_carrier] polynomial_def by auto
qed

lemma (in ring) degree_oneE[elim]:
  assumes "p ∈ carrier (K[X])" and "degree p = 1"
    and "⋀a b. ⟦ a ∈ K; a ≠ 𝟬; b ∈ K; p = [ a, b ] ⟧ ⟹ P"
  shows P
proof -
  from ‹degree p = 1› have "length p = Suc (Suc 0)"
    by simp
  then obtain a b where "p = [ a, b ]"
    by (metis length_0_conv length_Cons nat.inject neq_Nil_conv)
  with ‹p ∈ carrier (K[X])› have "a ∈ K" and "a ≠ 𝟬" and "b ∈ K"
    unfolding sym[OF univ_poly_carrier] polynomial_def by auto
  with ‹p = [ a, b ]› show ?thesis
    using assms(3) by simp
qed

lemma (in domain) subring_degree_one_associatedI:
  assumes "subring K R" and "a ∈ K" "a' ∈ K" and "b ∈ K" and "a ⊗ a'
= 𝟭"
  shows "[ a , b ] ∼K[X] [ 𝟭, a' ⊗ b ]"
proof -
  from ‹a ⊗ a' = 𝟭› have not_zero: "a ≠ 𝟬" "a' ≠ 𝟬"
    using subringE(1)[OF assms(1)] assms(2-3) by auto
  hence "[ a, b ] = [ a ] ⊗K[X] [ 𝟭, a' ⊗ b ]"
    using assms(2-4)[THEN subsetD[OF subringE(1)[OF assms(1)]]] assms(5)
m_assoc
    unfolding univ_poly_mult by fastforce
  moreover have "[ a, b ] ∈ carrier (K[X])" and "[ 𝟭, a' ⊗ b ] ∈ carrier
(K[X])"
    using subringE(1,3,6)[OF assms(1)] not_zero one_not_zero assms
    unfolding sym[OF univ_poly_carrier] polynomial_def by auto
  moreover have "[ a ] ∈ Units (K[X])"
  proof -
    from ‹a ≠ 𝟬› and ‹a' ≠ 𝟬› have "[ a ] ∈ carrier (K[X])" and "[
a' ] ∈ carrier (K[X])"
      using assms(2-3) unfolding sym[OF univ_poly_carrier] polynomial_def
```

```
by auto
    moreover have "a' ⊗ a = 1"
      using subsetD[OF subringE(1)[OF assms(1)]] assms m_comm by simp

    hence "[ a ] ⊗K[X] [ a' ] = [ 1 ]" and "[ a' ] ⊗K[X] [ a ] = [ 1
]"
      using assms unfolding univ_poly_mult by auto
    ultimately show ?thesis
      unfolding sym[OF univ_poly_one[of R K]] Units_def by blast
  qed
  ultimately show ?thesis
    using domain.ring_associated_iff[OF univ_poly_is_domain[OF assms(1)]]
by blast
qed

lemma (in domain) degree_one_associatedI:
  assumes "subfield K R" and "p ∈ carrier (K[X])" and "degree p = 1"
  shows "p ∼K[X] [ 1, inv (lead_coeff p) ⊗ (const_term p) ]"
proof -
  from ‹p ∈ carrier (K[X])› and ‹degree p = 1›
  obtain a b where "p = [ a, b ]" and "a ∈ K" "a ≠ 0" and "b ∈ K"
    by auto
  thus ?thesis
    using subring_degree_one_associatedI[OF subfieldE(1)[OF assms(1)]]
          subfield_m_inv[OF assms(1)] subsetD[OF subfieldE(3)[OF assms(1)]]
    unfolding const_term_def
    by auto
qed
```

## 35.5 Ideals

```
lemma (in domain) exists_unique_gen:
  assumes "subfield K R" "ideal I (K[X])" "I ≠ { [] }"
  shows "∃!p ∈ carrier (K[X]). lead_coeff p = 1 ∧ I = PIdlK[X] p"
    (is "∃!p. ?generator p")
proof -
  interpret UP: principal_domain "K[X]"
    using univ_poly_is_principal[OF assms(1)] .
  obtain q where q: "q ∈ carrier (K[X])" "I = PIdlK[X] q"
    using UP.exists_gen[OF assms(2)] by blast
  hence not_nil: "q ≠ []"
    using UP.genideal_zero UP.cgenideal_eq_genideal[OF UP.zero_closed]
assms(3)
    by (auto simp add: univ_poly_zero)
  hence "lead_coeff q ∈ K - { 0 }"
    using q(1) unfolding univ_poly_def polynomial_def by auto
  hence inv_lc_q: "inv (lead_coeff q) ∈ K - { 0 }" "inv (lead_coeff q)
⊗ lead_coeff q = 1"
    using subfield_m_inv[OF assms(1)] by auto
```

```
  define p where "p = [ inv (lead_coeff q) ] ⊗K[X] q"
  have is_poly: "polynomial K [ inv (lead_coeff q) ]" "polynomial K q"
    using inv_lc_q(1) q(1) unfolding univ_poly_def polynomial_def by auto
  hence in_carrier: "p ∈ carrier (K[X])"
    using UP.m_closed unfolding univ_poly_carrier p_def by simp
  have lc_p: "lead_coeff p = 1"
    using poly_mult_lead_coeff[OF subfieldE(1)[OF assms(1)] is_poly _
not_nil] inv_lc_q(2)
    unfolding p_def univ_poly_mult[of R K] by simp
  moreover have PIdl_p: "I = PIdlK[X] p"
    using UP.associated_iff_same_ideal[OF in_carrier q(1)] q(2) inv_lc_q(1)
p_def
          associated_polynomials_iff[OF assms(1) in_carrier q(1)]
    by auto
  ultimately have "?generator p"
    using in_carrier by simp

  moreover
  have "⋀r. ⟦ r ∈ carrier (K[X]); lead_coeff r = 1; I = PIdlK[X] r ⟧ ⟹
r = p"
  proof -
    fix r assume r: "r ∈ carrier (K[X])" "lead_coeff r = 1" "I = PIdlK[X]
r"
    have "subring K R"
      by (simp add: ‹subfield K R› subfieldE(1))
    obtain k where k: "k ∈ K - { 0 }" "r = [ k ] ⊗K[X] p"
      using UP.associated_iff_same_ideal[OF r(1) in_carrier] PIdl_p r(3)
            associated_polynomials_iff[OF assms(1) r(1) in_carrier]
      by auto
    hence "polynomial K [ k ]"
      unfolding polynomial_def by simp
    moreover have "p ≠ []"
      using not_nil UP.associated_iff_same_ideal[OF in_carrier q(1)] q(2)
PIdl_p
            associated_polynomials_imp_same_length[OF ‹subring K R› in_carrier
q(1)] by auto
    ultimately have "lead_coeff r = k ⊗ (lead_coeff p)"
      using poly_mult_lead_coeff[OF subfieldE(1)[OF assms(1)]] in_carrier
k(2)
      unfolding univ_poly_def by (auto simp del: poly_mult.simps)
    hence "k = 1"
      using lc_p r(2) k(1) subfieldE(3)[OF assms(1)] by auto
    hence "r = map ((⊗) 1) p"
      using poly_mult_const(1)[OF subfieldE(1)[OF assms(1)] _ k(1), of
p] in_carrier
      unfolding k(2) univ_poly_carrier[of R K] univ_poly_mult[of R K]
by auto
    moreover have "set p ⊆ carrier R"
```

```
      using polynomial_in_carrier[OF subfieldE(1)[OF assms(1)]]
            in_carrier univ_poly_carrier[of R K] by auto
    hence "map ((⊗) 1) p = p"
      by (induct p) (auto)
    ultimately show "r = p" by simp
  qed

  ultimately show ?thesis by blast
qed

proposition (in domain) exists_unique_pirreducible_gen:
  assumes "subfield K R" "ring_hom_ring (K[X]) R h"
    and "a_kernel (K[X]) R h ≠ { [] }" "a_kernel (K[X]) R h ≠ carrier
(K[X])"
  shows "∃!p ∈ carrier (K[X]). pirreducible K p ∧ lead_coeff p = 1 ∧
a_kernel (K[X]) R h = PIdl_K[X] p"
    (is "∃!p. ?generator p")
proof -
  interpret UP: principal_domain "K[X]"
    using univ_poly_is_principal[OF assms(1)] .

  have "ideal (a_kernel (K[X]) R h) (K[X])"
    using ring_hom_ring.kernel_is_ideal[OF assms(2)] .
  then obtain p
    where p: "p ∈ carrier (K[X])" "lead_coeff p = 1" "a_kernel (K[X])
R h = PIdl_K[X] p"
      and unique:
      "⋀q. ⟦ q ∈ carrier (K[X]); lead_coeff q = 1; a_kernel (K[X]) R
h = PIdl_K[X] q ⟧ ⟹ q = p"
    using exists_unique_gen[OF assms(1) _ assms(3)] by metis

  have "p ∈ carrier (K[X]) - { [] }"
      using UP.genideal_zero UP.cgenideal_eq_genideal[OF UP.zero_closed]
assms(3) p(1,3)
      by (auto simp add: univ_poly_zero)
  hence "pprime K p"
    using ring_hom_ring.primeideal_vimage[OF assms(2) UP.is_cring zeroprimeideal]
          UP.primeideal_iff_prime[of p]
    unfolding univ_poly_zero sym[OF p(3)] a_kernel_def' by simp
  hence "pirreducible K p"
    using pprime_iff_pirreducible[OF assms(1) p(1)] by simp
  thus ?thesis
    using p unique by metis
qed

lemma (in domain) cgenideal_pirreducible:
  assumes "subfield K R" and "p ∈ carrier (K[X])" "pirreducible K p"

  shows "⟦ pirreducible K q; q ∈ PIdl_K[X] p ⟧ ⟹ p ∼_K[X] q"
```

```
proof -
  interpret UP: principal_domain "K[X]"
    using univ_poly_is_principal[OF assms(1)] .

  assume q: "pirreducible K q" "q ∈ PIdl_K[X] p"
  hence in_carrier: "q ∈ carrier (K[X])"
    using additive_subgroup.a_subset[OF ideal.axioms(1)[OF UP.cgenideal_ideal[OF
assms(2)]]] by auto
  hence "p divides_K[X] q"
    by (meson q assms(2) UP.cgenideal_ideal UP.cgenideal_minimal UP.to_contain_is_to_divide
  then obtain r where r: "r ∈ carrier (K[X])" "q = p ⊗_K[X] r"
    by auto
  hence "r ∈ Units (K[X])"
    using pirreducibleE(3)[OF _ in_carrier q(1) assms(2) r(1)] subfieldE(1)[OF
assms(1)]
          pirreducibleE(2)[OF _ assms(2-3)] by auto
  thus "p ~_K[X] q"
    using UP.ring_associated_iff[OF in_carrier assms(2)] r(2) UP.associated_sym
    unfolding UP.m_comm[OF assms(2) r(1)] by auto
qed
```

## 35.6 Roots and Multiplicity

```
definition (in ring) is_root :: "'a list ⇒ 'a ⇒ bool"
  where "is_root p x ⟷ (x ∈ carrier R ∧ eval p x = 0 ∧ p ≠ [])"

definition (in ring) alg_mult :: "'a list ⇒ 'a ⇒ nat"
  where "alg_mult p x =
          (if p = [] then 0 else
            (if x ∈ carrier R then Greatest (λ n. ([ 1, ⊖ x ] [^]_poly_ring R
n) pdivides p) else 0))"

definition (in ring) roots :: "'a list ⇒ 'a multiset"
  where "roots p = Abs_multiset (alg_mult p)"

definition (in ring) roots_on :: "'a set ⇒ 'a list ⇒ 'a multiset"
  where "roots_on K p = roots p ∩# mset_set K"

definition (in ring) splitted :: "'a list ⇒ bool"
  where "splitted p ⟷ size (roots p) = degree p"

definition (in ring) splitted_on :: "'a set ⇒ 'a list ⇒ bool"
  where "splitted_on K p ⟷ size (roots_on K p) = degree p"

lemma (in domain) pdivides_imp_root_sharing:
  assumes "p ∈ carrier (poly_ring R)" "p pdivides q" and "a ∈ carrier
R"
  shows "eval p a = 0 ⟹ eval q a = 0"
proof -
```

```
  from ‹p pdivides q› obtain r where r: "q = p ⊗poly_ring R r" "r ∈
carrier (poly_ring R)"
    unfolding pdivides_def factor_def by auto
  hence "eval q a = (eval p a) ⊗ (eval r a)"
    using ring_hom_memE(2)[OF eval_is_hom[OF carrier_is_subring assms(3)]
assms(1) r(2)] by simp
  thus "eval p a = 0 ⟹ eval q a = 0"
    using ring_hom_memE(1)[OF eval_is_hom[OF carrier_is_subring assms(3)]
r(2)] by auto
qed

lemma (in domain) degree_one_root:
  assumes "subfield K R" and "p ∈ carrier (K[X])" and "degree p = 1"
  shows "eval p (⊖ (inv (lead_coeff p) ⊗ (const_term p))) = 0"
    and "inv (lead_coeff p) ⊗ (const_term p) ∈ K"
proof -
  from ‹degree p = 1› have "length p = Suc (Suc 0)"
    by simp
  then obtain a b where p: "p = [ a, b ]"
    by (metis (no_types, opaque_lifting) Suc_length_conv length_0_conv)
  hence "a ∈ K - { 0 }" "b ∈ K"  and in_carrier: "a ∈ carrier R" "b
∈ carrier R"
    using assms(2) subfieldE(3)[OF assms(1)] unfolding sym[OF univ_poly_carrier]
polynomial_def by auto
  hence inv_a: "inv a ∈ carrier R" "a ⊗ inv a = 1" and "inv a ∈ K"
    using subfield_m_inv(1-2)[OF assms(1), of a] subfieldE(3)[OF assms(1)]
by auto
  hence "eval p (⊖ (inv a ⊗ b)) = a ⊗ (⊖ (inv a ⊗ b)) ⊕ b"
    using in_carrier unfolding p by simp
  also have " ... = ⊖ (a ⊗ (inv a ⊗ b)) ⊕ b"
    using inv_a in_carrier by (simp add: r_minus)
  also have " ... = 0"
    using in_carrier(2) unfolding sym[OF m_assoc[OF in_carrier(1) inv_a(1)
in_carrier(2)]] inv_a(2) by algebra
  finally have "eval p (⊖ (inv a ⊗ b)) = 0" .
  moreover have ct: "const_term p = b"
    using in_carrier unfolding p const_term_def by auto
  ultimately show "eval p (⊖ (inv (lead_coeff p) ⊗ (const_term p))) =
0"
    unfolding p by simp
  from ‹inv a ∈ K› and ‹b ∈ K›
  show "inv (lead_coeff p) ⊗ (const_term p) ∈ K"
    using p subringE(6)[OF subfieldE(1)[OF assms(1)]] unfolding ct by
auto
qed
lemma (in domain) is_root_imp_pdivides:
  assumes "p ∈ carrier (poly_ring R)"
  shows "is_root p x ⟹ [ 1, ⊖ x ] pdivides p"
proof -
```

```
  let ?b = "[ 1 , ⊖ x ]"

  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .

  assume "is_root p x" hence x: "x ∈ carrier R" and is_root: "eval p
x = 0"
    unfolding is_root_def by auto
  hence b: "?b ∈ carrier (poly_ring R)"
    unfolding sym[OF univ_poly_carrier] polynomial_def by auto
  then obtain q r where q: "q ∈ carrier (poly_ring R)" and r: "r ∈ carrier
(poly_ring R)"
    and long_divides: "p = (?b ⊗poly_ring R q) ⊕poly_ring R r" "r = []
∨ degree r < degree ?b"
    using long_division_theorem[OF carrier_is_subring, of p ?b] assms
by (auto simp add: univ_poly_carrier)

  show ?thesis
  proof (cases "r = []")
    case True then have "r = 0poly_ring R"
      unfolding univ_poly_zero[of R "carrier R"] .
    thus ?thesis
      using long_divides(1) q r b dividesI[OF q, of p ?b] by (simp add:
pdivides_def)
  next
    case False then have "length r = Suc 0"
      using long_divides(2) le_SucE by fastforce
    then obtain a where "r = [ a ]" and a: "a ∈ carrier R" and "a ≠
0"
      using r unfolding sym[OF univ_poly_carrier] polynomial_def
      by (metis False length_0_conv length_Suc_conv list.sel(1) list.set_sel(1)
subset_code(1))

    have "eval p x = ((eval ?b x) ⊗ (eval q x)) ⊕ (eval r x)"
      using long_divides(1) ring_hom_memE[OF eval_is_hom[OF carrier_is_subring
x]] by (simp add: b q r)
    also have " ... = eval r x"
      using ring_hom_memE[OF eval_is_hom[OF carrier_is_subring x]] x b
q r by (auto, algebra)
    finally have "a = 0"
      using a unfolding ‹r = [ a ]› is_root by simp
    with ‹a ≠ 0› have False .. thus ?thesis ..
  qed
qed

lemma (in domain) pdivides_imp_is_root:
  assumes "p ≠ []" and "x ∈ carrier R"
  shows "[ 1, ⊖ x ] pdivides p ⟹ is_root p x"
proof -
```

```
  assume "[ 1, ⊖ x ] pdivides p"
  then obtain q where q: "q ∈ carrier (poly_ring R)" and pdiv: "p =
[ 1, ⊖ x ] ⊗poly_ring R q"
    unfolding pdivides_def by auto
  moreover have "[ 1, ⊖ x ] ∈ carrier (poly_ring R)"
    using assms(2) unfolding sym[OF univ_poly_carrier] polynomial_def
by simp
  ultimately have "eval p x = 0"
    using ring_hom_memE[OF eval_is_hom[OF carrier_is_subring, of x]] assms(2)
by (auto, algebra)
  with ‹p ≠ []› and ‹x ∈ carrier R› show "is_root p x"
    unfolding is_root_def by simp
qed

lemma (in domain) associated_polynomials_imp_same_is_root:
  assumes "p ∈ carrier (poly_ring R)" and "q ∈ carrier (poly_ring R)"
and "p ∼poly_ring R q"
  shows "is_root p x ⟷ is_root q x"
proof (cases "p = []")
  case True with ‹p ∼poly_ring R q› have "q = []"
    unfolding associated_def True factor_def univ_poly_def by auto
  thus ?thesis
    using True unfolding is_root_def by simp
next
  case False
  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .

  have "is_root q x"
    if p: "p ∈ carrier (poly_ring R)"
    and q: "q ∈ carrier (poly_ring R)"
    and pq: "p ∼poly_ring R q"
    and is_root: "is_root p x"
    for p q
  proof -
    from is_root have "[ 1, ⊖ x ] pdivides p" and "p ≠ []" and "x ∈
carrier R"
      using is_root_imp_pdivides[OF p] unfolding is_root_def by auto
    moreover have "[ 1, ⊖ x ] ∈ carrier (poly_ring R)"
      using is_root unfolding is_root_def sym[OF univ_poly_carrier] polynomial_def
by simp
    ultimately have "[ 1, ⊖ x ] pdivides q"
      using UP.divides_cong_r[OF _ pq ] unfolding pdivides_def by simp
    with ‹p ≠ []› and ‹x ∈ carrier R› show ?thesis
      using associated_polynomials_imp_same_length[OF carrier_is_subring
p q pq]
        pdivides_imp_is_root[of q x]
      by fastforce
  qed
```

```
  then show ?thesis
    using assms UP.associated_sym[OF assms(3)] by blast
qed

lemma (in ring) monic_degree_one_root_condition:
  assumes "a ∈ carrier R" shows "is_root [ 𝟭, ⊖ a ] b ⟷ a = b"
  using assms minus_equality r_neg[OF assms] unfolding is_root_def by
(auto, fastforce)

lemma (in field) degree_one_root_condition:
  assumes "p ∈ carrier (poly_ring R)" and "degree p = 1"
  shows "is_root p x ⟷ x = ⊖ (inv (lead_coeff p) ⊗ (const_term p))"
proof -
  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .

  from ‹degree p = 1› have "length p = Suc (Suc 0)"
    by simp
  then obtain a b where p: "p = [ a, b ]"
    by (metis length_0_conv length_Cons list.exhaust nat.inject)
  hence a: "a ∈ carrier R" "a ≠ 𝟬" and b: "b ∈ carrier R"
    using assms(1) unfolding sym[OF univ_poly_carrier] polynomial_def
by auto
  hence inv_a: "inv a ∈ carrier R" "(inv a) ⊗ a = 𝟭"
    using subfield_m_inv[OF carrier_is_subfield, of a] by auto
  hence in_carrier: "[ 𝟭, (inv a) ⊗ b ] ∈ carrier (poly_ring R)"
    using b unfolding sym[OF univ_poly_carrier] polynomial_def by auto

  have "p ∼poly_ring R [ 𝟭, (inv a) ⊗ b ]"
  proof (rule UP.associatedI2'[OF _ _ in_carrier, of _ "[ a ]"])
    have "p = [ a ] ⊗poly_ring R [ 𝟭, inv a ⊗ b ]"
      using a inv_a b m_assoc[of a "inv a" b] unfolding p univ_poly_mult
by (auto, algebra)
    also have " ... = [ 𝟭, inv a ⊗ b ] ⊗poly_ring R [ a ]"
      using UP.m_comm[OF in_carrier, of "[ a ]"] a
      by (auto simp add: sym[OF univ_poly_carrier] polynomial_def)
    finally show "p = [ 𝟭, inv a ⊗ b ] ⊗poly_ring R [ a ]" .
  next
    from ‹a ∈ carrier R› and ‹a ≠ 𝟬› show "[ a ] ∈ Units (poly_ring
R)"
      unfolding univ_poly_units[OF carrier_is_subfield] by simp
  qed

  moreover have "(inv a) ⊗ b = ⊖ (⊖ (inv (lead_coeff p) ⊗ (const_term
p)))"
    and "inv (lead_coeff p) ⊗ (const_term p) ∈ carrier R"
    using inv_a a b unfolding p const_term_def by auto

  ultimately show ?thesis
```

```
    using associated_polynomials_imp_same_is_root[OF assms(1) in_carrier]
          monic_degree_one_root_condition
    by (metis add.inv_closed)
qed

lemma (in domain) is_root_poly_mult_imp_is_root:
  assumes "p ∈ carrier (poly_ring R)" and "q ∈ carrier (poly_ring R)"
  shows "is_root (p ⊗poly_ring R q) x ⟹ (is_root p x) ∨ (is_root q x)"
proof -
  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .

  assume is_root: "is_root (p ⊗poly_ring R q) x"
  hence "p ≠ []" and "q ≠ []"
    unfolding is_root_def sym[OF univ_poly_zero[of R "carrier R"]]
    using UP.l_null[OF assms(2)] UP.r_null[OF assms(1)] by blast+
  moreover have x: "x ∈ carrier R" and "eval (p ⊗poly_ring R q) x = 0"
    using is_root unfolding is_root_def by simp+
  hence "eval p x = 0 ∨ eval q x = 0"
    using ring_hom_memE[OF eval_is_hom[OF carrier_is_subring], of x] assms
integral by auto
  ultimately show "(is_root p x) ∨ (is_root q x)"
    using x unfolding is_root_def by auto
qed

lemma (in domain) degree_zero_imp_not_is_root:
  assumes "p ∈ carrier (poly_ring R)" and "degree p = 0" shows "¬ is_root
p x"
proof (cases "p = []", simp add: is_root_def)
  case False with ‹degree p = 0› have "length p = Suc 0"
    using le_SucE by fastforce
  then obtain a where "p = [ a ]" and "a ∈ carrier R" and "a ≠ 0"
    using assms unfolding sym[OF univ_poly_carrier] polynomial_def
    by (metis False length_0_conv length_Suc_conv list.sel(1) list.set_sel(1)
subset_code(1))
  thus ?thesis
    unfolding is_root_def by auto
qed

lemma (in domain) finite_number_of_roots:
  assumes "p ∈ carrier (poly_ring R)" shows "finite { x. is_root p x
}"
  using assms
proof (induction "degree p" arbitrary: p)
  case 0 thus ?case
    by (simp add: degree_zero_imp_not_is_root)
next
  case (Suc n) show ?case
  proof (cases "{ x. is_root p x } = {}")
```

```
    case True thus ?thesis
      by (simp add: True)
  next
    interpret UP: domain "poly_ring R"
      using univ_poly_is_domain[OF carrier_is_subring] .

    case False
    then obtain a where is_root: "is_root p a"
      by blast
    hence a: "a ∈ carrier R" and eval: "eval p a = 𝟬" and p_not_zero:
"p ≠ []"
      unfolding is_root_def by auto
    hence in_carrier: "[ 𝟭, ⊖ a ] ∈ carrier (poly_ring R)"
      unfolding sym[OF univ_poly_carrier] polynomial_def by auto

    obtain q where q: "q ∈ carrier (poly_ring R)" and p: "p = [ 𝟭, ⊖
a ] ⊗poly_ring R q"
      using is_root_imp_pdivides[OF Suc(3) is_root] unfolding pdivides_def
by auto
    with ‹p ≠ []› have q_not_zero: "q ≠ []"
      using UP.r_null UP.integral in_carrier unfolding sym[OF univ_poly_zero[of
R "carrier R"]]
      by metis
    hence "degree q = n"
      using poly_mult_degree_eq[OF carrier_is_subring, of "[ 𝟭, ⊖ a ]"
q]
            in_carrier q p_not_zero p Suc(2)
      unfolding univ_poly_carrier
      by (metis One_nat_def Suc_eq_plus1 diff_Suc_1 list.distinct(1)
                list.size(3-4) plus_1_eq_Suc univ_poly_mult)
    hence "finite { x. is_root q x }"
      using Suc(1)[OF _ q] by simp

    moreover have "{ x. is_root p x } ⊆ insert a { x. is_root q x }"
      using is_root_poly_mult_imp_is_root[OF in_carrier q]
            monic_degree_one_root_condition[OF a]
      unfolding p by auto

    ultimately show ?thesis
      using finite_subset by auto
  qed
qed

lemma (in domain) alg_multE:
  assumes "x ∈ carrier R" and "p ∈ carrier (poly_ring R)" and "p ≠
[]"
  shows "([ 𝟭, ⊖ x ] [^]poly_ring R (alg_mult p x)) pdivides p"
    and "⋀n. ([ 𝟭, ⊖ x ] [^]poly_ring R n) pdivides p ⟹ n ≤ alg_mult
p x"
```

```
proof -
  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .

  let ?ppow = "λn :: nat. ([ 1, ⊖ x ] [^]poly_ring R n)"

  define S :: "nat set" where "S = { n. ?ppow n pdivides p }"
  have "?ppow 0 = 1poly_ring R"
    using UP.nat_pow_0 by simp
  hence "0 ∈ S"
    using UP.one_divides[OF assms(2)] unfolding S_def pdivides_def by
simp
  hence "S ≠ {}"
    by auto

  moreover have "n ≤ degree p" if "n ∈ S" for n :: nat
  proof -
    have "[ 1, ⊖ x ] ∈ carrier (poly_ring R)"
      using assms unfolding sym[OF univ_poly_carrier] polynomial_def by
auto
    hence "?ppow n ∈ carrier (poly_ring R)"
      using assms unfolding univ_poly_zero by auto
    with ‹n ∈ S› have "degree (?ppow n) ≤ degree p"
      using pdivides_imp_degree_le[OF carrier_is_subring _ assms(2-3),
of "?ppow n"] by (simp add: S_def)
    with ‹[ 1, ⊖ x ] ∈ carrier (poly_ring R)› show ?thesis
      using polynomial_pow_degree by simp
  qed
  hence "finite S"
    using finite_nat_set_iff_bounded_le by blast

  ultimately have MaxS: "⋀n. n ∈ S ⟹ n ≤ Max S" "Max S ∈ S"
    using Max_ge[of S] Max_in[of S] by auto
  with ‹x ∈ carrier R› have "alg_mult p x = Max S"
    using Greatest_equality[of "λn. ?ppow n pdivides p" "Max S"] assms(3)
    unfolding S_def alg_mult_def by auto
  thus "([ 1, ⊖ x ] [^]poly_ring R (alg_mult p x)) pdivides p"
   and "⋀n. ([ 1, ⊖ x ] [^]poly_ring R n) pdivides p ⟹ n ≤ alg_mult
p x"
    using MaxS unfolding S_def by auto
qed

lemma (in domain) le_alg_mult_imp_pdivides:
  assumes "x ∈ carrier R" and "p ∈ carrier (poly_ring R)"
  shows "n ≤ alg_mult p x ⟹ ([ 1, ⊖ x ] [^]poly_ring R n) pdivides
p"
proof -
  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .
```

```
  assume le_alg_mult: "n ≤ alg_mult p x"
  have in_carrier: "[ 1, ⊖ x ] ∈ carrier (poly_ring R)"
    using assms(1) unfolding sym[OF univ_poly_carrier] polynomial_def
by auto
  hence ppow_pdivides:
    "([ 1, ⊖ x ] [^]poly_ring R n) pdivides
     ([ 1, ⊖ x ] [^]poly_ring R (alg_mult p x))"
    using polynomial_pow_division[OF _ le_alg_mult] by simp

  show ?thesis
  proof (cases "p = []")
    case True thus ?thesis
      using in_carrier pdivides_zero[OF carrier_is_subring] by auto
  next
    case False thus ?thesis
      using ppow_pdivides UP.divides_trans UP.nat_pow_closed alg_multE(1)[OF
assms] in_carrier
      unfolding pdivides_def by meson
  qed
qed

lemma (in domain) alg_mult_gt_zero_iff_is_root:
  assumes "p ∈ carrier (poly_ring R)" shows "alg_mult p x > 0 ⟷ is_root
p x"
proof -
  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .
  show ?thesis
  proof
    assume is_root: "is_root p x" hence x: "x ∈ carrier R" and not_zero:
"p ≠ []"
      unfolding is_root_def by auto
    have "[1, ⊖ x] [^]poly_ring R (Suc 0) = [1, ⊖ x]"
      using x unfolding univ_poly_def by auto
    thus "alg_mult p x > 0"
      using is_root_imp_pdivides[OF _ is_root] alg_multE(2)[OF x, of p
"Suc 0"] not_zero assms by auto
  next
    assume gt_zero: "alg_mult p x > 0"
    hence x: "x ∈ carrier R" and not_zero: "p ≠ []"
      unfolding alg_mult_def by (cases "p = []", auto, cases "x ∈ carrier
R", auto)
    hence in_carrier: "[ 1, ⊖ x ] ∈ carrier (poly_ring R)"
      unfolding sym[OF univ_poly_carrier] polynomial_def by auto
    with ‹x ∈ carrier R› have "[ 1, ⊖ x ] pdivides p" and "eval [ 1,
⊖ x ] x = 0"
      using le_alg_mult_imp_pdivides[of x p "1::nat"] gt_zero assms by
(auto, algebra)
```

```
    thus "is_root p x"
      using pdivides_imp_root_sharing[OF in_carrier] not_zero x by (simp
add: is_root_def)
  qed
qed

lemma (in domain) alg_mult_eq_count_roots:
  assumes "p ∈ carrier (poly_ring R)" shows "alg_mult p = count (roots
p)"
  using finite_number_of_roots[OF assms]
  unfolding sym[OF alg_mult_gt_zero_iff_is_root[OF assms]]
  by (simp add: roots_def)

lemma (in domain) roots_mem_iff_is_root:
  assumes "p ∈ carrier (poly_ring R)" shows "x ∈# roots p ⟷ is_root
p x"
  using alg_mult_eq_count_roots[OF assms] count_greater_zero_iff
  unfolding roots_def sym[OF alg_mult_gt_zero_iff_is_root[OF assms]] by
metis

lemma (in domain) degree_zero_imp_empty_roots:
  assumes "p ∈ carrier (poly_ring R)" and "degree p = 0" shows "roots
p = {#}"
  using degree_zero_imp_not_is_root[of p] roots_mem_iff_is_root[of p]
assms by auto

lemma (in domain) degree_zero_imp_splitted:
  assumes "p ∈ carrier (poly_ring R)" and "degree p = 0" shows "splitted
p"
  unfolding splitted_def degree_zero_imp_empty_roots[OF assms] assms(2)
by simp

lemma (in domain) roots_inclI':
  assumes "p ∈ carrier (poly_ring R)" and "⋀a. ⟦ a ∈ carrier R; p ≠
[] ⟧ ⟹ alg_mult p a ≤ count m a"
  shows "roots p ⊆# m"
proof (intro mset_subset_eqI)
  fix a show "count (roots p) a ≤ count m a"
    using assms unfolding sym[OF alg_mult_eq_count_roots[OF assms(1)]]
alg_mult_def
    by (cases "p = []", simp, cases "a ∈ carrier R", auto)
qed

lemma (in domain) roots_inclI:
  assumes "p ∈ carrier (poly_ring R)" and "q ∈ carrier (poly_ring R)"
"q ≠ []"
    and "⋀a. ⟦ a ∈ carrier R; p ≠ [] ⟧ ⟹ ([ 𝟭, ⊖ a ] [^]poly_ring R
(alg_mult p a)) pdivides q"
  shows "roots p ⊆# roots q"
```

```
  using roots_inclI'[OF assms(1), of "roots q"] assms alg_multE(2)[OF
_ assms(2-3)]
  unfolding sym[OF alg_mult_eq_count_roots[OF assms(2)]] by auto

lemma (in domain) pdivides_imp_roots_incl:
  assumes "p ∈ carrier (poly_ring R)" and "q ∈ carrier (poly_ring R)"
"q ≠ []"
  shows "p pdivides q ⟹ roots p ⊆# roots q"
proof (rule roots_inclI[OF assms])
  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .

  fix a assume "p pdivides q" and a: "a ∈ carrier R"
  hence "[ 1 , ⊖ a ] ∈ carrier (poly_ring R)"
    unfolding sym[OF univ_poly_carrier] polynomial_def by simp
  with ‹p pdivides q› show "([1, ⊖ a] [^]poly_ring R (alg_mult p a))
pdivides q"
    using UP.divides_trans[of _p q] le_alg_mult_imp_pdivides[OF a assms(1)]
    by (auto simp add: pdivides_def)
qed

lemma (in domain) associated_polynomials_imp_same_roots:
  assumes "p ∈ carrier (poly_ring R)" and "q ∈ carrier (poly_ring R)"
and "p ∼poly_ring R q"
  shows "roots p = roots q"
  using assms pdivides_imp_roots_incl zero_pdivides
  unfolding pdivides_def associated_def
  by (metis subset_mset.eq_iff)

lemma (in domain) monic_degree_one_roots:
  assumes "a ∈ carrier R" shows "roots [ 1 , ⊖ a ] = {# a #}"
proof -
  let ?p = "[ 1 , ⊖ a ]"

  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .

  from ‹a ∈ carrier R› have in_carrier: "?p ∈ carrier (poly_ring R)"
    unfolding sym[OF univ_poly_carrier] polynomial_def by simp
  show ?thesis
  proof (rule subset_mset.antisym)
    show "{# a #} ⊆# roots ?p"
      using roots_mem_iff_is_root[OF in_carrier]
            monic_degree_one_root_condition[OF assms]
      by simp
  next
    show "roots ?p ⊆# {# a #}"
    proof (rule mset_subset_eqI, auto)
      fix b assume "a ≠ b" thus "count (roots ?p) b = 0"
```

```
        using alg_mult_gt_zero_iff_is_root[OF in_carrier]
              monic_degree_one_root_condition[OF assms]
        unfolding sym[OF alg_mult_eq_count_roots[OF in_carrier]]
        by fastforce
    next
      have "(?p [^]poly_ring R (alg_mult ?p a)) pdivides ?p"
        using le_alg_mult_imp_pdivides[OF assms in_carrier] by simp
      hence "degree (?p [^]poly_ring R (alg_mult ?p a)) ≤ degree ?p"
        using pdivides_imp_degree_le[OF carrier_is_subring, of _ ?p] in_carrier
by auto
      thus "count (roots ?p) a ≤ Suc 0"
        using polynomial_pow_degree[OF in_carrier]
        unfolding sym[OF alg_mult_eq_count_roots[OF in_carrier]]
        by auto
    qed
  qed
qed

lemma (in domain) degree_one_roots:
  assumes "a ∈ carrier R" "a' ∈ carrier R" and "b ∈ carrier R" and "a
⊗ a' = 1"
  shows "roots [ a , b ] = {# ⊖ (a' ⊗ b) #}"
proof -
  have "[ a, b ] ∈ carrier (poly_ring R)" and "[ 1, a' ⊗ b ] ∈ carrier
(poly_ring R)"
    using assms unfolding sym[OF univ_poly_carrier] polynomial_def by
auto
  thus ?thesis
    using subring_degree_one_associatedI[OF carrier_is_subring assms]
assms
          monic_degree_one_roots associated_polynomials_imp_same_roots
    by (metis add.inv_closed local.minus_minus m_closed)
qed

lemma (in field) degree_one_imp_singleton_roots:
  assumes "p ∈ carrier (poly_ring R)" and "degree p = 1"
  shows "roots p = {# ⊖ (inv (lead_coeff p) ⊗ (const_term p)) #}"
proof -
  from ‹p ∈ carrier (poly_ring R)› and ‹degree p = 1›
  obtain a b where "p = [ a, b ]" and "a ∈ carrier R" "a ≠ 0" and "b
∈ carrier R"
    by auto
  thus ?thesis
    using degree_one_roots[of a "inv a" b]
    by (auto simp add: const_term_def field_Units)
qed

lemma (in field) degree_one_imp_splitted:
  assumes "p ∈ carrier (poly_ring R)" and "degree p = 1" shows "splitted
```

```
p"
  using degree_one_imp_singleton_roots[OF assms] assms(2) unfolding splitted_def
by simp

lemma (in field) no_roots_imp_same_roots:
  assumes "p ∈ carrier (poly_ring R)" "p ≠ []" and "q ∈ carrier (poly_ring
R)"
  shows "roots p = {#} ⟹ roots (p ⊗poly_ring R q) = roots q"
proof -
  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .

  assume no_roots: "roots p = {#}" show "roots (p ⊗poly_ring R q) = roots
q"
  proof (intro subset_mset.antisym)
    have pdiv: "q pdivides (p ⊗poly_ring R q)"
      using UP.divides_prod_l assms unfolding pdivides_def by blast
    show "roots q ⊆# roots (p ⊗poly_ring R q)"
      using pdivides_imp_roots_incl[OF _ _ _ pdiv] assms
            degree_zero_imp_empty_roots[OF assms(3)]
      by (cases "q = []", auto, metis UP.l_null UP.m_rcancel UP.zero_closed
univ_poly_zero)
  next
    show "roots (p ⊗poly_ring R q) ⊆# roots q"
    proof (cases "p ⊗poly_ring R q = []")
      case True thus ?thesis
        using degree_zero_imp_empty_roots[OF UP.m_closed[OF assms(1,3)]]
by simp
    next
      case False with ‹p ≠ []› have q_not_zero: "q ≠ []"
        by (metis UP.r_null assms(1) univ_poly_zero)
      show ?thesis
      proof (rule roots_inclI[OF UP.m_closed[OF assms(1,3)] assms(3) q_not_zero])
        fix a assume a: "a ∈ carrier R"
        hence "¬ ([ 1, ⊖ a ] pdivides p)"
          using assms(1-2) no_roots pdivides_imp_is_root roots_mem_iff_is_root[of
p] by auto
        moreover have in_carrier: "[ 1, ⊖ a ] ∈ carrier (poly_ring R)"
          using a unfolding sym[OF univ_poly_carrier] polynomial_def by
auto
        hence "pirreducible (carrier R) [ 1, ⊖ a ]"
          using degree_one_imp_pirreducible[OF carrier_is_subfield] by
simp
        moreover
        have "([ 1, ⊖ a ] [^]poly_ring R (alg_mult (p ⊗poly_ring R q) a))
pdivides (p ⊗poly_ring R q)"
          using le_alg_mult_imp_pdivides[OF a UP.m_closed, of p q] assms
by simp
        ultimately show "([ 1, ⊖ a ] [^]poly_ring R (alg_mult (p ⊗poly_ring R
```

```
q) a)) pdivides q"
          using pirreducible_pow_pdivides_iff[OF carrier_is_subfield in_carrier]
assms by auto
      qed
    qed
  qed
qed

lemma (in field) poly_mult_degree_one_monic_imp_same_roots:
  assumes "a ∈ carrier R" and "p ∈ carrier (poly_ring R)" "p ≠ []"
  shows "roots ([ 1, ⊖ a ] ⊗poly_ring R p) = add_mset a (roots p)"
proof -
  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .

  from ‹a ∈ carrier R› have in_carrier: "[ 1, ⊖ a ] ∈ carrier (poly_ring
R)"
    unfolding sym[OF univ_poly_carrier] polynomial_def by simp

  show ?thesis
  proof (intro subset_mset.antisym[OF roots_inclI' mset_subset_eqI])
    show "([ 1, ⊖ a ] ⊗poly_ring R p) ∈ carrier (poly_ring R)"
      using in_carrier assms(2) by simp
  next
    fix b assume b: "b ∈ carrier R" and "[ 1, ⊖ a ] ⊗poly_ring R p ≠
[]"
    hence not_zero: "p ≠ []"
      unfolding univ_poly_def by auto
    from ‹b ∈ carrier R› have in_carrier': "[ 1, ⊖ b ] ∈ carrier (poly_ring
R)"
      unfolding sym[OF univ_poly_carrier] polynomial_def by simp
    show "alg_mult ([ 1, ⊖ a ] ⊗poly_ring R p) b ≤ count (add_mset a
(roots p)) b"
    proof (cases "a = b")
      case False
      hence "¬ [ 1, ⊖ b ] pdivides [ 1, ⊖ a ]"
        using assms(1) b monic_degree_one_root_condition pdivides_imp_is_root
by blast
      moreover have "pirreducible (carrier R) [ 1, ⊖ b ]"
        using degree_one_imp_pirreducible[OF carrier_is_subfield in_carrier']
by simp
      ultimately
      have "[ 1, ⊖ b ] [^]poly_ring R (alg_mult ([ 1, ⊖ a ] ⊗poly_ring R
p) b) pdivides p"
        using le_alg_mult_imp_pdivides[OF b UP.m_closed, of _ p] assms(2)
in_carrier
            pirreducible_pow_pdivides_iff[OF carrier_is_subfield in_carrier'
in_carrier, of p]
        by auto
```

```
      with ‹a ≠ b› show ?thesis
        using alg_mult_eq_count_roots[OF assms(2)] alg_multE(2)[OF b assms(2)
not_zero] by auto
    next
      case True
      have "[ 𝟭, ⊖ a ] pdivides ([ 𝟭, ⊖ a ] ⊗poly_ring R p)"
        using dividesI[OF assms(2)] unfolding pdivides_def by auto
      with ‹[ 𝟭, ⊖ a ] ⊗poly_ring R p ≠ []›
      have "alg_mult ([ 𝟭, ⊖ a ] ⊗poly_ring R p) a ≥ Suc 0"
        using alg_multE(2)[of a _ "Suc 0"] in_carrier assms by auto
      then obtain m where m: "alg_mult ([ 𝟭, ⊖ a ] ⊗poly_ring R p) a
= Suc m"
        using Suc_le_D by blast
      hence "([ 𝟭, ⊖ a ] ⊗poly_ring R ([ 𝟭, ⊖ a ] [^]poly_ring R m)) pdivides
             ([ 𝟭, ⊖ a ] ⊗poly_ring R p)"
        using le_alg_mult_imp_pdivides[OF _ UP.m_closed, of a _ p]
              in_carrier assms UP.nat_pow_Suc2 by force
      hence "([ 𝟭, ⊖ a ] [^]poly_ring R m) pdivides p"
        using UP.mult_divides in_carrier assms(2)
        unfolding univ_poly_zero pdivides_def factor_def
        by (simp add: UP.m_assoc UP.m_lcancel univ_poly_zero)
      with ‹a = b› show ?thesis
        using alg_mult_eq_count_roots assms in_carrier UP.nat_pow_Suc2

              alg_multE(2)[OF assms(1) _ not_zero] m
        by auto
    qed
  next
    fix b
    have not_zero: "[ 𝟭, ⊖ a ] ⊗poly_ring R p ≠ []"
      using assms in_carrier univ_poly_zero[of R] UP.integral by auto

    show "count (add_mset a (roots p)) b ≤ count (roots ([𝟭, ⊖ a] ⊗poly_ring R
p)) b"
    proof (cases "a = b")
      case True
      have "([ 𝟭, ⊖ a ] ⊗poly_ring R ([ 𝟭, ⊖ a ] [^]poly_ring R (alg_mult
p a))) pdivides
            ([ 𝟭, ⊖ a ] ⊗poly_ring R p)"
        using UP.divides_mult[OF _ in_carrier] le_alg_mult_imp_pdivides[OF
assms(1,2)] in_carrier assms
        by (auto simp add: pdivides_def)
      with ‹a = b› show ?thesis
        using alg_mult_eq_count_roots assms in_carrier UP.nat_pow_Suc2

              alg_multE(2)[OF assms(1) _ not_zero]
        by auto
    next
      case False
```

```
      have "p pdivides ([ 1, ⊖ a ] ⊗poly_ring R p)"
        using dividesI[OF in_carrier] UP.m_comm in_carrier assms unfold-
ing pdivides_def by auto
      thus ?thesis
        using False pdivides_imp_roots_incl assms in_carrier not_zero
        by (simp add: subseteq_mset_def)
    qed
  qed
qed

lemma (in domain) not_empty_rootsE[elim]:
  assumes "p ∈ carrier (poly_ring R)" and "roots p ≠ {#}"
    and "⋀a. ⟦ a ∈ carrier R; a ∈# roots p;
              [ 1, ⊖ a ] ∈ carrier (poly_ring R); [ 1, ⊖ a ] pdivides
p ⟧ ⟹ P"
  shows P
proof -
  from ‹roots p ≠ {#}› obtain a where "a ∈# roots p"
    by blast
  with ‹p ∈ carrier (poly_ring R)› have "[ 1, ⊖ a ] pdivides p"
    and "[ 1, ⊖ a ] ∈ carrier (poly_ring R)" and "a ∈ carrier R"
    using is_root_imp_pdivides[of p] roots_mem_iff_is_root[of p] is_root_def[of
p a]
    unfolding sym[OF univ_poly_carrier] polynomial_def by auto
  with ‹a ∈# roots p› show ?thesis
    using assms(3)[of a] by auto
qed

lemma (in field) associated_polynomials_imp_same_roots:
  assumes "p ∈ carrier (poly_ring R)" "p ≠ []" and "q ∈ carrier (poly_ring
R)" "q ≠ []"
  shows "roots (p ⊗poly_ring R q) = roots p + roots q"
proof -
  interpret UP: domain "poly_ring R"
    using univ_poly_is_domain[OF carrier_is_subring] .
  from assms show ?thesis
  proof (induction "degree p" arbitrary: p rule: less_induct)
    case less show ?case
    proof (cases "roots p = {#}")
      case True thus ?thesis
        using no_roots_imp_same_roots[of p q] less by simp
    next
      case False with ‹p ∈ carrier (poly_ring R)›
      obtain a where a: "a ∈ carrier R" and "a ∈# roots p" and pdiv:
"[ 1, ⊖ a ] pdivides p"
          and in_carrier: "[ 1, ⊖ a ] ∈ carrier (poly_ring R)"
        by blast
      show ?thesis
      proof (cases "degree p = 1")
```

```
        case True with ‹p ∈ carrier (poly_ring R)›
        obtain b c where p: "p = [ b, c ]" and b: "b ∈ carrier R" "b
≠ 0" and c: "c ∈ carrier R"
          by auto
        with ‹a ∈# roots p› have roots: "roots p = {# a #}" and a: "⊖
a = inv b ⊗ c" "a ∈ carrier R"
          and lead: "lead_coeff p = b" and const: "const_term p = c"
          using degree_one_imp_singleton_roots[of p] less(2) field_Units
          unfolding const_term_def by auto
        hence "(p ⊗poly_ring R q) ∼poly_ring R ([ 1, ⊖ a ] ⊗poly_ring R
q)"
          using UP.mult_cong_l[OF degree_one_associatedI[OF carrier_is_subfield
_ True]] less(2,4)
          by (auto simp add: a lead const)
        hence "roots (p ⊗poly_ring R q) = roots ([ 1, ⊖ a ] ⊗poly_ring R
q)"
          using associated_polynomials_imp_same_roots in_carrier less(2,4)
unfolding a by simp
        thus ?thesis
          unfolding poly_mult_degree_one_monic_imp_same_roots[OF a(2)
less(4,5)] roots by simp
      next
        case False
        from ‹[ 1, ⊖ a ] pdivides p›
        obtain r where p: "p = [ 1, ⊖ a ] ⊗poly_ring R r" and r: "r ∈
carrier (poly_ring R)"
          unfolding pdivides_def by auto
        with ‹p ≠ []› have not_zero: "r ≠ []"
          using in_carrier univ_poly_zero[of R "carrier R"] UP.integral_iff
by auto
        with ‹p = [ 1, ⊖ a ] ⊗poly_ring R r› have deg: "degree p = Suc
(degree r)"
          using poly_mult_degree_eq[OF carrier_is_subring, of _ r] in_carrier
r
          unfolding univ_poly_carrier sym[OF univ_poly_mult[of R "carrier
R"]] by auto
        with ‹r ≠ []› and ‹q ≠ []› have "r ⊗poly_ring R q ≠ []"
          using in_carrier univ_poly_zero[of R "carrier R"] UP.integral
less(4) r by auto
        hence "roots (p ⊗poly_ring R q) = add_mset a (roots (r ⊗poly_ring R
q))"
          using poly_mult_degree_one_monic_imp_same_roots[OF a UP.m_closed[OF
r less(4)]]
                UP.m_assoc[OF in_carrier r less(4)] p by auto
        also have " ... = add_mset a (roots r + roots q)"
          using less(1)[OF _ r not_zero less(4-5)] deg by simp
        also have " ... = (add_mset a (roots r)) + roots q"
          by simp
        also have " ... = roots p + roots q"
```

```
          using poly_mult_degree_one_monic_imp_same_roots[OF a r not_zero]
p by simp
        finally show ?thesis .
      qed
    qed
  qed
qed

lemma (in field) size_roots_le_degree:
  assumes "p ∈ carrier (poly_ring R)" shows "size (roots p) ≤ degree
p"
  using assms
proof (induction "degree p" arbitrary: p rule: less_induct)
  case less show ?case
  proof (cases "roots p = {#}", simp)
    interpret UP: domain "poly_ring R"
      using univ_poly_is_domain[OF carrier_is_subring] .

    case False with ‹p ∈ carrier (poly_ring R)›
    obtain a where a: "a ∈ carrier R" and "a ∈# roots p" and "[ 1, ⊖
a ] pdivides p"
      and in_carrier: "[ 1, ⊖ a ] ∈ carrier (poly_ring R)"
      by blast
    then obtain q where p: "p = [ 1, ⊖ a ] ⊗poly_ring R q" and q: "q
∈ carrier (poly_ring R)"
      unfolding pdivides_def by auto
    with ‹a ∈# roots p› have "p ≠ []"
      using degree_zero_imp_empty_roots[OF less(2)] by auto
    hence not_zero: "q ≠ []"
      using in_carrier univ_poly_zero[of R "carrier R"] UP.integral_iff
p by auto
    hence "degree p = Suc (degree q)"
      using poly_mult_degree_eq[OF carrier_is_subring, of _ q] in_carrier
p q
      unfolding univ_poly_carrier sym[OF univ_poly_mult[of R "carrier
R"]] by auto
    with ‹q ≠ []› show ?thesis
      using poly_mult_degree_one_monic_imp_same_roots[OF a q] p less(1)[OF
_ q]
      by (metis Suc_le_mono lessI size_add_mset)
  qed
qed

lemma (in domain) pirreducible_roots:
  assumes "p ∈ carrier (poly_ring R)" and "pirreducible (carrier R) p"
and "degree p ≠ 1"
  shows "roots p = {#}"
proof (rule ccontr)
  assume "roots p ≠ {#}" with ‹p ∈ carrier (poly_ring R)›
```

```
  obtain a where a: "a ∈ carrier R" and "a ∈# roots p" and "[ 1, ⊖
a ] pdivides p"
    and in_carrier: "[ 1, ⊖ a ] ∈ carrier (poly_ring R)"
    by blast
  hence "[ 1, ⊖ a ] ∼poly_ring R p"
    using divides_pirreducible_condition[OF assms(2) in_carrier]
          univ_poly_units_incl[OF carrier_is_subring]
    unfolding pdivides_def by auto
  hence "degree p = 1"
    using associated_polynomials_imp_same_length[OF carrier_is_subring
in_carrier assms(1)] by auto
  with ‹degree p ≠ 1› show False ..
qed

lemma (in field) pirreducible_imp_not_splitted:
  assumes "p ∈ carrier (poly_ring R)" and "pirreducible (carrier R) p"
and "degree p ≠ 1"
  shows "¬ splitted p"
  using pirreducible_roots[of p] pirreducible_degree[OF carrier_is_subfield,
of p] assms
  by (simp add: splitted_def)

lemma (in field)
  assumes "p ∈ carrier (poly_ring R)" and "q ∈ carrier (poly_ring R)"
    and "pirreducible (carrier R) p" and "degree p ≠ 1"
  shows "roots (p ⊗poly_ring R q) = roots q"
  using no_roots_imp_same_roots[of p q] pirreducible_roots[of p] assms
  unfolding ring_irreducible_def univ_poly_zero by auto

lemma (in field) trivial_factors_imp_splitted:
  assumes "p ∈ carrier (poly_ring R)"
    and "⋀q. ⟦ q ∈ carrier (poly_ring R); pirreducible (carrier R) q;
q pdivides p ⟧ ⟹ degree q ≤ 1"
  shows "splitted p"
  using assms
proof (induction "degree p" arbitrary: p rule: less_induct)
  interpret UP: principal_domain "poly_ring R"
    using univ_poly_is_principal[OF carrier_is_subfield] .
  case less show ?case
  proof (cases "degree p = 0", simp add: degree_zero_imp_splitted[OF less(2)])
    case False show ?thesis
    proof (cases "roots p = {#}")
      case True
      from ‹degree p ≠ 0› have "p ∉ Units (poly_ring R)" and "p ∈ carrier
(poly_ring R) - { [] }"
        using univ_poly_units'[OF carrier_is_subfield, of p] less(2) by
auto
      then obtain q where "q ∈ carrier (poly_ring R)" "pirreducible (carrier
R) q" and "q pdivides p"
```

```
        using UP.exists_irreducible_divisor[of p] unfolding univ_poly_zero
pdivides_def by auto
      with ‹degree p ≠ 0› have "roots p ≠ {#}"
        using degree_one_imp_singleton_roots[OF _ , of q] less(3)[of q]
              pdivides_imp_roots_incl[OF _ less(2), of q]
              pirreducible_degree[OF carrier_is_subfield, of q]
        by force
      from ‹roots p = {#}› and ‹roots p ≠ {#}› have False
        by simp
      thus ?thesis ..
    next
      case False with ‹p ∈ carrier (poly_ring R)›
      obtain a where a: "a ∈ carrier R" and "a ∈# roots p" and "[ 𝟭,
⊖ a ] pdivides p"
        and in_carrier: "[ 𝟭, ⊖ a ] ∈ carrier (poly_ring R)"
        by blast
      then obtain q where p: "p = [ 𝟭, ⊖ a ] ⊗poly_ring R q" and q:
"q ∈ carrier (poly_ring R)"
        unfolding pdivides_def by blast
      with ‹degree p ≠ 0› have "p ≠ []"
        by auto
      with ‹p = [ 𝟭, ⊖ a ] ⊗poly_ring R q› have "q ≠ []"
        using in_carrier q unfolding sym[OF univ_poly_zero[of R "carrier
R"]] by auto
      with ‹p = [ 𝟭, ⊖ a ] ⊗poly_ring R q› and ‹p ≠ []› have "degree
p = Suc (degree q)"
        using poly_mult_degree_eq[OF carrier_is_subring] in_carrier q
        unfolding univ_poly_carrier sym[OF univ_poly_mult[of R "carrier
R"]] by auto
      moreover have "q pdivides p"
        using p dividesI[OF in_carrier] UP.m_comm[OF in_carrier q] by
(auto simp add: pdivides_def)
      hence "degree r = 1" if "r ∈ carrier (poly_ring R)" and "pirreducible
(carrier R) r"
        and "r pdivides q" for r
        using less(3)[OF that(1-2)] UP.divides_trans[OF _ _ that(1), of
q p] that(3)
              pirreducible_degree[OF carrier_is_subfield that(1-2)]
        by (auto simp add: pdivides_def)
      ultimately have "splitted q"
        using less(1)[OF _ q] by auto
      with ‹degree p = Suc (degree q)› and ‹q ≠ []› show ?thesis
        using poly_mult_degree_one_monic_imp_same_roots[OF a q]
        unfolding sym[OF p] splitted_def
        by simp
    qed
  qed
qed
```

```
lemma (in field) pdivides_imp_splitted:
  assumes "p ∈ carrier (poly_ring R)" and "q ∈ carrier (poly_ring R)"
"q ≠ []" and "splitted q"
  shows "p pdivides q ⟹ splitted p"
proof (cases "p = []")
  case True thus ?thesis
    using degree_zero_imp_splitted[OF assms(1)] by simp
next
  interpret UP: principal_domain "poly_ring R"
    using univ_poly_is_principal[OF carrier_is_subfield] .

  case False
  assume "p pdivides q"
  then obtain b where b: "b ∈ carrier (poly_ring R)" and q: "q = p ⊗poly_ring R
b"
    unfolding pdivides_def by auto
  with ‹q ≠ []› have "p ≠ []" and "b ≠ []"
    using assms UP.integral_iff[of p b] unfolding sym[OF univ_poly_zero[of
R "carrier R"]] by auto
  hence "degree p + degree b = size (roots p) + size (roots b)"
    using associated_polynomials_imp_same_roots[of p b] assms b q splitted_def
          poly_mult_degree_eq[OF carrier_is_subring,of p b]
    unfolding univ_poly_carrier sym[OF univ_poly_mult[of R "carrier R"]]
    by auto
  moreover have "size (roots p) ≤ degree p" and "size (roots b) ≤ degree
b"
    using size_roots_le_degree assms(1) b by auto
  ultimately show ?thesis
    unfolding splitted_def by linarith
qed

lemma (in field) splitted_imp_trivial_factors:
  assumes "p ∈ carrier (poly_ring R)" "p ≠ []" and "splitted p"
  shows "⋀q. ⟦ q ∈ carrier (poly_ring R); pirreducible (carrier R) q;
q pdivides p ⟧ ⟹ degree q = 1"
  using pdivides_imp_splitted[OF _ assms] pirreducible_imp_not_splitted
  by auto
```

## 35.7 Link between pmod and rupture_surj

```
lemma (in domain) rupture_surj_composed_with_pmod:
  assumes "subfield K R" and "p ∈ carrier (K[X])" and "q ∈ carrier (K[X])"
  shows "rupture_surj K p q = rupture_surj K p (q pmod p)"
proof -
  interpret UP: principal_domain "K[X]"
    using univ_poly_is_principal[OF assms(1)] .
  interpret Rupt: ring "Rupt K p"
    using assms by (simp add: UP.cgenideal_ideal ideal.quotient_is_ring
rupture_def)
```

```
  let ?h = "rupture_surj K p"

  have "?h q = (?h p ⊗Rupt K p ?h (q pdiv p)) ⊕Rupt K p ?h (q pmod p)"
   and "?h (q pdiv p) ∈ carrier (Rupt K p)" "?h (q pmod p) ∈ carrier
(Rupt K p)"
    using pdiv_pmod[OF assms(1,3,2)] long_division_closed[OF assms(1,3,2)]
assms UP.m_closed
          ring_hom_memE[OF rupture_surj_hom(1)[OF subfieldE(1)[OF assms(1)]
assms(2)]]
    by metis+
  moreover have "?h p = PIdlK[X] p"
    using assms by (simp add: UP.a_rcos_zero UP.cgenideal_ideal UP.cgenideal_self)
  hence "?h p = 0Rupt K p"
    unfolding rupture_def FactRing_def by simp
  ultimately show ?thesis
    by simp
qed

corollary (in domain) rupture_carrier_as_pmod_image:
  assumes "subfield K R" and "p ∈ carrier (K[X])"
  shows "(rupture_surj K p) ‘ ((λq. q pmod p) ‘ (carrier (K[X]))) = carrier
(Rupt K p)"
    (is "?lhs = ?rhs")
proof
  have "(λq. q pmod p) ‘ carrier (K[X]) ⊆ carrier (K[X])"
    using long_division_closed(2)[OF assms(1) _ assms(2)] by auto
  thus "?lhs ⊆ ?rhs"
    using ring_hom_memE(1)[OF rupture_surj_hom(1)[OF subfieldE(1)[OF assms(1)]
assms(2)]] by auto
next
  show "?rhs ⊆ ?lhs"
  proof
    fix a assume "a ∈ carrier (Rupt K p)"
    then obtain q where "q ∈ carrier (K[X])" and "a = rupture_surj K
p q"
      unfolding rupture_def FactRing_def A_RCOSETS_def’ by auto
    thus "a ∈ ?lhs"
      using rupture_surj_composed_with_pmod[OF assms] by auto
  qed
qed

lemma (in domain) rupture_surj_inj_on:
  assumes "subfield K R" and "p ∈ carrier (K[X])"
  shows "inj_on (rupture_surj K p) ((λq. q pmod p) ‘ (carrier (K[X])))"
proof (intro inj_onI)
  interpret UP: principal_domain "K[X]"
    using univ_poly_is_principal[OF assms(1)] .
```

```
  fix a b
  assume "a ∈ (λq. q pmod p) ‘ carrier (K[X])"
     and "b ∈ (λq. q pmod p) ‘ carrier (K[X])"
  then obtain q s
    where q: "q ∈ carrier (K[X])" "a = q pmod p"
      and s: "s ∈ carrier (K[X])" "b = s pmod p"
    by auto
  moreover assume "rupture_surj K p a = rupture_surj K p b"
  ultimately have "q ⊖K[X] s ∈ (PIdlK[X] p)"
    using UP.quotient_eq_iff_same_a_r_cos[OF UP.cgenideal_ideal[OF assms(2)],
of q s]
          rupture_surj_composed_with_pmod[OF assms] by auto
  hence "p pdivides (q ⊖K[X] s)"
    using assms q(1) s(1) UP.to_contain_is_to_divide pdivides_iff_shell
    by (meson UP.cgenideal_ideal UP.cgenideal_minimal UP.minus_closed)
  thus "a = b"
    unfolding q s same_pmod_iff_pdivides[OF assms(1) q(1) s(1) assms(2)]
.
qed
```

## 35.8 Dimension

```
definition (in ring) exp_base :: "'a ⇒ nat ⇒ 'a list"
  where "exp_base x n = map (λi. x [^] i) (rev [0..< n])"

lemma (in ring) exp_base_closed:
  assumes "x ∈ carrier R" shows "set (exp_base x n) ⊆ carrier R"
  using assms by (induct n) (auto simp add: exp_base_def)

lemma (in ring) exp_base_append:
  shows "exp_base x (n + m) = (map (λi. x [^] i) (rev [n..< n + m]))
@ exp_base x n"
  unfolding exp_base_def by (metis map_append rev_append upt_add_eq_append
zero_le)

lemma (in ring) drop_exp_base:
  shows "drop n (exp_base x m) = exp_base x (m - n)"
proof -
  have ?thesis if "n > m"
    using that by (simp add: exp_base_def)
  moreover have ?thesis if "n ≤ m"
    using exp_base_append[of x "m - n" n] that by auto
  ultimately show ?thesis
    by linarith
qed

lemma (in ring) combine_eq_eval:
  shows "combine Ks (exp_base x (length Ks)) = eval Ks x"
  unfolding exp_base_def by (induct Ks) (auto)
```

```
lemma (in domain) pmod_image_characterization:
  assumes "subfield K R" and "p ∈ carrier (K[X])" and "p ≠ []"
  shows "(λq. q pmod p) ‘ carrier (K[X]) = { q ∈ carrier (K[X]). length
q ≤ degree p }"
proof -
  interpret UP: principal_domain "K[X]"
    using univ_poly_is_principal[OF assms(1)] .

  show ?thesis
  proof (rule order_antisym; rule subsetI)
    fix q assume "q ∈ { q ∈ carrier (K[X]). length q ≤ degree p }"
    then have "q ∈ carrier (K[X])" and "length q ≤ degree p"
      by simp+

    show "q ∈ (λq. q pmod p) ‘ carrier (K[X])"
    proof (cases "q = []")
      case True
      have "p pmod p = q"
        unfolding True pmod_zero_iff_pdivides[OF assms(1,2,2)]
        using assms(1-2) pdivides_iff_shell by auto
      thus ?thesis
        using assms(2) by blast
    next
      case False
      with ‹length q ≤ degree p› have "degree q < degree p"
        using le_eq_less_or_eq by fastforce
      with ‹q ∈ carrier (K[X])› show ?thesis
        using pmod_const(2)[OF assms(1) _ assms(2), of q] by (metis imageI)

    qed
  next
    fix q assume "q ∈ (λq. q pmod p) ‘ carrier (K[X])"
    then obtain q’ where "q’ ∈ carrier (K[X])" and "q = q’ pmod p"
      by auto
    thus "q ∈ { q ∈ carrier (K[X]). length q ≤ degree p }"
      using long_division_closed(2)[OF assms(1) _ assms(2), of q’]
            pmod_degree[OF assms(1) _ assms(2-3), of q’]
      by auto
  qed
qed

lemma (in domain) Span_var_pow_base:
  assumes "subfield K R"
  shows "ring.Span (K[X]) (poly_of_const ‘ K) (ring.exp_base (K[X]) X
n) =
         { q ∈ carrier (K[X]). length q ≤ n }" (is "?lhs = ?rhs")
proof -
  note subring = subfieldE(1)[OF assms]
```

```
  note subfield = univ_poly_subfield_of_consts[OF assms]

  interpret UP: domain "K[X]"
    using univ_poly_is_domain[OF subring] .

  show ?thesis
  proof (rule order_antisym; rule subsetI)
    fix q assume "q ∈ { q ∈ carrier (K[X]). length q ≤ n }"
    then have q: "q ∈ carrier (K[X])" "length q ≤ n"
      by simp+

    let ?repl = "replicate (n - length q) 0_K[X]"
    let ?map = "map poly_of_const q"
    let ?comb = "UP.combine"
    define Ks where "Ks = ?repl @ ?map"

    have "q = ?comb ?map (UP.exp_base X (length q))"
      using q eval_rewrite[OF subring q(1)] unfolding sym[OF UP.combine_eq_eval]
by auto
    moreover from ‹length q ≤ n›
    have "?comb (?repl @ Ks) (UP.exp_base X n) =  ?comb Ks (UP.exp_base
X (length q))"
      if "set Ks ⊆ carrier (K[X])" for Ks
      using UP.combine_prepend_replicate[OF that UP.exp_base_closed[OF
var_closed(1)[OF subring]]]
      unfolding UP.drop_exp_base by auto

    moreover have "set ?map ⊆ carrier (K[X])"
      using map_norm_in_poly_ring_carrier[OF subring q(1)]
      unfolding sym[OF univ_poly_carrier] polynomial_def by auto

    moreover have "?repl = map poly_of_const (replicate (n - length q)
0)"
      unfolding poly_of_const_def univ_poly_zero by (induct "n - length
q") (auto)
    hence "set ?repl ⊆ poly_of_const ‘ K"
      using subringE(2)[OF subring] by auto
    moreover from ‹q ∈ carrier (K[X])› have "set q ⊆ K"
      unfolding sym[OF univ_poly_carrier] polynomial_def by auto
    hence "set ?map ⊆ poly_of_const ‘ K"
      by auto

    ultimately have "q = ?comb Ks (UP.exp_base X n)" and "set Ks ⊆ poly_of_const
‘ K"
      by (simp add: Ks_def)+
    thus "q ∈ UP.Span (poly_of_const ‘ K) (UP.exp_base X n)"
      using UP.Span_eq_combine_set[OF subfield UP.exp_base_closed[OF var_closed(1)[OF
subring]]] by auto
  next
```

```
    fix q assume "q ∈ UP.Span (poly_of_const ‘ K) (UP.exp_base X n)"
    thus "q ∈ { q ∈ carrier (K[X]). length q ≤ n }"
    proof (induction n arbitrary: q)
      case 0 thus ?case
        unfolding UP.exp_base_def by (auto simp add: univ_poly_zero)
    next
      case (Suc n)
      then obtain k p where k: "k ∈ K" and p: "p ∈ UP.Span (poly_of_const
‘ K) (UP.exp_base X n)"
        and q: "q = ((poly_of_const k) ⊗K[X] (X [^]K[X] n)) ⊕K[X] p"
        unfolding UP.exp_base_def using UP.line_extension_mem_iff by auto
      have p_in_carrier: "p ∈ carrier (K[X])" and "length p ≤ n"
        using Suc(1)[OF p] by simp+
      moreover from ‹k ∈ K› have "poly_of_const k ∈ carrier (K[X])"
        unfolding poly_of_const_def sym[OF univ_poly_carrier] polynomial_def
by auto
      ultimately have "q ∈ carrier (K[X])"
        unfolding q using var_pow_closed[OF subring, of n] by algebra

      moreover have "poly_of_const k = 0K[X]" if "k = 0"
        unfolding poly_of_const_def that univ_poly_zero by simp
      with ‹p ∈ carrier (K[X])› have "q = p" if "k = 0"
        unfolding q using var_pow_closed[OF subring, of n] that by algebra
      with ‹length p ≤ n› have "length q ≤ Suc n" if "k = 0"
        using that by simp

      moreover have "poly_of_const k = [ k ]" if "k ≠ 0"
        unfolding poly_of_const_def using that by simp
      hence monom: "monom k n = (poly_of_const k) ⊗K[X] (X [^]K[X] n)"
if "k ≠ 0"
        using that monom_eq_var_pow[OF subring] subfieldE(3)[OF assms]
k by auto
      with ‹p ∈ carrier (K[X])› and ‹k ∈ K› and ‹length p ≤ n›
      have "length q = Suc n" if "k ≠ 0"
        using that poly_add_length_eq[OF subring monom_is_polynomial[OF
subring, of k n], of p]
        unfolding univ_poly_carrier monom_def univ_poly_add sym[OF monom[OF
that]] q by auto
      ultimately show ?case
        by (cases "k = 0", auto)
    qed
  qed
qed

lemma (in domain) var_pow_base_independent:
  assumes "subfield K R"
  shows "ring.independent (K[X]) (poly_of_const ‘ K) (ring.exp_base (K[X])
X n)"
proof -
```

```
  note subring = subfieldE(1)[OF assms]
  interpret UP: domain "K[X]"
    using univ_poly_is_domain[OF subring] .

  show ?thesis
  proof (induction n, simp add: UP.exp_base_def)
    case (Suc n)
    have "X [^]K[X] n ∉ UP.Span (poly_of_const ‘ K) (ring.exp_base (K[X])
X n)"
      unfolding sym[OF unitary_monom_eq_var_pow[OF subring]] monom_def
                Span_var_pow_base[OF assms] by auto
    moreover have "X [^]K[X] n # UP.exp_base X n = UP.exp_base X (Suc
n)"
      unfolding UP.exp_base_def by simp
    ultimately show ?case
      using UP.li_Cons[OF var_pow_closed[OF subring, of n] _Suc] by simp
  qed
qed

lemma (in domain) bounded_degree_dimension:
  assumes "subfield K R"
  shows "ring.dimension (K[X]) n (poly_of_const ‘ K) { q ∈ carrier (K[X]).
length q ≤ n }"
proof -
  interpret UP: domain "K[X]"
    using univ_poly_is_domain[OF subfieldE(1)[OF assms]] .
  have "length (UP.exp_base X n) = n"
    unfolding UP.exp_base_def by simp
  thus ?thesis
    using UP.dimension_independent[OF var_pow_base_independent[OF assms],
of n]
    unfolding Span_var_pow_base[OF assms] by simp
qed

corollary (in domain) univ_poly_infinite_dimension:
  assumes "subfield K R" shows "ring.infinite_dimension (K[X]) (poly_of_const
‘ K) (carrier (K[X]))"
proof (rule ccontr)
  interpret UP: domain "K[X]"
    using univ_poly_is_domain[OF subfieldE(1)[OF assms]] .

  assume "¬ UP.infinite_dimension (poly_of_const ‘ K) (carrier (K[X]))"
  then obtain n where n: "UP.dimension n (poly_of_const ‘ K) (carrier
(K[X]))"
    by blast
  show False
    using UP.independent_length_le_dimension[OF univ_poly_subfield_of_consts[OF
assms] n
          var_pow_base_independent[OF assms, of "Suc n"]
```

```
        UP.exp_base_closed[OF var_closed(1)[OF subfieldE(1)[OF assms]]]]
      unfolding UP.exp_base_def by simp
qed

corollary (in domain) rupture_dimension:
  assumes "subfield K R" and "p ∈ carrier (K[X])" and "degree p > 0"
  shows "ring.dimension (Rupt K p) (degree p) ((rupture_surj K p) ‘ poly_of_const
‘ K) (carrier (Rupt K p))"
proof -
  interpret UP: domain "K[X]"
    using univ_poly_is_domain[OF subfieldE(1)[OF assms(1)]] .
  interpret Hom: ring_hom_ring "K[X]" "Rupt K p" "rupture_surj K p"
    using rupture_surj_hom(2)[OF subfieldE(1)[OF assms(1)] assms(2)] .

  have not_nil: "p ≠ []"
    using assms(3) by auto

  show ?thesis
    using Hom.inj_hom_dimension[OF univ_poly_subfield_of_consts rupture_one_not_zero
          rupture_surj_inj_on] bounded_degree_dimension assms
    unfolding sym[OF rupture_carrier_as_pmod_image[OF assms(1-2)]]
              pmod_image_characterization[OF assms(1-2) not_nil]
    by simp
qed

end


theory Indexed_Polynomials
  imports Weak_Morphisms "HOL-Library.Multiset" Polynomial_Divisibility

begin
```

# 36 Indexed Polynomials

In this theory, we build a basic framework to the study of polynomials on letters indexed by a set. The main interest is to then apply these concepts to the construction of the algebraic closure of a field.

## 36.1 Definitions

We formalize indexed monomials as multisets with its support a subset of the index set. On top of those, we build indexed polynomials which are simply functions mapping a monomial to its coefficient.

```
definition (in ring) indexed_const :: "'a ⇒ ('c multiset ⇒ 'a)"
  where "indexed_const k = (λm. if m = {#} then k else 0)"
```

```
definition (in ring) indexed_pmult :: "('c multiset ⇒ 'a) ⇒ 'c ⇒ ('c
multiset ⇒ 'a)" (infixl ‹⨂› 65)
  where "indexed_pmult P i = (λm. if i ∈# m then P (m - {# i #}) else
0)"

definition (in ring) indexed_padd :: "_ ⇒ _ ⇒ ('c multiset ⇒ 'a)" (infixl
‹⨁› 65)
  where "indexed_padd P Q = (λm. (P m) ⊕ (Q m))"

definition (in ring) indexed_var :: "'c ⇒ ('c multiset ⇒ 'a)" (‹𝒳ı›)
  where "indexed_var i = (indexed_const 1) ⨂ i"

definition (in ring) index_free :: "('c multiset ⇒ 'a) ⇒ 'c ⇒ bool"
  where "index_free P i ⟷ (∀m. i ∈# m ⟶ P m = 0)"

definition (in ring) carrier_coeff :: "('c multiset ⇒ 'a) ⇒ bool"
  where "carrier_coeff P ⟷ (∀m. P m ∈ carrier R)"

inductive_set (in ring) indexed_pset :: "'c set ⇒ 'a set ⇒ ('c multiset
⇒ 'a) set"
  (‹(‹open_block notation=‹postfix 𝒳››_ [𝒳ı])› 80)
  for I and K where
    indexed_const:  "k ∈ K ⟹ indexed_const k ∈ (K[𝒳_I])"
  | indexed_padd:  "⟦ P ∈ (K[𝒳_I]); Q ∈ (K[𝒳_I]) ⟧ ⟹ P ⨁ Q ∈ (K[𝒳_I])"
  | indexed_pmult: "⟦ P ∈ (K[𝒳_I]); i ∈ I ⟧ ⟹ P ⨂ i ∈ (K[𝒳_I])"

fun (in ring) indexed_eval_aux :: "('c multiset ⇒ 'a) list ⇒ 'c ⇒ ('c
multiset ⇒ 'a)"
  where "indexed_eval_aux Ps i = foldr (λP Q. (Q ⨂ i) ⨁ P) Ps (indexed_const
0)"

fun (in ring) indexed_eval :: "('c multiset ⇒ 'a) list ⇒ 'c ⇒ ('c multiset
⇒ 'a)"
  where "indexed_eval Ps i = indexed_eval_aux (rev Ps) i"
```

## 36.2 Basic Properties

```
lemma (in ring) carrier_coeffE:
  assumes "carrier_coeff P" shows "P m ∈ carrier R"
  using assms unfolding carrier_coeff_def by simp

lemma (in ring) indexed_zero_def: "indexed_const 0 = (λ_. 0)"
  unfolding indexed_const_def by simp

lemma (in ring) indexed_const_index_free: "index_free (indexed_const
k) i"
  unfolding index_free_def indexed_const_def by auto

lemma (in domain) indexed_var_not_index_free: "¬ index_free 𝒳_i i"
```

```
proof -
  have "𝒳ᵢ {# i #} = 1"
    unfolding indexed_var_def indexed_pmult_def indexed_const_def by simp
  thus ?thesis
    using one_not_zero unfolding index_free_def by fastforce
qed

lemma (in ring) indexed_pmult_zero [simp]:
  shows "indexed_pmult (indexed_const 0) i = indexed_const 0"
  unfolding indexed_zero_def indexed_pmult_def by auto

lemma (in ring) indexed_padd_zero:
  assumes "carrier_coeff P" shows "P ⨁ (indexed_const 0) = P" and "(indexed_const
0) ⨁ P = P"
  using assms unfolding carrier_coeff_def indexed_zero_def indexed_padd_def
by auto

lemma (in ring) indexed_padd_const:
  shows "(indexed_const k1) ⨁ (indexed_const k2) = indexed_const (k1
⊕ k2)"
  unfolding indexed_padd_def indexed_const_def by auto

lemma (in ring) indexed_const_in_carrier:
  assumes "K ⊆ carrier R" and "k ∈ K" shows "⋀m. (indexed_const k)
m ∈ carrier R"
  using assms unfolding indexed_const_def by auto

lemma (in ring) indexed_padd_in_carrier:
  assumes "carrier_coeff P" and "carrier_coeff Q" shows "carrier_coeff
(indexed_padd P Q)"
  using assms unfolding carrier_coeff_def indexed_padd_def by simp

lemma (in ring) indexed_pmult_in_carrier:
  assumes "carrier_coeff P" shows "carrier_coeff (P ⨂ i)"
  using assms unfolding carrier_coeff_def indexed_pmult_def by simp

lemma (in ring) indexed_eval_aux_in_carrier:
  assumes "list_all carrier_coeff Ps" shows "carrier_coeff (indexed_eval_aux
Ps i)"
  using assms unfolding carrier_coeff_def
  by (induct Ps) (auto simp add: indexed_zero_def indexed_padd_def indexed_pmult_def)

lemma (in ring) indexed_eval_in_carrier:
  assumes "list_all carrier_coeff Ps" shows "carrier_coeff (indexed_eval
Ps i)"
  using assms indexed_eval_aux_in_carrier[of "rev Ps"] by auto

lemma (in ring) indexed_pset_in_carrier:
  assumes "K ⊆ carrier R" and "P ∈ (K[𝒳ᵢ])" shows "carrier_coeff P"
```

```
  using assms(2,1) indexed_const_in_carrier unfolding carrier_coeff_def
  by (induction) (auto simp add: indexed_zero_def indexed_padd_def indexed_pmult_def)
```

## 36.3 Indexed Eval

```
lemma (in ring) exists_indexed_eval_aux_monomial:
  assumes "carrier_coeff P" and "list_all carrier_coeff Qs"
    and "count n i = k" and "P n ≠ 0" and "list_all (λQ. index_free
Q i) Qs"
  obtains m where "count m i = length Qs + k" and "(indexed_eval_aux
(Qs @ [ P ]) i) m ≠ 0"
proof -
  from assms(2,5) have "∃m. count m i = length Qs + k ∧ (indexed_eval_aux
(Qs @ [ P ]) i) m ≠ 0"
  proof (induct Qs)
    case Nil thus ?case
      using indexed_padd_zero(2)[OF assms(1)] assms(3-4) by auto
  next
    case (Cons Q Qs)
    then obtain m where m: "count m i = length Qs + k" "(indexed_eval_aux
(Qs @ [ P ]) i) m ≠ 0"
      by auto
    define m' where "m' = m + {# i #}"
    hence "Q m' = 0"
      using Cons(3) unfolding index_free_def by simp
    moreover have "(indexed_eval_aux (Qs @ [ P ]) i) m ∈ carrier R"
      using indexed_eval_aux_in_carrier[of "Qs @ [ P ]" i] Cons(2) assms(1)
carrier_coeffE by auto
    hence "((indexed_eval_aux (Qs @ [ P ]) i) ⊗ i) m' ∈ carrier R -
{ 0 }"
      using m unfolding indexed_pmult_def m'_def by simp
    ultimately have "(indexed_eval_aux (Q # (Qs @ [ P ])) i) m' ≠ 0"
      by (auto simp add: indexed_padd_def)
    moreover from ‹count m i = length Qs + k› have "count m' i = length
(Q # Qs) + k"
      unfolding m'_def by simp
    ultimately show ?case
      by auto
  qed
  thus thesis
    using that by blast
qed

lemma (in ring) indexed_eval_aux_monomial_degree_le:
  assumes "list_all carrier_coeff Ps" and "list_all (λP. index_free P
i) Ps"
    and "(indexed_eval_aux Ps i) m ≠ 0" shows "count m i ≤ length Ps
- 1"
  using assms(1-3)
```

```
proof (induct Ps arbitrary: m, simp add: indexed_zero_def)
  case (Cons P Ps) show ?case
  proof (cases "count m i = 0", simp)
    assume "count m i ≠ 0"
    hence "P m = 𝟬"
      using Cons(3) unfolding index_free_def by simp
    moreover have "(indexed_eval_aux Ps i) m ∈ carrier R"
      using carrier_coeffE[OF indexed_eval_aux_in_carrier[of Ps i]] Cons(2)
by simp
    ultimately have "((indexed_eval_aux Ps i) ⨂ i) m ≠ 𝟬"
      using Cons(4) by (auto simp add: indexed_padd_def)
    with ‹count m i ≠ 0› have "(indexed_eval_aux Ps i) (m - {# i #})
≠ 𝟬"
      unfolding indexed_pmult_def by (auto simp del: indexed_eval_aux.simps)
    hence "count m i - 1 ≤ length Ps - 1"
      using Cons(1)[of "m - {# i #}"] Cons(2-3) by auto
    moreover from ‹(indexed_eval_aux Ps i) (m - {# i #}) ≠ 𝟬› have
"length Ps > 0"
      by (auto simp add: indexed_zero_def)
    moreover from ‹count m i ≠ 0› have "count m i > 0"
      by simp
    ultimately show ?thesis
      by (simp add: Suc_leI le_diff_iff)
  qed
qed

lemma (in ring) indexed_eval_aux_is_inj:
  assumes "list_all carrier_coeff Ps" and "list_all (λP. index_free P
i) Ps"
      and "list_all carrier_coeff Qs" and "list_all (λQ. index_free Q
i) Qs"
    and "indexed_eval_aux Ps i = indexed_eval_aux Qs i" and "length Ps
= length Qs"
  shows "Ps = Qs"
  using assms
proof (induct Ps arbitrary: Qs, simp)
  case (Cons P Ps)
  from ‹length (P # Ps) = length Qs› obtain Q' Qs' where Qs: "Qs = Q'
# Qs'" and "length Ps = length Qs'"
    by (metis Suc_length_conv)

  have in_carrier:
    "((indexed_eval_aux Ps  i) ⨂ i) m ∈ carrier R" "P  m ∈ carrier R"
    "((indexed_eval_aux Qs' i) ⨂ i) m ∈ carrier R" "Q' m ∈ carrier R"
for m
    using indexed_eval_aux_in_carrier[of Ps  i]
          indexed_eval_aux_in_carrier[of Qs' i] Cons(2,4) carrier_coeffE
    unfolding Qs indexed_pmult_def by auto
```

```
  have "(indexed_eval_aux (P # Ps) i) m = (indexed_eval_aux (Q' # Qs')
i) m" for m
    using Cons(6) unfolding Qs by simp
  hence eq: "((indexed_eval_aux Ps i) ⨂ i) m ⊕ P m = ((indexed_eval_aux
Qs' i) ⨂ i) m ⊕ Q' m" for m
    by (simp add: indexed_padd_def)

  have "P m = Q' m" if "i ∈# m" for m
    using that Cons(3,5) unfolding index_free_def Qs by auto
  moreover have "P m = Q' m" if "i ∉# m" for m
    using in_carrier(2,4) eq[of m] that by (auto simp add: indexed_pmult_def)
  ultimately have "P = Q'"
    by auto

  hence "(indexed_eval_aux Ps i) m = (indexed_eval_aux Qs' i) m" for m
    using eq[of "m + {# i #}"] in_carrier[of "m + {# i #}"] unfolding
indexed_pmult_def by auto
  with ‹length Ps = length Qs'› have "Ps = Qs'"
    using Cons(1)[of Qs'] Cons(2-5) unfolding Qs by auto
  with ‹P = Q'› show ?case
    unfolding Qs by simp
qed

lemma (in ring) indexed_eval_aux_is_inj':
  assumes "list_all carrier_coeff Ps" and "list_all (λP. index_free P
i) Ps"
      and "list_all carrier_coeff Qs" and "list_all (λQ. index_free Q
i) Qs"
      and "carrier_coeff P" and "index_free P i" "P ≠ indexed_const 0"
      and "carrier_coeff Q" and "index_free Q i" "Q ≠ indexed_const 0"
    and "indexed_eval_aux (Ps @ [ P ]) i = indexed_eval_aux (Qs @ [ Q
]) i"
  shows "Ps = Qs" and "P = Q"
proof -
  obtain m n where "P m ≠ 0" and "Q n ≠ 0"
    using assms(7,10) unfolding indexed_zero_def by blast
  hence "count m i = 0" and "count n i = 0"
    using assms(6,9) unfolding index_free_def by (meson count_inI)+
  with ‹P m ≠ 0› and ‹Q n ≠ 0› obtain m' n'
    where m': "count m' i = length Ps" "(indexed_eval_aux (Ps @ [ P ])
i) m' ≠ 0"
      and n': "count n' i = length Qs" "(indexed_eval_aux (Qs @ [ Q ])
i) n' ≠ 0"
    using exists_indexed_eval_aux_monomial[of P Ps m i 0]
          exists_indexed_eval_aux_monomial[of Q Qs n i 0] assms(1-5,8)
    by (metis (no_types, lifting) add.right_neutral)
  have "(indexed_eval_aux (Qs @ [ Q ]) i) m' ≠ 0"
    using m'(2) assms(11) by simp
  with ‹count m' i = length Ps› have "length Ps ≤ length Qs"
```

```
    using indexed_eval_aux_monomial_degree_le[of "Qs @ [ Q ]" i m'] assms(3-4,8-9)
by auto
  moreover have "(indexed_eval_aux (Ps @ [ P ]) i) n' ≠ 0"
    using n'(2) assms(11) by simp
  with ‹count n' i = length Qs› have "length Qs ≤ length Ps"
    using indexed_eval_aux_monomial_degree_le[of "Ps @ [ P ]" i n'] assms(1-2,5-6)
by auto
  ultimately have same_len: "length (Ps @ [ P ]) = length (Qs @ [ Q ])"
    by simp
  thus "Ps = Qs" and "P = Q"
    using indexed_eval_aux_is_inj[of "Ps @ [ P ]" i "Qs @ [ Q ]"] assms(1-6,8-9,11)
by auto
qed

lemma (in ring) exists_indexed_eval_monomial:
  assumes "carrier_coeff P" and "list_all carrier_coeff Qs"
    and "P n ≠ 0" and "list_all (λQ. index_free Q i) Qs"
  obtains m where "count m i = length Qs + (count n i)" and "(indexed_eval
(P # Qs) i) m ≠ 0"
  using exists_indexed_eval_aux_monomial[OF assms(1) _ _ assms(3), of
"rev Qs"] assms(2,4) by auto

corollary (in ring) exists_indexed_eval_monomial':
  assumes "carrier_coeff P" and "list_all carrier_coeff Qs"
    and "P ≠ indexed_const 0" and "list_all (λQ. index_free Q i) Qs"
  obtains m where "count m i ≥ length Qs" and "(indexed_eval (P # Qs)
i) m ≠ 0"
proof -
  from ‹P ≠ indexed_const 0› obtain n where "P n ≠ 0"
    unfolding indexed_const_def by auto
  then obtain m where "count m i = length Qs + (count n i)" and "(indexed_eval
(P # Qs) i) m ≠ 0"
    using exists_indexed_eval_monomial[OF assms(1-2) _ assms(4)] by auto
  thus thesis
    using that by force
qed

lemma (in ring) indexed_eval_monomial_degree_le:
  assumes "list_all carrier_coeff Ps" and "list_all (λP. index_free P
i) Ps"
    and "(indexed_eval Ps i) m ≠ 0" shows "count m i ≤ length Ps - 1"
  using indexed_eval_aux_monomial_degree_le[of "rev Ps"] assms by auto

lemma (in ring) indexed_eval_is_inj:
  assumes "list_all carrier_coeff Ps" and "list_all (λP. index_free P
i) Ps"
      and "list_all carrier_coeff Qs" and "list_all (λQ. index_free Q
i) Qs"
      and "carrier_coeff P" and "index_free P i" "P ≠ indexed_const 0"
```

```
      and "carrier_coeff Q" and "index_free Q i" "Q ≠ indexed_const 0"
    and "indexed_eval (P # Ps) i = indexed_eval (Q # Qs) i"
  shows "Ps = Qs" and "P = Q"
proof -
  have rev_cond:
    "list_all carrier_coeff (rev Ps)" "list_all (λP. index_free P i) (rev
Ps)"
    "list_all carrier_coeff (rev Qs)" "list_all (λQ. index_free Q i) (rev
Qs)"
    using assms(1-4) by auto
  show "Ps = Qs" and "P = Q"
    using indexed_eval_aux_is_inj'[OF rev_cond assms(5-10)] assms(11)
by auto
qed

lemma (in ring) indexed_eval_inj_on_carrier:
  assumes "⋀P. P ∈ carrier L ⟹ carrier_coeff P" and "⋀P. P ∈ carrier
L ⟹ index_free P i" and "0_L = indexed_const 0"
  shows "inj_on (λPs. indexed_eval Ps i) (carrier (poly_ring L))"
proof -
  have aux_lemma: "Ps = []"
    if "Ps ∈ carrier (poly_ring L)" and "indexed_eval Ps i = indexed_const
0" for Ps
  proof (rule ccontr)
    assume "¬ ?thesis"
    then obtain P' Ps' where Ps: "Ps = P' # Ps'"
      using list.exhaust by blast
    with ‹Ps ∈ carrier (poly_ring L)›
    have "P' ≠ indexed_const 0" and "list_all carrier_coeff Ps" and
"list_all (λP. index_free P i) Ps"
      using assms unfolding sym[OF univ_poly_carrier[of L "carrier L"]]
polynomial_def
      by (simp add: list.pred_set subset_code(1))+
    then obtain m where "(indexed_eval Ps i) m ≠ 0"
      using exists_indexed_eval_monomial'[of P' Ps'] unfolding Ps by auto
    hence "indexed_eval Ps i ≠ indexed_const 0"
      unfolding indexed_const_def by auto
    with ‹indexed_eval Ps i = indexed_const 0› show False by simp
  qed

  show ?thesis
  proof (rule inj_onI)
    fix Ps Qs
    assume "Ps ∈ carrier (poly_ring L)" and "Qs ∈ carrier (poly_ring
L)"
    show "indexed_eval Ps i = indexed_eval Qs i ⟹ Ps = Qs"
    proof (cases)
      assume "Qs = []" and "indexed_eval Ps i = indexed_eval Qs i"
      with ‹Ps ∈ carrier (poly_ring L)› show "Ps = Qs"
```

```
        using aux_lemma by simp
    next
      assume "Qs ≠ []" and eq: "indexed_eval Ps i = indexed_eval Qs
i"
      with ‹Qs ∈ carrier (poly_ring L)› have "Ps ≠ []"
        using aux_lemma by auto
      from ‹Ps ≠ []› and ‹Qs ≠ []› obtain P' Ps' Q' Qs' where Ps:
"Ps = P' # Ps'" and Qs: "Qs = Q' # Qs'"
        using list.exhaust by metis

      from ‹Ps ∈ carrier (poly_ring L)› and ‹Ps = P' # Ps'›
      have "carrier_coeff P'" and "index_free P' i" "P' ≠ indexed_const
0"
       and "list_all carrier_coeff Ps'" and "list_all (λP. index_free
P i) Ps'"
        using assms unfolding sym[OF univ_poly_carrier[of L "carrier L"]]
polynomial_def
        by (simp add: list.pred_set subset_code(1))+
      moreover
      from ‹Qs ∈ carrier (poly_ring L)› and ‹Qs = Q' # Qs'›
      have "carrier_coeff Q'" and "index_free Q' i" "Q' ≠ indexed_const
0"
       and "list_all carrier_coeff Qs'" and "list_all (λP. index_free
P i) Qs'"
        using assms unfolding sym[OF univ_poly_carrier[of L "carrier L"]]
polynomial_def
        by (simp add: list.pred_set subset_code(1))+
      ultimately show ?thesis
        using indexed_eval_is_inj[of Ps' i Qs' P' Q'] eq unfolding Ps
Qs by auto
    qed
  qed
qed
```

## 36.4 Link with Weak Morphisms

We study some elements of the contradiction needed in the algebraic closure existence proof.

```
context ring
begin

lemma (in ring) indexed_padd_index_free:
  assumes "index_free P i" and "index_free Q i" shows "index_free (P
⨁ Q) i"
  using assms unfolding indexed_padd_def index_free_def by auto

lemma (in ring) indexed_pmult_index_free:
  assumes "index_free P j" and "i ≠ j" shows "index_free (P ⨂ i) j"
  using assms unfolding index_free_def indexed_pmult_def
```

```
  by (metis insert_DiffM insert_noteq_member)

lemma (in ring) indexed_eval_index_free:
  assumes "list_all (λP. index_free P j) Ps" and "i ≠ j" shows "index_free
(indexed_eval Ps i) j"
proof -
  have "index_free (indexed_eval_aux Ps i) j" if "list_all (λP. index_free
P j) Ps" for Ps
    using that indexed_padd_index_free[OF indexed_pmult_index_free[OF
_ assms(2)]]
    by (induct Ps) (auto simp add: indexed_zero_def index_free_def)
  thus ?thesis
    using assms(1) by auto
qed

context
  fixes L :: "(('c multiset) ⇒ 'a) ring" and i :: 'c
  assumes hyps:
    — i   "field L"
    — ii  "⋀P. P ∈ carrier L ⟹ carrier_coeff P"
    — iii "⋀P. P ∈ carrier L ⟹ index_free P i"
    — iv  "𝟬_L = indexed_const 𝟬"
begin

interpretation L: field L
  using ‹field L› .

interpretation UP: principal_domain "poly_ring L"
  using L.univ_poly_is_principal[OF L.carrier_is_subfield] .


abbreviation eval_pmod
  where "eval_pmod q ≡ (λp. indexed_eval (L.pmod p q) i)"

abbreviation image_poly
  where "image_poly q ≡ image_ring (eval_pmod q) (poly_ring L)"


lemma indexed_eval_is_weak_ring_morphism:
  assumes "q ∈ carrier (poly_ring L)" shows "weak_ring_morphism (eval_pmod
q) (PIdl_poly_ring L q) (poly_ring L)"
proof (rule weak_ring_morphismI)
  show "ideal (PIdl_poly_ring L q) (poly_ring L)"
    using UP.cgenideal_ideal[OF assms] .
next
  fix a b assume in_carrier: "a ∈ carrier (poly_ring L)" "b ∈ carrier
(poly_ring L)"
  note ldiv_closed = in_carrier[THEN L.long_division_closed(2)[OF L.carrier_is_subfield
_ assms]]
```

```
  have "(eval_pmod q) a = (eval_pmod q) b ⟷ L.pmod a q = L.pmod b q"
    using inj_onD[OF indexed_eval_inj_on_carrier[OF hyps(2-4)] _ ldiv_closed]
by fastforce
  also have " ... ⟷ q pdivides_L (a ⊖_poly_ring L b)"
    unfolding L.same_pmod_iff_pdivides[OF L.carrier_is_subfield in_carrier
assms] ..
  also have " ... ⟷ PIdl_poly_ring L (a ⊖_poly_ring L b) ⊆ PIdl_poly_ring L
q"
    unfolding UP.to_contain_is_to_divide[OF assms UP.minus_closed[OF in_carrier]]
pdivides_def ..
  also have " ... ⟷ a ⊖_poly_ring L b ∈ PIdl_poly_ring L q"
    unfolding UP.cgenideal_eq_genideal[OF assms] UP.cgenideal_eq_genideal[OF
UP.minus_closed[OF in_carrier]]
              UP.Idl_subset_ideal'[OF UP.minus_closed[OF in_carrier] assms]
..
  finally show "(eval_pmod q) a = (eval_pmod q) b ⟷ a ⊖_poly_ring L b
∈ PIdl_poly_ring L q" .
qed

lemma eval_norm_eq_id:
  assumes "q ∈ carrier (poly_ring L)" and "degree q > 0" and "a ∈ carrier
L"
  shows "((eval_pmod q) ∘ (ring.poly_of_const L)) a = a"
proof (cases)
  assume "a = 0_L" thus ?thesis
    using L.long_division_zero(2)[OF L.carrier_is_subfield assms(1)] hyps(4)
    unfolding ring.poly_of_const_def[OF L.ring_axioms] by auto
next
  assume "a ≠ 0_L" then have in_carrier: "[ a ] ∈ carrier (poly_ring
L)"
    using assms(3) unfolding sym[OF univ_poly_carrier[of L "carrier L"]]
polynomial_def by simp
  from ‹a ≠ 0_L› show ?thesis
    using L.pmod_const(2)[OF L.carrier_is_subfield in_carrier assms(1)]
assms(2)
          indexed_padd_zero(2)[OF hyps(2)[OF assms(3)]]
    unfolding ring.poly_of_const_def[OF L.ring_axioms] by auto
qed

lemma image_poly_iso_incl:
  assumes "q ∈ carrier (poly_ring L)" and "degree q > 0" shows "id ∈
ring_hom L (image_poly q)"
proof -
  have "((eval_pmod q) ∘ L.poly_of_const) ∈ ring_hom L (image_poly q)"
    using ring_hom_trans[OF L.canonical_embedding_is_hom[OF L.carrier_is_subring]
          UP.weak_ring_morphism_is_hom[OF indexed_eval_is_weak_ring_morphism[OF
assms(1)]]]
    by simp
```

```
  thus ?thesis
    using eval_norm_eq_id[OF assms(1-2)] L.ring_hom_restrict[of _ "image_poly
q" id] by auto
qed

lemma image_poly_is_field:
  assumes "q ∈ carrier (poly_ring L)" and "pirreducible_L (carrier L)
q" shows "field (image_poly q)"
  using UP.image_ring_is_field[OF indexed_eval_is_weak_ring_morphism[OF
assms(1)]] assms(2)
  unfolding sym[OF L.rupture_is_field_iff_pirreducible[OF L.carrier_is_subfield
assms(1)]] rupture_def
  by simp

lemma image_poly_index_free:
  assumes "q ∈ carrier (poly_ring L)" and "P ∈ carrier (image_poly q)"
and "¬ index_free P j" "i ≠ j"
  obtains Q where "Q ∈ carrier L" and "¬ index_free Q j"
proof -
  from ‹P ∈ carrier (image_poly q)› obtain p where p: "p ∈ carrier (poly_ring
L)" and P: "P = (eval_pmod q) p"
    unfolding image_ring_carrier by blast
  from ‹¬ index_free P j› have "¬ list_all (λP. index_free P j) (L.pmod
p q)"
    using indexed_eval_index_free[OF _ assms(4), of "L.pmod p q"] un-
folding sym[OF P] by auto
  then obtain Q where "Q ∈ set (L.pmod p q)" and "¬ index_free Q j"
    unfolding list_all_iff by auto
  thus ?thesis
    using L.long_division_closed(2)[OF L.carrier_is_subfield p assms(1)]
that
    unfolding sym[OF univ_poly_carrier[of L "carrier L"]] polynomial_def
    by auto
qed

lemma eval_pmod_var:
  assumes "indexed_const ∈ ring_hom R L" and "q ∈ carrier (poly_ring
L)" and "degree q > 1"
  shows "(eval_pmod q) X_L = 𝒳_i" and "𝒳_i ∈ carrier (image_poly q)"
proof -
  have "X_L = [ indexed_const 1, indexed_const 0 ]" and "X_L ∈ carrier
(poly_ring L)"
    using ring_hom_one[OF assms(1)] hyps(4) L.var_closed(1) L.carrier_is_subring
unfolding var_def by auto
  thus "(eval_pmod q) X_L = 𝒳_i"
    using L.pmod_const(2)[OF L.carrier_is_subfield _ assms(2), of "X_L"]
assms(3)
    by (auto simp add: indexed_pmult_def indexed_padd_def indexed_const_def
indexed_var_def)
```

```
  with ‹XL ∈ carrier (poly_ring L)› show "𝒳i ∈ carrier (image_poly
q)"
    using image_iff unfolding image_ring_carrier by fastforce
qed

lemma image_poly_eval_indexed_var:
  assumes "indexed_const ∈ ring_hom R L"
    and "q ∈ carrier (poly_ring L)" and "degree q > 1" and "pirreducibleL
(carrier L) q"
  shows "(ring.eval (image_poly q)) q 𝒳i = 0image_poly q"
proof -
  let ?surj = "L.rupture_surj (carrier L) q"
  let ?Rupt = "RuptL (carrier L) q"
  let ?f = "eval_pmod q"

  interpret UP: ring "poly_ring L"
    using L.univ_poly_is_ring[OF L.carrier_is_subring] .
  from ‹pirreducibleL (carrier L) q› interpret Rupt: field ?Rupt
    using L.rupture_is_field_iff_pirreducible[OF L.carrier_is_subfield
assms(2)] by simp

  have weak_morphism: "weak_ring_morphism ?f (PIdlpoly_ring L q) (poly_ring
L)"
    using indexed_eval_is_weak_ring_morphism[OF assms(2)] .
  then interpret I: ideal "PIdlpoly_ring L q" "poly_ring L"
    using weak_ring_morphism.axioms(1) by auto
  interpret Hom: ring_hom_ring ?Rupt "image_poly q" "λx. the_elem (?f
` x)"
    using ring_hom_ring.intro[OF I.quotient_is_ring UP.image_ring_is_ring[OF
weak_morphism]]
          UP.weak_ring_morphism_is_iso[OF weak_morphism]
    unfolding ring_iso_def symmetric[OF ring_hom_ring_axioms_def] rupture_def
    by auto

  have "set q ⊆ carrier L" and lc: "q ≠ [] ⟹ lead_coeff q ∈ carrier
L - { 0L }"
    using assms(2) unfolding sym[OF univ_poly_carrier] polynomial_def
by auto

  have map_surj: "set (map (?surj ∘ L.poly_of_const) q) ⊆ carrier ?Rupt"
  proof -
    have "L.poly_of_const a ∈ carrier (poly_ring L)" if "a ∈ carrier L"
for a
      using that L.normalize_gives_polynomial[of "[ a ]"]
      unfolding univ_poly_carrier ring.poly_of_const_def[OF L.ring_axioms]
by simp
    hence "(?surj ∘ L.poly_of_const) a ∈ carrier ?Rupt" if "a ∈ carrier
L" for a
      using ring_hom_memE(1)[OF L.rupture_surj_hom(1)[OF L.carrier_is_subring
```

```
assms(2)]] that by simp
    with ‹set q ⊆ carrier L› show ?thesis
      by (induct q) (auto)
  qed

  have "?surj X_L ∈ carrier ?Rupt"
    using ring_hom_memE(1)[OF L.rupture_surj_hom(1)[OF _ assms(2)] L.var_closed(1)]
L.carrier_is_subring by simp
  moreover have "map (λx. the_elem (?f ‘ x)) (map (?surj ∘ L.poly_of_const)
q) = q"
  proof -
    define g where "g = (?surj ∘ L.poly_of_const)"
    define f where "f = (λx. the_elem (?f ‘ x))"

    have "the_elem (?f ‘ ((?surj ∘ L.poly_of_const) a)) = ((eval_pmod
q) ∘ L.poly_of_const) a"
      if "a ∈ carrier L" for a
      using that L.normalize_gives_polynomial[of "[ a ]"] UP.weak_ring_morphism_range[OF
weak_morphism]
      unfolding univ_poly_carrier ring.poly_of_const_def[OF L.ring_axioms]
by auto
    hence "the_elem (?f ‘ ((?surj ∘ L.poly_of_const) a)) = a" if "a ∈
carrier L" for a
      using eval_norm_eq_id[OF assms(2)] that assms(3) by simp
    hence "f (g a) = a" if "a ∈ carrier L" for a
      using that unfolding f_def g_def by simp
    with ‹set q ⊆ carrier L› have "map f (map g q) = q"
      by (induct q) (auto)
    thus ?thesis
      unfolding f_def g_def by simp
  qed
  moreover have "(λx. the_elem (?f ‘ x)) (?surj X_L) = 𝒳_i"
    using UP.weak_ring_morphism_range[OF weak_morphism L.var_closed(1)[OF
L.carrier_is_subring]]
    unfolding eval_pmod_var(1)[OF assms(1-3)] by simp
  ultimately have "Hom.S.eval q 𝒳_i = (λx. the_elem (?f ‘ x)) (Rupt.eval
(map (?surj ∘ L.poly_of_const) q) (?surj X_L))"
    using Hom.eval_hom'[OF _ map_surj] by auto
  moreover have "0_?Rupt = ?surj 0_poly_ring L"
    unfolding rupture_def FactRing_def by (simp add: I.a_rcos_const)
  hence "the_elem (?f ‘ 0_?Rupt) = 0_image_poly q"
    using UP.weak_ring_morphism_range[OF weak_morphism UP.zero_closed]
    unfolding image_ring_zero by simp
  hence "(λx. the_elem (?f ‘ x)) (Rupt.eval (map (?surj ∘ L.poly_of_const)
q) (?surj X_L)) = 0_image_poly q"
    using L.polynomial_rupture[OF L.carrier_is_subring assms(2)] by simp
  ultimately show ?thesis
    by simp
qed
```

**end**

**end**

**end**

```
theory Finite_Extensions
  imports Embedded_Algebras Polynomials Polynomial_Divisibility

begin
```

# 37 Finite Extensions

## 37.1 Definitions

```
definition (in ring) transcendental :: "'a set ⇒ 'a ⇒ bool"
  where "transcendental K x ⟷ inj_on (λp. eval p x) (carrier (K[X]))"

abbreviation (in ring) algebraic :: "'a set ⇒ 'a ⇒ bool"
  where "algebraic K x ≡ ¬ transcendental K x"

definition (in ring) Irr :: "'a set ⇒ 'a ⇒ 'a list"
  where "Irr K x = (THE p. p ∈ carrier (K[X]) ∧ pirreducible K p ∧ eval
p x = 0 ∧ lead_coeff p = 1)"

inductive_set (in ring) simple_extension :: "'a set ⇒ 'a ⇒ 'a set"
  for K and x where
    zero [simp, intro]: "0 ∈ simple_extension K x" |
    lin:  "⟦ k1 ∈ simple_extension K x; k2 ∈ K ⟧ ⟹ (k1 ⊗ x) ⊕ k2 ∈
simple_extension K x"

fun (in ring) finite_extension :: "'a set ⇒ 'a list ⇒ 'a set"
  where "finite_extension K xs = foldr (λx K'. simple_extension K' x)
xs K"
```

## 37.2 Basic Properties

```
lemma (in ring) transcendental_consistent:
  assumes "subring K R" shows "transcendental = ring.transcendental (R
⦇ carrier := K ⦈)"
  unfolding transcendental_def ring.transcendental_def[OF subring_is_ring[OF
assms]]
            univ_poly_consistent[OF assms] eval_consistent[OF assms] ..

lemma (in ring) algebraic_consistent:
  assumes "subring K R" shows "algebraic = ring.algebraic (R ⦇ carrier
:= K ⦈)"
```

```
  unfolding over_def transcendental_consistent[OF assms] ..

lemma (in ring) eval_transcendental:
  assumes "(transcendental over K) x" "p ∈ carrier (K[X])" "eval p x
= 0" shows "p = []"
proof -
  have "[] ∈ carrier (K[X])" and "eval [] x = 0"
    by (auto simp add: univ_poly_def)
  thus ?thesis
    using assms unfolding over_def transcendental_def inj_on_def by auto
qed

lemma (in ring) transcendental_imp_trivial_ker:
  shows "(transcendental over K) x ⟹ a_kernel (K[X]) R (λp. eval p
x) = { [] }"
  using eval_transcendental unfolding a_kernel_def' by (auto simp add:
univ_poly_def)

lemma (in ring) non_trivial_ker_imp_algebraic:
  shows "a_kernel (K[X]) R (λp. eval p x) ≠ { [] } ⟹ (algebraic over
K) x"
  using transcendental_imp_trivial_ker unfolding over_def by auto

lemma (in domain) trivial_ker_imp_transcendental:
  assumes "subring K R" and "x ∈ carrier R"
  shows "a_kernel (K[X]) R (λp. eval p x) = { [] } ⟹ (transcendental
over K) x"
  using ring_hom_ring.trivial_ker_imp_inj[OF eval_ring_hom[OF assms]]
  unfolding transcendental_def over_def by (simp add: univ_poly_zero)

lemma (in domain) algebraic_imp_non_trivial_ker:
  assumes "subring K R" and "x ∈ carrier R"
  shows "(algebraic over K) x ⟹ a_kernel (K[X]) R (λp. eval p x) ≠
{ [] }"
  using trivial_ker_imp_transcendental[OF assms] unfolding over_def by
auto

lemma (in domain) algebraicE:
  assumes "subring K R" and "x ∈ carrier R" "(algebraic over K) x"
  obtains p where "p ∈ carrier (K[X])" "p ≠ []" "eval p x = 0"
proof -
  have "[] ∈ a_kernel (K[X]) R (λp. eval p x)"
    unfolding a_kernel_def' univ_poly_def by auto
  then obtain p where "p ∈ carrier (K[X])" "p ≠ []" "eval p x = 0"
    using algebraic_imp_non_trivial_ker[OF assms] unfolding a_kernel_def'
by blast
  thus thesis using that by auto
qed
```

```
lemma (in ring) algebraicI:
  assumes "p ∈ carrier (K[X])" "p ≠ []" and "eval p x = 0" shows "(algebraic
over K) x"
  using assms non_trivial_ker_imp_algebraic unfolding a_kernel_def' by
auto

lemma (in ring) transcendental_mono:
  assumes "K ⊆ K'" "(transcendental over K') x" shows "(transcendental
over K) x"
proof -
  have "carrier (K[X]) ⊆ carrier (K'[X])"
    using assms(1) unfolding univ_poly_def polynomial_def by auto
  thus ?thesis
    using assms unfolding over_def transcendental_def by (metis inj_on_subset)
qed

corollary (in ring) algebraic_mono:
  assumes "K ⊆ K'" "(algebraic over K) x" shows "(algebraic over K')
x"
  using transcendental_mono[OF assms(1)] assms(2) unfolding over_def by
blast

lemma (in domain) zero_is_algebraic:
  assumes "subring K R" shows "(algebraic over K) 0"
  using algebraicI[OF var_closed(1)[OF assms]] unfolding var_def by auto

lemma (in domain) algebraic_self:
  assumes "subring K R" and "k ∈ K" shows "(algebraic over K) k"
proof (rule algebraicI[of "[ 1, ⊖ k ]"])
  show "[ 1, ⊖ k ] ∈ carrier (K [X])" and "[ 1, ⊖ k ] ≠ []"
    using subringE(2-3,5)[OF assms(1)] assms(2) unfolding univ_poly_def
polynomial_def by auto
  have "k ∈ carrier R"
    using subringE(1)[OF assms(1)] assms(2) by auto
  thus "eval [ 1, ⊖ k ] k = 0"
    by (auto, algebra)
qed

lemma (in domain) ker_diff_carrier:
  assumes "subring K R"
  shows "a_kernel (K[X]) R (λp. eval p x) ≠ carrier (K[X])"
proof -
  have "eval [ 1 ] x ≠ 0" and "[ 1 ] ∈ carrier (K[X])"
    using subringE(3)[OF assms] unfolding univ_poly_def polynomial_def
by auto
  thus ?thesis
    unfolding a_kernel_def' by blast
qed
```

### 37.3 Minimal Polynomial

```
lemma (in domain) minimal_polynomial_is_unique:
  assumes "subfield K R" and "x ∈ carrier R" "(algebraic over K) x"
  shows "∃!p ∈ carrier (K[X]). pirreducible K p ∧ eval p x = 0 ∧ lead_coeff
p = 1"
    (is "∃!p. ?minimal_poly p")
proof -
  interpret UP: principal_domain "K[X]"
    using univ_poly_is_principal[OF assms(1)] .

  let ?ker_gen = "λp. p ∈ carrier (K[X]) ∧ pirreducible K p ∧ lead_coeff
p = 1 ∧
                   a_kernel (K[X]) R (λp. eval p x) = PIdl_K[X] p"

  obtain p where p: "?ker_gen p" and unique: "⋀q. ?ker_gen q ⟹ q =
p"
    using exists_unique_pirreducible_gen[OF assms(1) eval_ring_hom[OF
_ assms(2)]
          algebraic_imp_non_trivial_ker[OF _ assms(2-3)]
          ker_diff_carrier] subfieldE(1)[OF assms(1)] by auto
  hence "?minimal_poly p"
    using UP.cgenideal_self p unfolding a_kernel_def' by auto
  moreover have "⋀q. ?minimal_poly q ⟹ q = p"
  proof -
    fix q assume q: "?minimal_poly q"
    then have "q ∈ PIdl_K[X] p"
      using p unfolding a_kernel_def' by auto
    hence "p ∼_K[X] q"
      using cgenideal_pirreducible[OF assms(1)] p q by simp
    hence "a_kernel (K[X]) R (λp. eval p x) = PIdl_K[X] q"
      using UP.associated_iff_same_ideal q p by simp
    thus "q = p"
      using unique q by simp
  qed
  ultimately show ?thesis by blast
qed

lemma (in domain) IrrE:
  assumes "subfield K R" and "x ∈ carrier R" "(algebraic over K) x"
  shows "Irr K x ∈ carrier (K[X])" and "pirreducible K (Irr K x)"
    and "lead_coeff (Irr K x) = 1" and "eval (Irr K x) x = 0"
  using theI'[OF minimal_polynomial_is_unique[OF assms]] unfolding Irr_def
by auto

lemma (in domain) Irr_generates_ker:
  assumes "subfield K R" and "x ∈ carrier R" "(algebraic over K) x"
  shows "a_kernel (K[X]) R (λp. eval p x) = PIdl_K[X] (Irr K x)"
proof -
  obtain q
```

```
    where q: "q ∈ carrier (K[X])" "pirreducible K q"
      and ker: "a_kernel (K[X]) R (λp. eval p x) = PIdl_K[X] q"
    using exists_unique_pirreducible_gen[OF assms(1) eval_ring_hom[OF
_ assms(2)]
          algebraic_imp_non_trivial_ker[OF _ assms(2-3)]
          ker_diff_carrier] subfieldE(1)[OF assms(1)] by auto
  have "Irr K x ∈ PIdl_K[X] q"
    using IrrE(1,4)[OF assms] ker unfolding a_kernel_def' by auto
  thus ?thesis
    using cgenideal_pirreducible[OF assms(1) q(1-2) IrrE(2)[OF assms]]
q(1) IrrE(1)[OF assms]
          cring.associated_iff_same_ideal[OF univ_poly_is_cring[OF subfieldE(1)[OF
assms(1)]]]
    unfolding ker
    by simp
qed

lemma (in domain) Irr_minimal:
  assumes "subfield K R" and "x ∈ carrier R" "(algebraic over K) x"
    and "p ∈ carrier (K[X])" "eval p x = 0" shows "(Irr K x) pdivides
p"
proof -
  interpret UP: principal_domain "K[X]"
    using univ_poly_is_principal[OF assms(1)] .

  have "p ∈ PIdl_K[X] (Irr K x)"
    using Irr_generates_ker[OF assms(1-3)] assms(4-5) unfolding a_kernel_def'
by auto
  hence "(Irr K x) divides_K[X] p"
    using UP.to_contain_is_to_divide IrrE(1)[OF assms(1-3)]
    by (meson UP.cgenideal_ideal UP.cgenideal_minimal assms(4))
  thus ?thesis
    unfolding pdivides_iff_shell[OF assms(1) IrrE(1)[OF assms(1-3)] assms(4)]
.
qed

lemma (in domain) rupture_of_Irr:
  assumes "subfield K R" and "x ∈ carrier R" "(algebraic over K) x" shows
"field (Rupt K (Irr K x))"
  using rupture_is_field_iff_pirreducible[OF assms(1)] IrrE(1-2)[OF assms]
by simp
```

## 37.4 Simple Extensions

```
lemma (in ring) simple_extension_consistent:
  assumes "subring K R" shows "ring.simple_extension (R ⦇ carrier :=
K ⦈) = simple_extension"
proof -
  interpret K: ring "R ⦇ carrier := K ⦈"
```

```
    using subring_is_ring[OF assms] .

  have "⋀K' x. K.simple_extension  K' x ⊆ simple_extension K' x"
  proof
    fix K' x a show "a ∈ K.simple_extension  K' x ⟹ a ∈ simple_extension
K' x"
      by (induction rule: K.simple_extension.induct) (auto simp add: simple_extension.lin)
  qed
  moreover
  have "⋀K' x. simple_extension K' x ⊆ K.simple_extension  K' x"
  proof
    fix K' x a assume a: "a ∈ simple_extension K' x" thus "a ∈ K.simple_extension
K' x"
      using K.simple_extension.zero K.simple_extension.lin
      by (induction rule: simple_extension.induct) (simp)+
  qed
  ultimately show ?thesis by blast
qed

lemma (in ring) mono_simple_extension:
  assumes "K ⊆ K'" shows "simple_extension K x ⊆ simple_extension K'
x"
proof
  fix a assume "a ∈ simple_extension K x" thus "a ∈ simple_extension
K' x"
  proof (induct a rule: simple_extension.induct, simp)
    case lin thus ?case using simple_extension.lin assms by blast
  qed
qed

lemma (in ring) simple_extension_incl:
  assumes "K ⊆ carrier R" and "x ∈ carrier R" shows "K ⊆ simple_extension
K x"
proof
  fix k assume "k ∈ K" thus "k ∈ simple_extension K x"
    using simple_extension.lin[OF simple_extension.zero, of k K x] assms
by auto
qed

lemma (in ring) simple_extension_mem:
  assumes "subring K R" and "x ∈ carrier R" shows "x ∈ simple_extension
K x"
proof -
  have "𝟭 ∈ simple_extension K x"
    using simple_extension_incl[OF _ assms(2)] subringE(1,3)[OF assms(1)]
by auto
  thus ?thesis
    using simple_extension.lin[OF _ subringE(2)[OF assms(1)], of 𝟭 x]
assms(2) by auto
```

```
qed

lemma (in ring) simple_extension_carrier:
  assumes "x ∈ carrier R" shows "simple_extension (carrier R) x = carrier
R"
proof
  show "carrier R ⊆ simple_extension (carrier R) x"
    using simple_extension_incl[OF _ assms] by auto
next
  show "simple_extension (carrier R) x ⊆ carrier R"
  proof
    fix a assume "a ∈ simple_extension (carrier R) x" thus "a ∈ carrier
R"
      by (induct a rule: simple_extension.induct) (auto simp add: assms)
  qed
qed

lemma (in ring) simple_extension_in_carrier:
  assumes "K ⊆ carrier R" and "x ∈ carrier R" shows "simple_extension
K x ⊆ carrier R"
  using mono_simple_extension[OF assms(1), of x] simple_extension_carrier[OF
assms(2)] by auto

lemma (in ring) simple_extension_subring_incl:
  assumes "subring K' R" and "K ⊆ K'" "x ∈ K'" shows "simple_extension
K x ⊆ K'"
  using ring.simple_extension_in_carrier[OF subring_is_ring[OF assms(1)]]
assms(2-3)
  unfolding simple_extension_consistent[OF assms(1)] by simp

lemma (in ring) simple_extension_as_eval_img:
  assumes "K ⊆ carrier R" "x ∈ carrier R"
  shows "simple_extension K x = (λp. eval p x) ‘ carrier (K[X])"
proof
  show "simple_extension K x ⊆ (λp. eval p x) ‘ carrier (K[X])"
  proof
    fix a assume "a ∈ simple_extension K x" thus "a ∈ (λp. eval p x)
‘ carrier (K[X])"
    proof (induction rule: simple_extension.induct)
      case zero
      have "polynomial K []" and "eval [] x = 0"
        unfolding polynomial_def by simp+
      thus ?case
        unfolding univ_poly_carrier by force
    next
      case (lin k1 k2)
      then obtain p where p: "p ∈ carrier (K[X])" "polynomial K p" "eval
p x = k1"
        by (auto simp add: univ_poly_carrier)
```

```
      hence "set p ⊆ carrier R" and "k2 ∈ carrier R"
        using assms(1) lin(2) unfolding polynomial_def by auto
      hence "eval (normalize (p @ [ k2 ])) x = k1 ⊗ x ⊕ k2"
        using eval_append_aux[of p k2 x] eval_normalize[of "p @ [ k2 ]"
x] assms(2) p(3) by auto
      moreover have "set (p @ [k2]) ⊆ K"
        using polynomial_incl[OF p(2)] ‹k2 ∈ K› by auto
      then have "local.normalize (p @ [k2]) ∈ carrier (K [X])"
        using normalize_gives_polynomial univ_poly_carrier by blast
      ultimately show ?case
        unfolding univ_poly_carrier by force
    qed
  qed
next
  show "(λp. eval p x) ‘ carrier (K[X]) ⊆ simple_extension K x"
  proof
    fix a assume "a ∈ (λp. eval p x) ‘ carrier (K[X])"
    then obtain p where p: "set p ⊆ K" "eval p x = a"
      using polynomial_incl unfolding univ_poly_def by auto
    thus "a ∈ simple_extension K x"
    proof (induct "length p" arbitrary: p a)
      case 0 thus ?case
        using simple_extension.zero by simp
    next
      case (Suc n)
      obtain p’ k where p: "p = p’ @ [ k ]"
        using Suc(2) by (metis list.size(3) nat.simps(3) rev_exhaust)
      hence "a = (eval p’ x) ⊗ x ⊕ k"
        using eval_append_aux[of p’ k x] Suc(3-4) assms unfolding p by
auto
      moreover have "eval p’ x ∈ simple_extension K x"
        using Suc(1-3) unfolding p by auto
      ultimately show ?case
        using simple_extension.lin Suc(3) unfolding p by auto
    qed
  qed
qed

corollary (in domain) simple_extension_is_subring:
  assumes "subring K R" "x ∈ carrier R" shows "subring (simple_extension
K x) R"
  using ring_hom_ring.img_is_subring[OF eval_ring_hom[OF assms]
        ring.carrier_is_subring[OF univ_poly_is_ring[OF assms(1)]]]
        simple_extension_as_eval_img[OF subringE(1)[OF assms(1)] assms(2)]
  by simp

corollary (in domain) simple_extension_minimal:
  assumes "subring K R" "x ∈ carrier R"
  shows "simple_extension K x = ⋂ { K’. subring K’ R ∧ K ⊆ K’ ∧ x ∈
```

```
K' }"
  using simple_extension_is_subring[OF assms] simple_extension_mem[OF
assms]
        simple_extension_incl[OF subringE(1)[OF assms(1)] assms(2)] simple_extension_subring
  by blast

corollary (in domain) simple_extension_isomorphism:
  assumes "subring K R" "x ∈ carrier R"
  shows "(K[X]) Quot (a_kernel (K[X]) R (λp. eval p x)) ≃ R ⦇ carrier
:= simple_extension K x ⦈"
  using ring_hom_ring.FactRing_iso_set_aux[OF eval_ring_hom[OF assms]]
        simple_extension_as_eval_img[OF subringE(1)[OF assms(1)] assms(2)]
  unfolding is_ring_iso_def by auto

corollary (in domain) simple_extension_of_algebraic:
  assumes "subfield K R" and "x ∈ carrier R" "(algebraic over K) x"
  shows "Rupt K (Irr K x) ≃ R ⦇ carrier := simple_extension K x ⦈"
  using simple_extension_isomorphism[OF subfieldE(1)[OF assms(1)] assms(2)]
  unfolding Irr_generates_ker[OF assms] rupture_def by simp

corollary (in domain) simple_extension_of_transcendental:
  assumes "subring K R" and "x ∈ carrier R" "(transcendental over K)
x"
  shows "K[X] ≃ R ⦇ carrier := simple_extension K x ⦈"
  using simple_extension_isomorphism[OF _ assms(2), of K] assms(1)
        ring_iso_trans[OF ring.FactRing_zeroideal(2)[OF univ_poly_is_ring]]
  unfolding transcendental_imp_trivial_ker[OF assms(3)] univ_poly_zero
  by auto

proposition (in domain) simple_extension_subfield_imp_algebraic:
  assumes "subring K R" "x ∈ carrier R"
  shows "subfield (simple_extension K x) R ⟹ (algebraic over K) x"
proof -
  assume simple_ext: "subfield (simple_extension K x) R" show "(algebraic
over K) x"
  proof (rule ccontr)
    assume "¬ (algebraic over K) x" then have "(transcendental over
K) x"
      unfolding over_def by simp
    then obtain h where h: "h ∈ ring_iso (R ⦇ carrier := simple_extension
K x ⦈) (K[X])"
      using ring_iso_sym[OF univ_poly_is_ring simple_extension_of_transcendental]
assms
      unfolding is_ring_iso_def by blast
    then interpret Hom: ring_hom_ring "R ⦇ carrier := simple_extension
K x ⦈" "K[X]" h
      using subring_is_ring[OF simple_extension_is_subring[OF assms]]
            univ_poly_is_ring[OF assms(1)] assms h
      by (auto simp add: ring_hom_ring_def ring_hom_ring_axioms_def ring_iso_def)
```

```
    have "field (K[X])"
      using field.ring_iso_imp_img_field[OF subfield_iff(2)[OF simple_ext]
h]
      unfolding Hom.hom_one Hom.hom_zero by simp
    moreover have "¬ field (K[X])"
      using univ_poly_not_field[OF assms(1)] .
    ultimately show False by simp
  qed
qed

proposition (in domain) simple_extension_is_subfield:
  assumes "subfield K R" "x ∈ carrier R"
  shows "subfield (simple_extension K x) R ⟷ (algebraic over K) x"
proof
  assume alg: "(algebraic over K) x"
  then obtain h where h: "h ∈ ring_iso (Rupt K (Irr K x)) (R ⦇ carrier
:= simple_extension K x ⦈)"
    using simple_extension_of_algebraic[OF assms] unfolding is_ring_iso_def
by blast
  have rupt_field: "field (Rupt K (Irr K x))" and "ring (R ⦇ carrier
:= simple_extension K x ⦈)"
    using subring_is_ring[OF simple_extension_is_subring[OF subfieldE(1)]]
          rupture_of_Irr[OF assms alg] assms by simp+
  then interpret Hom: ring_hom_ring "Rupt K (Irr K x)" "R ⦇ carrier :=
simple_extension K x ⦈" h
    using h cring.axioms(1)[OF domain.axioms(1)[OF field.axioms(1)]]
    by (auto simp add: ring_hom_ring_def ring_hom_ring_axioms_def ring_iso_def)
  show "subfield (simple_extension K x) R"
    using field.ring_iso_imp_img_field[OF rupt_field h] subfield_iff(1)[OF
_
          simple_extension_in_carrier[OF subfieldE(3)[OF assms(1)] assms(2)]]
    by simp
next
  assume simple_ext: "subfield (simple_extension K x) R" thus "(algebraic
over K) x"
    using simple_extension_subfield_imp_algebraic[OF subfieldE(1)[OF assms(1)]
assms(2)] by simp
qed
```

## 37.5 Link between dimension of K-algebras and algebraic extensions

```
lemma (in domain) exp_base_independent:
  assumes "subfield K R" "x ∈ carrier R" "(algebraic over K) x"
  shows "independent K (exp_base x (degree (Irr K x)))"
proof -
  have "⋀n. n ≤ degree (Irr K x) ⟹ independent K (exp_base x n)"
  proof -
    fix n show "n ≤ degree (Irr K x) ⟹ independent K (exp_base x n)"
```

```
    proof (induct n, simp add: exp_base_def)
      case (Suc n)
      have "x [^] n ∉ Span K (exp_base x n)"
      proof (rule ccontr)
        assume "¬ x [^] n ∉ Span K (exp_base x n)"
        then obtain a Ks
          where Ks: "a ∈ K - { 0 }" "set Ks ⊆ K" "length Ks = n" "combine
(a # Ks) (exp_base x (Suc n)) = 0"
          using Span_mem_imp_non_trivial_combine[OF assms(1) exp_base_closed[OF
assms(2), of n]]
          by (auto simp add: exp_base_def)
        hence "eval (a # Ks) x = 0"
          using combine_eq_eval by (auto simp add: exp_base_def)
        moreover have "(a # Ks) ∈ carrier (K[X]) - { [] }"
          unfolding univ_poly_def polynomial_def using Ks(1-2) by auto
        ultimately have "degree (Irr K x) ≤ n"
          using pdivides_imp_degree_le[OF subfieldE(1)[OF assms(1)]
                IrrE(1)[OF assms] _ _ Irr_minimal[OF assms, of "a # Ks"]]
Ks(3) by auto
        from ‹Suc n ≤ degree (Irr K x)› and this show False by simp
      qed
      thus ?case
        using independent.li_Cons assms(2) Suc by (auto simp add: exp_base_def)
    qed
  qed
  thus ?thesis
    by simp
qed

lemma (in ring) Span_eq_eval_img:
  assumes "subfield K R" "x ∈ carrier R"
  shows "Span K (exp_base x n) = (λp. eval p x) ‘ { p ∈ carrier (K[X]).
length p ≤ n }"
    (is "?Span = ?eval_img")
proof
  show "?Span ⊆ ?eval_img"
  proof
    fix u assume "u ∈ Span K (exp_base x n)"
    then obtain Ks where Ks: "set Ks ⊆ K" "length Ks = n" "u = combine
Ks (exp_base x n)"
      using Span_eq_combine_set_length_version[OF assms(1) exp_base_closed[OF
assms(2)]]
      by (auto simp add: exp_base_def)
    hence "u = eval (normalize Ks) x"
      using combine_eq_eval eval_normalize[OF _ assms(2)] subfieldE(3)[OF
assms(1)] by auto
    moreover have "normalize Ks ∈ carrier (K[X])"
      using normalize_gives_polynomial[OF Ks(1)] unfolding univ_poly_def
by auto
```

```
    moreover have "length (normalize Ks) ≤ n"
      using normalize_length_le[of Ks] Ks(2) by auto
    ultimately show "u ∈ ?eval_img" by auto
  qed
next
  show "?eval_img ⊆ ?Span"
  proof
    fix u assume "u ∈ ?eval_img"
    then obtain p where p: "p ∈ carrier (K[X])" "length p ≤ n" "u =
eval p x"
      by blast
    hence "combine p (exp_base x (length p)) = u"
      using combine_eq_eval by auto
    moreover have set_p: "set p ⊆ K"
      using polynomial_incl[of K p] p(1) unfolding univ_poly_carrier by
auto
    hence "set p ⊆ carrier R"
      using subfieldE(3)[OF assms(1)] by auto
    moreover have "drop (n - length p) (exp_base x n) = exp_base x (length
p)"
      using p(2) drop_exp_base by auto
    ultimately have "combine ((replicate (n - length p) 0) @ p) (exp_base
x n) = u"
      using combine_prepend_replicate[OF _ exp_base_closed[OF assms(2),
of n]] by auto
    moreover have "set ((replicate (n - length p) 0) @ p) ⊆ K"
      using subringE(2)[OF subfieldE(1)[OF assms(1)]] set_p by auto
    ultimately show "u ∈ ?Span"
      using Span_eq_combine_set[OF assms(1) exp_base_closed[OF assms(2),
of n]] by blast
  qed
qed

lemma (in domain) Span_exp_base:
  assumes "subfield K R" "x ∈ carrier R" "(algebraic over K) x"
  shows "Span K (exp_base x (degree (Irr K x))) = simple_extension K x"
  unfolding simple_extension_as_eval_img[OF subfieldE(3)[OF assms(1)]
assms(2)]
            Span_eq_eval_img[OF assms(1-2)]
proof (auto)
  interpret UP: principal_domain "K[X]"
    using univ_poly_is_principal[OF assms(1)] .
  note hom_simps = ring_hom_memE[OF eval_is_hom[OF subfieldE(1)[OF assms(1)]
assms(2)]]

  fix p assume p: "p ∈ carrier (K[X])"
  have Irr: "Irr K x ∈ carrier (K[X])" "Irr K x ≠ []"
    using IrrE(1-2)[OF assms] unfolding ring_irreducible_def univ_poly_zero
by auto
```

```
  then obtain q r
    where q: "q ∈ carrier (K[X])" and r: "r ∈ carrier (K[X])"
      and dvd: "p = Irr K x ⊗K [X] q ⊕K [X] r" "r = [] ∨ degree r < degree
(Irr K x)"
    using subfield_long_division_theorem_shell[OF assms(1) p Irr(1)] un-
folding univ_poly_zero by auto
  hence "eval p x = (eval (Irr K x) x) ⊗ (eval q x) ⊕ (eval r x)"
    using hom_simps(2-3) Irr(1) by simp
  hence "eval p x = eval r x"
    using hom_simps(1) q r unfolding IrrE(4)[OF assms] by simp
  moreover have "length r < length (Irr K x)"
    using dvd(2) Irr(2) by auto
  ultimately
  show "eval p x ∈ (λp. local.eval p x) ` { p ∈ carrier (K [X]). length
p ≤ length (Irr K x) - Suc 0 }"
    using r by auto
qed

corollary (in domain) dimension_simple_extension:
  assumes "subfield K R" "x ∈ carrier R" "(algebraic over K) x"
  shows "dimension (degree (Irr K x)) K (simple_extension K x)"
  using dimension_independent[OF exp_base_independent[OF assms]] Span_exp_base[OF
assms]
  by (simp add: exp_base_def)

lemma (in ring) finite_dimension_imp_algebraic:
  assumes "subfield K R" "subring F R" and "finite_dimension K F"
  shows "x ∈ F ⟹ (algebraic over K) x"
proof -
  let ?Us = "λn. map (λi. x [^] i) (rev [0..< Suc n])"

  assume x: "x ∈ F" then have in_carrier: "x ∈ carrier R"
    using subringE[OF assms(2)] by auto
  obtain n where n: "dimension n K F"
    using assms(3) by auto
  have set_Us: "set (?Us n) ⊆ F"
    using x subringE(3,6)[OF assms(2)] by (induct n) (auto)
  hence "set (?Us n) ⊆ carrier R"
    using subringE(1)[OF assms(2)] by auto
  moreover have "dependent K (?Us n)"
    using independent_length_le_dimension[OF assms(1) n _ set_Us] by auto
  ultimately
  obtain Ks where Ks: "length Ks = Suc n" "combine Ks (?Us n) = 0" "set
Ks ⊆ K" "set Ks ≠ { 0 }"
    using dependent_imp_non_trivial_combine[OF assms(1), of "?Us n"] by
auto
  have "set Ks ⊆ carrier R"
    using subring_props(1)[OF assms(1)] Ks(3) by auto
  hence "eval (normalize Ks) x = 0"
```

```
    using combine_eq_eval[of Ks] eval_normalize[OF _ in_carrier] Ks(1-2)
by (simp add: exp_base_def)
  moreover have "normalize Ks = [] ⟹ set Ks ⊆ { 0 }"
    by (induct Ks) (auto, meson list.discI,
                    metis all_not_in_conv list.discI list.sel(3) singletonD
subset_singletonD)
  hence "normalize Ks ≠ []"
    using Ks(1,4) by (metis list.size(3) nat.distinct(1) set_empty subset_singleton_iff)
  moreover have "normalize Ks ∈ carrier (K[X])"
    using normalize_gives_polynomial[OF Ks(3)] unfolding univ_poly_def
by auto
  ultimately show ?thesis
    using algebraicI by auto
qed

corollary (in domain) simple_extension_dim:
  assumes "subfield K R" "x ∈ carrier R" "(algebraic over K) x"
  shows "(dim over K) (simple_extension K x) = degree (Irr K x)"
  using dimI[OF assms(1) dimension_simple_extension[OF assms]] .

corollary (in domain) finite_dimension_simple_extension:
  assumes "subfield K R" "x ∈ carrier R"
  shows "finite_dimension K (simple_extension K x) ⟷ (algebraic over
K) x"
  using finite_dimensionI[OF dimension_simple_extension[OF assms]]
        finite_dimension_imp_algebraic[OF _ simple_extension_is_subring[OF
subfieldE(1)]]
        simple_extension_mem[OF subfieldE(1)] assms
  by auto
```

## 37.6 Finite Extensions

```
lemma (in ring) finite_extension_consistent:
  assumes "subring K R" shows "ring.finite_extension (R ⦇ carrier :=
K ⦈) = finite_extension"
proof -
  have "⋀K' xs. ring.finite_extension (R ⦇ carrier := K ⦈) K' xs = finite_extension
K' xs"
  proof -
    fix K' xs show "ring.finite_extension (R ⦇ carrier := K ⦈) K' xs =
finite_extension K' xs"
      using ring.finite_extension.simps[OF subring_is_ring[OF assms]]
            simple_extension_consistent[OF assms] by (induct xs) (auto)
  qed
  thus ?thesis by blast
qed

lemma (in ring) mono_finite_extension:
  assumes "K ⊆ K'" shows "finite_extension K xs ⊆ finite_extension K'
```

```
xs"
  using mono_simple_extension assms by (induct xs) (auto)

lemma (in ring) finite_extension_carrier:
  assumes "set xs ⊆ carrier R" shows "finite_extension (carrier R) xs
= carrier R"
  using assms simple_extension_carrier by (induct xs) (auto)

lemma (in ring) finite_extension_in_carrier:
  assumes "K ⊆ carrier R" and "set xs ⊆ carrier R" shows "finite_extension
K xs ⊆ carrier R"
  using assms simple_extension_in_carrier by (induct xs) (auto)

lemma (in ring) finite_extension_subring_incl:
  assumes "subring K' R" and "K ⊆ K'" "set xs ⊆ K'" shows "finite_extension
K xs ⊆ K'"
  using ring.finite_extension_in_carrier[OF subring_is_ring[OF assms(1)]]
assms(2-3)
  unfolding finite_extension_consistent[OF assms(1)] by simp

lemma (in ring) finite_extension_incl_aux:
  assumes "K ⊆ carrier R" and "x ∈ carrier R" "set xs ⊆ carrier R"
  shows "finite_extension K xs ⊆ finite_extension K (x # xs)"
  using simple_extension_incl[OF finite_extension_in_carrier[OF assms(1,3)]
assms(2)] by simp

lemma (in ring) finite_extension_incl:
  assumes "K ⊆ carrier R" and "set xs ⊆ carrier R" shows "K ⊆ finite_extension
K xs"
  using finite_extension_incl_aux[OF assms(1)] assms(2) by (induct xs)
(auto)

lemma (in ring) finite_extension_as_eval_img:
  assumes "K ⊆ carrier R" and "x ∈ carrier R" "set xs ⊆ carrier R"
  shows "finite_extension K (x # xs) = (λp. eval p x) ` carrier ((finite_extension
K xs) [X])"
  using simple_extension_as_eval_img[OF finite_extension_in_carrier[OF
assms(1,3)] assms(2)] by simp

lemma (in domain) finite_extension_is_subring:
  assumes "subring K R" "set xs ⊆ carrier R" shows "subring (finite_extension
K xs) R"
  using assms simple_extension_is_subring by (induct xs) (auto)

corollary (in domain) finite_extension_mem:
  assumes subring: "subring K R"
  shows "set xs ⊆ carrier R ⟹ set xs ⊆ finite_extension K xs"
proof (induct xs)
  case Nil
```

```
  then show ?case by simp
next
  case (Cons a xs)
  from Cons(2) have a: "a ∈ carrier R" and xs: "set xs ⊆ carrier R"
by auto
  show ?case
  proof
    fix x assume "x ∈ set (a # xs)"
    then consider "x = a" | "x ∈ set xs" by auto
    then show "x ∈ finite_extension K (a # xs)"
    proof cases
      case 1
      with a have "x ∈ carrier R" by simp
      with xs have "x ∈ finite_extension K (x # xs)"
        using simple_extension_mem[OF finite_extension_is_subring[OF subring]]
by simp
      with 1 show ?thesis by simp
    next
      case 2
      with Cons have *: "x ∈ finite_extension K xs" by auto
      from a xs have "finite_extension K xs ⊆ finite_extension K (a #
xs)"
        by (rule finite_extension_incl_aux[OF subringE(1)[OF subring]])
      with * show ?thesis by auto
    qed
  qed
qed

corollary (in domain) finite_extension_minimal:
  assumes "subring K R" "set xs ⊆ carrier R"
  shows "finite_extension K xs = ⋂ { K'. subring K' R ∧ K ⊆ K' ∧ set
xs ⊆ K' }"
  using finite_extension_is_subring[OF assms] finite_extension_mem[OF
assms]
        finite_extension_incl[OF subringE(1)[OF assms(1)] assms(2)] finite_extension_subring
  by blast

corollary (in domain) finite_extension_same_set:
  assumes "subring K R" "set xs ⊆ carrier R" "set xs = set ys"
  shows "finite_extension K xs = finite_extension K ys"
  using finite_extension_minimal[OF assms(1)] assms(2-3) by auto
```

The reciprocal is also true, but it is more subtle.

```
proposition (in domain) finite_extension_is_subfield:
  assumes "subfield K R" "set xs ⊆ carrier R"
  shows "(⋀x. x ∈ set xs ⟹ (algebraic over K) x) ⟹ subfield (finite_extension
K xs) R"
  using simple_extension_is_subfield algebraic_mono assms
  by (induct xs) (auto, metis finite_extension.simps finite_extension_incl
```

```
subring_props(1))

proposition (in domain) finite_extension_finite_dimension:
  assumes "subfield K R" "set xs ⊆ carrier R"
  shows "(⋀x. x ∈ set xs ⟹ (algebraic over K) x) ⟹ finite_dimension
K (finite_extension K xs)"
    and "finite_dimension K (finite_extension K xs) ⟹ (⋀x. x ∈ set
xs ⟹ (algebraic over K) x)"
proof -
  show "finite_dimension K (finite_extension K xs) ⟹ (⋀x. x ∈ set
xs ⟹ (algebraic over K) x)"
    using finite_dimension_imp_algebraic[OF assms(1)
          finite_extension_is_subring[OF subfieldE(1)[OF assms(1)] assms(2)]]
          finite_extension_mem[OF subfieldE(1)[OF assms(1)] assms(2)]
by auto
next
  show "(⋀x. x ∈ set xs ⟹ (algebraic over K) x) ⟹ finite_dimension
K (finite_extension K xs)"
    using assms(2)
  proof (induct xs, simp add: finite_dimensionI[OF dimension_one[OF assms(1)]])
    case (Cons x xs)
    hence "finite_dimension K (finite_extension K xs)"
      by auto
    moreover have "(algebraic over (finite_extension K xs)) x"
      using algebraic_mono[OF finite_extension_incl[OF subfieldE(3)[OF
assms(1)]]] Cons(2-3) by auto
    moreover have "subfield (finite_extension K xs) R"
      using finite_extension_is_subfield[OF assms(1)] Cons(2-3) by auto
    ultimately show ?case
      using telescopic_base_dim(1)[OF assms(1) _ _
            finite_dimensionI[OF dimension_simple_extension, of _ x]]
Cons(3) by auto
  qed
qed

corollary (in domain) finite_extesion_mem_imp_algebraic:
  assumes "subfield K R" "set xs ⊆ carrier R" and "⋀x. x ∈ set xs ⟹
(algebraic over K) x"
  shows "y ∈ finite_extension K xs ⟹ (algebraic over K) y"
  using finite_dimension_imp_algebraic[OF assms(1)
        finite_extension_is_subring[OF subfieldE(1)[OF assms(1)] assms(2)]]
        finite_extension_finite_dimension(1)[OF assms(1-2)] assms(3) by
auto

corollary (in domain) simple_extesion_mem_imp_algebraic:
  assumes "subfield K R" "x ∈ carrier R" "(algebraic over K) x"
  shows "y ∈ simple_extension K x ⟹ (algebraic over K) y"
  using finite_extesion_mem_imp_algebraic[OF assms(1), of "[ x ]"] assms(2-3)
by auto
```

## 37.7 Arithmetic of algebraic numbers

We show that the set of algebraic numbers of a field over a subfield K is a subfield itself.

```
lemma (in field) subfield_of_algebraics:
  assumes "subfield K R" shows "subfield { x ∈ carrier R. (algebraic
over K) x } R"
proof -
  let ?set_of_algebraics = "{ x ∈ carrier R. (algebraic over K) x }"

  show ?thesis
  proof (rule subfieldI'[OF subringI])
    show "?set_of_algebraics ⊆ carrier R" and "1 ∈ ?set_of_algebraics"
      using algebraic_self[OF _ subringE(3)] subfieldE(1)[OF assms(1)]
by auto
  next
    fix x y assume x: "x ∈ ?set_of_algebraics" and y: "y ∈ ?set_of_algebraics"
    have "⊖ x ∈ simple_extension K x"
      using subringE(5)[OF simple_extension_is_subring[OF subfieldE(1)]]
            simple_extension_mem[OF subfieldE(1)] assms(1) x by auto
    thus "⊖ x ∈ ?set_of_algebraics"
      using simple_extesion_mem_imp_algebraic[OF assms] x by auto

    have "x ⊕ y ∈ finite_extension K [ x, y ]" and "x ⊗ y ∈ finite_extension
K [ x, y ]"
      using subringE(6-7)[OF finite_extension_is_subring[OF subfieldE(1)[OF
assms(1)]], of "[ x, y ]"]
            finite_extension_mem[OF subfieldE(1)[OF assms(1)], of "[ x,
y ]"] x y by auto
    thus "x ⊕ y ∈ ?set_of_algebraics" and "x ⊗ y ∈ ?set_of_algebraics"
      using finite_extesion_mem_imp_algebraic[OF assms, of "[ x, y ]"]
x y by auto
  next
    fix z assume z: "z ∈ ?set_of_algebraics - { 0 }"
    have "inv z ∈ simple_extension K z"
      using subfield_m_inv(1)[of "simple_extension K z"]
            simple_extension_is_subfield[OF assms, of z]
            simple_extension_mem[OF subfieldE(1)] assms(1) z by auto
    thus "inv z ∈ ?set_of_algebraics"
      using simple_extesion_mem_imp_algebraic[OF assms] field_Units z
by auto
  qed
qed

end


theory Algebraic_Closure
  imports Indexed_Polynomials Polynomial_Divisibility Finite_Extensions
```

**begin**

# 38 Algebraic Closure

## 38.1 Definitions

**inductive** iso_incl :: "'a ring ⇒ 'a ring ⇒ bool" (**infixl** ‹$\lesssim$› 65) **for** A B
  **where** iso_inclI [intro]: "id ∈ ring_hom A B ⟹ iso_incl A B"

**definition** law_restrict :: "('a, 'b) ring_scheme ⇒ 'a ring"
  **where** "law_restrict R ≡ (ring.truncate R)
          ⦇ mult := (λa ∈ carrier R. λb ∈ carrier R. a $\otimes_R$ b),
            add := (λa ∈ carrier R. λb ∈ carrier R. a $\oplus_R$ b) ⦈"

**definition** (**in** ring) $\sigma$ :: "'a list ⇒ ((('a list × nat) multiset) ⇒ 'a) list"
  **where** "$\sigma$ P = map indexed_const P"

**definition** (**in** ring) extensions :: "((('a list × nat) multiset) ⇒ 'a) ring set"
  **where** "extensions ≡ { L — such that.
          — i (field L) ∧
          — ii (indexed_const ∈ ring_hom R L) ∧
          — iii (∀$\mathcal{P}$ ∈ carrier L. carrier_coeff $\mathcal{P}$) ∧
          — iv (∀$\mathcal{P}$ ∈ carrier L. ∀P ∈ carrier (poly_ring R). ∀i.
                ¬ index_free $\mathcal{P}$ (P, i) ⟶
                  $\mathcal{X}_{(P,\ i)}$ ∈ carrier L ∧ (ring.eval L) ($\sigma$ P) $\mathcal{X}_{(P,\ i)}$ = $0_L$) }"

**abbreviation** (**in** ring) restrict_extensions :: "((('a list × nat) multiset) ⇒ 'a) ring set" (‹$\mathcal{S}$›)
  **where** "$\mathcal{S}$ ≡ law_restrict ` extensions"

## 38.2 Basic Properties

**lemma** law_restrict_carrier: "carrier (law_restrict R) = carrier R"
  **by** (simp add: law_restrict_def ring.defs)

**lemma** law_restrict_one: "one (law_restrict R) = one R"
  **by** (simp add: law_restrict_def ring.defs)

**lemma** law_restrict_zero: "zero (law_restrict R) = zero R"
  **by** (simp add: law_restrict_def ring.defs)

**lemma** law_restrict_mult: "monoid.mult (law_restrict R) = (λa ∈ carrier R. λb ∈ carrier R. a $\otimes_R$ b)"
  **by** (simp add: law_restrict_def ring.defs)

```
lemma law_restrict_add: "add (law_restrict R) = (λa ∈ carrier R. λb
∈ carrier R. a ⊕R b)"
  by (simp add: law_restrict_def ring.defs)

lemma (in ring) law_restrict_is_ring: "ring (law_restrict R)"
  by (unfold_locales) (auto simp add: law_restrict_def Units_def ring.defs,
      simp_all add: a_assoc a_comm m_assoc l_distr r_distr a_lcomm)

lemma (in field) law_restrict_is_field: "field (law_restrict R)"
proof -
  have "comm_monoid_axioms (law_restrict R)"
    using m_comm unfolding comm_monoid_axioms_def law_restrict_carrier
law_restrict_mult by auto
  then interpret L: cring "law_restrict R"
    using cring.intro law_restrict_is_ring comm_monoid.intro ring.is_monoid
by auto
  have "Units R = Units (law_restrict R)"
    unfolding Units_def law_restrict_carrier law_restrict_mult law_restrict_one
by auto
  thus ?thesis
    using L.cring_fieldI unfolding field_Units law_restrict_carrier law_restrict_zero
by simp
qed

lemma law_restrict_iso_imp_eq:
  assumes "id ∈ ring_iso (law_restrict A) (law_restrict B)" and "ring
A" and "ring B"
  shows "law_restrict A = law_restrict B"
proof -
  have "carrier A = carrier B"
    using ring_iso_memE(5)[OF assms(1)] unfolding bij_betw_def law_restrict_def
by (simp add: ring.defs)
  hence mult: "a ⊗law_restrict A b = a ⊗law_restrict B b"
    and add:  "a ⊕law_restrict A b = a ⊕law_restrict B b" for a b
    using ring_iso_memE(2-3)[OF assms(1)] unfolding law_restrict_def by
(auto simp add: ring.defs)
  have "monoid.mult (law_restrict A) = monoid.mult (law_restrict B)"
    using mult by auto
  moreover have "add (law_restrict A) = add (law_restrict B)"
    using add by auto
  moreover from ‹carrier A = carrier B› have "carrier (law_restrict
A) = carrier (law_restrict B)"
    unfolding law_restrict_def by (simp add: ring.defs)
  moreover have "0law_restrict A = 0law_restrict B"
    using ring_hom_zero[OF _ assms(2-3)[THEN ring.law_restrict_is_ring]]
assms(1)
    unfolding ring_iso_def by auto
  moreover have "1law_restrict A = 1law_restrict B"
```

```
    using ring_iso_memE(4)[OF assms(1)] by simp
  ultimately show ?thesis by simp
qed

lemma law_restrict_hom: "h ∈ ring_hom A B ⟷ h ∈ ring_hom (law_restrict
A) (law_restrict B)"
proof
  assume "h ∈ ring_hom A B" thus "h ∈ ring_hom (law_restrict A) (law_restrict
B)"
    by (auto intro!: ring_hom_memI dest: ring_hom_memE simp: law_restrict_def
ring.defs)
next
  assume h: "h ∈ ring_hom (law_restrict A) (law_restrict B)" show "h
∈ ring_hom A B"
    using ring_hom_memE[OF h] by (auto intro!: ring_hom_memI simp: law_restrict_def
ring.defs)
qed

lemma iso_incl_hom: "A ≲ B ⟷ (law_restrict A) ≲ (law_restrict B)"
  using law_restrict_hom iso_incl.simps by blast
```

## 38.3 Partial Order

```
lemma iso_incl_backwards:
  assumes "A ≲ B" shows "id ∈ ring_hom A B"
  using assms by cases

lemma iso_incl_antisym_aux:
  assumes "A ≲ B" and "B ≲ A" shows "id ∈ ring_iso A B"
proof -
  have hom: "id ∈ ring_hom A B" "id ∈ ring_hom B A"
    using assms(1-2)[THEN iso_incl_backwards] by auto
  thus ?thesis
    using hom[THEN ring_hom_memE(1)] by (auto simp add: ring_iso_def bij_betw_def
inj_on_def)
qed

lemma iso_incl_refl: "A ≲ A"
  by (rule iso_inclI[OF ring_hom_memI], auto)

lemma iso_incl_trans:
  assumes "A ≲ B" and "B ≲ C" shows "A ≲ C"
  using ring_hom_trans[OF assms[THEN iso_incl_backwards]] by auto

lemma (in ring) iso_incl_antisym:
  assumes "A ∈ 𝒮" "B ∈ 𝒮" and "A ≲ B" "B ≲ A" shows "A = B"
proof -
  obtain A' B' :: "(('a list × nat) multiset ⇒ 'a) ring"
    where A: "A = law_restrict A'" "ring A'" and B: "B = law_restrict
```

```
B'" "ring B'"
    using assms(1-2) field.is_ring by (auto simp add: extensions_def)
  thus ?thesis
    using law_restrict_iso_imp_eq iso_incl_antisym_aux[OF assms(3-4)]
by simp
qed

lemma (in ring) iso_incl_partial_order: "partial_order_on 𝒮 (relation_of
(≲) 𝒮)"
  using iso_incl_refl iso_incl_trans iso_incl_antisym by (rule partial_order_on_relation_of

lemma iso_inclE:
  assumes "ring A" and "ring B" and "A ≲ B" shows "ring_hom_ring A
B id"
  using iso_incl_backwards[OF assms(3)] ring_hom_ring.intro[OF assms(1-2)]
  unfolding symmetric[OF ring_hom_ring_axioms_def] by simp

lemma iso_incl_imp_same_eval:
  assumes "ring A" and "ring B" and "A ≲ B" and "a ∈ carrier A" and
"set p ⊆ carrier A"
  shows "(ring.eval A) p a = (ring.eval B) p a"
  using ring_hom_ring.eval_hom'[OF iso_inclE[OF assms(1-3)] assms(4-5)]
by simp
```

## 38.4 Extensions Non Empty

```
lemma (in ring) indexed_const_is_inj: "inj indexed_const"
  unfolding indexed_const_def by (rule inj_onI, metis)

lemma (in ring) indexed_const_inj_on: "inj_on indexed_const (carrier
R)"
  unfolding indexed_const_def by (rule inj_onI, metis)

lemma (in field) extensions_non_empty: "𝒮 ≠ {}"
proof -
  have "image_ring indexed_const R ∈ extensions"
  proof (auto simp add: extensions_def)
    show "field (image_ring indexed_const R)"
      using inj_imp_image_ring_is_field[OF indexed_const_inj_on] .
  next
    show "indexed_const ∈ ring_hom R (image_ring indexed_const R)"
      using inj_imp_image_ring_iso[OF indexed_const_inj_on] unfolding
ring_iso_def by auto
  next
    fix 𝒫 :: "(('a list × nat) multiset) ⇒ 'a" and P and i
    assume "𝒫 ∈ carrier (image_ring indexed_const R)"
    then obtain k where "k ∈ carrier R" and "𝒫 = indexed_const k"
      unfolding image_ring_carrier by blast
    hence "index_free 𝒫 (P, i)" for P i
```

```
      unfolding index_free_def indexed_const_def by auto
    thus "¬ index_free 𝒫 (P, i) ⟹ 𝒳(P, i) ∈ carrier (image_ring indexed_const
R)"
     and "¬ index_free 𝒫 (P, i) ⟹ ring.eval (image_ring indexed_const
R) (σ P) 𝒳(P, i) = 0image_ring indexed_const R"
      by auto
    from ‹k ∈ carrier R› and ‹𝒫 = indexed_const k› show "carrier_coeff
𝒫"
      unfolding indexed_const_def carrier_coeff_def by auto
  qed
  thus ?thesis
    by blast
qed
```

## 38.5 Chains

```
definition union_ring :: "(('a, 'c) ring_scheme) set ⇒ 'a ring"
  where "union_ring C =
          ⦇ carrier = (⋃(carrier ‘ C)),
        monoid.mult = (λa b. (monoid.mult (SOME R. R ∈ C ∧ a ∈ carrier
R ∧ b ∈ carrier R) a b)),
                one = one (SOME R. R ∈ C),
               zero = zero (SOME R. R ∈ C),
                add = (λa b. (add (SOME R. R ∈ C ∧ a ∈ carrier R ∧ b
∈ carrier R) a b)) ⦈"


lemma union_ring_carrier: "carrier (union_ring C) = (⋃(carrier ‘ C))"
  unfolding union_ring_def by simp

context
  fixes C :: "'a ring set"
  assumes field_chain: "⋀R. R ∈ C ⟹ field R" and chain: "⋀R S. ⟦ R
∈ C; S ∈ C ⟧ ⟹ R ≲ S ∨ S ≲ R"
begin

lemma ring_chain: "R ∈ C ⟹ ring R"
  using field.is_ring[OF field_chain] by blast

lemma same_one_same_zero:
  assumes "R ∈ C" shows "1union_ring C = 1R" and "0union_ring C = 0R"
proof -
  have "1R = 1S" if "R ∈ C" and "S ∈ C" for R S
    using ring_hom_one[of id] chain[OF that] unfolding iso_incl.simps
by auto
  moreover have "0R = 0S" if "R ∈ C" and "S ∈ C" for R S
    using chain[OF that] ring_hom_zero[OF _ ring_chain ring_chain] that
unfolding iso_incl.simps by auto
  ultimately have "one (SOME R. R ∈ C) = 1R" and "zero (SOME R. R ∈ C)
```

```
= 𝟬R"
    using assms by (metis (mono_tags) someI)+
  thus "𝟭union_ring C = 𝟭R" and "𝟬union_ring C = 𝟬R"
    unfolding union_ring_def by auto
qed

lemma same_laws:
  assumes "R ∈ C" and "a ∈ carrier R" and "b ∈ carrier R"
  shows "a ⊗union_ring C b = a ⊗R b" and "a ⊕union_ring C b = a ⊕R b"
proof -
  have "a ⊗R b = a ⊗S b"
    if "R ∈ C" "a ∈ carrier R" "b ∈ carrier R" and "S ∈ C" "a ∈ carrier
S" "b ∈ carrier S" for R S
    using ring_hom_memE(2)[of id R S] ring_hom_memE(2)[of id S R] that
chain[OF that(1,4)]
    unfolding iso_incl.simps by auto
  moreover have "a ⊕R b = a ⊕S b"
    if "R ∈ C" "a ∈ carrier R" "b ∈ carrier R" and "S ∈ C" "a ∈ carrier
S" "b ∈ carrier S" for R S
    using ring_hom_memE(3)[of id R S] ring_hom_memE(3)[of id S R] that
chain[OF that(1,4)]
    unfolding iso_incl.simps by auto
  ultimately
  have "monoid.mult (SOME R. R ∈ C ∧ a ∈ carrier R ∧ b ∈ carrier R)
a b = a ⊗R b"
   and        "add (SOME R. R ∈ C ∧ a ∈ carrier R ∧ b ∈ carrier R)
a b = a ⊕R b"
    using assms by (metis (mono_tags, lifting) someI)+
  thus "a ⊗union_ring C b = a ⊗R b" and "a ⊕union_ring C b = a ⊕R b"
    unfolding union_ring_def by auto
qed

lemma exists_superset_carrier:
  assumes "finite S" and "S ≠ {}" and "S ⊆ carrier (union_ring C)"
  shows "∃R ∈ C. S ⊆ carrier R"
  using assms
proof (induction, simp)
  case (insert s S)
  obtain R where R: "s ∈ carrier R" "R ∈ C"
    using insert(5) unfolding union_ring_def by auto
  show ?case
  proof (cases)
    assume "S = {}" thus ?thesis
      using R by blast
  next
    assume "S ≠ {}"
    then obtain T where T: "S ⊆ carrier T" "T ∈ C"
      using insert(3,5) by blast
    have "carrier R ⊆ carrier T ∨ carrier T ⊆ carrier R"
```

```
      using ring_hom_memE(1)[of id R] ring_hom_memE(1)[of id T] chain[OF
R(2) T(2)]
      unfolding iso_incl.simps by auto
    thus ?thesis
      using R T by auto
  qed
qed

lemma union_ring_is_monoid:
  assumes "C ≠ {}" shows "comm_monoid (union_ring C)"
proof
  fix a b c
  assume "a ∈ carrier (union_ring C)" "b ∈ carrier (union_ring C)" "c
∈ carrier (union_ring C)"
  then obtain R where R: "R ∈ C" "a ∈ carrier R" "b ∈ carrier R" "c
∈ carrier R"
    using exists_superset_carrier[of "{ a, b, c }"] by auto
  then interpret field R
    using field_chain by simp

  show "a ⊗union_ring C b ∈ carrier (union_ring C)"
    using R(1-3) unfolding same_laws(1)[OF R(1-3)] unfolding union_ring_def
by auto
  show "(a ⊗union_ring C b) ⊗union_ring C c = a ⊗union_ring C (b ⊗union_ring C
c)"
   and "a ⊗union_ring C b = b ⊗union_ring C a"
   and "1union_ring C ⊗union_ring C a = a"
   and "a ⊗union_ring C 1union_ring C = a"
    using same_one_same_zero[OF R(1)] same_laws(1)[OF R(1)] R(2-4) m_assoc
m_comm by auto
next
  show "1union_ring C ∈ carrier (union_ring C)"
    using ring.ring_simprules(6)[OF ring_chain] assms same_one_same_zero(1)
    unfolding union_ring_carrier by auto
qed

lemma union_ring_is_abelian_group:
  assumes "C ≠ {}" shows "cring (union_ring C)"
proof (rule cringI[OF abelian_groupI union_ring_is_monoid[OF assms]])
  fix a b c
  assume "a ∈ carrier (union_ring C)" "b ∈ carrier (union_ring C)" "c
∈ carrier (union_ring C)"
  then obtain R where R: "R ∈ C" "a ∈ carrier R" "b ∈ carrier R" "c
∈ carrier R"
    using exists_superset_carrier[of "{ a, b, c }"] by auto
  then interpret field R
    using field_chain by simp

  show "a ⊕union_ring C b ∈ carrier (union_ring C)"
```

```
    using R(1-3) unfolding same_laws(2)[OF R(1-3)] unfolding union_ring_def
by auto
  show "(a ⊕union_ring C b) ⊗union_ring C c = (a ⊗union_ring C c) ⊕union_ring C
(b ⊗union_ring C c)"
   and "(a ⊕union_ring C b) ⊕union_ring C c = a ⊕union_ring C (b ⊕union_ring C
c)"
   and "a ⊕union_ring C b = b ⊕union_ring C a"
   and "0union_ring C ⊕union_ring C a = a"
    using same_one_same_zero[OF R(1)] same_laws[OF R(1)] R(2-4) l_distr
a_assoc a_comm by auto
  have "∃a' ∈ carrier R. a' ⊕union_ring C a = 0union_ring C"
    using same_laws(2)[OF R(1)] R(2) same_one_same_zero[OF R(1)] by simp
  with ‹R ∈ C› show "∃y ∈ carrier (union_ring C). y ⊕union_ring C a
= 0union_ring C"
    unfolding union_ring_carrier by auto
next
  show "0union_ring C ∈ carrier (union_ring C)"
    using ring.ring_simprules(2)[OF ring_chain] assms same_one_same_zero(2)
    unfolding union_ring_carrier by auto
qed

lemma union_ring_is_field :
  assumes "C ≠ {}" shows "field (union_ring C)"
proof (rule cring.cring_fieldI[OF union_ring_is_abelian_group[OF assms]])
  have "carrier (union_ring C) - { 0union_ring C } ⊆ Units (union_ring
C)"
  proof
    fix a assume "a ∈ carrier (union_ring C) - { 0union_ring C }"
    hence "a ∈ carrier (union_ring C)" and "a ≠ 0union_ring C"
      by auto
    then obtain R where R: "R ∈ C" "a ∈ carrier R"
      using exists_superset_carrier[of "{ a }"] by auto
    then interpret field R
      using field_chain by simp

    from ‹a ∈ carrier R› and ‹a ≠ 0union_ring C› have "a ∈ Units R"
      unfolding same_one_same_zero[OF R(1)] field_Units by auto
    hence "∃a' ∈ carrier R. a' ⊗union_ring C a = 1union_ring C ∧ a ⊗union_ring C
a' = 1union_ring C"
      using same_laws[OF R(1)] same_one_same_zero[OF R(1)] R(2) unfold-
ing Units_def by auto
    with ‹R ∈ C› and ‹a ∈ carrier (union_ring C)› show "a ∈ Units
(union_ring C)"
      unfolding Units_def union_ring_carrier by auto
  qed
  moreover have "0union_ring C ∉ Units (union_ring C)"
  proof (rule ccontr)
    assume "¬ 0union_ring C ∉ Units (union_ring C)"
    then obtain a where a: "a ∈ carrier (union_ring C)" "a ⊗union_ring C
```

```
0union_ring C = 1union_ring C"
      unfolding Units_def by auto
    then obtain R where R: "R ∈ C" "a ∈ carrier R"
      using exists_superset_carrier[of "{ a }"] by auto
    then interpret field R
      using field_chain by simp
    have "1R = 0R"
      using a R same_laws(1)[OF R(1)] same_one_same_zero[OF R(1)] by auto
    thus False
      using one_not_zero by simp
  qed
  hence "Units (union_ring C) ⊆ carrier (union_ring C) - { 0union_ring C
}"
    unfolding Units_def by auto
  ultimately show "Units (union_ring C) = carrier (union_ring C) - { 0union_ring C
}"
    by simp
qed

lemma union_ring_is_upper_bound:
  assumes "R ∈ C" shows "R ≲ union_ring C"
  using ring_hom_memI[of R id "union_ring C"] same_laws[of R] same_one_same_zero[of
R] assms
  unfolding union_ring_carrier by auto

end
```

## 38.6 Zorn

```
lemma (in ring) exists_core_chain:
  assumes "C ∈ Chains (relation_of (≲) 𝒮)" obtains C' where "C' ⊆ extensions"
and "C = law_restrict ` C'"
  using Chains_relation_of[OF assms] by (meson subset_image_iff)

lemma (in ring) core_chain_is_chain:
  assumes "law_restrict ` C ∈ Chains (relation_of (≲) 𝒮)" shows "⋀R
S. ⟦ R ∈ C; S ∈ C ⟧ ⟹ R ≲ S ∨ S ≲ R"
proof -
  fix R S assume "R ∈ C" and "S ∈ C" thus "R ≲ S ∨ S ≲ R"
    using assms(1) unfolding iso_incl_hom[of R] iso_incl_hom[of S] Chains_def
relation_of_def
    by auto
qed

lemma (in field) exists_maximal_extension:
  shows "∃M ∈ 𝒮. ∀L ∈ 𝒮. M ≲ L ⟶ L = M"
proof (rule predicate_Zorn[OF iso_incl_partial_order])
  fix C assume C: "C ∈ Chains (relation_of (≲) 𝒮)"
  show "∃L ∈ 𝒮. ∀R ∈ C. R ≲ L"
```

```
  proof (cases)
    assume "C = {}" thus ?thesis
      using extensions_non_empty by auto
  next
    assume "C ≠ {}"
    from ‹C ∈ Chains (relation_of (≲) 𝒮)›
    obtain C' where C': "C' ⊆ extensions" "C = law_restrict ` C'"
      using exists_core_chain by auto
    with ‹C ≠ {}› obtain S where S: "S ∈ C'" and "C' ≠ {}"
      by auto

    have core_chain: "⋀R. R ∈ C' ⟹ field R" "⋀R S. ⟦ R ∈ C'; S ∈ C'
⟧ ⟹ R ≲ S ∨ S ≲ R"
      using core_chain_is_chain[of C'] C' C unfolding extensions_def by
auto
    from ‹C' ≠ {}› interpret Union: field "union_ring C'"
        using union_ring_is_field[OF core_chain] C'(1) by blast

    have "union_ring C' ∈ extensions"
    proof (auto simp add: extensions_def)
      show "field (union_ring C')"
        using Union.field_axioms .
    next
      from ‹S ∈ C'› have "indexed_const ∈ ring_hom R S"
        using C'(1) unfolding extensions_def by auto
      thus "indexed_const ∈ ring_hom R (union_ring C')"
        using ring_hom_trans[of _ R S id] union_ring_is_upper_bound[OF
core_chain S]
        unfolding iso_incl.simps by auto
    next
      show "a ∈ carrier (union_ring C') ⟹ carrier_coeff a" for a
        using C'(1) unfolding union_ring_carrier extensions_def by auto
    next
      fix 𝒫 P i
      assume "𝒫 ∈ carrier (union_ring C')"
        and P: "P ∈ carrier (poly_ring R)"
        and not_index_free: "¬ index_free 𝒫 (P, i)"
      from ‹𝒫 ∈ carrier (union_ring C')› obtain T where T: "T ∈ C'"
"𝒫 ∈ carrier T"
        using exists_superset_carrier[of C' "{ 𝒫 }"] core_chain by auto
      hence "𝒳(P, i) ∈ carrier T" and "(ring.eval T) (σ P) 𝒳(P, i) =
0T"
        and field: "field T" and hom: "indexed_const ∈ ring_hom R T"
        using P not_index_free C'(1) unfolding extensions_def by auto
      with ‹T ∈ C'› show "𝒳(P, i) ∈ carrier (union_ring C')"
        unfolding union_ring_carrier by auto
      have "set P ⊆ carrier R"
        using P unfolding sym[OF univ_poly_carrier] polynomial_def by
auto
```

```
      hence "set (σ P) ⊆ carrier T"
        using ring_hom_memE(1)[OF hom] unfolding σ_def by (induct P) (auto)
      with ‹𝒳(P, i) ∈ carrier T› and ‹(ring.eval T) (σ P) 𝒳(P, i) =
𝟬T›
      show "(ring.eval (union_ring C')) (σ P) 𝒳(P, i) = 𝟬union_ring C'"
        using iso_incl_imp_same_eval[OF field.is_ring[OF field] Union.is_ring
              union_ring_is_upper_bound[OF core_chain T(1)]] same_one_same_zero(2)[OF
core_chain T(1)]
        by auto
    qed
    moreover have "R ≲ law_restrict (union_ring C')" if "R ∈ C" for R
      using that union_ring_is_upper_bound[OF core_chain] iso_incl_hom
unfolding C' by auto
    ultimately show ?thesis
      by blast
  qed
qed
```

## 38.7 Existence of roots

```
lemma polynomial_hom:
  assumes "h ∈ ring_hom R S" and "field R" and "field S"
  shows "p ∈ carrier (poly_ring R) ⟹ (map h p) ∈ carrier (poly_ring
S)"
proof -
  assume "p ∈ carrier (poly_ring R)"
  interpret ring_hom_ring R S h
    using ring_hom_ringI2[OF assms(2-3)[THEN field.is_ring] assms(1)]
.

  from ‹p ∈ carrier (poly_ring R)› have "set p ⊆ carrier R" and lc:
"p ≠ [] ⟹ lead_coeff p ≠ 𝟬R"
    unfolding sym[OF univ_poly_carrier] polynomial_def by auto
  hence "set (map h p) ⊆ carrier S"
    by (induct p) (auto)
  moreover have "h a = 𝟬S ⟹ a = 𝟬R" if "a ∈ carrier R" for a
    using non_trivial_field_hom_is_inj[OF assms(1-3)] that unfolding inj_on_def
by simp
  with ‹set p ⊆ carrier R› have "lead_coeff (map h p) ≠ 𝟬S" if "p ≠
[]"
    using lc[OF that] that by (cases p) (auto)
  ultimately show ?thesis
    unfolding sym[OF univ_poly_carrier] polynomial_def by auto
qed

lemma (in ring_hom_ring) subfield_polynomial_hom:
  assumes "subfield K R" and "𝟭S ≠ 𝟬S"
  shows "p ∈ carrier (K[X]R) ⟹ (map h p) ∈ carrier ((h ` K)[X]S)"
proof -
```

```
  assume "p ∈ carrier (K[X]R)"
  hence "p ∈ carrier (poly_ring (R ⦇ carrier := K ⦈))"
    using R.univ_poly_consistent[OF subfieldE(1)[OF assms(1)]] by simp
  moreover have "h ∈ ring_hom (R ⦇ carrier := K ⦈) (S ⦇ carrier := h
‘ K ⦈)"
    using hom_mult subfieldE(3)[OF assms(1)] unfolding ring_hom_def subset_iff
by auto
  moreover have "field (R ⦇ carrier := K ⦈)" and "field (S ⦇ carrier
:= (h ‘ K) ⦈)"
    using R.subfield_iff(2)[OF assms(1)] S.subfield_iff(2)[OF img_is_subfield(2)[OF
assms]] by simp+
  ultimately have "(map h p) ∈ carrier (poly_ring (S ⦇ carrier := h ‘
K ⦈))"
    using polynomial_hom[of h "R ⦇ carrier := K ⦈" "S ⦇ carrier := h ‘
K ⦈"] by auto
  thus ?thesis
    using S.univ_poly_consistent[OF subfieldE(1)[OF img_is_subfield(2)[OF
assms]]] by simp
qed


lemma (in field) exists_root:
  assumes "M ∈ extensions" and "⋀L. ⟦ L ∈ extensions; M ≲ L ⟧ ⟹ law_restrict
L = law_restrict M"
    and "P ∈ carrier (poly_ring R)"
  shows "(ring.splitted M) (σ P)"
proof (rule ccontr)
  from ‹M ∈ extensions› interpret M: field M + Hom: ring_hom_ring R M
"indexed_const"
    using ring_hom_ringI2[OF ring_axioms field.is_ring] unfolding extensions_def
by auto
  interpret UP: principal_domain "poly_ring M"
    using M.univ_poly_is_principal[OF M.carrier_is_subfield] .

  assume not_splitted: "¬ (ring.splitted M) (σ P)"
  have "(σ P) ∈ carrier (poly_ring M)"
    using polynomial_hom[OF Hom.homh field_axioms M.field_axioms assms(3)]
unfolding σ_def by simp
  then obtain Q
    where Q: "Q ∈ carrier (poly_ring M)" "pirreducibleM (carrier M) Q"
"Q pdividesM (σ P)"
      and degree_gt: "degree Q > 1"
    using M.trivial_factors_imp_splitted[of "σ P"] not_splitted by force

  from ‹(σ P) ∈ carrier (poly_ring M)› have "(σ P) ≠ []"
    using M.degree_zero_imp_splitted[of "σ P"] not_splitted unfolding
σ_def by auto

  have "∃i. ∀𝒫 ∈ carrier M. index_free 𝒫 (P, i)"
```

```
  proof (rule ccontr)
    assume "∄i. ∀𝒫 ∈ carrier M. index_free 𝒫 (P, i)"
    then have "𝒳(P, i) ∈ carrier M" and "(ring.eval M) (σ P) 𝒳(P, i)
= 0M" for i
      using assms(1,3) unfolding extensions_def by blast+
    with ‹(σ P) ≠ []› have "((λi :: nat. 𝒳(P, i)) ‘ UNIV) ⊆ { a. (ring.is_root
M) (σ P) a }"
      unfolding M.is_root_def by auto
    moreover have "inj (λi :: nat. 𝒳(P, i))"
      unfolding indexed_var_def indexed_const_def indexed_pmult_def inj_def
      by (metis (no_types, lifting) add_mset_eq_singleton_iff diff_single_eq_union
                                    multi_member_last prod.inject zero_not_one)
    hence "infinite ((λi :: nat. 𝒳(P, i)) ‘ UNIV)"
      unfolding infinite_iff_countable_subset by auto
    ultimately have "infinite { a. (ring.is_root M) (σ P) a }"
      using finite_subset by auto
    with ‹(σ P) ∈ carrier (poly_ring M)› show False
      using M.finite_number_of_roots by simp
  qed
  then obtain i :: nat where "∀𝒫 ∈ carrier M. index_free 𝒫 (P, i)"
    by blast

  then have hyps:
    — i   "field M"
    — ii  "⋀𝒫. 𝒫 ∈ carrier M ⟹ carrier_coeff 𝒫"
    — iii "⋀𝒫. 𝒫 ∈ carrier M ⟹ index_free 𝒫 (P, i)"
    — iv  "0M = indexed_const 0"
    using assms(1,3) unfolding extensions_def by auto

  define image_poly where "image_poly = image_ring (eval_pmod M (P, i)
Q) (poly_ring M)"
  with ‹degree Q > 1› have "M ≲ image_poly"
    using image_poly_iso_incl[OF hyps Q(1)] by auto
  moreover have is_field: "field image_poly"
    using image_poly_is_field[OF hyps Q(1-2)] unfolding image_poly_def
by simp
  moreover have "image_poly ∈ extensions"
  proof (auto simp add: extensions_def is_field)
    fix 𝒫 assume "𝒫 ∈ carrier image_poly"
    then obtain R where 𝒫: "𝒫 = eval_pmod M (P, i) Q R" and "R ∈ carrier
(poly_ring M)"
      unfolding image_poly_def image_ring_carrier by auto
    hence "M.pmod R Q ∈ carrier (poly_ring M)"
      using M.long_division_closed(2)[OF M.carrier_is_subfield _ Q(1)]
by simp
    hence "list_all carrier_coeff (M.pmod R Q)"
      using hyps(2) unfolding sym[OF univ_poly_carrier] list_all_iff polynomial_def
by auto
    thus "carrier_coeff 𝒫"
```

```
      using indexed_eval_in_carrier[of "M.pmod R Q"] unfolding 𝒫 by simp
  next
    from ‹M ≲ image_poly› show "indexed_const ∈ ring_hom R image_poly"
      using ring_hom_trans[OF Hom.homh, of id] unfolding iso_incl.simps
by simp
  next
    from ‹M ≲ image_poly› interpret Id: ring_hom_ring M image_poly id
      using iso_inclE[OF M.ring_axioms field.is_ring[OF is_field]] by
simp

    fix 𝒫 S j
    assume A: "𝒫 ∈ carrier image_poly" "¬ index_free 𝒫 (S, j)" "S ∈
carrier (poly_ring R)"
    have "𝒳_(S, j) ∈ carrier image_poly ∧ Id.eval (σ S) 𝒳_(S, j) = 0_image_poly"
    proof (cases)
      assume "(P, i) ≠ (S, j)"
      then obtain Q' where "Q' ∈ carrier M" and "¬ index_free Q' (S,
j)"
        using A(1) image_poly_index_free[OF hyps Q(1) _ A(2)] unfold-
ing image_poly_def by auto
      hence "𝒳_(S, j) ∈ carrier M" and "M.eval (σ S) 𝒳_(S, j) = 0_M"
        using assms(1) A(3) unfolding extensions_def by auto
      moreover have "σ S ∈ carrier (poly_ring M)"
        using polynomial_hom[OF Hom.homh field_axioms M.field_axioms A(3)]
unfolding σ_def .
      ultimately show ?thesis
        using Id.eval_hom[OF M.carrier_is_subring] Id.hom_closed Id.hom_zero
by auto
    next
      assume "¬ (P, i) ≠ (S, j)" hence S: "(P, i) = (S, j)"
        by simp
      have poly_hom: "R ∈ carrier (poly_ring image_poly)" if "R ∈ carrier
(poly_ring M)" for R
        using polynomial_hom[OF Id.homh M.field_axioms is_field that]
by simp
      have "𝒳_(S, j) ∈ carrier image_poly"
        using eval_pmod_var(2)[OF hyps Hom.homh Q(1) degree_gt] unfold-
ing image_poly_def S by simp
      moreover have "Id.eval Q 𝒳_(S, j) = 0_image_poly"
        using image_poly_eval_indexed_var[OF hyps Hom.homh Q(1) degree_gt
Q(2)] unfolding image_poly_def S by simp
      moreover have "Q pdivides_image_poly (σ S)"
      proof -
        obtain R where R: "R ∈ carrier (poly_ring M)" "σ S = Q ⊗_poly_ring M
R"
          using Q(3) S unfolding pdivides_def by auto
        moreover have "set Q ⊆ carrier M" and "set R ⊆ carrier M"
          using Q(1) R(1) unfolding sym[OF univ_poly_carrier] polynomial_def
by auto
```

```
        ultimately have "Id.normalize (σ S) = Q ⊗poly_ring image_poly R"
          using Id.poly_mult_hom'[of Q R] unfolding univ_poly_mult by
simp
        moreover have "σ S ∈ carrier (poly_ring M)"
          using polynomial_hom[OF Hom.homh field_axioms M.field_axioms
A(3)] unfolding σ_def .
        hence "σ S ∈ carrier (poly_ring image_poly)"
          using polynomial_hom[OF Id.homh M.field_axioms is_field] by
simp
        hence "Id.normalize (σ S) = σ S"
          using Id.normalize_polynomial unfolding sym[OF univ_poly_carrier]
by simp
        ultimately show ?thesis
          using poly_hom[OF Q(1)] poly_hom[OF R(1)]
          unfolding pdivides_def factor_def univ_poly_mult by auto
      qed
      moreover have "Q ∈ carrier (poly_ring (image_poly))"
        using poly_hom[OF Q(1)] by simp
      ultimately show ?thesis
        using domain.pdivides_imp_root_sharing[OF field.axioms(1)[OF is_field],
of Q] by auto
    qed
    thus "𝒳(S, j) ∈ carrier image_poly" and "Id.eval (σ S) 𝒳(S, j) =
0image_poly"
      by auto
  qed
  ultimately have "law_restrict M = law_restrict image_poly"
    using assms(2) by simp
  hence "carrier M = carrier image_poly"
    unfolding law_restrict_def by (simp add:ring.defs)
  moreover have "𝒳(P, i) ∈ carrier image_poly"
    using eval_pmod_var(2)[OF hyps Hom.homh Q(1) degree_gt] unfolding
image_poly_def by simp
  moreover have "𝒳(P, i) ∉ carrier M"
    using indexed_var_not_index_free[of "(P, i)"] hyps(3) by blast
  ultimately show False by simp
qed

lemma (in field) exists_extension_with_roots:
  shows "∃L ∈ extensions. ∀P ∈ carrier (poly_ring R). (ring.splitted
L) (σ P)"
proof -
  obtain M where "M ∈ extensions" and "∀L ∈ extensions. M ≲ L ⟶ law_restrict
L = law_restrict M"
    using exists_maximal_extension iso_incl_hom by blast
  thus ?thesis
    using exists_root[of M] by auto
qed
```

## 38.8 Existence of Algebraic Closure

```
locale algebraic_closure = field L + subfield K L for L (structure) and
K +
  assumes algebraic_extension: "x ∈ carrier L ⟹ (algebraic over K)
x"
    and roots_over_subfield: "P ∈ carrier (K[X]) ⟹ splitted P"

locale algebraically_closed = field L for L (structure) +
  assumes roots_over_carrier: "P ∈ carrier (poly_ring L) ⟹ splitted
P"

definition (in field) alg_closure :: "((’a list × nat) multiset ⇒ ’a)
ring"
  where "alg_closure = (SOME L — such that.
          — i  algebraic_closure L (indexed_const ‘ (carrier R)) ∧
          — ii indexed_const ∈ ring_hom R L)"

lemma algebraic_hom:
  assumes "h ∈ ring_hom R S" and "field R" and "field S" and "subfield
K R" and "x ∈ carrier R"
  shows "((ring.algebraic R) over K) x ⟹ ((ring.algebraic S) over (h
‘ K)) (h x)"
proof -
  interpret Hom: ring_hom_ring R S h
    using ring_hom_ringI2[OF assms(2-3)[THEN field.is_ring] assms(1)]
.
  assume "(Hom.R.algebraic over K) x"
  then obtain p where p: "p ∈ carrier (K[X]_R)" and "p ≠ []" and eval:
"Hom.R.eval p x = 0_R"
    using domain.algebraicE[OF field.axioms(1) subfieldE(1), of R K x]
assms(2,4-5) by auto
  hence "(map h p) ∈ carrier ((h ‘ K)[X]_S)" and "(map h p) ≠ []"
    using Hom.subfield_polynomial_hom[OF assms(4) one_not_zero[OF assms(3)]]
by auto
  moreover have "Hom.S.eval (map h p) (h x) = 0_S"
    using Hom.eval_hom[OF subfieldE(1)[OF assms(4)] assms(5) p] unfold-
ing eval by simp
  ultimately show ?thesis
    using Hom.S.non_trivial_ker_imp_algebraic[of "h ‘ K" "h x"] unfold-
ing a_kernel_def’ by auto
qed

lemma (in field) exists_closure:
  obtains L :: "(((’a list × nat) multiset) ⇒ ’a) ring"
  where "algebraic_closure L (indexed_const ‘ (carrier R))" and "indexed_const
∈ ring_hom R L"
proof -
  obtain L where "L ∈ extensions"
    and roots: "⋀P. P ∈ carrier (poly_ring R) ⟹ (ring.splitted L) (σ
```

```
P)"
    using exists_extension_with_roots by auto

  let ?K = "indexed_const ` (carrier R)"
  let ?set_of_algs = "{ x ∈ carrier L. ((ring.algebraic L) over ?K) x
}"
  let ?M = "L ⦇ carrier := ?set_of_algs ⦈"

  from ‹L ∈ extensions›
  have L: "field L" and  hom: "ring_hom_ring R L indexed_const"
    using ring_hom_ringI2[OF ring_axioms field.is_ring] unfolding extensions_def
by auto
  have "subfield ?K L"
    using ring_hom_ring.img_is_subfield(2)[OF hom carrier_is_subfield
          domain.one_not_zero[OF field.axioms(1)[OF L]]] by auto
  hence set_of_algs: "subfield ?set_of_algs L"
    using field.subfield_of_algebraics[OF L, of ?K] by simp
  have M: "field ?M"
    using ring.subfield_iff(2)[OF field.is_ring[OF L] set_of_algs] by
simp

  interpret Id: ring_hom_ring ?M L id
    using ring_hom_ringI[OF field.is_ring[OF M] field.is_ring[OF L]] by
auto

  have is_subfield: "subfield ?K ?M"
  proof (intro ring.subfield_iff(1)[OF field.is_ring[OF M]])
    have "L ⦇ carrier := ?K ⦈ = ?M ⦇ carrier := ?K ⦈"
      by simp
    moreover from ‹subfield ?K L› have "field (L ⦇ carrier := ?K ⦈)"
      using ring.subfield_iff(2)[OF field.is_ring[OF L]] by simp
    ultimately show "field (?M ⦇ carrier := ?K ⦈)"
      by simp
  next
    show "?K ⊆ carrier ?M"
    proof
      fix x :: "(('a list × nat) multiset) ⇒ 'a"
      assume "x ∈ ?K"
      hence "x ∈ carrier L"
        using ring_hom_memE(1)[OF ring_hom_ring.homh[OF hom]] by auto
      moreover from ‹subfield ?K L› and ‹x ∈ ?K› have "(Id.S.algebraic
over ?K) x"
        using domain.algebraic_self[OF field.axioms(1)[OF L] subfieldE(1)]
by auto
      ultimately show "x ∈ carrier ?M"
        by auto
    qed
  qed
```

```
  have "algebraic_closure ?M ?K"
  proof (intro algebraic_closure.intro[OF M is_subfield])
    have "(Id.R.algebraic over ?K) x" if "x ∈ carrier ?M" for x
      using that Id.S.algebraic_consistent[OF subfieldE(1)[OF set_of_algs]]
by simp
    moreover have "Id.R.splitted P" if "P ∈ carrier (?K[X]?M)" for P
    proof -
      from ‹P ∈ carrier (?K[X]?M)› have "P ∈ carrier (poly_ring ?M)"
        using Id.R.carrier_polynomial_shell[OF subfieldE(1)[OF is_subfield]]
by simp
      show ?thesis
      proof (cases "degree P = 0")
        case True with ‹P ∈ carrier (poly_ring ?M)› show ?thesis
          using domain.degree_zero_imp_splitted[OF field.axioms(1)[OF
M]]
          by fastforce
      next
        case False then have "degree P > 0"
          by simp
        from ‹P ∈ carrier (?K[X]?M)› have "P ∈ carrier (?K[X]L)"
          unfolding Id.S.univ_poly_consistent[OF subfieldE(1)[OF set_of_algs]]
.
        hence "set P ⊆ ?K"
          unfolding sym[OF univ_poly_carrier] polynomial_def by auto
        hence "∃Q. set Q ⊆ carrier R ∧ P = σ Q"
        proof (induct P, simp add: σ_def)
          case (Cons p P)
          then obtain q Q where "q ∈ carrier R" "set Q ⊆ carrier R"
            and "σ Q = P" "indexed_const q = p"
            unfolding σ_def by auto
          hence "set (q # Q) ⊆ carrier R" and "σ (q # Q) = (p # P)"
            unfolding σ_def by auto
          thus ?case
            by metis
        qed
        then obtain Q where "set Q ⊆ carrier R" and "σ Q = P"
          by auto
        moreover have "lead_coeff Q ≠ 0"
        proof (rule ccontr)
          assume "¬ lead_coeff Q ≠ 0" then have "lead_coeff Q = 0"
            by simp
          with ‹σ Q = P› and ‹degree P > 0› have "lead_coeff P = indexed_const
0"
            unfolding σ_def by (metis diff_0_eq_0 length_map less_irrefl_nat
list.map_sel(1) list.size(3))
          hence "lead_coeff P = 0L"
            using ring_hom_zero[OF ring_hom_ring.homh ring_hom_ring.axioms(1-2)]
hom by auto
          with ‹degree P > 0› have "¬ P ∈ carrier (?K[X]?M)"
```

```
            unfolding sym[OF univ_poly_carrier] polynomial_def by auto
          with ‹P ∈ carrier (?K[X]?M)› show False
            by simp
        qed
        ultimately have "Q ∈ carrier (poly_ring R)"
          unfolding sym[OF univ_poly_carrier] polynomial_def by auto
        with ‹σ Q = P› have "Id.S.splitted P"
          using roots[of Q] by simp

        from ‹P ∈ carrier (poly_ring ?M)› show ?thesis
        proof (rule field.trivial_factors_imp_splitted[OF M])
          fix R
          assume R: "R ∈ carrier (poly_ring ?M)" "pirreducible?M (carrier
?M) R" and "R pdivides?M P"

          from ‹P ∈ carrier (poly_ring ?M)› and ‹R ∈ carrier (poly_ring
?M)›
          have "P ∈ carrier ((?set_of_algs)[X]L)" and "R ∈ carrier ((?set_of_algs)[X]L)"
            unfolding Id.S.univ_poly_consistent[OF subfieldE(1)[OF set_of_algs]]
by auto
          hence in_carrier: "P ∈ carrier (poly_ring L)" "R ∈ carrier
(poly_ring L)"
            using Id.S.carrier_polynomial_shell[OF subfieldE(1)[OF set_of_algs]]
by auto

          from ‹R pdivides?M P› have "R divides((?set_of_algs)[X]L) P"
            unfolding pdivides_def Id.S.univ_poly_consistent[OF subfieldE(1)[OF
set_of_algs]]
            by simp
          with ‹P ∈ carrier ((?set_of_algs)[X]L)› and ‹R ∈ carrier ((?set_of_algs)[X]L)›
          have "R pdividesL P"
            using domain.pdivides_iff_shell[OF field.axioms(1)[OF L] set_of_algs,
of R P] by simp
          with ‹Id.S.splitted P› and ‹degree P ≠ 0› have "Id.S.splitted
R"
            using field.pdivides_imp_splitted[OF L in_carrier(2,1)] by
fastforce
          show "degree R ≤ 1"
          proof (cases "Id.S.roots R = {#}")
            case True with ‹Id.S.splitted R› show ?thesis
              unfolding Id.S.splitted_def by simp
          next
            case False with ‹R ∈ carrier (poly_ring L)›
            obtain a where "a ∈ carrier L" and "a ∈# Id.S.roots R"
              and "[ 1L, ⊖L a ] ∈ carrier (poly_ring L)" and pdiv: "[
1L, ⊖L a ] pdividesL R"
              using domain.not_empty_rootsE[OF field.axioms(1)[OF L],
of R] by blast
```

```
          from ‹P ∈ carrier (?K[X]L)›
          have "(Id.S.algebraic over ?K) a"
          proof (rule Id.S.algebraicI)
            from ‹degree P ≠ 0› show "P ≠ []"
              by auto
          next
            from ‹a ∈# Id.S.roots R› and ‹R ∈ carrier (poly_ring L)›
            have "Id.S.eval R a = 0L"
              using domain.roots_mem_iff_is_root[OF field.axioms(1)[OF
L]]
              unfolding Id.S.is_root_def by auto
            with ‹R pdividesL P› and ‹a ∈ carrier L› show "Id.S.eval
P a = 0L"
              using domain.pdivides_imp_root_sharing[OF field.axioms(1)[OF
L] in_carrier(2)] by simp
          qed
          with ‹a ∈ carrier L› have "a ∈ ?set_of_algs"
            by simp
          hence "[ 1L, ⊖L a ] ∈ carrier ((?set_of_algs)[X]L)"
            using subringE(3,5)[of ?set_of_algs L] subfieldE(1,6)[OF
set_of_algs]
            unfolding sym[OF univ_poly_carrier] polynomial_def by simp
          hence "[ 1L, ⊖L a ] ∈ carrier (poly_ring ?M)"
            unfolding Id.S.univ_poly_consistent[OF subfieldE(1)[OF set_of_algs]]
by simp

          from ‹[ 1L, ⊖L a ] ∈ carrier ((?set_of_algs)[X]L)›
           and ‹R ∈ carrier ((?set_of_algs)[X]L)›
          have "[ 1L, ⊖L a ] divides(?set_of_algs)[X]L R"
            using pdiv domain.pdivides_iff_shell[OF field.axioms(1)[OF
L] set_of_algs] by simp
          hence "[ 1L, ⊖L a ] dividespoly_ring ?M R"
            unfolding pdivides_def Id.S.univ_poly_consistent[OF subfieldE(1)[OF
set_of_algs]]
            by simp

          have "[ 1L, ⊖L a ] ∉ Units (poly_ring ?M)"
            using Id.R.univ_poly_units[OF field.carrier_is_subfield[OF
M]] by force
          with ‹[ 1L, ⊖L a ] ∈ carrier (poly_ring ?M)› and ‹R ∈ carrier
(poly_ring ?M)›
           and ‹[ 1L, ⊖L a ] dividespoly_ring ?M R›
          have "[ 1L, ⊖L a ] ∼poly_ring ?M R"
            using Id.R.divides_pirreducible_condition[OF R(2)] by auto
          with ‹[ 1L, ⊖L a ] ∈ carrier (poly_ring ?M)› and ‹R ∈ carrier
(poly_ring ?M)›
          have "degree R = 1"
            using domain.associated_polynomials_imp_same_length[OF field.axioms(1)[OF
M]
```

```
                    Id.R.carrier_is_subring, of "[ 𝟭L, ⊖L a ]" R] by
force
            thus ?thesis
              by simp
          qed
        qed
      qed
    qed
    ultimately show "algebraic_closure_axioms ?M ?K"
      unfolding algebraic_closure_axioms_def by auto
  qed
  moreover have "indexed_const ∈ ring_hom R ?M"
    using ring_hom_ring.homh[OF hom] subfieldE(3)[OF is_subfield]
    unfolding subset_iff ring_hom_def by auto
  ultimately show thesis
    using that by auto
qed

lemma (in field) alg_closureE:
  shows "algebraic_closure alg_closure (indexed_const ` (carrier R))"
    and "indexed_const ∈ ring_hom R alg_closure"
  using exists_closure unfolding alg_closure_def
  by (metis (mono_tags, lifting) someI2)+

lemma (in field) algebraically_closedI':
  assumes "⋀p. ⟦ p ∈ carrier (poly_ring R); degree p > 1 ⟧ ⟹ splitted
p"
  shows "algebraically_closed R"
proof
  fix p assume "p ∈ carrier (poly_ring R)" show "splitted p"
  proof (cases "degree p ≤ 1")
    case True with ‹p ∈ carrier (poly_ring R)› show ?thesis
      using degree_zero_imp_splitted degree_one_imp_splitted by fastforce
  next
    case False with ‹p ∈ carrier (poly_ring R)› show ?thesis
      using assms by fastforce
  qed
qed

lemma (in field) algebraically_closedI:
  assumes "⋀p. ⟦ p ∈ carrier (poly_ring R); degree p > 1 ⟧ ⟹ ∃x ∈
carrier R. eval p x = 𝟬"
  shows "algebraically_closed R"
proof
  fix p assume "p ∈ carrier (poly_ring R)" thus "splitted p"
  proof (induction "degree p" arbitrary: p rule: less_induct)
    case less show ?case
    proof (cases "degree p ≤ 1")
      case True with ‹p ∈ carrier (poly_ring R)› show ?thesis
```

```
        using degree_zero_imp_splitted degree_one_imp_splitted by fastforce
    next
      case False then have "degree p > 1"
        by simp
      with ‹p ∈ carrier (poly_ring R)› have "roots p ≠ {#}"
        using assms[of p] roots_mem_iff_is_root[of p] unfolding is_root_def
by force
      then obtain a where a: "a ∈ carrier R" "a ∈# roots p"
        and pdiv: "[ 𝟭, ⊖ a ] pdivides p" and in_carrier: "[ 𝟭, ⊖ a
] ∈ carrier (poly_ring R)"
        using less(2) by blast
      then obtain q where q: "q ∈ carrier (poly_ring R)" and p: "p =
[ 𝟭, ⊖ a ] ⊗poly_ring R q"
        unfolding pdivides_def by blast
      with ‹degree p > 1› have not_zero: "q ≠ []" and "p ≠ []"
        using domain.integral_iff[OF univ_poly_is_domain[OF carrier_is_subring]
in_carrier, of q]
        by (auto simp add: univ_poly_zero[of R "carrier R"])
      hence deg: "degree p = Suc (degree q)"
        using poly_mult_degree_eq[OF carrier_is_subring] in_carrier q
p
        unfolding univ_poly_carrier sym[OF univ_poly_mult[of R "carrier
R"]] by auto
      hence "splitted q"
        using less(1)[OF _ q] by simp
      moreover have "roots p = add_mset a (roots q)"
        using poly_mult_degree_one_monic_imp_same_roots[OF a(1) q not_zero]
p by simp
      ultimately show ?thesis
        unfolding splitted_def deg by simp
    qed
  qed
qed

sublocale algebraic_closure ⊆ algebraically_closed
proof (rule algebraically_closedI')
  fix P assume in_carrier: "P ∈ carrier (poly_ring L)" and gt_one: "degree
P > 1"
  then have gt_zero: "degree P > 0"
    by simp

  define A where "A = finite_extension K P"

  from ‹P ∈ carrier (poly_ring L)› have "set P ⊆ carrier L"
    by (simp add: polynomial_incl univ_poly_carrier)
  hence A: "subfield A L" and P: "P ∈ carrier (A[X])"
    using finite_extension_mem[OF subfieldE(1)[OF subfield_axioms], of
P] in_carrier
          algebraic_extension finite_extension_is_subfield[OF subfield_axioms,
```

```
of P]
    unfolding sym[OF A_def] sym[OF univ_poly_carrier] polynomial_def by
auto
  from ‹set P ⊆ carrier L› have incl: "K ⊆ A"
    using finite_extension_incl[OF subfieldE(3)[OF subfield_axioms]] un-
folding A_def by simp

  interpret UP_K: domain "K[X]"
    using univ_poly_is_domain[OF subfieldE(1)[OF subfield_axioms]] .
  interpret UP_A: domain "A[X]"
    using univ_poly_is_domain[OF subfieldE(1)[OF A]] .
  interpret Rupt: ring "Rupt A P"
    unfolding rupture_def using ideal.quotient_is_ring[OF UP_A.cgenideal_ideal[OF
P]] .
  interpret Hom: ring_hom_ring "L ⦇ carrier := A ⦈" "Rupt A P" "rupture_surj
A P ∘ poly_of_const"
    using ring_hom_ringI2[OF subring_is_ring[OF subfieldE(1)] Rupt.ring_axioms
          rupture_surj_norm_is_hom[OF subfieldE(1) P]] A by simp
  let ?h = "rupture_surj A P ∘ poly_of_const"

  have h_simp: "rupture_surj A P ‘ poly_of_const ‘ E = ?h ‘ E" for E
    by auto
  hence aux_lemmas:
    "subfield (rupture_surj A P ‘ poly_of_const ‘ K) (Rupt A P)"
    "subfield (rupture_surj A P ‘ poly_of_const ‘ A) (Rupt A P)"
    using Hom.img_is_subfield(2)[OF _ rupture_one_not_zero[OF A P gt_zero]]
          ring.subfield_iff(1)[OF subring_is_ring[OF subfieldE(1)[OF A]]]
          subfield_iff(2)[OF subfield_axioms] subfield_iff(2)[OF A] incl
    by auto

  have "carrier (K[X]) ⊆ carrier (A[X])"
    using subsetI[of "carrier (K[X])" "carrier (A[X])"] incl
    unfolding sym[OF univ_poly_carrier] polynomial_def by auto
  hence "id ∈ ring_hom (K[X]) (A[X])"
    unfolding ring_hom_def unfolding univ_poly_mult univ_poly_add univ_poly_one
by (simp add: subsetD)
  hence "rupture_surj A P ∈ ring_hom (K[X]) (Rupt A P)"
    using ring_hom_trans[OF _ rupture_surj_hom(1)[OF subfieldE(1)[OF A]
P], of id] by simp
  then interpret Hom’: ring_hom_ring "K[X]" "Rupt A P" "rupture_surj A
P"
    using ring_hom_ringI2[OF UP_K.ring_axioms Rupt.ring_axioms] by simp

  from ‹id ∈ ring_hom (K[X]) (A[X])› have Id: "ring_hom_ring (K[X])
(A[X]) id"
    using ring_hom_ringI2[OF UP_K.ring_axioms UP_A.ring_axioms] by simp
  hence "subalgebra (poly_of_const ‘ K) (carrier (K[X])) (A[X])"
    using ring_hom_ring.img_is_subalgebra[OF Id _ UP_K.carrier_is_subalgebra[OF
subfieldE(3)]]
```

```
          univ_poly_subfield_of_consts[OF subfield_axioms] by auto

  moreover from ‹carrier (K[X]) ⊆ carrier (A[X])› have "poly_of_const
` K ⊆ carrier (A[X])"
    using subfieldE(3)[OF univ_poly_subfield_of_consts[OF subfield_axioms]]
by simp

  ultimately
  have "subalgebra (rupture_surj A P ` poly_of_const ` K) (rupture_surj
A P ` carrier (K[X])) (Rupt A P)"
    using ring_hom_ring.img_is_subalgebra[OF rupture_surj_hom(2)[OF subfieldE(1)[OF
A] P]] by simp

  moreover have "Rupt.finite_dimension (rupture_surj A P ` poly_of_const
` K) (carrier (Rupt A P))"
  proof (intro Rupt.telescopic_base_dim(1)[where
            ?K = "rupture_surj A P ` poly_of_const ` K" and
            ?F = "rupture_surj A P ` poly_of_const ` A" and
            ?E = "carrier (Rupt A P)", OF aux_lemmas])
    show "Rupt.finite_dimension (rupture_surj A P ` poly_of_const ` A)
(carrier (Rupt A P))"
      using Rupt.finite_dimensionI[OF rupture_dimension[OF A P gt_zero]]
.
  next
    let ?h = "rupture_surj A P ∘ poly_of_const"

    from ‹set P ⊆ carrier L› have "finite_dimension K A"
      using finite_extension_finite_dimension(1)[OF subfield_axioms, of
P] algebraic_extension
      unfolding A_def by auto
    then obtain Us where Us: "set Us ⊆ carrier L" "A = Span K Us"
      using exists_base subfield_axioms by blast
    hence "?h ` A = Rupt.Span (?h ` K) (map ?h Us)"
      using Hom.Span_hom[of K Us] incl Span_base_incl[OF subfield_axioms,
of Us]
      unfolding Span_consistent[OF subfieldE(1)[OF A]] by simp
    moreover have "set (map ?h Us) ⊆ carrier (Rupt A P)"
      using Span_base_incl[OF subfield_axioms Us(1)] ring_hom_memE(1)[OF
Hom.homh]
      unfolding sym[OF Us(2)] by auto
    ultimately
    show "Rupt.finite_dimension (rupture_surj A P ` poly_of_const ` K)
(rupture_surj A P ` poly_of_const ` A)"
      using Rupt.Span_finite_dimension[OF aux_lemmas(1)] unfolding h_simp
by simp
  qed

  moreover have "rupture_surj A P ` carrier (A[X]) = carrier (Rupt A
P)"
```

```
    unfolding rupture_def FactRing_def A_RCOSETS_def' by auto
  with ‹carrier (K[X]) ⊆ carrier (A[X])› have "rupture_surj A P ‘ carrier
(K[X]) ⊆ carrier (Rupt A P)"
    by auto

  ultimately
  have "Rupt.finite_dimension (rupture_surj A P ‘ poly_of_const ‘ K) (rupture_surj
A P ‘ carrier (K[X]))"
    using Rupt.subalbegra_incl_imp_finite_dimension[OF aux_lemmas(1)]
by simp

  hence "¬ inj_on (rupture_surj A P) (carrier (K[X]))"
    using Hom'.infinite_dimension_hom[OF _ rupture_one_not_zero[OF A P
gt_zero] _
          UP_K.carrier_is_subalgebra[OF subfieldE(3)] univ_poly_infinite_dimension[OF
subfield_axioms]]
          univ_poly_subfield_of_consts[OF subfield_axioms]
    by auto
  then obtain Q where Q: "Q ∈ carrier (K[X])" "Q ≠ []" and "rupture_surj
A P Q = 0Rupt A P"
    using Hom'.trivial_ker_imp_inj Hom'.hom_zero unfolding a_kernel_def'
univ_poly_zero by blast
  with ‹carrier (K[X]) ⊆ carrier (A[X])› have "Q ∈ PIdlA[X] P"
    using ideal.rcos_const_imp_mem[OF UP_A.cgenideal_ideal[OF P]]
    unfolding rupture_def FactRing_def by auto
  then obtain R where "R ∈ carrier (A[X])" and "Q = R ⊗A[X] P"
    unfolding cgenideal_def by blast
  with ‹P ∈ carrier (A[X])› have "P pdivides Q"
    using dividesI[of _ "A[X]"] UP_A.m_comm pdivides_iff_shell[OF A] by
simp
  thus "splitted P"
    using pdivides_imp_splitted[OF in_carrier
          carrier_polynomial_shell[OF subfieldE(1)[OF subfield_axioms]
Q(1)] Q(2)
          roots_over_subfield[OF Q(1)]] Q
    by simp
qed

end


theory Algebraic_Closure_Type
imports
  Algebraic_Closure
  "HOL-Computational_Algebra.Computational_Algebra"
  "HOL-Computational_Algebra.Field_as_Ring"
begin
```

```
definition (in ring_1) ring_of_type_algebra :: "'a ring"
  where "ring_of_type_algebra = ⦇
    carrier = UNIV, monoid.mult = (λx y. x * y),
    one = 1,
    ring.zero = 0,
    add = (λ x y. x + y) ⦈"

lemma (in comm_ring_1) ring_from_type_algebra [intro]:
  "ring (ring_of_type_algebra :: 'a ring)"
proof -
  have "∃y. x + y = 0" for x :: 'a
    using add.right_inverse by blast
  thus ?thesis
    unfolding ring_of_type_algebra_def using add.right_inverse
    by unfold_locales (auto simp:algebra_simps Units_def)
qed

lemma (in comm_ring_1) cring_from_type_algebra [intro]:
  "cring (ring_of_type_algebra :: 'a ring)"
proof -
  have "∃y. x + y = 0" for x :: 'a
    using add.right_inverse by blast
  thus ?thesis
    unfolding ring_of_type_algebra_def using add.right_inverse
    by unfold_locales (auto simp:algebra_simps Units_def)
qed

lemma (in Fields.field) field_from_type_algebra [intro]:
  "field (ring_of_type_algebra :: 'a ring)"
proof -
  have "∃y. x + y = 0" for x :: 'a
    using add.right_inverse by blast

  moreover have "x ≠ 0 ⟹ ∃y. x * y = 1" for x :: 'a
    by (rule exI[of _ "inverse x"]) auto

  ultimately show ?thesis
    unfolding ring_of_type_algebra_def using add.right_inverse
    by unfold_locales (auto simp:algebra_simps Units_def)
qed
```

## 38.9 Definition

```
typedef (overloaded) 'a :: field alg_closure =
  "carrier (field.alg_closure (ring_of_type_algebra :: 'a :: field ring))"
proof -
  define K where "K ≡ (ring_of_type_algebra :: 'a ring)"
  define L where "L ≡ field.alg_closure K"
```

```
  interpret K: field K
    unfolding K_def by rule

  interpret algebraic_closure L "range K.indexed_const"
  proof -
    have *: "carrier K = UNIV"
      by (auto simp: K_def ring_of_type_algebra_def)
    show "algebraic_closure L (range K.indexed_const)"
      unfolding * [symmetric] L_def by (rule K.alg_closureE)
  qed

  show "∃x. x ∈ carrier L"
    using zero_closed by blast
qed

setup_lifting type_definition_alg_closure

instantiation alg_closure :: (field) field
begin

context
  fixes L K
  defines "K ≡ (ring_of_type_algebra :: 'a :: field ring)"
  defines "L ≡ field.alg_closure K"
begin

interpretation K: field K
  unfolding K_def by rule

interpretation algebraic_closure L "range K.indexed_const"
proof -
  have *: "carrier K = UNIV"
    by (auto simp: K_def ring_of_type_algebra_def)
  show "algebraic_closure L (range K.indexed_const)"
    unfolding * [symmetric] L_def by (rule K.alg_closureE)
qed

lift_definition zero_alg_closure :: "'a alg_closure" is "ring.zero L"
  by (fold K_def, fold L_def) (rule ring_simprules)

lift_definition one_alg_closure :: "'a alg_closure" is "monoid.one L"
  by (fold K_def, fold L_def) (rule ring_simprules)

lift_definition plus_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒
'a alg_closure"
  is "ring.add L"
  by (fold K_def, fold L_def) (rule ring_simprules)

lift_definition minus_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒
```

```
'a alg_closure"
  is "a_minus L"
  by (fold K_def, fold L_def) (rule ring_simprules)

lift_definition times_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒
'a alg_closure"
  is "monoid.mult L"
  by (fold K_def, fold L_def) (rule ring_simprules)

lift_definition uminus_alg_closure :: "'a alg_closure ⇒ 'a alg_closure"
  is "a_inv L"
  by (fold K_def, fold L_def) (rule ring_simprules)

lift_definition inverse_alg_closure :: "'a alg_closure ⇒ 'a alg_closure"
  is "λx. if x = ring.zero L then ring.zero L else m_inv L x"
  by (fold K_def, fold L_def) (auto simp: field_Units)

lift_definition divide_alg_closure :: "'a alg_closure ⇒ 'a alg_closure
⇒ 'a alg_closure"
  is "λx y. if y = ring.zero L then ring.zero L else monoid.mult L x (m_inv
L y)"
  by (fold K_def, fold L_def) (auto simp: field_Units)

end

instance proof -
  define K where "K ≡ (ring_of_type_algebra :: 'a ring)"
  define L where "L ≡ field.alg_closure K"

  interpret K: field K
    unfolding K_def by rule

  interpret algebraic_closure L "range K.indexed_const"
  proof -
    have *: "carrier K = UNIV"
      by (auto simp: K_def ring_of_type_algebra_def)
    show "algebraic_closure L (range K.indexed_const)"
      unfolding * [symmetric] L_def by (rule K.alg_closureE)
  qed

  show "OFCLASS('a alg_closure, field_class)"
  proof (standard, goal_cases)
    case 1
    show ?case
      by (transfer, fold K_def, fold L_def) (rule m_assoc)
  next
    case 2
    show ?case
      by (transfer, fold K_def, fold L_def) (rule m_comm)
```

```
  next
    case 3
    show ?case
      by (transfer, fold K_def, fold L_def) (rule l_one)
  next
    case 4
    show ?case
      by (transfer, fold K_def, fold L_def) (rule a_assoc)
  next
    case 5
    show ?case
      by (transfer, fold K_def, fold L_def) (rule a_comm)
  next
    case 6
    show ?case
      by (transfer, fold K_def, fold L_def) (rule l_zero)
  next
    case 7
    show ?case
      by (transfer, fold K_def, fold L_def) (rule ring_simprules)
  next
    case 8
    show ?case
      by (transfer, fold K_def, fold L_def) (rule ring_simprules)
  next
    case 9
    show ?case
      by (transfer, fold K_def, fold L_def) (rule ring_simprules)
  next
    case 10
    show ?case
      by (transfer, fold K_def, fold L_def) (rule zero_not_one)
  next
    case 11
    thus ?case
      by (transfer, fold K_def, fold L_def) (auto simp: field_Units)
  next
    case 12
    thus ?case
      by (transfer, fold K_def, fold L_def) auto
  next
    case 13
    thus ?case
      by transfer auto
  qed
qed

end
```

## 38.10 The algebraic closure is algebraically closed

```
instance alg_closure :: (field) alg_closed_field
proof
  define K where "K ≡ (ring_of_type_algebra :: 'a ring)"
  define L where "L ≡ field.alg_closure K"

  interpret K: field K
    unfolding K_def by rule

  interpret algebraic_closure L "range K.indexed_const"
  proof -
    have *: "carrier K = UNIV"
      by (auto simp: K_def ring_of_type_algebra_def)
    show "algebraic_closure L (range K.indexed_const)"
      unfolding * [symmetric] L_def by (rule K.alg_closureE)
  qed

  have [simp]: "Rep_alg_closure x ∈ carrier L" for x
    using Rep_alg_closure[of x] by (simp only: L_def K_def)

  have [simp]: "Rep_alg_closure x = Rep_alg_closure y ⟷ x = y" for x
y
    by (simp add: Rep_alg_closure_inject)
  have [simp]: "Rep_alg_closure x = 𝟬L ⟷ x = 0" for x
  proof -
    have "Rep_alg_closure x = Rep_alg_closure 0 ⟷ x = 0"
      by simp
    also have "Rep_alg_closure 0 = 𝟬L"
      by (simp add: zero_alg_closure.rep_eq L_def K_def)
    finally show ?thesis .
  qed

  have [simp]: "Rep_alg_closure (x ^ n) = Rep_alg_closure x [^]L n"
    for x :: "'a alg_closure" and n
    by (induction n)
       (auto simp: one_alg_closure.rep_eq times_alg_closure.rep_eq m_comm
             simp flip: L_def K_def)
  have [simp]: "Rep_alg_closure (Abs_alg_closure x) = x" if "x ∈ carrier
L" for x
    using that unfolding L_def K_def by (rule Abs_alg_closure_inverse)

  show "∃x. poly p x = 0" if p: "Polynomial.lead_coeff p = 1" "Polynomial.degree
p > 0"
    for p :: "'a alg_closure poly"
  proof -
    define P where "P = rev (map Rep_alg_closure (Polynomial.coeffs p))"
    have deg: "Polynomials.degree P = Polynomial.degree p"
      by (auto simp: P_def degree_eq_length_coeffs)
    have carrier_P: "P ∈ carrier (poly_ring L)"
```

```
      by (auto simp: univ_poly_def polynomial_def P_def hd_map hd_rev
last_map
                    last_coeffs_eq_coeff_degree)
    hence "splitted P"
      using roots_over_carrier by blast
    hence "roots P ≠ {#}"
      unfolding splitted_def using deg p by auto
    then obtain x where "x ∈# roots P"
      by blast
    hence x: "is_root P x"
      using roots_mem_iff_is_root[OF carrier_P] by auto
    hence [simp]: "x ∈ carrier L"
      by (auto simp: is_root_def)
    define x' where "x' = Abs_alg_closure x"
    define xs where "xs = rev (coeffs p)"

    have "cr_alg_closure (eval (map Rep_alg_closure xs) x) (poly (Poly
(rev xs)) x')"
      by (induction xs)
         (auto simp flip: K_def L_def simp: cr_alg_closure_def
                zero_alg_closure.rep_eq plus_alg_closure.rep_eq
                times_alg_closure.rep_eq Poly_append poly_monom
                a_comm m_comm x'_def)
    also have "map Rep_alg_closure xs = P"
      by (simp add: xs_def P_def rev_map)
    also have "Poly (rev xs) = p"
      by (simp add: xs_def)
    finally have "poly p x' = 0"
      using x by (auto simp: is_root_def cr_alg_closure_def)
    thus "∃x. poly p x = 0" ..
  qed
qed
```

## 38.11 Converting between the base field and the closure

```
context
  fixes L K
  defines "K ≡ (ring_of_type_algebra :: 'a :: field ring)"
  defines "L ≡ field.alg_closure K"
begin

interpretation K: field K
  unfolding K_def by rule

interpretation algebraic_closure L "range K.indexed_const"
proof -
  have *: "carrier K = UNIV"
    by (auto simp: K_def ring_of_type_algebra_def)
  show "algebraic_closure L (range K.indexed_const)"
```

```
    unfolding * [symmetric] L_def by (rule K.alg_closureE)
qed

lemma alg_closure_hom: "K.indexed_const ∈ Ring.ring_hom K L"
  unfolding L_def using K.alg_closureE(2) .

lift_definition to_ac :: "'a :: field ⇒ 'a alg_closure"
  is "ring.indexed_const K"
  by (fold K_def, fold L_def) (use mem_carrier in blast)

lemma to_ac_0 [simp]: "to_ac (0 :: 'a) = 0"
proof -
  have "to_ac (𝟬K) = 0"
  proof (transfer fixing: K, fold K_def, fold L_def)
    show "K.indexed_const 𝟬K = 𝟬L"
      using Ring.ring_hom_zero[OF alg_closure_hom] K.ring_axioms is_ring
      by simp
  qed
  thus ?thesis
    by (simp add: K_def ring_of_type_algebra_def)
qed

lemma to_ac_1 [simp]: "to_ac (1 :: 'a) = 1"
proof -
  have "to_ac (𝟭K) = 1"
  proof (transfer fixing: K, fold K_def, fold L_def)
    show "K.indexed_const 𝟭K = 𝟭L"
      using Ring.ring_hom_one[OF alg_closure_hom] K.ring_axioms is_ring
      by simp
  qed
  thus ?thesis
    by (simp add: K_def ring_of_type_algebra_def)
qed

lemma to_ac_add [simp]: "to_ac (x + y :: 'a) = to_ac x + to_ac y"
proof -
  have "to_ac (x ⊕K y) = to_ac x + to_ac y"
  proof (transfer fixing: K x y, fold K_def, fold L_def)
    show "K.indexed_const (x ⊕K y) = K.indexed_const x ⊕L K.indexed_const
y"
      using Ring.ring_hom_add[OF alg_closure_hom, of x y] K.ring_axioms
is_ring
      by (simp add: K_def ring_of_type_algebra_def)
  qed
  thus ?thesis
    by (simp add: K_def ring_of_type_algebra_def)
qed

lemma to_ac_minus [simp]: "to_ac (-x :: 'a) = -to_ac x"
```

```
  using to_ac_add to_ac_0 add_eq_0_iff by metis

lemma to_ac_diff [simp]: "to_ac (x - y :: 'a) = to_ac x - to_ac y"
  using to_ac_add[of x "-y"] by simp

lemma to_ac_mult [simp]: "to_ac (x * y :: 'a) = to_ac x * to_ac y"
proof -
  have "to_ac (x ⊗K y) = to_ac x * to_ac y"
  proof (transfer fixing: K x y, fold K_def, fold L_def)
    show "K.indexed_const (x ⊗K y) = K.indexed_const x ⊗L K.indexed_const
y"
      using Ring.ring_hom_mult[OF alg_closure_hom, of x y] K.ring_axioms
is_ring
      by (simp add: K_def ring_of_type_algebra_def)
  qed
  thus ?thesis
    by (simp add: K_def ring_of_type_algebra_def)
qed

lemma to_ac_inverse [simp]: "to_ac (inverse x :: 'a) = inverse (to_ac
x)"
  using to_ac_mult[of x "inverse x"] to_ac_1 to_ac_0
  by (metis divide_self_if field_class.field_divide_inverse field_class.field_inverse_zero
inverse_unique)

lemma to_ac_divide [simp]: "to_ac (x / y :: 'a) = to_ac x / to_ac y"
  using to_ac_mult[of x "inverse y"] to_ac_inverse[of y]
  by (simp add: field_class.field_divide_inverse)

lemma to_ac_power [simp]: "to_ac (x ^ n) = to_ac x ^ n"
  by (induction n) auto

lemma to_ac_of_nat [simp]: "to_ac (of_nat n) = of_nat n"
  by (induction n) auto

lemma to_ac_of_int [simp]: "to_ac (of_int n) = of_int n"
  by (induction n) auto

lemma to_ac_numeral [simp]: "to_ac (numeral n) = numeral n"
  using to_ac_of_nat[of "numeral n"] by (simp del: to_ac_of_nat)

lemma to_ac_sum: "to_ac (∑x∈A. f x) = (∑x∈A. to_ac (f x))"
  by (induction A rule: infinite_finite_induct) auto

lemma to_ac_prod: "to_ac (∏x∈A. f x) = (∏x∈A. to_ac (f x))"
  by (induction A rule: infinite_finite_induct) auto

lemma to_ac_sum_list: "to_ac (sum_list xs) = (∑x←xs. to_ac x)"
  by (induction xs) auto
```

```
lemma to_ac_prod_list: "to_ac (prod_list xs) = (∏x←xs. to_ac x)"
  by (induction xs) auto

lemma to_ac_sum_mset: "to_ac (sum_mset xs) = (∑x∈#xs. to_ac x)"
  by (induction xs) auto

lemma to_ac_prod_mset: "to_ac (prod_mset xs) = (∏x∈#xs. to_ac x)"
  by (induction xs) auto

end

lemma (in ring) indexed_const_eq_iff [simp]:
  "indexed_const x = (indexed_const y :: 'c multiset ⇒ 'a) ⟷ x = y"
proof
  assume "indexed_const x = (indexed_const y :: 'c multiset ⇒ 'a)"
  hence "indexed_const x ({#} :: 'c multiset) = indexed_const y ({#} ::
'c multiset)"
    by metis
  thus "x = y"
    by (simp add: indexed_const_def)
qed auto

lemma inj_to_ac: "inj to_ac"
  by (transfer, intro injI, subst (asm) ring.indexed_const_eq_iff) auto

lemma to_ac_eq_iff [simp]: "to_ac x = to_ac y ⟷ x = y"
  using inj_to_ac by (auto simp: inj_on_def)

lemma to_ac_eq_0_iff [simp]: "to_ac x = 0 ⟷ x = 0"
  and to_ac_eq_0_iff' [simp]: "0 = to_ac x ⟷ x = 0"
  and to_ac_eq_1_iff [simp]: "to_ac x = 1 ⟷ x = 1"
  and to_ac_eq_1_iff' [simp]: "1 = to_ac x ⟷ x = 1"
  using to_ac_eq_iff to_ac_0 to_ac_1 by metis+


definition of_ac :: "'a :: field alg_closure ⇒ 'a" where
  "of_ac x = (if x ∈ range to_ac then inv_into UNIV to_ac x else 0)"

lemma of_ac_eqI: "to_ac x = y ⟹ of_ac y = x"
  unfolding of_ac_def by (meson inj_to_ac inv_f_f range_eqI)

lemma of_ac_0 [simp]: "of_ac 0 = 0"
  and of_ac_1 [simp]: "of_ac 1 = 1"
  by (rule of_ac_eqI; simp; fail)+

lemma of_ac_to_ac [simp]: "of_ac (to_ac x) = x"
  by (rule of_ac_eqI) auto
```

```
lemma to_ac_of_ac: "x ∈ range to_ac ⟹ to_ac (of_ac x) = x"
  by auto


lemma CHAR_alg_closure [simp]:
  "CHAR('a :: field alg_closure) = CHAR('a)"
proof (rule CHAR_eqI)
  show "of_nat CHAR('a) = (0 :: 'a alg_closure)"
    by (metis of_nat_CHAR to_ac_0 to_ac_of_nat)
next
  show "CHAR('a) dvd n" if "of_nat n = (0 :: 'a alg_closure)" for n
    using that by (metis of_nat_eq_0_iff_char_dvd to_ac_eq_0_iff' to_ac_of_nat)
qed

instance alg_closure :: (field_char_0) field_char_0
proof
  show "inj (of_nat :: nat ⇒ 'a alg_closure)"
    by (metis injD inj_of_nat inj_on_def inj_to_ac to_ac_of_nat)
qed


bundle alg_closure_syntax
begin
notation to_ac (‹(‹open_block notation=‹postfix ↑›› _↑)› [1000] 999)
notation of_ac (‹(‹open_block notation=‹postfix ↓›› _↓)› [1000] 999)
end


bundle alg_closure_syntax'
begin
notation (output) to_ac (‹_›)
notation (output) of_ac (‹_›)
end
```

## 38.12 The algebraic closure is an algebraic extension

The algebraic closure is an algebraic extension, i.e. every element in it is a root of some non-zero polynomial in the base field.

```
theorem alg_closure_algebraic:
  fixes x :: "'a :: field alg_closure"
  obtains p :: "'a poly" where "p ≠ 0" "poly (map_poly to_ac p) x = 0"
proof -
  define K where "K ≡ (ring_of_type_algebra :: 'a ring)"
  define L where "L ≡ field.alg_closure K"

  interpret K: field K
    unfolding K_def by rule

  interpret algebraic_closure L "range K.indexed_const"
```

```
  proof -
    have *: "carrier K = UNIV"
      by (auto simp: K_def ring_of_type_algebra_def)
    show "algebraic_closure L (range K.indexed_const)"
      unfolding * [symmetric] L_def by (rule K.alg_closureE)
  qed

  let ?K = "range K.indexed_const"
  have sr: "subring ?K L"
    by (rule subring_axioms)
  define x' where "x' = Rep_alg_closure x"
  have "x' ∈ carrier L"
    unfolding x'_def L_def K_def by (rule Rep_alg_closure)
  hence alg: "(algebraic over range K.indexed_const) x'"
    using algebraic_extension by blast
  then obtain p where p: "p ∈ carrier (?K[X]_L)" "p ≠ []" "eval p x'
= 0_L"
    using algebraicE[OF sr ‹x' ∈ carrier L› alg] by blast

  have [simp]: "Rep_alg_closure x ∈ carrier L" for x
    using Rep_alg_closure[of x] by (simp only: L_def K_def)
  have [simp]: "Abs_alg_closure x = 0 ⟷ x = 0_L" if "x ∈ carrier L"
for x
    using that unfolding L_def K_def
    by (metis Abs_alg_closure_inverse zero_alg_closure.rep_eq zero_alg_closure_def)
  have [simp]: "Rep_alg_closure (x ^ n) = Rep_alg_closure x [^]_L n"
    for x :: "'a alg_closure" and n
    by (induction n)
       (auto simp: one_alg_closure.rep_eq times_alg_closure.rep_eq m_comm
             simp flip: L_def K_def)
  have [simp]: "Rep_alg_closure (Abs_alg_closure x) = x" if "x ∈ carrier
L" for x
    using that unfolding L_def K_def by (rule Abs_alg_closure_inverse)
  have [simp]: "Rep_alg_closure x = 0_L ⟷ x = 0" for x
    by (metis K_def L_def Rep_alg_closure_inverse zero_alg_closure.rep_eq)

  define p' where "p' = Poly (map Abs_alg_closure (rev p))"
  have "p' ≠ 0"
  proof
    assume "p' = 0"
    then obtain n where n: "map Abs_alg_closure (rev p) = replicate n
0"
      by (auto simp: p'_def Poly_eq_0)
    with ‹p ≠ []› have "n > 0"
      by (auto intro!: Nat.gr0I)
    have "last (map Abs_alg_closure (rev p)) = 0"
      using ‹n > 0› by (subst n) auto
    moreover have "Polynomials.lead_coeff p ≠ 0_L" "Polynomials.lead_coeff
p ∈ carrier L"
```

```
      using p ‹p ≠ []› local.subset
      by (fastforce simp: polynomial_def univ_poly_def)+
    ultimately show False
      using ‹p ≠ []› by (auto simp: last_map last_rev)
  qed

  have "set p ⊆ carrier L"
    using local.subset p by (auto simp: univ_poly_def polynomial_def)
  hence "cr_alg_closure (eval p x') (poly p' x)"
    unfolding p'_def
    by (induction p)
       (auto simp flip: K_def L_def simp: cr_alg_closure_def
               zero_alg_closure.rep_eq plus_alg_closure.rep_eq
               times_alg_closure.rep_eq Poly_append poly_monom
               a_comm m_comm x'_def)
  hence "poly p' x = 0"
    using p by (auto simp: cr_alg_closure_def x'_def)

  have coeff_p': "Polynomial.coeff p' i ∈ range to_ac" for i
  proof (cases "i ≥ length p")
    case False
    have "Polynomial.coeff p' i = Abs_alg_closure (rev p ! i)"
      unfolding p'_def using False
      by (auto simp: nth_default_def)
    moreover have "rev p ! i ∈ ?K"
      using p(1) False by (auto simp: univ_poly_def polynomial_def rev_nth)
    ultimately show ?thesis
      unfolding to_ac.abs_eq K_def by fastforce
  qed (auto simp: p'_def nth_default_def)


  define p'' where "p'' = map_poly of_ac p'"
  have p'_eq: "p' = map_poly to_ac p''"
    by (rule poly_eqI) (auto simp: coeff_map_poly p''_def to_ac_of_ac[OF
coeff_p'])

  show ?thesis
  proof (rule that)
    show "p'' ≠ 0"
      using ‹p' ≠ 0› by (auto simp: p'_eq)
  next
    show "poly (map_poly to_ac p'') x = 0"
      using ‹poly p' x = 0› by (simp add: p'_eq)
  qed
qed


instantiation alg_closure :: (field)
  "{unique_euclidean_ring, normalization_euclidean_semiring, normalization_semidom_multiplic
```

**begin**

**definition** [simp]: "normalize_alg_closure = (normalize_field :: 'a alg_closure ⇒ _)"
**definition** [simp]: "unit_factor_alg_closure = (unit_factor_field :: 'a alg_closure ⇒ _)"
**definition** [simp]: "modulo_alg_closure = (mod_field :: 'a alg_closure ⇒ _)"
**definition** [simp]: "euclidean_size_alg_closure = (euclidean_size_field :: 'a alg_closure ⇒ _)"
**definition** [simp]: "division_segment (x :: 'a alg_closure) = 1"

**instance**
  **by** standard
    (simp_all add: dvd_field_iff field_split_simps split: if_splits)

**end**

**instantiation** alg_closure :: (field) euclidean_ring_gcd
**begin**

**definition** gcd_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure" **where**
  "gcd_alg_closure = Euclidean_Algorithm.gcd"
**definition** lcm_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure" **where**
  "lcm_alg_closure = Euclidean_Algorithm.lcm"
**definition** Gcd_alg_closure :: "'a alg_closure set ⇒ 'a alg_closure" **where**
 "Gcd_alg_closure = Euclidean_Algorithm.Gcd"
**definition** Lcm_alg_closure :: "'a alg_closure set ⇒ 'a alg_closure" **where**
 "Lcm_alg_closure = Euclidean_Algorithm.Lcm"

**instance by** standard (simp_all add: gcd_alg_closure_def lcm_alg_closure_def Gcd_alg_closure_def Lcm_alg_closure_def)

**end**

**instance** alg_closure :: (field) semiring_gcd_mult_normalize
  ..

**end**

**theory** Ideal_Product
  **imports** Ideal
**begin**

# 39 Product of Ideals

In this section, we study the structure of the set of ideals of a given ring.

```
inductive_set
  ideal_prod :: "[ ('a, 'b) ring_scheme, 'a set, 'a set ] ⇒ 'a set" (infixl
‹·ı› 80)
  for R and I and J  where
    prod: "⟦ i ∈ I; j ∈ J ⟧ ⟹ i ⊗R j ∈ ideal_prod R I J"
  |  sum: "⟦ s1 ∈ ideal_prod R I J; s2 ∈ ideal_prod R I J ⟧ ⟹ s1 ⊕R
s2 ∈ ideal_prod R I J"

definition ideals_set :: "('a, 'b) ring_scheme ⇒ ('a set) ring"
  where "ideals_set R = ⦇ carrier = { I. ideal I R },
                           mult = ideal_prod R,
                            one = carrier R,
                           zero = { 𝟬R },
                            add = set_add R ⦈"
```

## 39.1 Basic Properties

```
lemma (in ring) ideal_prod_in_carrier:
  assumes "ideal I R" "ideal J R"
  shows "I · J ⊆ carrier R"
proof
  fix s assume "s ∈ I · J" thus "s ∈ carrier R"
    by (induct s rule: ideal_prod.induct) (auto, meson assms ideal.I_l_closed
ideal.Icarr)
qed

lemma (in ring) ideal_prod_inter:
  assumes "ideal I R" "ideal J R"
  shows "I · J ⊆ I ∩ J"
proof
  fix s assume "s ∈ I · J" thus "s ∈ I ∩ J"
    apply (induct s rule: ideal_prod.induct)
    apply (auto, (meson assms ideal.I_r_closed ideal.I_l_closed ideal.Icarr)+)
    apply (simp_all add: additive_subgroup.a_closed assms ideal.axioms(1))
    done
qed

lemma (in ring) ideal_prod_is_ideal:
  assumes "ideal I R" "ideal J R"
  shows "ideal (I · J) R"
proof (rule idealI)
  show "ring R" using ring_axioms .
next
  show "subgroup (I · J) (add_monoid R)"
    unfolding subgroup_def
  proof (auto)
```

```
    show "𝟬 ∈ I · J" using ideal_prod.prod[of 𝟬 I 𝟬 J R]
      by (simp add: additive_subgroup.zero_closed assms ideal.axioms(1))
  next
    fix s1 s2 assume s1: "s1 ∈ I · J" and s2: "s2 ∈ I · J"
    have IJcarr: "⋀a. a ∈ I · J ⟹ a ∈ carrier R"
      by (meson assms subsetD ideal_prod_in_carrier)
    show "s1 ∈ carrier R" using ideal_prod_in_carrier[OF assms] s1 by
blast
    show "s1 ⊕ s2 ∈ I · J" by (simp add: ideal_prod.sum[OF s1 s2])
    show "inv_add_monoid R s1 ∈ I · J" using s1
    proof (induct s1 rule: ideal_prod.induct)
      case (prod i j)
      hence "inv_add_monoid R (i ⊗ j) = (inv_add_monoid R i) ⊗ j"
        by (metis a_inv_def assms(1) assms(2) ideal.Icarr l_minus)
      thus ?case using ideal_prod.prod[of "inv_add_monoid R i" I j J R]
assms
        by (simp add: additive_subgroup.a_subgroup ideal.axioms(1) prod.hyps
subgroup.m_inv_closed)
    next
      case (sum s1 s2) thus ?case
        by (metis (no_types) IJcarr a_inv_def add.inv_mult_group ideal_prod.sum
sum.hyps)
    qed
  qed
next
  fix s x assume s: "s ∈ I · J" and x: "x ∈ carrier R"
  show "x ⊗ s ∈ I · J" using s
  proof (induct s rule: ideal_prod.induct)
    case (prod i j) thus ?case using ideal_prod.prod[of "x ⊗ i" I j J
R] assms
      by (simp add: x ideal.I_l_closed ideal.Icarr m_assoc)
  next
    case (sum s1 s2) thus ?case
    proof -
      have IJ: "I · J ⊆ carrier R"
        by (metis (no_types) assms(1) assms(2) ideal.axioms(2) ring.ideal_prod_in_carrier)
      then have "s2 ∈ carrier R"
        using sum.hyps(3) by blast
      moreover have "s1 ∈ carrier R"
        using IJ sum.hyps(1) by blast
      ultimately show ?thesis
        by (simp add: ideal_prod.sum r_distr sum.hyps x)
    qed
  qed
  show "s ⊗ x ∈ I · J" using s
  proof (induct s rule: ideal_prod.induct)
    case (prod i j) thus ?case using ideal_prod.prod[of i I "j ⊗ x" J
R] assms x
      by (simp add: x ideal.I_r_closed ideal.Icarr m_assoc)
```

```
  next
    case (sum s1 s2) thus ?case
    proof -
      have "s1 ∈ carrier R" "s2 ∈ carrier R"
        by (meson assms subsetD ideal_prod_in_carrier sum.hyps)+
      then show ?thesis
        by (metis ideal_prod.sum l_distr sum.hyps(2) sum.hyps(4) x)
    qed
  qed
qed

lemma (in ring) ideal_prod_eq_genideal:
  assumes "ideal I R" "ideal J R"
  shows "I · J = Idl (I <#> J)"
proof
  have "I <#> J ⊆ I · J"
  proof
    fix s assume "s ∈ I <#> J"
    then obtain i j where "i ∈ I" "j ∈ J" "s = i ⊗ j"
      unfolding set_mult_def by blast
    thus "s ∈ I · J" using ideal_prod.prod by simp
  qed
  thus "Idl (I <#> J) ⊆ I · J"
    unfolding genideal_def using ideal_prod_is_ideal[OF assms] by blast
next
  show "I · J ⊆ Idl (I <#> J)"
  proof
    fix s assume "s ∈ I · J" thus "s ∈ Idl (I <#> J)"
    proof (induct s rule: ideal_prod.induct)
      case (prod i j) hence "i ⊗ j ∈ I <#> J" unfolding set_mult_def
by blast
      thus ?case unfolding genideal_def by blast
    next
      case (sum s1 s2) thus ?case
        by (simp add: additive_subgroup.a_closed additive_subgroup.a_subset
            assms genideal_ideal ideal.axioms(1) set_mult_closed)
    qed
  qed
qed


lemma (in ring) ideal_prod_simp:
  assumes "ideal I R" "ideal J R"
  shows "I = I <+> (I · J)"
proof
  show "I ⊆ I <+> I · J"
  proof
    fix i assume "i ∈ I" hence "i ⊕ 0 ∈ I <+> I · J"
      using set_add_def'[of R I "I · J"] ideal_prod_is_ideal[OF assms]
```

```
            additive_subgroup.zero_closed[OF ideal.axioms(1), of "I · J"
R] by auto
    thus "i ∈ I <+> I · J"
      using ‹i ∈ I› assms(1) ideal.Icarr by fastforce
  qed
next
  show "I <+> I · J ⊆ I"
  proof
    fix s assume "s ∈ I <+> I · J"
    then obtain i ij where "i ∈ I" "ij ∈ I · J" "s = i ⊕ ij"
      using set_add_def'[of R I "I · J"] by auto
    thus "s ∈ I"
      using ideal_prod_inter[OF assms]
      by (meson additive_subgroup.a_closed assms(1) ideal.axioms(1) inf_sup_ord(1)
subsetCE)
  qed
qed

lemma (in ring) ideal_prod_one:
  assumes "ideal I R"
  shows "I · (carrier R) = I"
proof
  show "I · (carrier R) ⊆ I"
  proof
    fix s assume "s ∈ I · (carrier R)" thus "s ∈ I"
      by (induct s rule: ideal_prod.induct)
         (simp_all add: assms ideal.I_r_closed additive_subgroup.a_closed
ideal.axioms(1))
  qed
next
  show "I ⊆ I · (carrier R)"
  proof
    fix i assume "i ∈ I" thus "i ∈  I · (carrier R)"
      by (metis assms ideal.Icarr ideal_prod.simps one_closed r_one)
  qed
qed

lemma (in ring) ideal_prod_zero:
  assumes "ideal I R"
  shows "I · { 0 } = { 0 }"
proof
  show "I · { 0 } ⊆ { 0 }"
  proof
    fix s assume "s ∈ I · {0}" thus "s ∈ { 0 }"
      using assms ideal.Icarr by (induct s rule: ideal_prod.induct) (fastforce,
simp)
  qed
next
  show "{ 0 } ⊆ I · { 0 }"
```

```
    by (simp add: additive_subgroup.zero_closed assms
                  ideal.axioms(1) ideal_prod_is_ideal zeroideal)
qed

lemma (in ring) ideal_prod_assoc:
  assumes "ideal I R" "ideal J R" "ideal K R"
  shows "(I · J) · K = I · (J · K)"
proof
  show "(I · J) · K ⊆ I · (J · K)"
  proof
    fix s assume "s ∈ (I · J) · K" thus "s ∈ I · (J · K)"
    proof (induct s rule: ideal_prod.induct)
      case (sum s1 s2) thus ?case
        by (simp add: ideal_prod.sum)
    next
      case (prod i k) thus ?case
      proof (induct i rule: ideal_prod.induct)
        case (prod i j) thus ?case
          using ideal_prod.prod[OF prod(1) ideal_prod.prod[OF prod(2-3),of
R], of R]
          by (metis assms ideal.Icarr m_assoc)
      next
        case (sum s1 s2) thus ?case
        proof -
          have "s1 ∈ carrier R" "s2 ∈ carrier R"
            by (meson assms subsetD ideal.axioms(2) ring.ideal_prod_in_carrier
sum.hyps)+
          moreover have "k ∈ carrier R"
            by (meson additive_subgroup.a_Hcarr assms(3) ideal.axioms(1)
sum.prems)
          ultimately show ?thesis
            by (metis ideal_prod.sum l_distr sum.hyps(2) sum.hyps(4) sum.prems)
        qed
      qed
    qed
  qed
next
  show "I · (J · K) ⊆ (I · J) · K"
  proof
    fix s assume "s ∈ I · (J · K)" thus "s ∈ (I · J) · K"
    proof (induct s rule: ideal_prod.induct)
      case (sum s1 s2) thus ?case by (simp add: ideal_prod.sum)
    next
      case (prod i j) show ?case using prod(2) prod(1)
      proof (induct j rule: ideal_prod.induct)
        case (prod j k) thus ?case
          using ideal_prod.prod[OF ideal_prod.prod[OF prod(3) prod(1),
of R] prod (2), of R]
          by (metis assms ideal.Icarr m_assoc)
```

```
      next
        case (sum s1 s2) thus ?case
        proof -
          have "⋀a A B. ⟦a ∈ B · A; ideal A R; ideal B R⟧ ⟹ a ∈ carrier
R"
            by (meson subsetD ideal_prod_in_carrier)
          moreover have "i ∈ carrier R"
            by (meson additive_subgroup.a_Hcarr assms(1) ideal.axioms(1)
sum.prems)
          ultimately show ?thesis
            by (metis (no_types) assms(2) assms(3) ideal_prod.sum r_distr
sum)
        qed
      qed
    qed
  qed
qed

lemma (in ring) ideal_prod_r_distr:
  assumes "ideal I R" "ideal J R" "ideal K R"
  shows "I · (J <+> K) = (I · J) <+>  (I · K)"
proof
  show "I · (J <+> K) ⊆ I · J <+> I · K"
  proof
    fix s assume "s ∈ I · (J <+> K)" thus "s ∈ I · J <+> I · K"
    proof(induct s rule: ideal_prod.induct)
      case (prod i jk)
      then obtain j k where j: "j ∈ J" and k: "k ∈ K" and jk: "jk =
j ⊕ k"
        using set_add_def'[of R J K] by auto
      hence "i ⊗ j ⊕ i ⊗ k ∈ I · J <+> I · K"
        using ideal_prod.prod[OF prod(1) j,of R]
              ideal_prod.prod[OF prod(1) k,of R]
              set_add_def'[of R "I · J" "I · K"] by auto
      thus ?case
        using assms ideal.Icarr r_distr jk j k prod(1) by metis
    next
      case (sum s1 s2) thus ?case
        by (simp add: add_ideals additive_subgroup.a_closed assms ideal.axioms(1)
                      local.ring_axioms ring.ideal_prod_is_ideal)
    qed
  qed

  have aux_lemma: "s ∈ I · (J <+> K) ∧ s ∈ I · (K <+> J)"
    if A: "ideal J R" "ideal K R" "s ∈ I · J" for s J K
  proof -
    from ‹s ∈ I · J› have "s ∈ I · (J <+> K)"
    proof (induct s rule: ideal_prod.induct)
      case (prod i j)
```

```
      hence "(j ⊕ 0) ∈ J <+> K"
        using set_add_def'[of R J K]
          additive_subgroup.zero_closed[OF ideal.axioms(1), of K R] A(2)
by auto
      thus ?case
        by (metis A(1) additive_subgroup.a_Hcarr ideal.axioms(1) ideal_prod.prod
prod r_zero)
    next
      case (sum s1 s2) thus ?case
        by (simp add: ideal_prod.sum)
    qed
    thus ?thesis
      by (metis A(1) A(2) ideal_def ring.union_genideal sup_commute)
  qed
  show "I · J <+> I · K ⊆ I · (J <+> K)"
  proof
    fix s assume "s ∈ I · J <+> I · K"
    then obtain s1 s2 where s1: "s1 ∈ I · J" and s2: "s2 ∈ I · K" and
s: "s = s1 ⊕ s2"
      using set_add_def'[of R "I · J" "I · K"] by auto
    thus "s ∈ I · (J <+> K)"
      using aux_lemma[OF assms(2) assms(3) s1]
            aux_lemma[OF assms(3) assms(2) s2] by (simp add: ideal_prod.sum)
  qed
qed

lemma (in cring) ideal_prod_commute:
  assumes "ideal I R" "ideal J R"
  shows "I · J = J · I"
proof -
  have "I · J ⊆ J · I" if A: "ideal I R" "ideal J R" for I J
  proof
    fix s
    assume "s ∈ I · J"
    thus "s ∈ J · I"
    proof (induct s rule: ideal_prod.induct)
      case (prod i j)
      thus ?case
        using m_comm[OF ideal.Icarr[OF A(1) prod(1)] ideal.Icarr[OF A(2)
prod(2)]]
        by (simp add: ideal_prod.prod)
    next
      case (sum s1 s2)
      thus ?case by (simp add: ideal_prod.sum)
    qed
  qed
  with assms show ?thesis by blast
qed
```

The following result would also be true for locale ring

```
lemma (in cring) ideal_prod_distr:
  assumes "ideal I R" "ideal J R" "ideal K R"
  shows "I · (J <+> K) = (I · J) <+>  (I · K)"
    and "(J <+> K) · I = (J · I) <+>  (K · I)"
  by (simp_all add: assms ideal_prod_commute local.ring_axioms
                    ring.add_ideals ring.ideal_prod_r_distr)

lemma (in cring) ideal_prod_eq_inter:
  assumes "ideal I R" "ideal J R"
    and "I <+> J = carrier R"
  shows "I · J = I ∩ J"
proof
  show "I · J ⊆ I ∩ J"
    using assms ideal_prod_inter by auto
next
  show "I ∩ J ⊆ I · J"
  proof
    have "1 ∈ I <+> J" using assms(3) one_closed by simp
    then obtain i j where ij: "i ∈ I" "j ∈ J" "1 = i ⊕ j"
      using set_add_def'[of R I J] by auto

    fix s assume s: "s ∈ I ∩ J"
    hence "(i ⊗ s ∈ I · J) ∧ (s ⊗ j ∈ I · J)"
      using ij(1-2) by (simp add: ideal_prod.prod)
    moreover have "s = (i ⊗ s) ⊕ (s ⊗ j)"
      using ideal.Icarr[OF assms(1) ij(1)]
            ideal.Icarr[OF assms(2) ij(2)]
            ideal.Icarr[OF assms(1), of s]
      by (metis ij(3) s m_comm[of s i] Int_iff r_distr r_one)
    ultimately show "s ∈  I · J"
      using ideal_prod.sum by fastforce
  qed
qed
```

## 39.2 Structure of the Set of Ideals

We focus on commutative rings for convenience.

```
lemma (in cring) ideals_set_is_semiring: "semiring (ideals_set R)"
proof -
  have "abelian_monoid (ideals_set R)"
    apply (rule abelian_monoidI) unfolding ideals_set_def
    apply (simp_all add: add_ideals zeroideal)
    apply (simp add: add.set_mult_assoc additive_subgroup.a_subset ideal.axioms(1)
set_add_defs(1))
    apply (metis Un_absorb1 additive_subgroup.a_subset additive_subgroup.zero_closed
        cgenideal_minimal cgenideal_self empty_iff genideal_minimal ideal.axioms(1)
        local.ring_axioms order_refl ring.genideal_self subset_antisym
subset_singletonD
        union_genideal zero_closed zeroideal)
```

```
    by (metis sup_commute union_genideal)

  moreover have "monoid (ideals_set R)"
    apply (rule monoidI) unfolding ideals_set_def
    apply (simp_all add: ideal_prod_is_ideal oneideal
                         ideal_prod_commute ideal_prod_one)
    by (metis ideal_prod_assoc ideal_prod_commute)

  ultimately show ?thesis
    unfolding semiring_def semiring_axioms_def ideals_set_def
    by (simp_all add: ideal_prod_distr ideal_prod_commute ideal_prod_zero
zeroideal)
qed

lemma (in cring) ideals_set_is_comm_monoid: "comm_monoid (ideals_set
R)"
proof -
  have "monoid (ideals_set R)"
    apply (rule monoidI) unfolding ideals_set_def
    apply (simp_all add: ideal_prod_is_ideal oneideal
                         ideal_prod_commute ideal_prod_one)
    by (metis ideal_prod_assoc ideal_prod_commute)
  thus ?thesis
    unfolding comm_monoid_def comm_monoid_axioms_def
    by (simp add: ideal_prod_commute ideals_set_def)
qed

lemma (in cring) ideal_prod_eq_Inter_aux:
  assumes "I: {..(Suc n)} → { J. ideal J R }"
    and "⋀i j. ⟦ i ≤ Suc n; j ≤ Suc n ⟧ ⟹
                 i ≠ j ⟹ (I i) <+> (I j) = carrier R"
  shows "(⨂(ideals_set R) k ∈ {..n}. I k) <+> (I (Suc n)) = carrier R"
using assms
proof (induct n arbitrary: I)
  case 0
  hence "(⨂(ideals_set R) k ∈ {..0}. I k) <+> I (Suc 0) = (I 0) <+> (I
(Suc 0))"
    using comm_monoid.finprod_0[OF ideals_set_is_comm_monoid, of I]
    by (simp add: atMost_Suc ideals_set_def)
  also have " ... = carrier R"
    using 0(2)[of 0 "Suc 0"] by simp
  finally show ?case .
next
  interpret ISet: comm_monoid "ideals_set R"
    by (simp add: ideals_set_is_comm_monoid)

  case (Suc n)
  let ?I' = "λi. I (Suc i)"
  have "?I': {..(Suc n)} → { J. ideal J R }"
```

```
    using Suc.prems(1) by auto
  moreover have "⋀i j. ⟦ i ≤ Suc n; j ≤ Suc n ⟧ ⟹
                         i ≠ j ⟹ (?I' i) <+> (?I' j) = carrier R"
    by (simp add: Suc.prems(2))
  ultimately have "(⨂(ideals_set R) k ∈ {..n}. ?I' k) <+> (?I' (Suc n))
= carrier R"
    using Suc.hyps by metis

  moreover have I_carr: "I: {..Suc (Suc n)} → carrier (ideals_set R)"
    unfolding ideals_set_def using Suc by simp
  hence I'_carr: "I ∈ Suc ` {..n} → carrier (ideals_set R)" by auto
  ultimately have "(⨂(ideals_set R) k ∈ {(Suc 0)..Suc n}. I k) <+> (I
(Suc (Suc n))) = carrier R"
    using ISet.finprod_reindex[of I "λi. Suc i" "{..n}"] by (simp add:
atMost_atLeast0)

  hence "(carrier R) · (I 0) = ((⨂(ideals_set R) k ∈ {Suc 0..Suc n}. I
k) <+> I (Suc (Suc n))) · (I 0)"
    by auto
  moreover have fprod_cl1: "ideal (⨂(ideals_set R) k ∈ {Suc 0..Suc n}.
I k) R"
    by (metis I'_carr ISet.finprod_closed One_nat_def ideals_set_def image_Suc_atMost
        mem_Collect_eq partial_object.select_convs(1))
  ultimately
  have "I 0 = (⨂(ideals_set R) k ∈ {Suc 0..Suc n}. I k) · (I 0) <+> I
(Suc (Suc n)) · (I 0)"
    by (metis PiE Suc.prems(1) atLeast0_atMost_Suc atLeast0_atMost_Suc_eq_insert_0
        atMost_atLeast0 ideal_prod_commute ideal_prod_distr(2) ideal_prod_one
insertI1
        mem_Collect_eq oneideal)
  also have " ... = (I 0) · (⨂(ideals_set R) k ∈ {Suc 0..Suc n}. I k)
<+> I (Suc (Suc n)) · (I 0)"
    using fprod_cl1 ideal_prod_commute Suc.prems(1)
    by (simp add: atLeast0_atMost_Suc_eq_insert_0 atMost_atLeast0)
  also have " ... = (I 0) ⊗(ideals_set R) (⨂(ideals_set R) k ∈ {Suc 0..Suc
n}. I k) <+>
                    I (Suc (Suc n)) · (I 0)"
    by (simp add: ideals_set_def)
  finally have I0: "I 0 = (⨂(ideals_set R) k ∈ {..Suc n}. I k) <+> I (Suc
(Suc n)) · (I 0)"
    using ISet.finprod_insert[of "{Suc 0..Suc n}" 0 I]
          I_carr I'_carr atMost_atLeast0 ISet.finprod_0' atMost_Suc by
auto

  have I_SucSuc_I0: "ideal (I (Suc (Suc n))) R ∧ ideal (I 0) R"
    using Suc.prems(1) by auto
  have fprod_cl2: "ideal (⨂(ideals_set R) k ∈ {..Suc n}. I k) R"
    by (metis (no_types) ISet.finprod_closed I_carr Pi_split_insert_domain
atMost_Suc ideals_set_def mem_Collect_eq partial_object.select_convs(1))
```

```
  have "carrier R = I (Suc (Suc n)) <+> I 0"
    by (simp add: Suc.prems(2))
  also have " ... = I (Suc (Suc n)) <+>
                    ((⨂(ideals_set R) k ∈ {..Suc n}. I k) <+> I (Suc (Suc
n)) · (I 0))"
    using I0 by auto
  also have " ... = I (Suc (Suc n)) <+>
                    (I (Suc (Suc n)) · (I 0) <+> (⨂(ideals_set R) k ∈
{..Suc n}. I k))"
    using fprod_cl2 I_SucSuc_I0 by (metis Un_commute ideal_prod_is_ideal
union_genideal)
  also have " ... = (I (Suc (Suc n)) <+> I (Suc (Suc n)) · (I 0)) <+>
                    (⨂(ideals_set R) k ∈ {..Suc n}. I k)"
    using fprod_cl2 I_SucSuc_I0 by (metis add.set_mult_assoc ideal_def
ideal_prod_in_carrier
                                          oneideal ring.ideal_prod_one
set_add_defs(1))
  also have " ... = I (Suc (Suc n)) <+> (⨂(ideals_set R) k ∈ {..Suc n}.
I k)"
    using ideal_prod_simp[of "I (Suc (Suc n))" "I 0"] I_SucSuc_I0 by simp

  also have " ... = (⨂(ideals_set R) k ∈ {..Suc n}. I k) <+> I (Suc (Suc
n))"
    using fprod_cl2 I_SucSuc_I0 by (metis Un_commute union_genideal)
  finally show ?case by simp
qed

theorem (in cring) ideal_prod_eq_Inter:
  assumes "I: {..n :: nat} → { J. ideal J R }"
    and "⋀i j. ⟦ i ∈ {..n}; j ∈ {..n} ⟧ ⟹ i ≠ j ⟹ (I i) <+> (I j)
= carrier R"
  shows "(⨂(ideals_set R) k ∈ {..n}. I k) = (⋂ k ∈ {..n}. I k)" us-
ing assms
proof (induct n)
  case 0 thus ?case
    using comm_monoid.finprod_0[OF ideals_set_is_comm_monoid] by (simp
add: ideals_set_def)
next
  interpret ISet: comm_monoid "ideals_set R"
    by (simp add: ideals_set_is_comm_monoid)

  case (Suc n)
  hence IH: "(⨂(ideals_set R) k ∈ {..n}. I k) = (⋂ k ∈ {..n}. I k)"
    by (simp add: atMost_Suc)
  hence "(⨂(ideals_set R) k ∈ {..Suc n}. I k) = I (Suc n) ⊗(ideals_set R)
(⋂ k ∈ {..n}. I k)"
    using ISet.finprod_insert[of "{Suc 0..Suc n}" 0 I] atMost_Suc_eq_insert_0[of
n]
    by (metis ISet.finprod_Suc Suc.prems(1) ideals_set_def partial_object.select_convs(1))
```

```
  hence "(⨂(ideals_set R) k ∈ {..Suc n}. I k) = I (Suc n) · (⋂ k ∈ {..n}.
I k)"
    by (simp add: ideals_set_def)
  moreover have "(⋂ k ∈ {..n}. I k) <+> I (Suc n) = carrier R"
    using ideal_prod_eq_Inter_aux[of I n] by (simp add: Suc.prems IH)
  moreover have "ideal (⋂ k ∈ {..n}. I k) R"
    using ring.i_Intersect[of R "I ‘ {..n}"]
    by (metis IH ISet.finprod_closed Pi_split_insert_domain Suc.prems(1)
atMost_Suc
              ideals_set_def mem_Collect_eq partial_object.select_convs(1))
  ultimately
  have "(⨂(ideals_set R) k ∈ {..Suc n}. I k) = (⋂ k ∈ {..n}. I k) ∩
I (Suc n)"
    using ideal_prod_eq_inter[of "⋂ k ∈ {..n}. I k" "I (Suc n)"]
          ideal_prod_commute[of "⋂ k ∈ {..n}. I k" "I (Suc n)"]
    by (metis PiE Suc.prems(1) atMost_iff mem_Collect_eq order_refl)
  thus ?case by (simp add: Int_commute atMost_Suc)
qed

corollary (in cring) inter_plus_ideal_eq_carrier:
  assumes "⋀i. i ≤ Suc n ⟹ ideal (I i) R"
      and "⋀i j. ⟦ i ≤ Suc n; j ≤ Suc n; i ≠ j ⟧ ⟹ I i <+> I j = carrier
R"
  shows "(⋂ i ≤ n. I i) <+> (I (Suc n)) = carrier R"
  using ideal_prod_eq_Inter[of I n] ideal_prod_eq_Inter_aux[of I n] by
(auto simp add: assms)

corollary (in cring) inter_plus_ideal_eq_carrier_arbitrary:
  assumes "⋀i. i ≤ Suc n ⟹ ideal (I i) R"
      and "⋀i j. ⟦ i ≤ Suc n; j ≤ Suc n; i ≠ j ⟧ ⟹ I i <+> I j = carrier
R"
      and "j ≤ Suc n"
  shows "(⋂ i ∈ ({..(Suc n)} - { j }). I i) <+> (I j) = carrier R"
proof -
  define I’ where "I’ = (λi. if i = Suc n then (I j) else
                          if i = j     then (I (Suc n))
                                       else (I i))"
  have "⋀i. i ≤ Suc n ⟹ ideal (I’ i) R"
    using I’_def assms(1) assms(3) by auto
  moreover have "⋀i j. ⟦ i ≤ Suc n; j ≤ Suc n; i ≠ j ⟧ ⟹ I’ i <+>
I’ j = carrier R"
    using I’_def assms(2-3) by force
  ultimately have "(⋂ i ≤ n. I’ i) <+> (I’ (Suc n)) = carrier R"
    using inter_plus_ideal_eq_carrier by simp

  moreover have "I’ ‘ {..n} = I ‘ ({..(Suc n)} - { j })"
  proof
    show "I’ ‘ {..n} ⊆ I ‘ ({..Suc n} - {j})"
    proof
```

```
      fix x assume "x ∈ I' ` {..n}"
      then obtain i where i: "i ∈ {..n}" "I' i = x" by blast
      thus "x ∈ I ` ({..Suc n} - {j})"
      proof (cases)
        assume "i = j" thus ?thesis using i I'_def by auto
      next
        assume "i ≠ j" thus ?thesis using I'_def i insert_iff by auto
      qed
    qed
  next
    show "I ` ({..Suc n} - {j}) ⊆ I' ` {..n}"
    proof
      fix x assume "x ∈ I ` ({..Suc n} - {j})"
      then obtain i where i: "i ∈ {..Suc n}" "i ≠ j" "I i = x" by blast
      thus "x ∈ I' ` {..n}"
      proof (cases)
        assume "i = Suc n" thus ?thesis using I'_def assms(3) i(2-3)
by auto
      next
        assume "i ≠ Suc n" thus ?thesis using I'_def i by auto
      qed
    qed
  qed
  ultimately show ?thesis using I'_def by metis
qed
```

## 39.3 Another Characterization of Prime Ideals

With product of ideals being defined, we can give another definition of a prime ideal

```
lemma (in ring) primeideal_divides_ideal_prod:
  assumes "primeideal P R" "ideal I R" "ideal J R"
      and "I · J ⊆ P"
    shows "I ⊆ P ∨ J ⊆ P"
proof (cases)
  assume "∃ i ∈ I. i ∉ P"
  then obtain i where i: "i ∈ I" "i ∉ P" by blast
  have "J ⊆ P"
  proof
    fix j assume j: "j ∈ J"
    hence "i ⊗ j ∈ P"
      using ideal_prod.prod[OF i(1) j, of R] assms(4) by auto
    thus "j ∈ P"
      using primeideal.I_prime[OF assms(1), of i j] i j
      by (meson assms(2-3) ideal.Icarr)
  qed
  thus ?thesis by blast
next
  assume "¬ (∃ i ∈ I. i ∉ P)" thus ?thesis by blast
```

```
qed

lemma (in cring) divides_ideal_prod_imp_primeideal:
  assumes "ideal P R"
    and "P ≠ carrier R"
    and "⋀I J. ⟦ ideal I R; ideal J R; I · J ⊆ P ⟧ ⟹ I ⊆ P ∨ J ⊆ P"
  shows "primeideal P R"
proof -
  have "⋀a b. ⟦ a ∈ carrier R; b ∈ carrier R; a ⊗ b ∈ P ⟧ ⟹ a ∈ P
∨ b ∈ P"
  proof -
    fix a b assume A: "a ∈ carrier R" "b ∈ carrier R" "a ⊗ b ∈ P"
    have "(PIdl a) · (PIdl b) = Idl (PIdl (a ⊗ b))"
      using ideal_prod_eq_genideal[of "Idl { a }" "Idl { b }"]
            A(1-2) cgenideal_eq_genideal cgenideal_ideal cgenideal_prod
by auto
    hence "(PIdl a) · (PIdl b) = PIdl (a ⊗ b)"
      by (simp add: A Idl_subset_ideal cgenideal_ideal cgenideal_minimal
                    genideal_self oneideal subset_antisym)
    hence "(PIdl a) · (PIdl b) ⊆ P"
      by (simp add: A(3) assms(1) cgenideal_minimal)
    hence "(PIdl a) ⊆ P ∨ (PIdl b) ⊆ P"
      by (simp add: A assms(3) cgenideal_ideal)
    thus "a ∈ P ∨ b ∈ P"
      using A cgenideal_self by blast
  qed
  thus ?thesis
    using assms is_cring by (simp add: primeidealI)
qed

end


theory Chinese_Remainder
  imports Weak_Morphisms Ideal_Product

begin
```

# 40 Direct Product of Rings

## 40.1 Definitions

```
definition RDirProd :: "('a, 'n) ring_scheme ⇒ ('b, 'm) ring_scheme ⇒
('a × 'b) ring"
  where "RDirProd R S = monoid.extend (R ×× S)
          ⦇ zero = one ((add_monoid R) ×× (add_monoid S)),
             add = mult ((add_monoid R) ×× (add_monoid S)) ⦈ "

abbreviation nil_ring :: "('a list) ring"
```

```
  where "nil_ring ≡ monoid.extend nil_monoid ⦇ zero = [], add = (λa b.
[]) ⦈"

definition RDirProd_list :: "(('a, 'n) ring_scheme) list ⇒ ('a list) ring"
  where "RDirProd_list Rs = foldr (λR S. image_ring (λ(a, as). a # as)
(RDirProd R S)) Rs nil_ring"
```

## 40.2 Basic Properties

```
lemma RDirProd_carrier: "carrier (RDirProd R S) = carrier R × carrier
S"
  unfolding RDirProd_def DirProd_def by (simp add: monoid.defs)

lemma RDirProd_add_monoid [simp]: "add_monoid (RDirProd R S) = (add_monoid
R) ×× (add_monoid S)"
  by (simp add: RDirProd_def monoid.defs)

lemma RDirProd_ring:
  assumes "ring R" and "ring S" shows "ring (RDirProd R S)"
proof -
  have "monoid (RDirProd R S)"
    using DirProd_monoid[OF assms[THEN ring.axioms(2)]] unfolding monoid_def
    by (auto simp add: DirProd_def RDirProd_def monoid.defs)
  then interpret Prod: group "add_monoid (RDirProd R S)" + monoid "RDirProd
R S"
    using DirProd_group[OF assms[THEN abelian_group.a_group[OF ring.is_abelian_group]]]
    unfolding RDirProd_add_monoid by auto
  show ?thesis
    by (unfold_locales, auto simp add: RDirProd_def DirProd_def monoid.defs
assms ring.ring_simprules)
qed

lemma RDirProd_iso1:
  "(λ(x, y). (y, x)) ∈ ring_iso (RDirProd R S) (RDirProd S R)"
  unfolding ring_iso_def ring_hom_def bij_betw_def inj_on_def
  by (auto simp add: RDirProd_def DirProd_def monoid.defs)

lemma RDirProd_iso2:
  "(λ(x, (y, z)). ((x, y), z)) ∈ ring_iso (RDirProd R (RDirProd S T))
(RDirProd (RDirProd R S) T)"
  unfolding ring_iso_def ring_hom_def bij_betw_def inj_on_def
  by (auto simp add: image_iff RDirProd_def DirProd_def monoid.defs)

lemma RDirProd_iso3:
  "(λ((x, y), z). (x, (y, z))) ∈ ring_iso (RDirProd (RDirProd R S) T)
(RDirProd R (RDirProd S T))"
  unfolding ring_iso_def ring_hom_def bij_betw_def inj_on_def
  by (auto simp add: image_iff RDirProd_def DirProd_def monoid.defs)
```

```
lemma RDirProd_iso4:
  assumes "f ∈ ring_iso R S" shows "(λ(r, t). (f r, t)) ∈ ring_iso (RDirProd
R T) (RDirProd S T)"
  using assms unfolding ring_iso_def ring_hom_def bij_betw_def inj_on_def
  by (auto simp add: image_iff RDirProd_def DirProd_def monoid.defs)+

lemma RDirProd_iso5:
  assumes "f ∈ ring_iso S T" shows "(λ(r, s). (r, f s)) ∈ ring_iso (RDirProd
R S) (RDirProd R T)"
  using ring_iso_set_trans[OF ring_iso_set_trans[OF RDirProd_iso1 RDirProd_iso4[OF
assms]] RDirProd_iso1]
  by (simp add: case_prod_unfold comp_def)

lemma RDirProd_iso6:
  assumes "f ∈ ring_iso R R'" and "g ∈ ring_iso S S'"
  shows "(λ(r, s). (f r, g s)) ∈ ring_iso (RDirProd R S) (RDirProd R'
S')"
  using ring_iso_set_trans[OF RDirProd_iso4[OF assms(1)] RDirProd_iso5[OF
assms(2)]]
  by (simp add: case_prod_beta' comp_def)

lemma RDirProd_iso7:
  shows "(λa. (a, [])) ∈ ring_iso R (RDirProd R nil_ring)"
  unfolding ring_iso_def ring_hom_def bij_betw_def inj_on_def
  by (auto simp add: RDirProd_def DirProd_def monoid.defs)

lemma RDirProd_hom1:
  shows "(λa. (a, a)) ∈ ring_hom R (RDirProd R R)"
  by (auto simp add: ring_hom_def RDirProd_def DirProd_def monoid.defs)

lemma RDirProd_hom2:
  assumes "f ∈ ring_hom S T"
  shows "(λ(x, y). (x, f y)) ∈ ring_hom (RDirProd R S) (RDirProd R T)"
    and "(λ(x, y). (f x, y)) ∈ ring_hom (RDirProd S R) (RDirProd T R)"
  using assms by (auto simp add: ring_hom_def RDirProd_def DirProd_def
monoid.defs)

lemma RDirProd_hom3:
  assumes "f ∈ ring_hom R R'" and "g ∈ ring_hom S S'"
  shows "(λ(r, s). (f r, g s)) ∈ ring_hom (RDirProd R S) (RDirProd R'
S')"
  using ring_hom_trans[OF RDirProd_hom2(2)[OF assms(1)] RDirProd_hom2(1)[OF
assms(2)]]
  by (simp add: case_prod_beta' comp_def)
```

## 40.3 Direct Product of a List of Rings

```
lemma RDirProd_list_nil [simp]: "RDirProd_list [] = nil_ring"
  unfolding RDirProd_list_def by simp
```

```
lemma nil_ring_simprules [simp]:
  "carrier nil_ring = { [] }" and "one nil_ring = []" and "zero nil_ring
= []"
  by (auto simp add: monoid.defs)

lemma RDirProd_list_truncate:
  shows "monoid.truncate (RDirProd_list Rs) = DirProd_list Rs"
proof (induct Rs, simp add: RDirProd_list_def DirProd_list_def monoid.defs)
  case (Cons R Rs)
  have "monoid.truncate (RDirProd_list (R # Rs)) =
        monoid.truncate (image_ring (λ(a, as). a # as) (RDirProd R (RDirProd_list
Rs)))"
    unfolding RDirProd_list_def by simp
  also have " ... = image_group (λ(a, as). a # as) (monoid.truncate (RDirProd
R (RDirProd_list Rs)))"
  by (simp add: image_ring_def image_group_def monoid.defs)
  also have " ... = image_group (λ(a, as). a # as) (R ×× (monoid.truncate
(RDirProd_list Rs)))"
    by (simp add: RDirProd_def DirProd_def monoid.defs)
  also have " ... = DirProd_list (R # Rs)"
    unfolding Cons DirProd_list_def by simp
  finally show ?case .
qed

lemma RDirProd_list_carrier_def':
  shows "carrier (RDirProd_list Rs) = carrier (DirProd_list Rs)"
proof -
  have "carrier (RDirProd_list Rs) = carrier (monoid.truncate (RDirProd_list
Rs))"
    by (simp add: monoid.defs)
  thus ?thesis
    unfolding RDirProd_list_truncate .
qed

lemma RDirProd_list_carrier:
  shows "carrier (RDirProd_list (G # Gs)) = (λ(x, xs). x # xs) ` (carrier
G × carrier (RDirProd_list Gs))"
  unfolding RDirProd_list_carrier_def' using DirProd_list_carrier .

lemma RDirProd_list_one:
  shows "one (RDirProd_list Rs) = foldr (λR tl. (one R) # tl) Rs []"
  unfolding RDirProd_list_def RDirProd_def image_ring_def image_group_def
  by (induct Rs) (auto simp add: monoid.defs)

lemma RDirProd_list_zero:
  shows "zero (RDirProd_list Rs) = foldr (λR tl. (zero R) # tl) Rs []"
  unfolding RDirProd_list_def RDirProd_def image_ring_def
  by (induct Rs) (auto simp add: monoid.defs)
```

```
lemma RDirProd_list_zero':
  shows "zero (RDirProd_list (R # Rs)) = (zero R) # (zero (RDirProd_list
Rs))"
  unfolding RDirProd_list_zero by simp

lemma RDirProd_list_carrier_mem:
  assumes "as ∈ carrier (RDirProd_list Rs)"
  shows "length as = length Rs" and "⋀i. i < length Rs ⟹ (as ! i)
∈ carrier (Rs ! i)"
  using assms DirProd_list_carrier_mem unfolding RDirProd_list_carrier_def'
by auto

lemma RDirProd_list_carrier_memI:
  assumes "length as = length Rs" and "⋀i. i < length Rs ⟹ (as ! i)
∈ carrier (Rs ! i)"
  shows "as ∈ carrier (RDirProd_list Rs)"
  using assms DirProd_list_carrier_memI unfolding RDirProd_list_carrier_def'
by auto

lemma inj_on_RDirProd_carrier:
  shows "inj_on (λ(a, as). a # as) (carrier (RDirProd R (RDirProd_list
Rs)))"
  unfolding RDirProd_def DirProd_def inj_on_def by auto

lemma RDirProd_list_is_ring:
  assumes "⋀i. i < length Rs ⟹ ring (Rs ! i)" shows "ring (RDirProd_list
Rs)"
  using assms
proof (induct Rs)
  case Nil thus ?case
    unfolding RDirProd_list_def by (unfold_locales, auto simp add: monoid.defs
Units_def)
next
  case (Cons R Rs)
  hence is_ring: "ring (RDirProd R (RDirProd_list Rs))"
    using RDirProd_ring[of R "RDirProd_list Rs"] by force
  show ?case
    using ring.inj_imp_image_ring_is_ring[OF is_ring inj_on_RDirProd_carrier]
    unfolding RDirProd_list_def by auto
qed

lemma RDirProd_list_iso1:
  "(λ(a, as). a # as) ∈ ring_iso (RDirProd R (RDirProd_list Rs)) (RDirProd_list
(R # Rs))"
  using inj_imp_image_ring_iso[OF inj_on_RDirProd_carrier] unfolding RDirProd_list_def
by auto

lemma RDirProd_list_iso2:
```

```
  "Hilbert_Choice.inv (λ(a, as). a # as) ∈ ring_iso (RDirProd_list (R
# Rs)) (RDirProd R (RDirProd_list Rs))"
  unfolding RDirProd_list_def by (auto intro: inj_imp_image_ring_inv_iso
simp add: inj_def)

lemma RDirProd_list_iso3:
  "(λa. [ a ]) ∈ ring_iso R (RDirProd_list [ R ])"
proof -
  have [simp]: "(λa. [ a ]) = (λ(a, as). a # as) ∘ (λa. (a, []))" by
auto
  show ?thesis
    using ring_iso_set_trans[OF RDirProd_iso7] RDirProd_list_iso1[of R
"[]"]
    unfolding RDirProd_list_def by simp
qed

lemma RDirProd_list_hom1:
  "(λ(a, as). a # as) ∈ ring_hom (RDirProd R (RDirProd_list Rs)) (RDirProd_list
(R # Rs))"
  using RDirProd_list_iso1 unfolding ring_iso_def by auto

lemma RDirProd_list_hom2:
  assumes "f ∈ ring_hom R S" shows "(λa. [ f a ]) ∈ ring_hom R (RDirProd_list
[ S ])"
proof -
  have hom1: "(λa. (a, [])) ∈ ring_hom R (RDirProd R nil_ring)"
    using RDirProd_iso7 unfolding ring_iso_def by auto
  have hom2: "(λ(a, as). a # as) ∈ ring_hom (RDirProd S nil_ring) (RDirProd_list
[ S ])"
    using RDirProd_list_hom1[of _ "[]"] unfolding RDirProd_list_def by
auto
  have [simp]: "(λ(a, as). a # as) ∘ ((λ(x, y). (f x, y)) ∘ (λa. (a, [])))
= (λa. [ f a ])" by auto
  show ?thesis
    using ring_hom_trans[OF ring_hom_trans[OF hom1 RDirProd_hom2(2)[OF
assms]] hom2] by simp
qed
```

# 41 Chinese Remainder Theorem

## 41.1 Definitions

```
abbreviation (in ring) canonical_proj :: "'a set ⇒ 'a set ⇒ 'a ⇒ 'a
set × 'a set"
  where "canonical_proj I J ≡ (λa. (I +> a, J +> a))"

definition (in ring) canonical_proj_ext :: "(nat ⇒ 'a set) ⇒ nat ⇒ 'a
⇒ ('a set) list"
  where "canonical_proj_ext I n = (λa. map (λi. (I i) +> a) [0..< Suc
```

```
n])"
```

### 41.2 Chinese Remainder Simple

```
lemma (in ring) canonical_proj_is_surj:
  assumes "ideal I R" "ideal J R" and "I <+> J = carrier R"
  shows "(canonical_proj I J) ` carrier R = carrier (RDirProd (R Quot
I) (R Quot J))"
  unfolding RDirProd_def DirProd_def FactRing_def A_RCOSETS_def'
proof (auto simp add: monoid.defs)
  have aux_lemma1: "I +> i = 0R Quot I" if "ideal I R" "i ∈ I" for I i
    using that a_rcos_zero by (simp add: FactRing_def)

  have aux_lemma2: "I +> j = 1R Quot I"
    if A: "ideal I R" "i ∈ I" "j ∈ carrier R" "i ⊕ j = 1"
    for I i j
  proof -
    have "(I +> i) ⊕R Quot I (I +> j) = I +> 1"
      using ring_hom_memE(3)[OF ideal.rcos_ring_hom ideal.Icarr[OF _ A(2)]
A(3)] A(1,4) by simp
    moreover have "I +> i = I"
      using abelian_subgroupI3[OF ideal.axioms(1) is_abelian_group]
      by (simp add: A(1-2) abelian_subgroup.a_rcos_const)
    moreover have "I +> j ∈ carrier (R Quot I)" and "I = 0R Quot I" and
"I +> 1 = 1R Quot I"
      by (auto simp add: FactRing_def A_RCOSETS_def' A(3))
    ultimately show ?thesis
      using ring.ring_simprules(8)[OF ideal.quotient_is_ring[OF A(1)]]
by simp
  qed

  interpret I: ring "R Quot I" + J: ring "R Quot J"
    using assms(1-2)[THEN ideal.quotient_is_ring] by auto

  fix a b assume a: "a ∈ carrier R" and b: "b ∈ carrier R"
  have "1 ∈ I <+> J"
    using assms(3) by blast
  then obtain i j where i: "i ∈ carrier R" "i ∈ I" and j: "j ∈ carrier
R" "j ∈ J" and ij: "i ⊕ j = 1"
    using assms(1-2)[THEN ideal.Icarr] unfolding set_add_def' by auto
  hence rcos_j: "I +> j = 1R Quot I" and rcos_i: "J +> i = 1R Quot J"
    using assms(1-2)[THEN aux_lemma2] a_comm by simp+

  define s where "s = (a ⊗ j) ⊕ (b ⊗ i)"
  hence "s ∈ carrier R"
    using a b i j by simp

  have "I +> s = ((I +> a) ⊗R Quot I (I +> j)) ⊕R Quot I (I +> (b ⊗ i))"
    using ring_hom_memE(2-3)[OF ideal.rcos_ring_hom[OF assms(1)]]
```

```
    by (simp add: a b i(1) j(1) s_def)
  moreover have "I +> a ∈ carrier (R Quot I)"
    by (auto simp add: FactRing_def A_RCOSETS_def' a)
  ultimately have "I +> s = I +> a"
    unfolding rcos_j aux_lemma1[OF assms(1) ideal.I_l_closed[OF assms(1)
i(2) b]] by simp

  have "J +> s = (J +> (a ⊗ j)) ⊕R Quot J ((J +> b) ⊗R Quot J (J +> i))"
    using ring_hom_memE(2-3)[OF ideal.rcos_ring_hom[OF assms(2)]]
    by (simp add: a b i(1) j(1) s_def)
  moreover have "J +> b ∈ carrier (R Quot J)"
    by (auto simp add: FactRing_def A_RCOSETS_def' b)
  ultimately have "J +> s = J +> b"
    unfolding rcos_i aux_lemma1[OF assms(2) ideal.I_l_closed[OF assms(2)
j(2) a]] by simp

  from ‹I +> s = I +> a› and ‹J +> s = J +> b› and ‹s ∈ carrier R›
  show "(I +> a, J +> b) ∈ (canonical_proj I J) ‘ carrier R" by blast
qed

lemma (in ring) canonical_proj_ker:
  assumes "ideal I R" and "ideal J R"
  shows "a_kernel R (RDirProd (R Quot I) (R Quot J)) (canonical_proj I
J) = I ∩ J"
proof
  show "a_kernel R (RDirProd (R Quot I) (R Quot J)) (canonical_proj I
J) ⊆ I ∩ J"
    unfolding FactRing_def RDirProd_def DirProd_def a_kernel_def'
    by (auto simp add: assms[THEN ideal.rcos_const_imp_mem] monoid.defs)
next
  show "I ∩ J ⊆ a_kernel R (RDirProd (R Quot I) (R Quot J)) (canonical_proj
I J)"
  proof
    fix s assume s: "s ∈ I ∩ J" then have "I +> s = I" and "J +> s =
J"
      using abelian_subgroupI3[OF ideal.axioms(1) is_abelian_group]
      by (simp add: abelian_subgroup.a_rcos_const assms)+
    thus "s ∈ a_kernel R (RDirProd (R Quot I) (R Quot J)) (canonical_proj
I J)"
      unfolding FactRing_def RDirProd_def DirProd_def a_kernel_def'
      using ideal.Icarr[OF assms(1)] s by (simp add: monoid.defs)
  qed
qed

lemma (in ring) canonical_proj_is_hom:
  assumes "ideal I R" and "ideal J R"
  shows "(canonical_proj I J) ∈ ring_hom R (RDirProd (R Quot I) (R Quot
J))"
  unfolding RDirProd_def DirProd_def FactRing_def A_RCOSETS_def'
```

```
  by (auto intro!: ring_hom_memI
         simp add: assms[THEN ideal.rcoset_mult_add]
                   assms[THEN ideal.a_rcos_sum] monoid.defs)

lemma (in ring) canonical_proj_ring_hom:
  assumes "ideal I R" and "ideal J R"
  shows "ring_hom_ring R (RDirProd (R Quot I) (R Quot J)) (canonical_proj
I J)"
  using ring_hom_ring.intro[OF ring_axioms RDirProd_ring[OF assms[THEN
ideal.quotient_is_ring]]]
  by (simp add: ring_hom_ring_axioms_def canonical_proj_is_hom[OF assms])

theorem (in ring) chinese_remainder_simple:
  assumes "ideal I R" "ideal J R" and "I <+> J = carrier R"
  shows "R Quot (I ∩ J) ≃ RDirProd (R Quot I) (R Quot J)"
  using ring_hom_ring.FactRing_iso[OF canonical_proj_ring_hom canonical_proj_is_surj]
  by (simp add: canonical_proj_ker assms)
```

## 41.3 Chinese Remainder Generalized

```
lemma (in ring) canonical_proj_ext_zero [simp]: "(canonical_proj_ext
I 0) = (λa. [ (I 0) +> a ])"
  unfolding canonical_proj_ext_def by simp

lemma (in ring) canonical_proj_ext_tl:
  "(λa. canonical_proj_ext I (Suc n) a) = (λa. ((I 0) +> a) # (canonical_proj_ext
(λi. I (Suc i)) n a))"
  unfolding canonical_proj_ext_def by (induct n) (auto, metis (lifting)
append.assoc append_Cons append_Nil)

lemma (in ring) canonical_proj_ext_is_hom:
  assumes "⋀i. i ≤ n ⟹ ideal (I i) R"
  shows "(canonical_proj_ext I n) ∈ ring_hom R (RDirProd_list (map (λi.
R Quot (I i)) [0..< Suc n]))"
  using assms
proof (induct n arbitrary: I)
  case 0 thus ?case
    using RDirProd_list_hom2[OF ideal.rcos_ring_hom[of _ R]] by (simp
add: canonical_proj_ext_def)
next
  let ?DirProd = "λI n. RDirProd_list (map (λi. R Quot (I i)) [0..<Suc
n])"
  let ?proj = "λI n. canonical_proj_ext I n"

  case (Suc n)
  hence I: "ideal (I 0) R" by simp
  have hom: "(?proj (λi. I (Suc i)) n) ∈ ring_hom R (?DirProd (λi. I
(Suc i)) n)"
    using Suc(1)[of "λi. I (Suc i)"] Suc(2) by simp
```

```
  have [simp]:
    "(λ(a, as). a # as) ∘ ((λ(r, s). (I 0 +> r, ?proj (λi. I (Suc i))
n s)) ∘ (λa. (a, a))) = ?proj I (Suc n)"
    unfolding canonical_proj_ext_tl by auto
  moreover have
    "(R Quot I 0) # (map (λi. R Quot I (Suc i)) [0..< Suc n]) = map (λi.
R Quot (I i)) [0..< Suc (Suc n)]"
    by (induct n) (auto)
  moreover show ?case
    using ring_hom_trans[OF ring_hom_trans[OF RDirProd_hom1
          RDirProd_hom3[OF ideal.rcos_ring_hom[OF I] hom]] RDirProd_list_hom1]
    unfolding calculation(2) by simp
qed

lemma (in ring) RDirProd_Quot_list_is_ring:
  assumes "⋀i. i ≤ n ⟹ ideal (I i) R" shows "ring (RDirProd_list (map
(λi. R Quot (I i)) [0..< Suc n]))"
proof -
  have ring_list: "⋀i. i < Suc n ⟹ ring ((map (λi. R Quot I i) [0..<
Suc n]) ! i)"
    using ideal.quotient_is_ring[OF assms]
    by (metis add.left_neutral diff_zero le_simps(2) nth_map_upt)
  show ?thesis
    using RDirProd_list_is_ring[OF ring_list] by simp
qed

lemma (in ring) canonical_proj_ext_ring_hom:
  assumes "⋀i. i ≤ n ⟹ ideal (I i) R"
  shows "ring_hom_ring R (RDirProd_list (map (λi. R Quot (I i)) [0..<
Suc n])) (canonical_proj_ext I n)"
proof -
  have ring: "ring (RDirProd_list (map (λi. R Quot (I i)) [0..< Suc n]))"
    using RDirProd_Quot_list_is_ring[OF assms] by simp
  show ?thesis
    using canonical_proj_ext_is_hom assms ring_hom_ring.intro[OF ring_axioms
ring]
    unfolding ring_hom_ring_axioms_def by blast
qed

lemma (in ring) canonical_proj_ext_ker:
  assumes "⋀i. i ≤ (n :: nat) ⟹ ideal (I i) R"
  shows "a_kernel R (RDirProd_list (map (λi. R Quot (I i)) [0..< Suc
n])) (canonical_proj_ext I n) = (⋂i ≤ n. I i)"
proof -
  let ?map_Quot = "λI n. map (λi. R Quot (I i)) [0..< Suc n]"
  let ?ker = "λI n. a_kernel R (RDirProd_list (?map_Quot I n)) (canonical_proj_ext
I n)"

  from assms show ?thesis
```

```
  proof (induct n arbitrary: I)
    case 0 then have I: "ideal (I 0) R" by simp
    show ?case
      unfolding a_kernel_def' RDirProd_list_zero canonical_proj_ext_def
FactRing_def
      using ideal.rcos_const_imp_mem a_rcos_zero ideal.Icarr I by auto

  next
    case (Suc n)
    hence I: "ideal (I 0) R" by simp
    have map_simp: "?map_Quot I (Suc n) = (R Quot I 0) # (?map_Quot (λi.
I (Suc i)) n)"
      by (induct n) (auto)
    have ker_I0: "I 0 = a_kernel R (R Quot (I 0)) (λa. (I 0) +> a)"
      using ideal.rcos_const_imp_mem[OF I] a_rcos_zero[OF I] ideal.Icarr[OF
I]
      unfolding a_kernel_def' FactRing_def by auto
    hence "?ker I (Suc n) = (?ker (λi. I (Suc i)) n) ∩ (I 0)"
      unfolding a_kernel_def' map_simp RDirProd_list_zero' canonical_proj_ext_tl
by auto
    moreover have "?ker (λi. I (Suc i)) n = (⋂i ≤ n. I (Suc i))"
      using Suc(1)[of "λi. I (Suc i)"] Suc(2) by auto
    ultimately show ?case
      by (auto simp add: INT_extend_simps(10) atMost_atLeast0)
         (metis atLeastAtMost_iff le_zero_eq not_less_eq_eq)
  qed
qed

lemma (in cring) canonical_proj_ext_is_surj:
  assumes "⋀i. i ≤ n ⟹ ideal (I i) R" and "⋀i j. ⟦ i ≤ n; j ≤ n
⟧ ⟹ i ≠ j ⟹ I i <+> I j = carrier R"
  shows "(canonical_proj_ext I n) ` carrier R = carrier (RDirProd_list
(map (λi. R Quot (I i)) [0..< Suc n]))"
  using assms
proof (induct n arbitrary: I)
  case 0 show ?case
    by (auto simp add: RDirProd_list_carrier FactRing_def A_RCOSETS_def')
next
  have aux_lemma: "(λa. (f a, g a)) ` carrier R = (f ` carrier R) × (g
` carrier R)"
    if A: "ring T" "f ∈ ring_hom R S" "g ∈ ring_hom R T" "f ` carrier
R ⊆ f ` (a_kernel R T g)"
    for S :: "'c ring" and T :: "'d ring" and f g
  proof
    show "(λa. (f a, g a)) ` carrier R ⊆ (f ` carrier R) × (g ` carrier
R)"
      by blast
  next
    show "(f ` carrier R) × (g ` carrier R) ⊆ (λa. (f a, g a)) ` carrier
```

```
R"
    proof
      fix t assume "t ∈ (f ‘ carrier R) × (g ‘ carrier R)"
      then obtain a b where a: "a ∈ carrier R" "f a = fst t" and b:
"b ∈ carrier R" "g b = snd t"
        by auto
      obtain c where c: "c ∈ a_kernel R T g" "f c = f (a ⊖ b)"
        using A(4) minus_closed[OF a(1) b (1)] by auto
      have "f (c ⊕ b) = f (a ⊖ b) ⊕S  f b"
        using ring_hom_memE(3)[OF A(2)] b c unfolding a_kernel_def’ by
auto
      hence "f (c ⊕ b) = f a"
        using ring_hom_memE(3)[OF A(2) minus_closed[of a b], of b] a b
by algebra
      moreover have "g (c ⊕ b) = g b"
        using ring_hom_memE(1,3)[OF A(3)] b(1) c ring.ring_simprules(8)[OF
A(1)]
        unfolding a_kernel_def’ by auto
      ultimately have "(λa. (f a, g a)) (c ⊕ b) = t" and "c ⊕ b ∈ carrier
R"
        using a b c unfolding a_kernel_def’ by auto
      thus "t ∈ (λa. (f a, g a)) ‘ carrier R"
        by blast
    qed
  qed

  let ?map_Quot = "λI n. map (λi. R Quot (I i)) [0..< Suc n]"
  let ?DirProd = "λI n. RDirProd_list (?map_Quot I n)"
  let ?proj = "λI n. canonical_proj_ext I n"

  case (Suc n)
  interpret I: ideal "I 0" R + Inter: ideal "⋂i ≤ n. I (Suc i)" R
    using i_Intersect[of "(λi. I (Suc i)) ‘ {..n}"] Suc(2) by auto

  have map_simp: "?map_Quot I (Suc n) = (R Quot I 0) # (?map_Quot (λi.
I (Suc i)) n)"
    by (induct n) (auto)

  have IH: "(?proj (λi. I (Suc i)) n) ‘ carrier R = carrier (?DirProd
(λi. I (Suc i)) n)"
   and ring: "ring (?DirProd (λi. I (Suc i)) n)"
   and hom: "?proj (λi. I (Suc i)) n ∈ ring_hom R (?DirProd (λi. I (Suc
i)) n)"
    using RDirProd_Quot_list_is_ring[of n "λi. I (Suc i)"] Suc(1)[of "λi.
I (Suc i)"]
          canonical_proj_ext_is_hom[of n "λi. I (Suc i)"] Suc(2-3) by
auto

  have ker: "a_kernel R (?DirProd (λi. I (Suc i)) n) (?proj (λi. I (Suc
```

```
i)) n) = (⋂i ≤ n. I (Suc i))"
    using canonical_proj_ext_ker[of n "λi. I (Suc i)"] Suc(2) by auto
  have carrier_Quot: "carrier (R Quot (I 0)) = (λa. (I 0) +> a) ` carrier
R"
    by (auto simp add: RDirProd_list_carrier FactRing_def A_RCOSETS_def')
  have ring: "ring (?DirProd (λi. I (Suc i)) n)"
   and hom: "?proj (λi. I (Suc i)) n ∈ ring_hom R (?DirProd (λi. I (Suc
i)) n)"
    using RDirProd_Quot_list_is_ring[of n "λi. I (Suc i)"]
          canonical_proj_ext_is_hom[of n "λi. I (Suc i)"] Suc(2) by auto
  have "carrier (R Quot (I 0)) ⊆ (λa. (I 0) +> a) ` (⋂i ≤ n. I (Suc
i))"
  proof
    have "(⋂i ∈ {Suc 0.. Suc n}. I i) <+> (I 0) = carrier R"
      using inter_plus_ideal_eq_carrier_arbitrary[of n I 0]
      by (simp add: Suc(2-3) atLeast1_atMost_eq_remove0)
    hence eq_carrier: "(I 0) <+> (⋂i ≤ n. I (Suc i)) = carrier R"
      using set_add_comm[OF I.a_subset Inter.a_subset]
      by (metis INT_extend_simps(10) atMost_atLeast0 image_Suc_atLeastAtMost)

    fix b assume "b ∈ carrier (R Quot (I 0))"
    hence "(b, (⋂i ≤ n. I (Suc i))) ∈ carrier (R Quot I 0) × carrier
(R Quot (⋂i≤n. I (Suc i)))"
      using ring.ring_simprules(2)[OF Inter.quotient_is_ring] by (simp
add: FactRing_def)
    then obtain s
      where "s ∈ carrier R" "(canonical_proj (I 0) (⋂i ≤ n. I (Suc i)))
s = (b, (⋂i ≤ n. I (Suc i)))"
      using canonical_proj_is_surj[OF I.is_ideal Inter.is_ideal eq_carrier]
      unfolding RDirProd_carrier by (metis (no_types, lifting) imageE)
    hence "s ∈ (⋂i ≤ n. I (Suc i))" and "(λa. (I 0) +> a) s = b"
      using Inter.rcos_const_imp_mem by auto
    thus "b ∈ (λa. (I 0) +> a) ` (⋂i ≤ n. I (Suc i))"
      by auto
  qed
  hence "(λa. ((I 0) +> a, ?proj (λi. I (Suc i)) n a)) ` carrier R =
         carrier (R Quot (I 0)) × carrier (?DirProd (λi. I (Suc i)) n)"
    using aux_lemma[OF ring I.rcos_ring_hom hom] unfolding carrier_Quot
ker IH by simp
  moreover show ?case
    unfolding map_simp RDirProd_list_carrier sym[OF calculation(1)] canonical_proj_ext_tl
by auto
qed

theorem (in cring) chinese_remainder:
  assumes "⋀i. i ≤ n ⟹ ideal (I i) R" and "⋀i j. ⟦ i ≤ n; j ≤ n
⟧ ⟹ i ≠ j ⟹ I i <+> I j = carrier R"
  shows "R Quot (⋂i ≤ n. I i) ≃ RDirProd_list (map (λi. R Quot (I i))
[0..< Suc n])"
```

```
  using ring_hom_ring.FactRing_iso[OF canonical_proj_ext_ring_hom, of
n I]
        canonical_proj_ext_is_surj[of n I] canonical_proj_ext_ker[of n
I] assms
  by auto

end
```

```
theory Generated_Rings
  imports Subrings
begin
```

# 42 Generated Rings

```
inductive_set
  generate_ring :: "('a, 'b) ring_scheme ⇒ 'a set ⇒ 'a set"
  for R and H where
    one:   "𝟭R ∈ generate_ring R H"
  | incl:  "h ∈ H ⟹ h ∈ generate_ring R H"
  | a_inv: "h ∈ generate_ring R H ⟹ ⊖R h ∈ generate_ring R H"
  | eng_add : "⟦ h1 ∈ generate_ring R H; h2 ∈ generate_ring R H ⟧ ⟹
h1 ⊕R h2 ∈ generate_ring R H"
  | eng_mult: "⟦ h1 ∈ generate_ring R H; h2 ∈ generate_ring R H ⟧ ⟹
h1 ⊗R h2 ∈ generate_ring R H"
```

## 42.1 Basic Properties of Generated Rings - First Part

```
lemma (in ring) generate_ring_in_carrier:
  assumes "H ⊆ carrier R"
  shows "h ∈ generate_ring R H ⟹ h ∈ carrier R"
  apply (induction rule: generate_ring.induct) using assms
  by blast+

lemma (in ring) generate_ring_incl:
  assumes "H ⊆ carrier R"
  shows "generate_ring R H ⊆ carrier R"
  using generate_ring_in_carrier[OF assms] by auto

lemma (in ring) zero_in_generate: "𝟬R ∈ generate_ring R H"
  using one a_inv by (metis generate_ring.eng_add one_closed r_neg)

lemma (in ring) generate_ring_is_subring:
  assumes "H ⊆ carrier R"
  shows "subring (generate_ring R H) R"
  by (auto intro!: subringI[of "generate_ring R H"]
```

```
        simp add: generate_ring_in_carrier[OF assms] one a_inv eng_mult
eng_add)

lemma (in ring) generate_ring_is_ring:
  assumes "H ⊆ carrier R"
  shows "ring (R ⦇ carrier := generate_ring R H ⦈)"
  using subring_iff[OF generate_ring_incl[OF assms]] generate_ring_is_subring[OF
assms] by simp

lemma (in ring) generate_ring_min_subring1:
  assumes "H ⊆ carrier R" and "subring E R" "H ⊆ E"
  shows "generate_ring R H ⊆ E"
proof
  fix h assume h: "h ∈ generate_ring R H"
  show "h ∈ E"
    using h and assms(3)
      by (induct rule: generate_ring.induct)
        (auto simp add: subringE(3,5-7)[OF assms(2)])
qed

lemma (in ring) generate_ringI:
  assumes "H ⊆ carrier R"
    and "subring E R" "H ⊆ E"
    and "⋀K. ⟦ subring K R; H ⊆ K ⟧ ⟹ E ⊆ K"
  shows "E = generate_ring R H"
proof
  show "E ⊆ generate_ring R H"
    using assms generate_ring_is_subring generate_ring.incl by (metis
subset_iff)
  show "generate_ring R H ⊆ E"
    using generate_ring_min_subring1[OF assms(1-3)] by simp
qed

lemma (in ring) generate_ringE:
  assumes "H ⊆ carrier R" and "E = generate_ring R H"
  shows "subring E R" and "H ⊆ E" and "⋀K. ⟦ subring K R; H ⊆ K ⟧ ⟹
E ⊆ K"
proof -
  show "subring E R" using assms generate_ring_is_subring by simp
  show "H ⊆ E" using assms(2) by (simp add: generate_ring.incl subsetI)
  show "⋀K. subring K R  ⟹ H ⊆ K ⟹ E ⊆ K"
    using assms generate_ring_min_subring1 by auto
qed

lemma (in ring) generate_ring_min_subring2:
  assumes "H ⊆ carrier R"
  shows "generate_ring R H = ⋂{K. subring K R ∧ H ⊆ K}"
proof
  have "subring (generate_ring R H) R ∧ H ⊆ generate_ring R H"
```

```
    by (simp add: assms generate_ringE(2) generate_ring_is_subring)
  thus "⋂{K. subring K R ∧ H ⊆ K} ⊆ generate_ring R H" by blast
next
  have "⋀K. subring K R ∧ H ⊆ K ⟹ generate_ring R H ⊆ K"
    by (simp add: assms generate_ring_min_subring1)
  thus "generate_ring R H ⊆ ⋂{K. subring K R ∧ H ⊆ K}" by blast
qed

lemma (in ring) mono_generate_ring:
  assumes "I ⊆ J" and "J ⊆ carrier R"
  shows "generate_ring R I ⊆ generate_ring R J"
proof-
  have "I ⊆ generate_ring R J "
    using assms generate_ringE(2) by blast
  thus "generate_ring R I ⊆ generate_ring R J"
    using generate_ring_min_subring1[of I "generate_ring R J"] assms generate_ring_is_subri
assms(2)]
    by blast
qed

lemma (in ring) subring_gen_incl :
  assumes "subring H R"
    and  "subring K R"
    and "I ⊆ H"
    and "I ⊆ K"
  shows "generate_ring (R⦇carrier := K⦈) I ⊆ generate_ring (R⦇carrier
:= H⦈) I"
proof
  have incl_HK: "generate_ring (R ⦇ carrier := J ⦈) I ⊆ J" if J_def :
"subring J R" "I ⊆ J" for J
    using ring.mono_generate_ring[of "(R⦇carrier := J⦈)" I J ] subring_is_ring[OF
J_def(1)]
      ring.generate_ring_in_carrier[of "R⦇carrier := J⦈"]  ring_axioms
J_def(2)
    by auto

  fix x
  have "x ∈ generate_ring (R⦇carrier := K⦈) I ⟹ x ∈ generate_ring (R⦇carrier
:= H⦈) I"
  proof (induction  rule : generate_ring.induct)
    case one
    have "𝟭R⦇carrier := H⦈ ⊗ 𝟭R⦇carrier := K⦈ = 𝟭R⦇carrier := H⦈" by simp
    moreover have "𝟭R⦇carrier := H⦈ ⊗ 𝟭R⦇carrier := K⦈ = 𝟭R⦇carrier := K⦈"
by simp
    ultimately show ?case using assms generate_ring.one by metis
  next
    case (incl h) thus ?case using generate_ring.incl by force
  next
    case (a_inv h)
```

```
    have "a_inv (R⦇carrier := K⦈) h = a_inv R h"
      using assms group.m_inv_consistent[of "add_monoid R" K] a_comm_group
incl_HK[of K] a_inv
      unfolding subring_def comm_group_def a_inv_def by auto
    moreover have "a_inv (R⦇carrier := H⦈) h = a_inv R h"
      using assms group.m_inv_consistent[of "add_monoid R" H] a_comm_group
incl_HK[of H] a_inv
      unfolding subring_def comm_group_def a_inv_def by auto
    ultimately show ?case using generate_ring.a_inv a_inv.IH by fastforce
  next
    case (eng_add h1 h2)
    thus ?case using incl_HK assms generate_ring.eng_add by force
  next
    case (eng_mult h1 h2)
    thus ?case using generate_ring.eng_mult by force
  qed
  thus "x ∈ generate_ring (R⦇carrier := K⦈) I ⟹ x ∈ generate_ring (R⦇carrier
:= H⦈) I"
    by auto
qed

lemma (in ring) subring_gen_equality:
  assumes "subring H R" "K ⊆ H"
  shows "generate_ring R K = generate_ring (R ⦇ carrier := H ⦈) K"
  using subring_gen_incl[OF assms(1)carrier_is_subring assms(2)] assms
subringE(1)
        subring_gen_incl[OF carrier_is_subring assms(1) _ assms(2)]
  by force

end


theory Generated_Fields
imports Generated_Rings Subrings Multiplicative_Group
begin

inductive_set
  generate_field :: "('a, 'b) ring_scheme ⇒ 'a set ⇒ 'a set"
  for R and H where
    one  : "𝟭⇘R⇙ ∈ generate_field R H"
  | incl : "h ∈ H ⟹ h ∈ generate_field R H"
  | a_inv: "h ∈ generate_field R H ⟹ ⊖⇘R⇙ h ∈ generate_field R H"
  | m_inv: "⟦ h ∈ generate_field R H; h ≠ 𝟬⇘R⇙ ⟧ ⟹ inv⇘R⇙ h ∈ generate_field
R H"
  | eng_add : "⟦ h1 ∈ generate_field R H; h2 ∈ generate_field R H ⟧ ⟹
h1 ⊕⇘R⇙ h2 ∈ generate_field R H"
  | eng_mult: "⟦ h1 ∈ generate_field R H; h2 ∈ generate_field R H ⟧ ⟹
h1 ⊗⇘R⇙ h2 ∈ generate_field R H"
```

## 42.2 Basic Properties of Generated Rings - First Part

```
lemma (in field) generate_field_in_carrier:
  assumes "H ⊆ carrier R"
  shows "h ∈ generate_field R H ⟹ h ∈ carrier R"
  apply (induction rule: generate_field.induct)
  using assms field_Units
  by blast+

lemma (in field) generate_field_incl:
  assumes "H ⊆ carrier R"
  shows "generate_field R H ⊆ carrier R"
  using generate_field_in_carrier[OF assms] by auto

lemma (in field) zero_in_generate: "𝟬R ∈ generate_field R H"
  using one a_inv generate_field.eng_add one_closed r_neg
  by metis

lemma (in field) generate_field_is_subfield:
  assumes "H ⊆ carrier R"
  shows "subfield (generate_field R H) R"
proof (intro subfieldI', simp_all add: m_inv)
  show "subring (generate_field R H) R"
    by (auto intro: subringI[of "generate_field R H"]
             simp add: eng_add a_inv eng_mult one generate_field_in_carrier[OF
assms])
qed

lemma (in field) generate_field_is_add_subgroup:
  assumes "H ⊆ carrier R"
  shows "subgroup (generate_field R H) (add_monoid R)"
  using subring.axioms(1)[OF subfieldE(1)[OF generate_field_is_subfield[OF
assms]]] .

lemma (in field) generate_field_is_field :
  assumes "H ⊆ carrier R"
  shows "field (R ⦇ carrier := generate_field R H ⦈)"
  using subfield_iff generate_field_is_subfield assms by simp

lemma (in field) generate_field_min_subfield1:
  assumes "H ⊆ carrier R"
    and "subfield E R" "H ⊆ E"
  shows "generate_field R H ⊆ E"
proof
  fix h
  assume h: "h ∈ generate_field R H"
  show "h ∈ E"
    using h and assms(3) and subfield_m_inv[OF assms(2)]
    by (induct rule: generate_field.induct)
       (auto simp add: subringE(3,5-7)[OF subfieldE(1)[OF assms(2)]])
```

```
qed

lemma (in field) generate_fieldI:
  assumes "H ⊆ carrier R"
    and "subfield E R" "H ⊆ E"
    and "⋀K. ⟦ subfield K R; H ⊆ K ⟧ ⟹ E ⊆ K"
  shows "E = generate_field R H"
proof
  show "E ⊆ generate_field R H"
    using assms generate_field_is_subfield generate_field.incl by (metis
subset_iff)
  show "generate_field R H ⊆ E"
    using generate_field_min_subfield1[OF assms(1-3)] by simp
qed

lemma (in field) generate_fieldE:
  assumes "H ⊆ carrier R" and "E = generate_field R H"
  shows "subfield E R" and "H ⊆ E" and "⋀K. ⟦ subfield K R; H ⊆ K ⟧
⟹ E ⊆ K"
proof -
  show "subfield E R" using assms generate_field_is_subfield by simp
  show "H ⊆ E" using assms(2) by (simp add: generate_field.incl subsetI)
  show "⋀K. subfield K R  ⟹ H ⊆ K ⟹ E ⊆ K"
    using assms generate_field_min_subfield1 by auto
qed

lemma (in field) generate_field_min_subfield2:
  assumes "H ⊆ carrier R"
  shows "generate_field R H = ⋂{K. subfield K R ∧ H ⊆ K}"
proof
  have "subfield (generate_field R H) R ∧ H ⊆ generate_field R H"
    by (simp add: assms generate_fieldE(2) generate_field_is_subfield)
  thus "⋂{K. subfield K R ∧ H ⊆ K} ⊆ generate_field R H" by blast
next
  have "⋀K. subfield K R ∧ H ⊆ K ⟹ generate_field R H ⊆ K"
    by (simp add: assms generate_field_min_subfield1)
  thus "generate_field R H ⊆ ⋂{K. subfield K R ∧ H ⊆ K}" by blast
qed

lemma (in field) mono_generate_field:
  assumes "I ⊆ J" and "J ⊆ carrier R"
  shows "generate_field R I ⊆ generate_field R J"
proof-
  have "I ⊆ generate_field R J "
    using assms generate_fieldE(2) by blast
  thus "generate_field R I ⊆ generate_field R J"
    using generate_field_min_subfield1[of I "generate_field R J"] assms
generate_field_is_subfield[OF assms(2)]
    by blast
```

**qed**

```
lemma (in field) subfield_gen_incl :
  assumes "subfield H R"
    and  "subfield K R"
    and "I ⊆ H"
    and "I ⊆ K"
  shows "generate_field (R⦇carrier := K⦈) I ⊆ generate_field (R⦇carrier
:= H⦈) I"
proof
  have incl_HK: "generate_field (R ⦇ carrier := J ⦈) I ⊆ J"
    if J_def : "subfield J R" "I ⊆ J" for J
    using field.mono_generate_field[of "(R⦇carrier := J⦈)" I J] subfield_iff(2)[OF
J_def(1)]
      field.generate_field_in_carrier[of "R⦇carrier := J⦈"]  field_axioms
J_def
    by auto

  fix x
  have "x ∈ generate_field (R⦇carrier := K⦈) I ⟹ x ∈ generate_field
(R⦇carrier := H⦈) I"
  proof (induction  rule : generate_field.induct)
    case one
    have "𝟭R⦇carrier := H⦈ ⊗ 𝟭R⦇carrier := K⦈ = 𝟭R⦇carrier := H⦈" by simp
    moreover have "𝟭R⦇carrier := H⦈ ⊗ 𝟭R⦇carrier := K⦈ = 𝟭R⦇carrier := K⦈"
by simp
    ultimately show ?case using assms generate_field.one by metis
  next
    case (incl h)
    thus ?case using generate_field.incl by force
  next
    case (a_inv h)
    have "a_inv (R⦇carrier := K⦈) h = a_inv R h"
      using assms group.m_inv_consistent[of "add_monoid R" K] a_comm_group
incl_HK[of K] a_inv
        subring.axioms(1)[OF subfieldE(1)[OF assms(2)]]
      unfolding comm_group_def a_inv_def by auto
    moreover have "a_inv (R⦇carrier := H⦈) h = a_inv R h"
      using assms group.m_inv_consistent[of "add_monoid R" H] a_comm_group
incl_HK[of H] a_inv
        subring.axioms(1)[OF subfieldE(1)[OF assms(1)]]
      unfolding  comm_group_def a_inv_def by auto
    ultimately show ?case using generate_field.a_inv a_inv.IH by fastforce
  next
    case (m_inv h)
    have h_K : "h ∈ (K - {𝟬})" using incl_HK[OF assms(2) assms(4)] m_inv
by auto
    hence "m_inv (R⦇carrier := K⦈) h = m_inv R h"
```

```
      using  field.m_inv_mult_of[OF subfield_iff(2)[OF assms(2)]]
        group.m_inv_consistent[of "mult_of R" "K - {0}"] field_mult_group
units_of_inv
        subgroup_mult_of subfieldE[OF assms(2)] unfolding mult_of_def
apply simp
      by (metis h_K mult_of_def mult_of_is_Units subgroup.mem_carrier
units_of_carrier assms(2))
    moreover have h_H : "h ∈ (H - {0})" using incl_HK[OF assms(1) assms(3)]
m_inv by auto
    hence "m_inv (R(|carrier := H|)) h = m_inv R h"
      using  field.m_inv_mult_of[OF subfield_iff(2)[OF assms(1)]]
        group.m_inv_consistent[of "mult_of R" "H - {0}"] field_mult_group

        subgroup_mult_of[OF assms(1)]  unfolding mult_of_def apply simp
      by (metis h_H field_Units m_inv_mult_of mult_of_is_Units subgroup.mem_carrier
units_of_def)
    ultimately show ?case using generate_field.m_inv m_inv.IH h_H by
fastforce
  next
    case (eng_add h1 h2)
    thus ?case using incl_HK assms generate_field.eng_add by force
  next
    case (eng_mult h1 h2)
    thus ?case using generate_field.eng_mult by force
  qed
  thus "x ∈ generate_field (R(|carrier := K|)) I ⟹ x ∈ generate_field
(R(|carrier := H|)) I"
    by auto
qed

lemma (in field) subfield_gen_equality:
  assumes "subfield H R" "K ⊆ H"
  shows "generate_field R K = generate_field (R (| carrier := H |)) K"
  using subfield_gen_incl[OF assms(1) carrier_is_subfield assms(2)] assms
subringE(1)
        subfield_gen_incl[OF carrier_is_subfield assms(1) _ assms(2)]
subfieldE(1)[OF assms(1)]
  by force

end
```

# 43 Product and Sum Groups

```
theory Product_Groups
  imports Elementary_Groups "HOL-Library.Equipollence"

begin
```

## 43.1 Product of a Family of Groups

```
definition product_group:: "'a set ⇒ ('a ⇒ ('b, 'c) monoid_scheme) ⇒
('a ⇒ 'b) monoid"
  where "product_group I G ≡ ⦇carrier = (Π_E i∈I. carrier (G i)),
                              monoid.mult = (λx y. (λi∈I. x i ⊗_G i y
i)),
                              one = (λi∈I. 1_G i)⦈"

lemma carrier_product_group [simp]: "carrier(product_group I G) = (Π_E
i∈I. carrier (G i))"
  by (simp add: product_group_def)

lemma one_product_group [simp]: "one(product_group I G) = (λi∈I. one
(G i))"
  by (simp add: product_group_def)

lemma mult_product_group [simp]: "(⊗_product_group I G) = (λx y. λi∈I.
x i ⊗_G i y i)"
  by (simp add: product_group_def)

lemma product_group [simp]:
  assumes "⋀i. i ∈ I ⟹ group (G i)" shows "group (product_group I
G)"
proof (rule groupI; simp)
  show "(λi. x i ⊗_G i y i) ∈ (Π i∈I. carrier (G i))"
    if "x ∈ (Π_E i∈I. carrier (G i))" "y ∈ (Π_E i∈I. carrier (G i))" for
x y
    using that assms group.subgroup_self subgroup.m_closed by fastforce
  show "(λi. 1_G i) ∈ (Π i∈I. carrier (G i))"
    by (simp add: assms group.is_monoid)
  show "(λi∈I. (if i ∈ I then x i ⊗_G i y i else undefined) ⊗_G i z i)
=
        (λi∈I. x i ⊗_G i (if i ∈ I then y i ⊗_G i z i else undefined))"
    if "x ∈ (Π_E i∈I. carrier (G i))" "y ∈ (Π_E i∈I. carrier (G i))" "z
∈ (Π_E i∈I. carrier (G i))" for x y z
    using that  by (auto simp: PiE_iff assms group.is_monoid monoid.m_assoc
intro: restrict_ext)
  show "(λi∈I. (if i ∈ I then 1_G i else undefined) ⊗_G i x i) = x"
    if "x ∈ (Π_E i∈I. carrier (G i))" for x
    using assms that by (fastforce simp: Group.group_def PiE_iff)
  show "∃y∈Π_E i∈I. carrier (G i). (λi∈I. y i ⊗_G i x i) = (λi∈I. 1_G i)"
    if "x ∈ (Π_E i∈I. carrier (G i))" for x
    by (rule_tac x="λi∈I. inv_G i x i" in bexI) (use assms that in ‹auto
simp: PiE_iff group.l_inv›)
qed

lemma inv_product_group [simp]:
  assumes "f ∈ (Π_E i∈I. carrier (G i))" "⋀i. i ∈ I ⟹ group (G i)"
  shows "inv_product_group I G f = (λi∈I. inv_G i f i)"
```

```
proof (rule group.inv_equality)
  show "Group.group (product_group I G)"
    by (simp add: assms)
  show "(λi∈I. inv_G i f i) ⊗_product_group I G f = 1_product_group I G"
    using assms by (auto simp: PiE_iff group.l_inv)
  show "f ∈ carrier (product_group I G)"
    using assms by simp
  show "(λi∈I. inv_G i f i) ∈ carrier (product_group I G)"
    using PiE_mem assms by fastforce
qed


lemma trivial_product_group: "trivial_group(product_group I G) ⟷ (∀i
∈ I. trivial_group(G i))"
 (is "?lhs = ?rhs")
proof
  assume L: ?lhs
  then have "inv_product_group I G (λa∈I. 1_G a) = 1_product_group I G"
    by (metis group.is_monoid monoid.inv_one one_product_group trivial_group_def)
  have [simp]: "1_G i ⊗_G i 1_G i = 1_G i" if "i ∈ I" for i
    unfolding trivial_group_def
  proof -
    have 1: "(λa∈I. 1_G a) i = 1_G i"
      by (simp add: that)
    have "(λa∈I. 1_G a) = (λa∈I. 1_G a) ⊗_product_group I G (λa∈I. 1_G a)"
      by (metis (no_types) L group.is_monoid monoid.l_one one_product_group
singletonI trivial_group_def)
    then show ?thesis
      using 1 by (simp add: that)
  qed
  show ?rhs
    using L
    by (auto simp: trivial_group_def product_group_def PiE_eq_singleton
intro: groupI)
next
  assume ?rhs
  then show ?lhs
    by (simp add: PiE_eq_singleton trivial_group_def)
qed


lemma PiE_subgroup_product_group:
  assumes "⋀i. i ∈ I ⟹ group (G i)"
  shows "subgroup (PiE I H) (product_group I G) ⟷ (∀i ∈ I. subgroup
(H i) (G i))"
    (is "?lhs = ?rhs")
proof
  assume L: ?lhs
  then have [simp]: "PiE I H ≠ {}"
```

```
    using subgroup_nonempty by force
  show ?rhs
  proof (clarify; unfold_locales)
    show sub: "H i ⊆ carrier (G i)" if "i ∈ I" for i
      using that L by (simp add: subgroup_def) (metis (no_types, lifting)
L subgroup_nonempty subset_PiE)
    show "x ⊗G i y ∈ H i" if "i ∈ I" "x ∈ H i" "y ∈ H i" for i x y
    proof -
      have *: "⋀x. x ∈ PiE I H ⟹ (∀y ∈ PiE I H. ∀i∈I. x i ⊗G i y
i ∈ H i)"
        using L by (auto simp: subgroup_def Pi_iff)
      have "∀y∈H i. f i ⊗G i y ∈ H i" if f: "f ∈ PiE I H" and "i ∈ I"
for i f
        using * [OF f] ‹i ∈ I›
        by (subst(asm) all_PiE_elements) auto
      then have "∀f ∈ PiE I H. ∀i ∈ I. ∀y∈H i. f i ⊗G i y ∈ H i"
        by blast
      with that show ?thesis
        by (subst(asm) all_PiE_elements) auto
    qed
    show "1G i ∈ H i" if "i ∈ I" for i
      using L subgroup.one_closed that by fastforce
    show "invG i x ∈ H i" if "i ∈ I" and x: "x ∈ H i" for i x
    proof -
      have *: "∀y ∈ PiE I H. ∀i∈I. invG i y i ∈ H i"
      proof
        fix y
        assume y: "y ∈ PiE I H"
        then have yc: "y ∈ carrier (product_group I G)"
          by (metis (no_types) L subgroup_def subsetCE)
        have "invproduct_group I G y ∈ PiE I H"
          by (simp add: y L subgroup.m_inv_closed)
        moreover have "invproduct_group I G y = (λi∈I. invG i y i)"
          using yc by (simp add: assms)
        ultimately show "∀i∈I. invG i y i ∈ H i"
          by auto
      qed
      then have "∀i∈I. ∀x∈H i. invG i x ∈ H i"
        by (subst(asm) all_PiE_elements) auto
      then show ?thesis
        using that(1) x by blast
    qed
  qed
next
  assume R: ?rhs
  show ?lhs
  proof
    show "PiE I H ⊆ carrier (product_group I G)"
      using R by (force simp: subgroup_def)
```

```
    show "x ⊗product_group I G y ∈ PiE I H" if "x ∈ PiE I H" "y ∈ PiE
I H" for x y
      using R that by (auto simp: PiE_iff subgroup_def)
    show "1product_group I G ∈ PiE I H"
      using R by (force simp: subgroup_def)
    show "invproduct_group I G x ∈ PiE I H" if "x ∈ PiE I H" for x
    proof -
      have x: "x ∈ (ΠE i∈I. carrier (G i))"
        using R that by (force simp:  subgroup_def)
      show ?thesis
        using assms R that by (fastforce simp: x assms subgroup_def)
    qed
  qed
qed

lemma product_group_subgroup_generated:
  assumes "⋀i. i ∈ I ⟹ subgroup (H i) (G i)" and gp: "⋀i. i ∈ I ⟹
group (G i)"
  shows "product_group I (λi. subgroup_generated (G i) (H i))
        = subgroup_generated (product_group I G) (PiE I H)"
proof (rule monoid.equality)
  have [simp]: "⋀i. i ∈ I ⟹ carrier (G i) ∩ H i = H i" "(ΠE i∈I. carrier
(G i)) ∩ PiE I H = PiE I H"
    using assms by (force simp: subgroup_def)+
  have "(ΠE i∈I. generate (G i) (H i)) = generate (product_group I G)
(PiE I H)"
  proof (rule group.generateI)
    show "Group.group (product_group I G)"
      using assms by simp
    show "subgroup (ΠE i∈I. generate (G i) (H i)) (product_group I G)"
      using assms by (simp add: PiE_subgroup_product_group group.generate_is_subgroup
subgroup.subset)
    show "PiE I H ⊆ (ΠE i∈I. generate (G i) (H i))"
      using assms by (auto simp: PiE_iff generate.incl)
    show "(ΠE i∈I. generate (G i) (H i)) ⊆ K"
      if "subgroup K (product_group I G)" "PiE I H ⊆ K" for K
      using assms that group.generate_subgroup_incl by fastforce
  qed
  with assms
  show "carrier (product_group I (λi. subgroup_generated (G i) (H i)))
=
        carrier (subgroup_generated (product_group I G) (PiE I H))"
    by (simp add: carrier_subgroup_generated cong: PiE_cong)
qed auto

lemma finite_product_group:
  assumes "⋀i. i ∈ I ⟹ group (G i)"
  shows
   "finite (carrier (product_group I G)) ⟷
```

```
    finite {i. i ∈ I ∧ ~ trivial_group(G i)} ∧ (∀i ∈ I. finite(carrier(G
i)))"
proof -
  have [simp]: "⋀i. i ∈ I ⟹ carrier (G i) ≠ {}"
    using assms group.is_monoid by blast
  show ?thesis
    by (auto simp: finite_PiE_iff PiE_eq_empty_iff group.trivial_group_alt
[OF assms] cong: Collect_cong conj_cong)
qed
```

## 43.2 Sum of a Family of Groups

```
definition sum_group :: "'a set ⇒ ('a ⇒ ('b, 'c) monoid_scheme) ⇒ ('a
⇒ 'b) monoid"
  where "sum_group I G ≡
        subgroup_generated
         (product_group I G)
         {x ∈ Π_E i∈I. carrier (G i). finite {i ∈ I. x i ≠ 1_G i}}"

lemma subgroup_sum_group:
  assumes "⋀i. i ∈ I ⟹ group (G i)"
  shows "subgroup {x ∈ Π_E i∈I. carrier (G i). finite {i ∈ I. x i ≠ 1_G i}}
                 (product_group I G)"
proof unfold_locales
  fix x y
  have *: "{i. (i ∈ I ⟶ x i ⊗_G i y i ≠ 1_G i) ∧ i ∈ I}
        ⊆ {i ∈ I. x i ≠ 1_G i} ∪ {i ∈ I. y i ≠ 1_G i}"
    by (auto simp: Group.group_def dest: assms)
  assume
    "x ∈ {x ∈ Π_E i∈I. carrier (G i). finite {i ∈ I. x i ≠ 1_G i}}"
    "y ∈ {x ∈ Π_E i∈I. carrier (G i). finite {i ∈ I. x i ≠ 1_G i}}"
  then
  show "x ⊗_product_group I G y ∈ {x ∈ Π_E i∈I. carrier (G i). finite {i
∈ I. x i ≠ 1_G i}}"
    using assms
    apply (auto simp: Group.group_def monoid.m_closed PiE_iff)
    apply (rule finite_subset [OF *])
    by blast
next
  fix x
  assume "x ∈ {x ∈ Π_E i∈I. carrier (G i). finite {i ∈ I. x i ≠ 1_G i}}"
  then show "inv_product_group I G x ∈ {x ∈ Π_E i∈I. carrier (G i). finite
{i ∈ I. x i ≠ 1_G i}}"
    using assms
    by (auto simp: PiE_iff assms group.inv_eq_1_iff [OF assms] conj_commute
cong: rev_conj_cong)
qed (use assms [unfolded Group.group_def] in auto)

lemma carrier_sum_group:
```

```
  assumes "⋀i. i ∈ I ⟹ group (G i)"
  shows "carrier(sum_group I G) = {x ∈ Π_E i∈I. carrier (G i). finite
{i ∈ I. x i ≠ 1_G i}}"
proof -
  interpret SG: subgroup "{x ∈ Π_E i∈I. carrier (G i). finite {i ∈ I.
x i ≠ 1_G i}}" "(product_group I G)"
    by (simp add: assms subgroup_sum_group)
  show ?thesis
    by (simp add: sum_group_def subgroup_sum_group carrier_subgroup_generated_alt)
qed

lemma one_sum_group [simp]: "1_sum_group I G = (λi∈I. 1_G i)"
  by (simp add: sum_group_def)

lemma mult_sum_group [simp]: "(⊗_sum_group I G) = (λx y. (λi∈I. x i ⊗_G i
y i))"
  by (auto simp: sum_group_def)

lemma sum_group [simp]:
  assumes "⋀i. i ∈ I ⟹ group (G i)" shows "group (sum_group I G)"
proof (rule groupI)
  note group.is_monoid [OF assms, simp]
  show "x ⊗_sum_group I G y ∈ carrier (sum_group I G)"
    if "x ∈ carrier (sum_group I G)" and
      "y ∈ carrier (sum_group I G)" for x y
  proof -
    have *: "{i ∈ I. x i ⊗_G i y i ≠ 1_G i} ⊆ {i ∈ I. x i ≠ 1_G i} ∪ {i
∈ I. y i ≠ 1_G i}"
      by auto
    show ?thesis
      using that
      apply (simp add: assms carrier_sum_group PiE_iff monoid.m_closed
conj_commute cong: rev_conj_cong)
      apply (blast intro: finite_subset [OF *])
      done
  qed
  show "1_sum_group I G ⊗_sum_group I G x = x"
    if "x ∈ carrier (sum_group I G)" for x
    using that by (auto simp: assms carrier_sum_group PiE_iff extensional_def)
  show "∃y∈carrier (sum_group I G). y ⊗_sum_group I G x = 1_sum_group I G"
    if "x ∈ carrier (sum_group I G)" for x
  proof
    let ?y = "λi∈I. m_inv (G i) (x i)"
    show "?y ⊗_sum_group I G x = 1_sum_group I G"
      using that assms
      by (auto simp: carrier_sum_group PiE_iff group.l_inv)
    show "?y ∈ carrier (sum_group I G)"
      using that assms
      by (auto simp: carrier_sum_group PiE_iff group.inv_eq_1_iff group.l_inv
```

```
cong: conj_cong)
  qed
qed (auto simp: assms carrier_sum_group PiE_iff group.is_monoid monoid.m_assoc)

lemma inv_sum_group [simp]:
  assumes "⋀i. i ∈ I ⟹ group (G i)" and x: "x ∈ carrier (sum_group
I G)"
  shows "m_inv (sum_group I G) x = (λi∈I. m_inv (G i) (x i))"
proof (rule group.inv_equality)
  show "(λi∈I. inv_G i x i) ⊗_sum_group I G x = 1_sum_group I G"
    using x by (auto simp: carrier_sum_group PiE_iff group.l_inv assms
intro: restrict_ext)
  show "(λi∈I. inv_G i x i) ∈ carrier (sum_group I G)"
    using x by (simp add: carrier_sum_group PiE_iff group.inv_eq_1_iff
assms conj_commute cong: rev_conj_cong)
qed (auto simp: assms)


thm group.subgroups_Inter
theorem subgroup_Inter:
  assumes subgr: "(⋀H. H ∈ A ⟹ subgroup H G)"
    and not_empty: "A ≠ {}"
  shows "subgroup (⋂A) G"
proof
  show "⋂ A ⊆ carrier G"
    by (simp add: Inf_less_eq not_empty subgr subgroup.subset)
qed (auto simp: subgr subgroup.m_closed subgroup.one_closed subgroup.m_inv_closed)

thm group.subgroups_Inter_pair
lemma subgroup_Int:
  assumes "subgroup I G" "subgroup J G"
  shows "subgroup (I ∩ J) G" using subgroup_Inter[ where ?A = "{I,J}"]
assms by auto


lemma sum_group_subgroup_generated:
  assumes "⋀i. i ∈ I ⟹ group (G i)" and sg: "⋀i. i ∈ I ⟹ subgroup
(H i) (G i)"
  shows "sum_group I (λi. subgroup_generated (G i) (H i)) = subgroup_generated
(sum_group I G) (PiE I H)"
proof (rule monoid.equality)
  have "subgroup (carrier (sum_group I G) ∩ Pi_E I H) (product_group I
G)"
    by (rule subgroup_Int) (auto simp: assms carrier_sum_group subgroup_sum_group
PiE_subgroup_product_group)
  moreover have "carrier (sum_group I G) ∩ Pi_E I H
              ⊆ carrier (subgroup_generated (product_group I G)
                    {x ∈ Π_E i∈I. carrier (G i). finite {i ∈ I. x i ≠
1_G i}})"
```

```
    by (simp add: assms subgroup_sum_group subgroup.carrier_subgroup_generated_subgroup
carrier_sum_group)
  ultimately
  have "subgroup (carrier (sum_group I G) ∩ Pi_E I H) (sum_group I G)"
    by (simp add: assms sum_group_def group.subgroup_subgroup_generated_iff)
  then have *: "{f ∈ Π_E i∈I. carrier (subgroup_generated (G i) (H i)).
finite {i ∈ I. f i ≠ 1_G i}}
      = carrier (subgroup_generated (sum_group I G) (carrier (sum_group
I G) ∩ Pi_E I H))"
    apply (simp only: subgroup.carrier_subgroup_generated_subgroup)
    using subgroup.subset [OF sg]
    apply (auto simp: set_eq_iff PiE_def Pi_def assms carrier_sum_group
subgroup.carrier_subgroup_generated_subgroup)
    done
  then show "carrier (sum_group I (λi. subgroup_generated (G i) (H i)))
=
        carrier (subgroup_generated (sum_group I G) (Pi_E I H))"
    by simp (simp add: assms group.subgroupE(1) group.group_subgroup_generated
carrier_sum_group)
qed (auto simp: sum_group_def subgroup_generated_def)


lemma iso_product_groupI:
  assumes iso: "⋀i. i ∈ I ⟹ G i ≅ H i"
    and G: "⋀i. i ∈ I ⟹ group (G i)" and H: "⋀i. i ∈ I ⟹ group
(H i)"
  shows "product_group I G ≅ product_group I H" (is "?IG ≅ ?IH")
proof -
  have "⋀i. i ∈ I ⟹ ∃h. h ∈ iso (G i) (H i)"
    using iso by (auto simp: is_iso_def)
  then obtain f where f: "⋀i. i ∈ I ⟹ f i ∈ iso (G i) (H i)"
    by metis
  define h where "h ≡ λx. (λi∈I. f i (x i))"
  have hom: "h ∈ iso ?IG ?IH"
  proof (rule isoI)
    show hom: "h ∈ hom ?IG ?IH"
    proof (rule homI)
      fix x
      assume "x ∈ carrier ?IG"
      with f show "h x ∈ carrier ?IH"
        using PiE by (fastforce simp add: h_def PiE_def iso_def hom_def)
    next
      fix x y
      assume "x ∈ carrier ?IG" "y ∈ carrier ?IG"
      with f show "h (x ⊗_?IG y) = h x ⊗_?IH h y"
        apply (simp add: h_def PiE_def iso_def hom_def)
        using PiE by (fastforce simp add: h_def PiE_def iso_def hom_def
intro: restrict_ext)
    qed
```

```
    with G H interpret GH : group_hom "?IG" "?IH" h
      by (simp add: group_hom_def group_hom_axioms_def)
    show "bij_betw h (carrier ?IG) (carrier ?IH)"
      unfolding bij_betw_def
    proof (intro conjI subset_antisym)
      have "γ i = 1G i"
        if γ: "γ ∈ (ΠE i∈I. carrier (G i))" and eq: "(λi∈I. f i (γ i))
= (λi∈I. 1H i)" and "i ∈ I"
        for γ i
      proof -
        have "inj_on (f i) (carrier (G i))" "f i ∈ hom (G i) (H i)"
          using ‹i ∈ I› f by (auto simp: iso_def bij_betw_def)
        then have *: "⋀x. ⟦f i x = 1H i; x ∈ carrier (G i)⟧ ⟹ x = 1G i"
          by (metis G Group.group_def H hom_one inj_onD monoid.one_closed
‹i ∈ I›)
        show ?thesis
          using eq ‹i ∈ I› * γ by (simp add: fun_eq_iff) (meson PiE_iff)
      qed
      then show "inj_on h (carrier ?IG)"
        apply (simp add: iso_def bij_betw_def GH.inj_on_one_iff flip:
carrier_product_group)
        apply (force simp: h_def)
        done
    next
      show "h ‘ carrier ?IG ⊆ carrier ?IH"
        unfolding h_def using f
        by (force simp: PiE_def Pi_def Group.iso_def dest!: bij_betwE)
    next
      show "carrier ?IH ⊆ h ‘ carrier ?IG"
        unfolding h_def
      proof (clarsimp simp: iso_def bij_betw_def)
        fix x
        assume "x ∈ (ΠE i∈I. carrier (H i))"
        with f have x: "x ∈ (ΠE i∈I. f i ‘ carrier (G i))"
          unfolding h_def by (auto simp: iso_def bij_betw_def)
        have "⋀i. i ∈ I ⟹ inj_on (f i) (carrier (G i))"
          using f by (auto simp: iso_def bij_betw_def)
        let ?g = "λi∈I. inv_into (carrier (G i)) (f i) (x i)"
        show "x ∈ (λg. λi∈I. f i (g i)) ‘ (ΠE i∈I. carrier (G i))"
        proof
          show "x = (λi∈I. f i (?g i))"
            using x by (auto simp: PiE_iff fun_eq_iff extensional_def
f_inv_into_f)
          show "?g ∈ (ΠE i∈I. carrier (G i))"
            using x by (auto simp: PiE_iff inv_into_into)
        qed
      qed
    qed
  qed
```

```
  then show ?thesis
    using is_iso_def by auto
qed

lemma iso_sum_groupI:
  assumes iso: "⋀i. i ∈ I ⟹ G i ≅ H i"
    and G: "⋀i. i ∈ I ⟹ group (G i)" and H: "⋀i. i ∈ I ⟹ group
(H i)"
  shows "sum_group I G ≅ sum_group I H" (is "?IG ≅ ?IH")
proof -
  have "⋀i. i ∈ I ⟹ ∃h. h ∈ iso (G i) (H i)"
    using iso by (auto simp: is_iso_def)
  then obtain f where f: "⋀i. i ∈ I ⟹ f i ∈ iso (G i) (H i)"
    by metis
  then have injf: "inj_on (f i) (carrier (G i))"
    and homf: "f i ∈ hom (G i) (H i)" if "i ∈ I" for i
    using ‹i ∈ I› f by (auto simp: iso_def bij_betw_def)
  then have one: "⋀x. ⟦f i x = 1H i; x ∈ carrier (G i)⟧ ⟹ x = 1G i"
if "i ∈ I" for i
    by (metis G H group.subgroup_self hom_one inj_on_eq_iff subgroup.one_closed
that)
  have fin1: "finite {i ∈ I. x i ≠ 1G i} ⟹ finite {i ∈ I. f i (x i)
≠ 1H i}" for x
    using homf by (auto simp: G H hom_one elim!: rev_finite_subset)
  define h where "h ≡ λx. (λi∈I. f i (x i))"
  have hom: "h ∈ iso ?IG ?IH"
  proof (rule isoI)
    show hom: "h ∈ hom ?IG ?IH"
    proof (rule homI)
      fix x
      assume "x ∈ carrier ?IG"
      with f fin1 show "h x ∈ carrier ?IH"
        by (force simp: h_def PiE_def iso_def hom_def carrier_sum_group
assms conj_commute cong: conj_cong)
    next
      fix x y
      assume "x ∈ carrier ?IG" "y ∈ carrier ?IG"
      with homf show "h (x ⊗?IG y) = h x ⊗?IH h y"
        by (fastforce simp add: h_def PiE_def hom_def carrier_sum_group
assms intro: restrict_ext)
    qed
    with G H interpret GH : group_hom "?IG" "?IH" h
      by (simp add: group_hom_def group_hom_axioms_def)
    show "bij_betw h (carrier ?IG) (carrier ?IH)"
      unfolding bij_betw_def
    proof (intro conjI subset_antisym)
      have γ: "γ i = 1G i"
        if "γ ∈ (ΠE i∈I. carrier (G i))" and eq: "(λi∈I. f i (γ i))
= (λi∈I. 1H i)" and "i ∈ I"
```

```
        for γ i
        using ‹i ∈ I› one that by (simp add: fun_eq_iff) (meson PiE_iff)
      show "inj_on h (carrier ?IG)"
        apply (simp add: iso_def bij_betw_def GH.inj_on_one_iff assms
one flip: carrier_sum_group)
        apply (auto simp: h_def fun_eq_iff carrier_sum_group assms PiE_def
Pi_def extensional_def one)
        done
    next
      show "h ‘ carrier ?IG ⊆ carrier ?IH"
        using homf GH.hom_closed
        by (fastforce simp: h_def PiE_def Pi_def dest!: bij_betwE)
    next
      show "carrier ?IH ⊆ h ‘ carrier ?IG"
        unfolding h_def
      proof (clarsimp simp: iso_def bij_betw_def carrier_sum_group assms)
        fix x
        assume x: "x ∈ (Π_E i∈I. carrier (H i))" and fin: "finite {i
∈ I. x i ≠ 1_H i}"
        with f have xf: "x ∈ (Π_E i∈I. f i ‘ carrier (G i))"
          unfolding h_def
          by (auto simp: iso_def bij_betw_def)
        have "⋀i. i ∈ I ⟹ inj_on (f i) (carrier (G i))"
          using f by (auto simp: iso_def bij_betw_def)
        let ?g = "λi∈I. inv_into (carrier (G i)) (f i) (x i)"
        show "x ∈ (λg. λi∈I. f i (g i))
                ‘ {x ∈ Π_E i∈I. carrier (G i). finite {i ∈ I. x i ≠
1_G i}}"
        proof
          show xeq: "x = (λi∈I. f i (?g i))"
            using x by (clarsimp simp: PiE_iff fun_eq_iff extensional_def)
(metis iso_iff f_inv_into_f f)
          have "finite {i ∈ I. inv_into (carrier (G i)) (f i) (x i) ≠
1_G i}"
            apply (rule finite_subset [OF _ fin])
            using G H group.subgroup_self hom_one homf injf inv_into_f_eq
subgroup.one_closed by fastforce
          with x show "?g ∈ {x ∈ Π_E i∈I. carrier (G i). finite {i ∈
I. x i ≠ 1_G i}}"
            apply (auto simp: PiE_iff inv_into_into conj_commute cong:
conj_cong)
            by (metis (no_types, opaque_lifting) iso_iff f inv_into_into)
        qed
      qed
    qed
  qed
  then show ?thesis
    using is_iso_def by auto
qed
```

**end**

# 44 Free Abelian Groups

```
theory Free_Abelian_Groups
  imports
    Product_Groups FiniteProduct "HOL-Cardinals.Cardinal_Arithmetic"
   "HOL-Library.Countable_Set" "HOL-Library.Poly_Mapping" "HOL-Library.Equipollence"

begin


lemma eqpoll_Fpow:
  assumes "infinite A" shows "Fpow A ≈ A"
  unfolding eqpoll_iff_card_of_ordIso
  by (metis assms card_of_Fpow_infinite)

lemma infinite_iff_card_of_countable: "⟦countable B; infinite B⟧ ⟹
infinite A ⟷ ( |B| ≤o |A| )"
  unfolding infinite_iff_countable_subset card_of_ordLeq countable_def
  by (force intro: card_of_ordLeqI ordLeq_transitive)

lemma iso_imp_eqpoll_carrier: "G ≅ H ⟹ carrier G ≈ carrier H"
  by (auto simp: is_iso_def iso_def eqpoll_def)
```

## 44.1 Generalised finite product

```
definition
  gfinprod :: "[('b, 'm) monoid_scheme, 'a ⇒ 'b, 'a set] ⇒ 'b"
  where "gfinprod G f A =
   (if finite {x ∈ A. f x ≠ 1G} then finprod G f {x ∈ A. f x ≠ 1G} else
1G)"

context comm_monoid begin

lemma gfinprod_closed [simp]:
  "f ∈ A → carrier G ⟹ gfinprod G f A ∈ carrier G"
  unfolding gfinprod_def
  by (auto simp: image_subset_iff_funcset intro: finprod_closed)

lemma gfinprod_cong:
  "⟦A = B; f ∈ B → carrier G;
    ⋀i. i ∈ B =simp=> f i = g i⟧ ⟹ gfinprod G f A = gfinprod G g B"
  unfolding gfinprod_def
  by (auto simp: simp_implies_def cong: conj_cong intro: finprod_cong)

lemma gfinprod_eq_finprod [simp]: "⟦finite A; f ∈ A → carrier G⟧ ⟹
gfinprod G f A = finprod G f A"
```

```
  by (auto simp: gfinprod_def intro: finprod_mono_neutral_cong_left)

lemma gfinprod_insert [simp]:
  assumes "finite {x ∈ A. f x ≠ 1G}" "f ∈ A → carrier G" "f i ∈ carrier
G"
  shows "gfinprod G f (insert i A) = (if i ∈ A then gfinprod G f A else
f i ⊗ gfinprod G f A)"
proof -
  have f: "f ∈ {x ∈ A. f x ≠ 1} → carrier G"
    using assms by (auto simp: image_subset_iff_funcset)
  have "{x. x = i ∧ f x ≠ 1 ∨ x ∈ A ∧ f x ≠ 1} = (if f i = 1 then {x
∈ A. f x ≠ 1} else insert i {x ∈ A. f x ≠ 1})"
    by auto
  then show ?thesis
    using assms
    unfolding gfinprod_def by (simp add: conj_disj_distribR insert_absorb
f split: if_split_asm)
qed

lemma gfinprod_distrib:
  assumes fin: "finite {x ∈ A. f x ≠ 1G}" "finite {x ∈ A. g x ≠ 1G}"
     and "f ∈ A → carrier G" "g ∈ A → carrier G"
  shows "gfinprod G (λi. f i ⊗ g i) A = gfinprod G f A ⊗ gfinprod G
g A"
proof -
  have "finite {x ∈ A. f x ⊗ g x ≠ 1}"
    by (auto intro: finite_subset [OF _ finite_UnI [OF fin]])
  then have "gfinprod G (λi. f i ⊗ g i) A = gfinprod G (λi. f i ⊗ g
i) ({i ∈ A. f i ≠ 1G} ∪ {i ∈ A. g i ≠ 1G})"
    unfolding gfinprod_def
    using assms by (force intro: finprod_mono_neutral_cong)
  also have "... = gfinprod G f A ⊗ gfinprod G g A"
  proof -
    have "finprod G f ({i ∈ A. f i ≠ 1G} ∪ {i ∈ A. g i ≠ 1G}) = gfinprod
G f A"
      "finprod G g ({i ∈ A. f i ≠ 1G} ∪ {i ∈ A. g i ≠ 1G}) = gfinprod
G g A"
      using assms by (auto simp: gfinprod_def intro: finprod_mono_neutral_cong_right)
    moreover have "(λi. f i ⊗ g i) ∈ {i ∈ A. f i ≠ 1} ∪ {i ∈ A. g i
≠ 1} → carrier G"
      using assms by (force simp: image_subset_iff_funcset)
    ultimately show ?thesis
      using assms
      apply simp
      apply (subst finprod_multf, auto)
      done
  qed
  finally show ?thesis .
qed
```

```
lemma gfinprod_mono_neutral_cong_left:
  assumes "A ⊆ B"
    and 1: "⋀i. i ∈ B - A ⟹ h i = 1"
    and gh: "⋀x. x ∈ A ⟹ g x = h x"
    and h: "h ∈ B → carrier G"
  shows "gfinprod G g A = gfinprod G h B"
proof (cases "finite {x ∈ B. h x ≠ 1}")
  case True
  then have "finite {x ∈ A. h x ≠ 1}"
    apply (rule rev_finite_subset)
    using ‹A ⊆ B› by auto
  with True assms show ?thesis
    apply (simp add: gfinprod_def cong: conj_cong)
    apply (auto intro!: finprod_mono_neutral_cong_left)
    done
next
  case False
  have "{x ∈ B. h x ≠ 1} ⊆ {x ∈ A. h x ≠ 1}"
    using 1 by auto
  with False have "infinite {x ∈ A. h x ≠ 1}"
    using infinite_super by blast
  with False assms show ?thesis
    by (simp add: gfinprod_def cong: conj_cong)
qed

lemma gfinprod_mono_neutral_cong_right:
  assumes "A ⊆ B" "⋀i. i ∈ B - A ⟹ g i = 1" "⋀x. x ∈ A ⟹ g x =
h x" "g ∈ B → carrier G"
  shows "gfinprod G g B = gfinprod G h A"
  using assms  by (auto intro!: gfinprod_mono_neutral_cong_left [symmetric])

lemma gfinprod_mono_neutral_cong:
  assumes [simp]: "finite B" "finite A"
    and *: "⋀i. i ∈ B - A ⟹ h i = 1" "⋀i. i ∈ A - B ⟹ g i = 1"
    and gh: "⋀x. x ∈ A ∩ B ⟹ g x = h x"
    and g: "g ∈ A → carrier G"
    and h: "h ∈ B → carrier G"
 shows "gfinprod G g A = gfinprod G h B"
proof-
  have "gfinprod G g A = gfinprod G g (A ∩ B)"
    by (rule gfinprod_mono_neutral_cong_right) (use assms in auto)
  also have "... = gfinprod G h (A ∩ B)"
    by (rule gfinprod_cong) (use assms in auto)
  also have "... = gfinprod G h B"
    by (rule gfinprod_mono_neutral_cong_left) (use assms in auto)
  finally show ?thesis .
qed
```

```
end

lemma (in comm_group) hom_group_sum:
  assumes hom: "⋀i. i ∈ I ⟹ f i ∈ hom (A i) G" and grp: "⋀i. i ∈
I ⟹ group (A i)"
  shows "(λx. gfinprod G (λi. (f i) (x i)) I) ∈ hom (sum_group I A) G"
  unfolding hom_def
proof (intro CollectI conjI ballI)
  show "(λx. gfinprod G (λi. f i (x i)) I) ∈ carrier (sum_group I A)
→ carrier G"
    using assms
    by (force simp: hom_def carrier_sum_group intro: gfinprod_closed simp
flip: image_subset_iff_funcset)
next
  fix x y
  assume x: "x ∈ carrier (sum_group I A)" and y: "y ∈ carrier (sum_group
I A)"
  then have finx: "finite {i ∈ I. x i ≠ 𝟭A i}" and finy: "finite {i
∈ I. y i ≠ 𝟭A i}"
    using assms by (auto simp: carrier_sum_group)
  have finfx: "finite {i ∈ I. f i (x i) ≠ 𝟭}"
    using assms by (auto simp: is_group hom_one [OF hom] intro: finite_subset
[OF _ finx])
  have finfy: "finite {i ∈ I. f i (y i) ≠ 𝟭}"
    using assms by (auto simp: is_group hom_one [OF hom] intro: finite_subset
[OF _ finy])
  have carr: "f i (x i) ∈ carrier G"  "f i (y i) ∈ carrier G" if "i ∈
I" for i
    using hom_carrier [OF hom] that x y assms
    by (fastforce simp add: carrier_sum_group)+
  have lam: "(λi. f i ( x i ⊗A i y i)) ∈ I → carrier G"
    using x y assms by (auto simp: hom_def carrier_sum_group PiE_def Pi_def)
  have lam': "(λi. f i (if i ∈ I then x i ⊗A i y i else undefined)) ∈
I → carrier G"
    by (simp add: lam Pi_cong)
  with lam x y assms
  show "gfinprod G (λi. f i ((x ⊗sum_group I A y) i)) I
      = gfinprod G (λi. f i (x i)) I ⊗ gfinprod G (λi. f i (y i)) I"
    by (simp add: carrier_sum_group PiE_def Pi_def hom_mult [OF hom]
                  gfinprod_distrib finfx finfy carr cong: gfinprod_cong)
qed
```

## 44.2 Free Abelian groups on a set, using the "frag" type constructor.

```
definition free_Abelian_group :: "'a set ⇒ ('a ⇒0 int) monoid"
  where "free_Abelian_group S = ⦇carrier = {c. Poly_Mapping.keys c ⊆
S}, monoid.mult = (+), one  = 0⦈"
```

```
lemma group_free_Abelian_group [simp]: "group (free_Abelian_group S)"
proof -
  have "⋀x. Poly_Mapping.keys x ⊆ S ⟹ x ∈ Units (free_Abelian_group
S)"
    unfolding free_Abelian_group_def Units_def
    by clarsimp (metis eq_neg_iff_add_eq_0 neg_eq_iff_add_eq_0 keys_minus)
  then show ?thesis
    unfolding free_Abelian_group_def
    by unfold_locales (auto simp: dest: subsetD [OF keys_add])
qed

lemma carrier_free_Abelian_group_iff [simp]:
  shows "x ∈ carrier (free_Abelian_group S) ⟷ Poly_Mapping.keys x ⊆
S"
  by (auto simp: free_Abelian_group_def)

lemma one_free_Abelian_group [simp]: "𝟭free_Abelian_group S = 0"
  by (auto simp: free_Abelian_group_def)

lemma mult_free_Abelian_group [simp]: "x ⊗free_Abelian_group S y = x +
y"
  by (auto simp: free_Abelian_group_def)

lemma inv_free_Abelian_group [simp]: "Poly_Mapping.keys x ⊆ S ⟹ invfree_Abelian_group S
x = -x"
  by (rule group.inv_equality [OF group_free_Abelian_group]) auto

lemma abelian_free_Abelian_group: "comm_group(free_Abelian_group S)"
  apply (rule group.group_comm_groupI [OF group_free_Abelian_group])
  by (simp add: free_Abelian_group_def)

lemma pow_free_Abelian_group [simp]:
  fixes n::nat
  shows "Group.pow (free_Abelian_group S) x n = frag_cmul (int n) x"
  by (induction n) (auto simp: nat_pow_def free_Abelian_group_def frag_cmul_distrib)

lemma int_pow_free_Abelian_group [simp]:
  fixes n::int
  assumes "Poly_Mapping.keys x ⊆ S"
  shows "Group.pow (free_Abelian_group S) x n = frag_cmul n x"
proof (induction n)
  case (nonneg n)
  then show ?case
    by (simp add: int_pow_int)
next
  case (neg n)
  have "x [^]free_Abelian_group S - int (Suc n)
     = invfree_Abelian_group S (x [^]free_Abelian_group S int (Suc n))"
    by (rule group.int_pow_neg [OF group_free_Abelian_group]) (use assms
```

```
in ‹simp add: free_Abelian_group_def›)
  also have "... = frag_cmul (- int (Suc n)) x"
    by (metis assms inv_free_Abelian_group pow_free_Abelian_group int_pow_int
minus_frag_cmul
              order_trans keys_cmul)
  finally show ?case .
qed

lemma frag_of_in_free_Abelian_group [simp]:
   "frag_of x ∈ carrier(free_Abelian_group S) ⟷ x ∈ S"
  by simp

lemma free_Abelian_group_induct:
  assumes major: "Poly_Mapping.keys x ⊆ S"
      and minor: "P(0)"
           "⋀x y. ⟦Poly_Mapping.keys x ⊆ S; Poly_Mapping.keys y ⊆ S;
P x; P y⟧ ⟹ P(x-y)"
           "⋀a. a ∈ S ⟹ P(frag_of a)"
         shows "P x"
proof -
  have "Poly_Mapping.keys x ⊆ S ∧ P x"
    using major
  proof (induction x rule: frag_induction)
    case (diff a b)
    then show ?case
      by (meson Un_least minor(2) order.trans keys_diff)
  qed (auto intro: minor)
  then show ?thesis ..
qed

lemma sum_closed_free_Abelian_group:
    "(⋀i. i ∈ I ⟹ x i ∈ carrier (free_Abelian_group S)) ⟹ sum x I
∈ carrier (free_Abelian_group S)"
  apply (induction I rule: infinite_finite_induct, auto)
  by (metis (no_types, opaque_lifting) UnE subsetCE keys_add)


lemma (in comm_group) free_Abelian_group_universal:
  fixes f :: "'c ⇒ 'a"
  assumes "f ` S ⊆ carrier G"
  obtains h where "h ∈ hom (free_Abelian_group S) G" "⋀x. x ∈ S ⟹
h(frag_of x) = f x"
proof
  have fin: "Poly_Mapping.keys u ⊆ S ⟹ finite {x ∈ S. f x [^] poly_mapping.lookup
u x ≠ 1}" for u :: "'c ⇒₀ int"
    apply (rule finite_subset [OF _ finite_keys [of u]])
    unfolding keys.rep_eq by force
  define h :: "('c ⇒₀ int) ⇒ 'a"
    where "h ≡ λx. gfinprod G (λa. f a [^] poly_mapping.lookup x a) S"
```

```
  show "h ∈ hom (free_Abelian_group S) G"
  proof (rule homI)
    fix x y
    assume xy: "x ∈ carrier (free_Abelian_group S)" "y ∈ carrier (free_Abelian_group
S)"
    then show "h (x ⊗free_Abelian_group S y) = h x ⊗ h y"
      using assms unfolding h_def free_Abelian_group_def
      by (simp add: fin gfinprod_distrib image_subset_iff Poly_Mapping.lookup_add
int_pow_mult cong: gfinprod_cong)
  qed (use assms in ‹force simp: free_Abelian_group_def h_def intro: gfinprod_closed›)
  show "h(frag_of x) = f x" if "x ∈ S" for x
  proof -
    have fin: "(λa. f x [^] (1::int)) ∈ {x} → carrier G" "f x [^] (1::int)
∈ carrier G"
      using assms that by force+
    show ?thesis
      by (cases " f x [^] (1::int) = 1") (use assms that in ‹auto simp:
h_def gfinprod_def finprod_singleton›)
  qed
qed

lemma eqpoll_free_Abelian_group_infinite:
  assumes "infinite A" shows "carrier(free_Abelian_group A) ≈ A"
proof (rule lepoll_antisym)
  have "carrier (free_Abelian_group A) ≲ {f::'a⇒int. f ‘ A ⊆ UNIV ∧
{x. f x ≠ 0} ⊆ A ∧ finite {x. f x ≠ 0}}"
    unfolding lepoll_def
    by (rule_tac x="Poly_Mapping.lookup" in exI) (auto simp: poly_mapping_eqI
lookup_not_eq_zero_eq_in_keys inj_onI)
  also have "... ≲ Fpow (A × (UNIV::int set))"
    by (rule lepoll_restricted_funspace)
  also have "... ≈ A × (UNIV::int set)"
  proof (rule eqpoll_Fpow)
    show "infinite (A × (UNIV::int set))"
      using assms finite_cartesian_productD1 by fastforce
  qed
  also have "... ≈ A"
    unfolding eqpoll_iff_card_of_ordIso
  proof -
    have "|A × (UNIV::int set)| <=o |A|"
      by (simp add: assms card_of_Times_ordLeq_infinite flip: infinite_iff_card_of_countabl
    moreover have "|A| ≤o |A × (UNIV::int set)|"
      by simp
    ultimately have "|A| *c |(UNIV::int set)| =o |A|"
      by (simp add: cprod_def ordIso_iff_ordLeq)
    then show "|A × (UNIV::int set)| =o |A|"
      by (metis Times_cprod ordIso_transitive)
  qed
  finally show "carrier (free_Abelian_group A) ≲ A" .
```

```
  have "inj_on frag_of A"
    by (simp add: frag_of_eq inj_on_def)
  moreover have "frag_of ` A ⊆ carrier (free_Abelian_group A)"
    by (simp add: image_subsetI)
  ultimately show "A ≲ carrier (free_Abelian_group A)"
    by (force simp: lepoll_def)
qed

proposition (in comm_group) eqpoll_homomorphisms_from_free_Abelian_group:
   "{f. f ∈ extensional (carrier(free_Abelian_group S)) ∧ f ∈ hom (free_Abelian_group
S) G}
    ≈ (S →E carrier G)"  (is "?lhs ≈ ?rhs")
  unfolding eqpoll_def bij_betw_def
proof (intro exI conjI)
  let ?f = "λf. restrict (f ∘ frag_of) S"
  show "inj_on ?f ?lhs"
  proof (clarsimp simp: inj_on_def)
    fix g h
    assume
      g: "g ∈ extensional (carrier (free_Abelian_group S))" "g ∈ hom (free_Abelian_group
S) G"
      and h: "h ∈ extensional (carrier (free_Abelian_group S))" "h ∈
hom (free_Abelian_group S) G"
      and eq: "restrict (g ∘ frag_of) S = restrict (h ∘ frag_of) S"
    have 0: "0 ∈ carrier (free_Abelian_group S)"
      by simp
    interpret hom_g: group_hom "free_Abelian_group S" G g
      using g by (auto simp: group_hom_def group_hom_axioms_def is_group)
    interpret hom_h: group_hom "free_Abelian_group S" G h
      using h by (auto simp: group_hom_def group_hom_axioms_def is_group)
    have "Poly_Mapping.keys c ⊆ S ⟹ Poly_Mapping.keys c ⊆ S ∧ g c
= h c" for c
    proof (induction c rule: frag_induction)
      case zero
      show ?case
        using hom_g.hom_one hom_h.hom_one by auto
    next
      case (one x)
      then show ?case
        using eq by (simp add: fun_eq_iff) (metis comp_def)
    next
      case (diff a b)
      then show ?case
        using hom_g.hom_mult hom_h.hom_mult hom_g.hom_inv hom_h.hom_inv
        apply (auto simp: dest: subsetD [OF keys_diff])
        by (metis keys_minus uminus_add_conv_diff)
    qed
    then show "g = h"
      by (meson g h carrier_free_Abelian_group_iff extensionalityI)
```

```
  qed
  have "f ∈ (λf. restrict (f ∘ frag_of) S) `
           {f ∈ extensional (carrier (free_Abelian_group S)). f ∈ hom
(free_Abelian_group S) G}"
    if f: "f ∈ S →E carrier G"
    for f :: "'c ⇒ 'a"
  proof -
    obtain h where h: "h ∈ hom (free_Abelian_group S) G" "⋀x. x ∈ S
⟹ h(frag_of x) = f x"
    proof (rule free_Abelian_group_universal)
      show "f ` S ⊆ carrier G"
        using f by blast
    qed auto
    let ?h = "restrict h (carrier (free_Abelian_group S))"
    show ?thesis
    proof
      show "f = restrict (?h ∘ frag_of) S"
        using f by (force simp: h)
      show "?h ∈ {f ∈ extensional (carrier (free_Abelian_group S)). f
∈ hom (free_Abelian_group S) G}"
        using h by (auto simp: hom_def dest!: subsetD [OF keys_add])
    qed
  qed
  then show "?f ` ?lhs = S →E carrier G"
    by (auto simp: hom_def Ball_def Pi_def)
qed

lemma hom_frag_minus:
  assumes "h ∈ hom (free_Abelian_group S) (free_Abelian_group T)" "Poly_Mapping.keys
a ⊆ S"
  shows "h (-a) = - (h a)"
proof -
  have "Poly_Mapping.keys (h a) ⊆ T"
    by (meson assms carrier_free_Abelian_group_iff hom_in_carrier)
  then show ?thesis
    by (metis (no_types) assms carrier_free_Abelian_group_iff group_free_Abelian_group
group_hom.hom_inv group_hom_axioms_def group_hom_def inv_free_Abelian_group)
qed

lemma hom_frag_add:
  assumes "h ∈ hom (free_Abelian_group S) (free_Abelian_group T)" "Poly_Mapping.keys
a ⊆ S" "Poly_Mapping.keys b ⊆ S"
  shows "h (a+b) = h a + h b"
proof -
  have "Poly_Mapping.keys (h a) ⊆ T"
    by (meson assms carrier_free_Abelian_group_iff hom_in_carrier)
  moreover
  have "Poly_Mapping.keys (h b) ⊆ T"
    by (meson assms carrier_free_Abelian_group_iff hom_in_carrier)
```

```
  ultimately show ?thesis
    using assms hom_mult by fastforce
qed

lemma hom_frag_diff:
  assumes "h ∈ hom (free_Abelian_group S) (free_Abelian_group T)" "Poly_Mapping.keys
a ⊆ S" "Poly_Mapping.keys b ⊆ S"
  shows "h (a-b) = h a - h b"
  by (metis (no_types, lifting) assms diff_conv_add_uminus hom_frag_add
hom_frag_minus keys_minus)


proposition isomorphic_free_Abelian_groups:
   "free_Abelian_group S ≅ free_Abelian_group T ⟷ S ≈ T"  (is "(?FS
≅ ?FT) = ?rhs")
proof
  interpret S: group "?FS"
    by simp
  interpret T: group "?FT"
    by simp
  interpret G2: comm_group "integer_mod_group 2"
    by (rule abelian_integer_mod_group)
  let ?Two = "{0..<2::int}"
  have [simp]: "¬ ?Two ⊆ {a}" for a
    by (simp add: subset_iff) presburger
  assume L: "?FS ≅ ?FT"
  let ?HS = "{h ∈ extensional (carrier ?FS). h ∈ hom ?FS (integer_mod_group
2)}"
  let ?HT = "{h ∈ extensional (carrier ?FT). h ∈ hom ?FT (integer_mod_group
2)}"
  have "S →E ?Two ≈ ?HS"
    apply (rule eqpoll_sym)
    using G2.eqpoll_homomorphisms_from_free_Abelian_group by (simp add:
carrier_integer_mod_group)
  also have "... ≈ ?HT"
  proof -
    obtain f g where "group_isomorphisms ?FS ?FT f g"
      using L S.iso_iff_group_isomorphisms by (force simp: is_iso_def)
    then have f: "f ∈ hom ?FS ?FT"
      and g: "g ∈ hom ?FT ?FS"
      and gf: "∀x ∈ carrier ?FS. g(f x) = x"
      and fg: "∀y ∈ carrier ?FT. f(g y) = y"
      by (auto simp: group_isomorphisms_def)
    let ?f = "λh. restrict (h ∘ g) (carrier ?FT)"
    let ?g = "λh. restrict (h ∘ f) (carrier ?FS)"
    show ?thesis
    proof (rule lepoll_antisym)
      show "?HS ≲ ?HT"
        unfolding lepoll_def
```

```
      proof (intro exI conjI)
        show "inj_on ?f ?HS"
          apply (rule inj_on_inverseI [where g = ?g])
          using hom_in_carrier [OF f]
          by (auto simp: gf fun_eq_iff carrier_integer_mod_group Ball_def
Pi_def extensional_def)
        show "?f ‘ ?HS ⊆ ?HT"
        proof clarsimp
          fix h
          assume h: "h ∈ hom ?FS (integer_mod_group 2)"
          have "h ∘ g ∈ hom ?FT (integer_mod_group 2)"
            by (rule hom_compose [OF g h])
          moreover have "restrict (h ∘ g) (carrier ?FT) x = (h ∘ g) x"
if "x ∈ carrier ?FT" for x
            using g that by (simp add: hom_def)
          ultimately show "restrict (h ∘ g) (carrier ?FT) ∈ hom ?FT (integer_mod_group
2)"
            using T.hom_restrict by fastforce
        qed
      qed
    next
      show "?HT ≲ ?HS"
        unfolding lepoll_def
      proof (intro exI conjI)
        show "inj_on ?g ?HT"
          apply (rule inj_on_inverseI [where g = ?f])
          using hom_in_carrier [OF g]
          by (auto simp: fg fun_eq_iff carrier_integer_mod_group Ball_def
Pi_def extensional_def)
        show "?g ‘ ?HT ⊆ ?HS"
        proof clarsimp
          fix k
          assume k: "k ∈ hom ?FT (integer_mod_group 2)"
          have "k ∘ f ∈ hom ?FS (integer_mod_group 2)"
            by (rule hom_compose [OF f k])
          moreover have "restrict (k ∘ f) (carrier ?FS) x = (k ∘ f) x"
if "x ∈ carrier ?FS" for x
            using f that by (simp add: hom_def)
          ultimately show "restrict (k ∘ f) (carrier ?FS) ∈ hom ?FS (integer_mod_group
2)"
            using S.hom_restrict by fastforce
        qed
      qed
    qed
  qed
  also have "... ≈ T →E ?Two"
    using G2.eqpoll_homomorphisms_from_free_Abelian_group by (simp add:
carrier_integer_mod_group)
  finally have *: "S →E ?Two ≈ T →E ?Two" .
```

```
  then have "finite (S →E ?Two) ⟷ finite (T →E ?Two)"
    by (rule eqpoll_finite_iff)
  then have "finite S ⟷ finite T"
    by (auto simp: finite_funcset_iff)
  then consider "finite S" "finite T" | "~ finite S" "~ finite T"
    by blast
  then show ?rhs
  proof cases
    case 1
    with * have "2 ^ card S = (2::nat) ^ card T"
      by (simp add: card_PiE finite_PiE eqpoll_iff_card)
    then have "card S = card T"
      by auto
    then show ?thesis
      using eqpoll_iff_card 1 by blast
  next
    case 2
    have "carrier (free_Abelian_group S) ≈ carrier (free_Abelian_group
T)"
      using L by (simp add: iso_imp_eqpoll_carrier)
    then show ?thesis
      using 2 eqpoll_free_Abelian_group_infinite eqpoll_sym eqpoll_trans
by metis
  qed
next
  assume ?rhs
  then obtain f g where f: "⋀x. x ∈ S ⟹ f x ∈ T ∧ g(f x) = x"
                    and g: "⋀y. y ∈ T ⟹ g y ∈ S ∧ f(g y) = y"
    using eqpoll_iff_bijections by metis
  interpret S: comm_group "?FS"
    by (simp add: abelian_free_Abelian_group)
  interpret T: comm_group "?FT"
    by (simp add: abelian_free_Abelian_group)
  have "(frag_of ∘ f) ‘ S ⊆ carrier (free_Abelian_group T)"
    using f by auto
  then obtain h where h: "h ∈ hom (free_Abelian_group S) (free_Abelian_group
T)"
    and h_frag: "⋀x. x ∈ S ⟹ h (frag_of x) = (frag_of ∘ f) x"
    using T.free_Abelian_group_universal [of "frag_of ∘ f" S] by blast
  interpret hhom: group_hom "free_Abelian_group S" "free_Abelian_group
T" h
    by (simp add: h group_hom_axioms_def group_hom_def)
  have "(frag_of ∘ g) ‘ T ⊆ carrier (free_Abelian_group S)"
    using g by auto
  then obtain k where k: "k ∈ hom (free_Abelian_group T) (free_Abelian_group
S)"
    and k_frag: "⋀x. x ∈ T ⟹ k (frag_of x) = (frag_of ∘ g) x"
    using S.free_Abelian_group_universal [of "frag_of ∘ g" T] by blast
  interpret khom: group_hom "free_Abelian_group T" "free_Abelian_group
```

```
S" k
    by (simp add: k group_hom_axioms_def group_hom_def)
  have kh: "Poly_Mapping.keys x ⊆ S ⟹ Poly_Mapping.keys x ⊆ S ∧ k
(h x) = x" for x
  proof (induction rule: frag_induction)
    case zero
    then show ?case
      apply auto
      by (metis group_free_Abelian_group h hom_one k one_free_Abelian_group)
  next
    case (one x)
    then show ?case
      by (auto simp: h_frag k_frag f)
  next
    case (diff a b)
    with keys_diff have "Poly_Mapping.keys (a - b) ⊆ S"
      by (metis Un_least order_trans)
    with diff hhom.hom_closed show ?case
      by (simp add: hom_frag_diff [OF h] hom_frag_diff [OF k])
  qed
  have hk: "Poly_Mapping.keys y ⊆ T ⟹ Poly_Mapping.keys y ⊆ T ∧ h
(k y) = y" for y
  proof (induction rule: frag_induction)
    case zero
    then show ?case
      apply auto
      by (metis group_free_Abelian_group h hom_one k one_free_Abelian_group)
  next
    case (one y)
    then show ?case
      by (auto simp: h_frag k_frag g)
  next
    case (diff a b)
    with keys_diff have "Poly_Mapping.keys (a - b) ⊆ T"
      by (metis Un_least order_trans)
    with diff khom.hom_closed show ?case
      by (simp add: hom_frag_diff [OF h] hom_frag_diff [OF k])
  qed
  have "h ∈ iso ?FS ?FT"
    unfolding iso_def bij_betw_iff_bijections mem_Collect_eq
  proof (intro conjI exI ballI h)
    fix x
    assume x: "x ∈ carrier (free_Abelian_group S)"
    show "h x ∈ carrier (free_Abelian_group T)"
      by (meson x h hom_in_carrier)
    show "k (h x) = x"
      using x by (simp add: kh)
  next
    fix y
```

```
    assume y: "y ∈ carrier (free_Abelian_group T)"
    show "k y ∈ carrier (free_Abelian_group S)"
      by (meson y k hom_in_carrier)
    show "h (k y) = y"
      using y by (simp add: hk)
  qed
  then show "?FS ≅ ?FT"
    by (auto simp: is_iso_def)
qed

lemma isomorphic_group_integer_free_Abelian_group_singleton:
  "integer_group ≅ free_Abelian_group {x}"
proof -
  have "(λn. frag_cmul n (frag_of x)) ∈ iso integer_group (free_Abelian_group
{x})"
  proof (rule isoI [OF homI])
    show "bij_betw (λn. frag_cmul n (frag_of x)) (carrier integer_group)
(carrier (free_Abelian_group {x}))"
      apply (rule bij_betwI [where g = "λy. Poly_Mapping.lookup y x"])
      by (auto simp: integer_group_def in_keys_iff intro!: poly_mapping_eqI)
  qed (auto simp: frag_cmul_distrib)
  then show ?thesis
    unfolding is_iso_def
    by blast
qed

lemma group_hom_free_Abelian_groups_id:
  "id ∈ hom (free_Abelian_group S) (free_Abelian_group T) ⟷ S ⊆ T"
proof -
  have "x ∈ T" if ST: "⋀c:: 'a ⇒₀ int. Poly_Mapping.keys c ⊆ S ⟶ Poly_Mapping.keys
c ⊆ T" and "x ∈ S" for x
    using ST [of "frag_of x"] ‹x ∈ S› by simp
  then show ?thesis
    by (auto simp: hom_def free_Abelian_group_def Pi_def)
qed

proposition iso_free_Abelian_group_sum:
  assumes "pairwise (λi j. disjnt (S i) (S j)) I"
  shows "(λf. sum' f I) ∈ iso (sum_group I (λi. free_Abelian_group(S
i))) (free_Abelian_group (⋃(S ` I)))"
    (is "?h ∈ iso ?G ?H")
proof (rule isoI)
  show hom: "?h ∈ hom ?G ?H"
  proof (rule homI)
    show "?h c ∈ carrier ?H" if "c ∈ carrier ?G" for c
      using that
      apply (simp add: sum.G_def carrier_sum_group)
      apply (rule order_trans [OF keys_sum])
      apply (auto simp: free_Abelian_group_def)
```

```
      done
    show "?h (x ⊗?G y) = ?h x ⊗?H ?h y"
      if "x ∈ carrier ?G" "y ∈ carrier ?G"  for x y
      using that by (simp add: sum.finite_Collect_op carrier_sum_group
sum.distrib')
  qed
  interpret GH: group_hom "?G" "?H" "?h"
    using hom by (simp add: group_hom_def group_hom_axioms_def)
  show "bij_betw ?h (carrier ?G) (carrier ?H)"
    unfolding bij_betw_def
  proof (intro conjI subset_antisym)
    show "?h ‘ carrier ?G ⊆ carrier ?H"
      apply (clarsimp simp: sum.G_def carrier_sum_group simp del: carrier_free_Abelian_grou
      by (force simp: PiE_def Pi_iff intro!: sum_closed_free_Abelian_group)
    have *: "poly_mapping.lookup (Abs_poly_mapping (λj. if j ∈ S i then
poly_mapping.lookup x j else 0)) k
          = (if k ∈ S i then poly_mapping.lookup x k else 0)" if "i ∈
I" for i k and x :: "'b ⇒0 int"
      using that by (auto simp: conj_commute cong: conj_cong)
    have eq: "Abs_poly_mapping (λj. if j ∈ S i then poly_mapping.lookup
x j else 0) = 0
     ⟷ (∀c ∈ S i. poly_mapping.lookup x c = 0)" if "i ∈ I" for i and
x :: "'b ⇒0 int"
      apply (auto simp: poly_mapping_eq_iff fun_eq_iff)
      apply (simp add: * Abs_poly_mapping_inverse conj_commute cong: conj_cong)
      apply (force dest!: spec split: if_split_asm)
      done
    have "x ∈ ?h ‘ {x ∈ ΠE i∈I. {c. Poly_Mapping.keys c ⊆ S i}. finite
{i ∈ I. x i ≠ 0}}"
      if x: "Poly_Mapping.keys x ⊆ ⋃ (S ‘ I)" for x :: "'b ⇒0 int"
    proof -
      let ?f = "(λi c. if c ∈ S i then Poly_Mapping.lookup x c else 0)"
      define J where "J ≡ {i ∈ I. ∃c∈S i. c ∈ Poly_Mapping.keys x}"
      have "J ⊆ (λc. THE i. i ∈ I ∧ c ∈ S i) ‘ Poly_Mapping.keys x"
      proof (clarsimp simp: J_def)
        show "i ∈ (λc. THE i. i ∈ I ∧ c ∈ S i) ‘ Poly_Mapping.keys x"
          if "i ∈ I" "c ∈ S i" "c ∈ Poly_Mapping.keys x" for i c
        proof
          show "i = (THE i. i ∈ I ∧ c ∈ S i)"
            using assms that by (auto simp: pairwise_def disjnt_def intro:
the_equality [symmetric])
        qed (simp add: that)
      qed
      then have fin: "finite J"
        using finite_subset finite_keys by blast
      have [simp]: "Poly_Mapping.keys (Abs_poly_mapping (?f i)) = {k.
?f i k ≠ 0}" if "i ∈ I" for i
        by (simp add: eq_onp_def keys.abs_eq conj_commute cong: conj_cong)
      have [simp]: "Poly_Mapping.lookup (Abs_poly_mapping (?f i)) c =
```

```
?f i c" if "i ∈ I" for i c
        by (auto simp: Abs_poly_mapping_inverse conj_commute cong: conj_cong)
      show ?thesis
      proof
        have "poly_mapping.lookup x c = poly_mapping.lookup (?h (λi∈I.
Abs_poly_mapping (?f i))) c"
          for c
        proof (cases "c ∈ Poly_Mapping.keys x")
          case True
          then obtain i where "i ∈ I" "c ∈ S i" "?f i c ≠ 0"
            using x by (auto simp: in_keys_iff)
          then have 1: "poly_mapping.lookup (sum' (λj. Abs_poly_mapping
(?f j)) (I - {i})) c = 0"
            using assms
            apply (simp add: sum.G_def Poly_Mapping.lookup_sum pairwise_def
disjnt_def)
            apply (force simp: eq split: if_split_asm intro!: comm_monoid_add_class.sum.neu
            done
          have 2: "poly_mapping.lookup x c = poly_mapping.lookup (Abs_poly_mapping
(?f i)) c"
            by (auto simp: ‹c ∈ S i› Abs_poly_mapping_inverse conj_commute
cong: conj_cong)
          have "finite {i ∈ I. Abs_poly_mapping (?f i) ≠ 0}"
            by (rule finite_subset [OF _ fin]) (use ‹i ∈ I› J_def eq
in ‹auto simp: in_keys_iff›)
          with ‹i ∈ I› have "?h (λj∈I. Abs_poly_mapping (?f j)) = Abs_poly_mapping
(?f i) + sum' (λj. Abs_poly_mapping (?f j)) (I - {i})"
            by (simp add: sum_diff1')
          then show ?thesis
            by (simp add: 1 2 Poly_Mapping.lookup_add)
        next
          case False
          then have "poly_mapping.lookup x c = 0"
            using keys.rep_eq by force
          then show ?thesis
            unfolding sum.G_def by (simp add: lookup_sum * comm_monoid_add_class.sum.neutra
        qed
        then show "x = ?h (λi∈I. Abs_poly_mapping (?f i))"
          by (rule poly_mapping_eqI)
        have "(λi. Abs_poly_mapping (?f i)) ∈ (Π i∈I. {c. Poly_Mapping.keys
c ⊆ S i})"
          by (auto simp: PiE_def Pi_def in_keys_iff)
        then show "(λi∈I. Abs_poly_mapping (?f i))
                 ∈ {x ∈ Π_E i∈I. {c. Poly_Mapping.keys c ⊆ S i}. finite
{i ∈ I. x i ≠ 0}}"
          using fin unfolding J_def by (force simp add: eq in_keys_iff
cong: conj_cong)
      qed
    qed
```

```
    then show "carrier ?H ⊆ ?h ‘ carrier ?G"
      by (simp add: carrier_sum_group) (auto simp: free_Abelian_group_def)
    show "inj_on ?h (carrier (sum_group I (λi. free_Abelian_group (S
i))))"
      unfolding GH.inj_on_one_iff
    proof clarify
      fix x
      assume "x ∈ carrier ?G" "?h x = 1?H"
      then have eq0: "sum’ x I = 0"
        and xs: "⋀i. i ∈ I ⟹ Poly_Mapping.keys (x i) ⊆ S i" and xext:
"x ∈ extensional I"
        and fin: "finite {i ∈ I. x i ≠ 0}"
        by (simp_all add: carrier_sum_group PiE_def Pi_def)
      have "x i = 0" if "i ∈ I" for i
      proof -
        have "sum’ x (insert i (I - {i})) = 0"
          using eq0 that by (simp add: insert_absorb)
        moreover have "Poly_Mapping.keys (sum’ x (I - {i})) = {}"
        proof -
          have "x i = - sum’ x (I - {i})"
            by (metis (mono_tags, lifting) diff_zero eq0 fin sum_diff1’
minus_diff_eq that)
          then have "Poly_Mapping.keys (x i) = Poly_Mapping.keys (sum’
x (I - {i}))"
            by simp
          then have "Poly_Mapping.keys (sum’ x (I - {i})) ⊆ S i"
            using that xs by metis
          moreover
          have "Poly_Mapping.keys (sum’ x (I - {i})) ⊆ (⋃j ∈ I - {i}.
S j)"
          proof -
            have "Poly_Mapping.keys (sum’ x (I - {i})) ⊆ (⋃i∈{j ∈ I.
j ≠ i ∧ x j ≠ 0}. Poly_Mapping.keys (x i))"
              using keys_sum [of x "{j ∈ I. j ≠ i ∧ x j ≠ 0}"] by (simp
add: sum.G_def)
            also have "... ⊆ ⋃ (S ‘ (I - {i}))"
              using xs by force
            finally show ?thesis .
          qed
          moreover have "A = {}" if "A ⊆ S i" "A ⊆ ⋃ (S ‘ (I - {i}))"
for A
            using assms that ‹i ∈ I›
            by (force simp: pairwise_def disjnt_def image_def subset_iff)
          ultimately show ?thesis
            by metis
        qed
        then have [simp]: "sum’ x (I - {i}) = 0"
          by (auto simp: sum.G_def)
        have "sum’ x (insert i (I - {i})) = x i"
```

```
        by (subst sum.insert' [OF finite_subset [OF _ fin]]) auto
      ultimately show ?thesis
        by metis
    qed
    with xext [unfolded extensional_def]
    show "x = 1sum_group I (λi. free_Abelian_group (S i))"
      by (force simp: free_Abelian_group_def)
  qed
 qed
qed

lemma isomorphic_free_Abelian_group_Union:
 "pairwise disjnt I ⟹ free_Abelian_group(⋃ I) ≅ sum_group I free_Abelian_group"
 using iso_free_Abelian_group_sum [of "λX. X" I]
 by (metis SUP_identity_eq empty_iff group.iso_sym group_free_Abelian_group
is_iso_def sum_group)

lemma isomorphic_sum_integer_group:
  "sum_group I (λi. integer_group) ≅ free_Abelian_group I"
proof -
 have "sum_group I (λi. integer_group) ≅ sum_group I (λi. free_Abelian_group
{i})"
   by (rule iso_sum_groupI) (auto simp: isomorphic_group_integer_free_Abelian_group_single
 also have "... ≅ free_Abelian_group I"
   using iso_free_Abelian_group_sum [of "λx. {x}" I] by (auto simp: is_iso_def)
 finally show ?thesis .
qed

end


theory Solvable_Groups
 imports Generated_Groups

begin
```

# 45 Solvable Groups

## 45.1 Definitions

```
inductive solvable_seq :: "('a, 'b) monoid_scheme ⇒ 'a set ⇒ bool"
 for G where
   unity: "solvable_seq G { 1G }"
 | extension: "⟦ solvable_seq G K; K ◁ (G ⦇ carrier := H ⦈); subgroup
H G;
                 comm_group ((G ⦇ carrier := H ⦈) Mod K) ⟧ ⟹ solvable_seq
G H"

definition solvable :: "('a, 'b) monoid_scheme ⇒ bool"
```

```
  where "solvable G ⟷ solvable_seq G (carrier G)"
```

## 45.2 Solvable Groups and Derived Subgroups

We show that a group G is solvable iff the subgroup (derived G ‘n) (carrier G) is trivial for a sufficiently large n.

```
lemma (in group) solvable_imp_subgroup:
  assumes "solvable_seq G H" shows "subgroup H G"
  using assms normal.axioms(1)[OF one_is_normal] by (induction) (auto)

lemma (in group) augment_solvable_seq:
  assumes "subgroup H G" and "solvable_seq G (derived G H)" shows "solvable_seq
G H"
  using extension[OF _ derived_subgroup_is_normal _ derived_quot_of_subgroup_is_comm_group]
assms by simp

theorem (in group) trivial_derived_seq_imp_solvable:
  assumes "subgroup H G" and "((derived G) ^^ n) H = { 1 }" shows "solvable_seq
G H"
  using assms
proof (induct n arbitrary: H, simp add: unity[of G])
  case (Suc n) thus ?case
    using augment_solvable_seq derived_is_subgroup[OF subgroup.subset]
by (simp add: funpow_swap1)
qed

theorem (in group) solvable_imp_trivial_derived_seq:
  assumes "solvable_seq G H" shows "∃n. (derived G ^^ n) H = { 1 }"
  using assms
proof (induction)
  case unity
  have "(derived G ^^ 0) { 1 } = { 1 }"
    by simp
  thus ?case by blast
next
  case (extension K H)
  obtain n where "(derived G ^^ n) K = { 1 }"
    using solvable_imp_subgroup extension(1,5) by auto
  hence "(derived G ^^ (Suc n)) H ⊆ { 1 }"
    using mono_exp_of_derived[OF derived_of_subgroup_minimal[OF extension(2-4)],
of n] by (simp add: funpow_swap1)
  moreover have "{ 1 } ⊆ (derived G ^^ (Suc n)) H"
    using subgroup.one_closed[OF exp_of_derived_is_subgroup[OF extension(3)],
of "Suc n"] by auto
  ultimately show ?case
    by blast
qed

theorem (in group) solvable_iff_trivial_derived_seq:
```

```
  "solvable G ⟷ (∃n. (derived G ^^ n) (carrier G) = { 1 })"
  using solvable_imp_trivial_derived_seq subgroup_self trivial_derived_seq_imp_solvable
  by (auto simp add: solvable_def)

corollary (in group) solvable_subgroup:
  assumes "subgroup H G" and "solvable G" shows "solvable_seq G H"
proof -
  obtain n where n: "(derived G ^^ n) (carrier G) = { 1 }"
    using assms(2) solvable_imp_trivial_derived_seq by (auto simp add:
solvable_def)
  show ?thesis
  proof (rule trivial_derived_seq_imp_solvable[OF assms(1), of n])
    show "(derived G ^^ n) H = { 1 }"
      using subgroup.one_closed[OF exp_of_derived_is_subgroup[OF assms(1)],
of n]
            mono_exp_of_derived[OF subgroup.subset[OF assms(1)], of n]
n
      by auto
  qed
qed
```

## 45.3 Short Exact Sequences

Even if we don't talk about short exact sequences explicitly, we show that given an injective homomorphism from a group H to a group G, if H isn't solvable the group G isn't neither.

```
theorem (in group_hom) solvable_img_imp_solvable:
  assumes "subgroup K G" and "inj_on h K" and "solvable_seq H (h ‘ K)"
shows "solvable_seq G K"
proof -
  obtain n where "(derived H ^^ n) (h ‘ K) = { 1_H }"
    using solvable_imp_trivial_derived_seq assms(1,3) by auto
  hence "h ‘ ((derived G ^^ n) K) = { 1_H }"
    unfolding exp_of_derived_img[OF subgroup.subset[OF assms(1)]] .
  moreover have "(derived G ^^ n) K ⊆ K"
    using G.mono_derived[of _ K] G.derived_incl[OF _ assms(1)] by (induct
n) (auto)
  hence "inj_on h ((derived G ^^ n) K)"
    using inj_on_subset[OF assms(2)] by blast
  moreover have "{ 1 } ⊆ (derived G ^^ n) K"
    using subgroup.one_closed[OF G.exp_of_derived_is_subgroup[OF assms(1)]]
by blast
  ultimately show ?thesis
    using G.trivial_derived_seq_imp_solvable[OF assms(1), of n]
    by (metis (no_types, lifting) hom_one image_empty image_insert inj_on_image_eq_iff
order_refl)
qed

corollary (in group_hom) inj_hom_imp_solvable:
```

```
  assumes "inj_on h (carrier G)" and "solvable H" shows "solvable G"
  using solvable_img_imp_solvable[OF _ assms(1)] G.subgroup_self
        solvable_subgroup[OF subgroup_img_is_subgroup assms(2)]
  unfolding solvable_def
  by simp

theorem (in group_hom) solvable_imp_solvable_img:
  assumes "solvable_seq G K" shows "solvable_seq H (h ‘ K)"
proof -
  obtain n where "(derived G ^^ n) K = { 1 }"
    using G.solvable_imp_trivial_derived_seq[OF assms] by blast
  thus ?thesis
    using trivial_derived_seq_imp_solvable[OF subgroup_img_is_subgroup,
of _ n]
          exp_of_derived_img[OF subgroup.subset, of _ n] G.solvable_imp_subgroup[OF
assms]
    by auto
qed

corollary (in group_hom) surj_hom_imp_solvable:
  assumes "h ‘ carrier G = carrier H" and "solvable G" shows "solvable
H"
  using assms solvable_imp_solvable_img[of "carrier G"] unfolding solvable_def
by simp

lemma solvable_seq_condition:
  assumes "group_hom G H f" "group_hom H K g" and "f ‘ I ⊆ J" and "kernel
H K g ⊆ f ‘ I"
    and "subgroup J H" and "solvable_seq G I" "solvable_seq K (g ‘ J)"
  shows "solvable_seq H J"
proof -
  interpret G: group G + H: group H + K: group K + J: subgroup J H + I:
subgroup I G
    using assms(1-2,5) group.solvable_imp_subgroup[OF _ assms(6)] un-
folding group_hom_def by auto

  obtain n m
    where n: "(derived G ^^ n) I = { 1_G }" and m: "(derived K ^^ m) (g
‘ J) = { 1_K }"
    using G.solvable_imp_trivial_derived_seq[OF assms(6)]
          K.solvable_imp_trivial_derived_seq[OF assms(7)]
    by auto
  have "(derived H ^^ m) J ⊆ f ‘ I"
    using m H.exp_of_derived_in_carrier[OF J.subset, of m] assms(4)
    by (auto simp add: group_hom.exp_of_derived_img[OF assms(2) J.subset]
kernel_def)
  hence "(derived H ^^ n) ((derived H ^^ m) J) ⊆ f ‘ ((derived G ^^ n)
I)"
    using n H.mono_exp_of_derived unfolding sym[OF group_hom.exp_of_derived_img[OF
```

```
assms(1) I.subset, of n]] by simp
  hence "(derived H ^^ (n + m)) J ⊆ { 1H }"
    using group_hom.hom_one[OF assms(1)] unfolding n by (simp add: funpow_add)
  moreover have "{ 1H } ⊆ (derived H ^^ (n + m)) J"
    using subgroup.one_closed[OF H.exp_of_derived_is_subgroup[OF assms(5),
of "n + m"]] by blast
  ultimately show ?thesis
    using H.trivial_derived_seq_imp_solvable[OF assms(5)] by simp
qed

lemma solvable_condition:
  assumes "group_hom G H f" "group_hom H K g"
    and "g ‘ (carrier H) = carrier K" and "kernel H K g ⊆ f ‘ (carrier
G)"
    and "solvable G" "solvable K" shows "solvable H"
  using solvable_seq_condition[OF assms(1-2) _ assms(4) group.subgroup_self]
assms(3,5-6)
        subgroup.subset[OF group_hom.img_is_subgroup[OF assms(1)]] group_hom.axioms(2)[OF
assms(1)]
  by (simp add: solvable_def)

end


theory Sym_Groups
  imports
    "HOL-Combinatorics.Cycles"
    Solvable_Groups
begin
```

# 46 Symmetric Groups

## 46.1 Definitions

```
abbreviation inv' :: "('a ⇒ 'b) ⇒ ('b ⇒ 'a)"
  where "inv' f ≡ Hilbert_Choice.inv f"

definition sym_group :: "nat ⇒ (nat ⇒ nat) monoid"
  where "sym_group n = (| carrier = { p. p permutes {1..n} }, mult = (∘),
one = id |)"

definition alt_group :: "nat ⇒ (nat ⇒ nat) monoid"
  where "alt_group n = (sym_group n) (| carrier := { p. p permutes {1..n}
∧ evenperm p } |)"

definition sign_img :: "int monoid"
  where "sign_img = (| carrier = { -1, 1 }, mult = (*), one = 1 |)"
```

## 46.2 Basic Properties

```
lemma sym_group_carrier: "p ∈ carrier (sym_group n) ⟷ p permutes {1..n}"
  unfolding sym_group_def by simp

lemma sym_group_mult: "mult (sym_group n) = (∘)"
  unfolding sym_group_def by simp

lemma sym_group_one: "one (sym_group n) = id"
  unfolding sym_group_def by simp

lemma sym_group_carrier': "p ∈ carrier (sym_group n) ⟹ permutation
p"
  unfolding sym_group_carrier permutation_permutes by auto

lemma alt_group_carrier: "p ∈ carrier (alt_group n) ⟷ p permutes {1..n}
∧ evenperm p"
  unfolding alt_group_def by auto

lemma alt_group_mult: "mult (alt_group n) = (∘)"
  unfolding alt_group_def using sym_group_mult by simp

lemma alt_group_one: "one (alt_group n) = id"
  unfolding alt_group_def using sym_group_one by simp

lemma alt_group_carrier': "p ∈ carrier (alt_group n) ⟹ permutation
p"
  unfolding alt_group_carrier permutation_permutes by auto

lemma sym_group_is_group: "group (sym_group n)"
  using permutes_inv permutes_inv_o(2)
  by (auto intro!: groupI
         simp add: sym_group_def permutes_compose
                   permutes_id comp_assoc, blast)

lemma sign_img_is_group: "group sign_img"
  unfolding sign_img_def by (unfold_locales, auto simp add: Units_def)

lemma sym_group_inv_closed:
  assumes "p ∈ carrier (sym_group n)" shows "inv' p ∈ carrier (sym_group
n)"
  using assms permutes_inv sym_group_def by auto

lemma alt_group_inv_closed:
  assumes "p ∈ carrier (alt_group n)" shows "inv' p ∈ carrier (alt_group
n)"
  using evenperm_inv[OF alt_group_carrier'] permutes_inv assms alt_group_carrier
by auto

lemma sym_group_inv_equality [simp]:
```

```
  assumes "p ∈ carrier (sym_group n)" shows "inv(sym_group n) p = inv'
p"
proof -
  have "inv' p ∘ p = id"
    using assms permutes_inv_o(2) sym_group_def by auto
  hence "(inv' p) ⊗(sym_group n) p = one (sym_group n)"
    by (simp add: sym_group_def)
  thus ?thesis  using group.inv_equality[OF sym_group_is_group]
    by (simp add: assms sym_group_inv_closed)
qed

lemma sign_is_hom: "sign ∈ hom (sym_group n) sign_img"
  unfolding hom_def sign_img_def sym_group_mult using sym_group_carrier'[of
_ n]
  by (auto simp add: sign_compose, meson sign_def)

lemma sign_group_hom: "group_hom (sym_group n) sign_img sign"
  using group_hom.intro[OF sym_group_is_group sign_img_is_group] sign_is_hom
  by (simp add: group_hom_axioms_def)

lemma sign_is_surj:
  assumes "n ≥ 2" shows "sign ` (carrier (sym_group n)) = carrier sign_img"
proof -
  have "swapidseq (Suc 0) (Fun.swap (1 :: nat) 2 id)"
    using comp_Suc[OF id, of "1 :: nat" "2"] by auto
  hence "sign (Fun.swap (1 :: nat) 2 id) = (-1 :: int)"
    by (simp add: sign_swap_id)
  moreover have "Fun.swap (1 :: nat) 2 id ∈ carrier (sym_group n)" and
"id ∈ carrier (sym_group n)"
    using assms permutes_swap_id[of "1 :: nat" "{1..n}" 2] permutes_id
    unfolding sym_group_carrier by auto
  ultimately have "carrier sign_img ⊆ sign ` (carrier (sym_group n))"
    using sign_id mk_disjoint_insert unfolding sign_img_def by fastforce
  moreover have "sign ` (carrier (sym_group n)) ⊆ carrier sign_img"
    using sign_is_hom unfolding hom_def by auto
  ultimately show ?thesis
    by blast
qed

lemma alt_group_is_sign_kernel:
  "carrier (alt_group n) = kernel (sym_group n) sign_img sign"
  unfolding alt_group_def sym_group_def sign_img_def kernel_def sign_def
by auto

lemma alt_group_is_subgroup: "subgroup (carrier (alt_group n)) (sym_group
n)"
  using group_hom.subgroup_kernel[OF sign_group_hom]
  unfolding alt_group_is_sign_kernel by blast
```

```
lemma alt_group_is_group: "group (alt_group n)"
  using group.subgroup_imp_group[OF sym_group_is_group alt_group_is_subgroup]
  by (simp add: alt_group_def)

lemma sign_iso:
  assumes "n ≥ 2" shows "(sym_group n) Mod (carrier (alt_group n)) ≅
sign_img"
  using group_hom.FactGroup_iso[OF sign_group_hom sign_is_surj[OF assms]]
  unfolding alt_group_is_sign_kernel .

lemma alt_group_inv_equality:
  assumes "p ∈ carrier (alt_group n)" shows "inv(alt_group n) p = inv'
p"
proof -
  have "inv' p ∘ p = id"
    using assms permutes_inv_o(2) alt_group_def by auto
  hence "(inv' p) ⊗(alt_group n) p = one (alt_group n)"
    by (simp add: alt_group_def sym_group_def)
  thus ?thesis  using group.inv_equality[OF alt_group_is_group]
    by (simp add: assms alt_group_inv_closed)
qed

lemma sym_group_card_carrier: "card (carrier (sym_group n)) = fact n"
  using card_permutations[of "{1..n}" n] unfolding sym_group_def by simp

lemma alt_group_card_carrier:
  assumes "n ≥ 2" shows "2 * card (carrier (alt_group n)) = fact n"
proof -
  have "card (rcosetssym_group n (carrier (alt_group n))) = 2"
    using iso_same_card[OF sign_iso[OF assms]] unfolding FactGroup_def
sign_img_def by auto
  thus ?thesis
    using group.lagrange[OF sym_group_is_group alt_group_is_subgroup,
of n]
    unfolding order_def sym_group_card_carrier by simp
qed
```

## 46.3 Transposition Sequences

In order to prove that the Alternating Group can be generated by 3-cycles, we need a stronger decomposition of permutations as transposition sequences than the one proposed at Permutations.thy.

```
inductive swapidseq_ext :: "'a set ⇒ nat ⇒ ('a ⇒ 'a) ⇒ bool"
  where
    empty:  "swapidseq_ext {} 0 id"
  | single: "⟦ swapidseq_ext S n p; a ∉ S ⟧ ⟹ swapidseq_ext (insert
a S) n p"
  | comp:   "⟦ swapidseq_ext S n p; a ≠ b ⟧ ⟹
```

```
              swapidseq_ext (insert a (insert b S)) (Suc n) ((transpose
a b) ∘ p)"


lemma swapidseq_ext_finite:
  assumes "swapidseq_ext S n p" shows "finite S"
  using assms by (induction) (auto)

lemma swapidseq_ext_zero:
  assumes "finite S" shows "swapidseq_ext S 0 id"
  using assms empty by (induct set: "finite", fastforce, simp add: single)

lemma swapidseq_ext_imp_swapidseq:
  ‹swapidseq n p› if ‹swapidseq_ext S n p›
using that proof induction
  case empty
  then show ?case
    by (simp add: fun_eq_iff)
next
  case (single S n p a)
  then show ?case by simp
next
  case (comp S n p a b)
  then have ‹swapidseq (Suc n) (transpose a b ∘ p)›
    by (simp add: comp_Suc)
  then show ?case by (simp add: comp_def)
qed

lemma swapidseq_ext_zero_imp_id:
  assumes "swapidseq_ext S 0 p" shows "p = id"
proof -
  have "⟦ swapidseq_ext S n p; n = 0 ⟧ ⟹ p = id" for n
    by (induction rule: swapidseq_ext.induct, auto)
  thus ?thesis
    using assms by simp
qed

lemma swapidseq_ext_finite_expansion:
  assumes "finite B" and "swapidseq_ext A n p" shows "swapidseq_ext
(A ∪ B) n p"
  using assms
proof (induct set: "finite", simp)
  case (insert b B) show ?case
    using insert single[OF insert(3), of b] by (metis Un_insert_right
insert_absorb)
qed

lemma swapidseq_ext_backwards:
  assumes "swapidseq_ext A (Suc n) p"
```

```
  shows "∃a b A' p'. a ≠ b ∧ A = (insert a (insert b A')) ∧
                    swapidseq_ext A' n p' ∧ p = (transpose a b) ∘ p'"
proof -
  have "∃a b A' p'. a ≠ b ∧ A = (insert a (insert b A')) ∧
                    swapidseq_ext A' k p' ∧ p = (transpose a b) ∘ p'"
    if "swapidseq_ext A n p" "n = Suc k"
    for A n k and p :: "'a ⇒ 'a"
    using that
  proof (induction)
    case empty
    thus ?case by simp
  next
    case single
    thus ?case by (metis Un_insert_right insert_iff insert_is_Un swapidseq_ext.single)
  next
    case comp
    thus ?case by blast
  qed
  thus ?thesis using assms by simp
qed

lemma swapidseq_ext_backwards':
  assumes "swapidseq_ext A (Suc n) p"
  shows "∃a b A' p'. a ∈ A ∧ b ∈ A ∧ a ≠ b ∧ swapidseq_ext A n p' ∧
p = (transpose a b) ∘ p'"
  using swapidseq_ext_backwards[OF assms] swapidseq_ext_finite_expansion
  by (metis Un_insert_left assms insertI1 sup.idem sup_commute swapidseq_ext_finite)

lemma swapidseq_ext_endswap:
  assumes "swapidseq_ext S n p" "a ≠ b"
  shows "swapidseq_ext (insert a (insert b S)) (Suc n) (p ∘ (transpose
a b))"
  using assms
proof (induction n arbitrary: S p a b)
  case 0 hence "p = id"
    using swapidseq_ext_zero_imp_id by blast
  thus ?case
    using 0 by (metis comp_id id_comp swapidseq_ext.comp)
next
  case (Suc n)
  then obtain c d S' and p' :: "'a ⇒ 'a"
    where cd: "c ≠ d" and S: "S = (insert c (insert d S'))" "swapidseq_ext
S' n p'"
      and p: "p = transpose c d ∘ p'"
    using swapidseq_ext_backwards[OF Suc(2)] by blast
  hence "swapidseq_ext (insert a (insert b S')) (Suc n) (p' ∘ (transpose
a b))"
    by (simp add: Suc.IH Suc.prems(2))
  hence "swapidseq_ext (insert c (insert d (insert a (insert b S'))))
```

```
(Suc (Suc n))
                 (transpose c d ∘ p' ∘ (transpose a b))"
    by (metis cd fun.map_comp swapidseq_ext.comp)
  thus ?case
    by (metis S(1) p insert_commute)
qed

lemma swapidseq_ext_extension:
  assumes "swapidseq_ext A n p" and "swapidseq_ext B m q" and "A ∩ B
= {}"
  shows "swapidseq_ext (A ∪ B) (n + m) (p ∘ q)"
  using assms(1,3)
proof (induction, simp add: assms(2))
  case single show ?case
    using swapidseq_ext.single[OF single(3)] single(2,4) by auto
next
  case comp show ?case
    using swapidseq_ext.comp[OF comp(3,2)] comp(4)
    by (metis Un_insert_left add_Suc insert_disjoint(1) o_assoc)
qed

lemma swapidseq_ext_of_cycles:
  assumes "cycle cs" shows "swapidseq_ext (set cs) (length cs - 1) (cycle_of_list
cs)"
  using assms
proof (induct cs rule: cycle_of_list.induct)
  case (1 i j cs) show ?case
    using comp[OF 1(1), of i j] 1(2) by (simp add: o_def)
next
  case "2_1" show ?case
    by (simp, metis eq_id_iff empty)
next
  case ("2_2" v) show ?case
    using single[OF empty, of v] by (simp, metis eq_id_iff)
qed

lemma cycle_decomp_imp_swapidseq_ext:
  assumes "cycle_decomp S p" shows "∃n. swapidseq_ext S n p"
  using assms
proof (induction)
  case empty show ?case
    using swapidseq_ext.empty by blast
next
  case (comp I p cs)
  then obtain m where m: "swapidseq_ext I m p" by blast
  hence "swapidseq_ext (set cs) (length cs - 1) (cycle_of_list cs)"
    using comp.hyps(2) swapidseq_ext_of_cycles by blast
  thus ?case using swapidseq_ext_extension m
    using comp.hyps(3) by blast
```

```
qed

lemma swapidseq_ext_of_permutation:
  assumes "p permutes S" and "finite S" shows "∃n. swapidseq_ext S n
p"
  using cycle_decomp_imp_swapidseq_ext[OF cycle_decomposition[OF assms]]
.

lemma split_swapidseq_ext:
  assumes "k ≤ n" and "swapidseq_ext S n p"
  obtains q r U V where "swapidseq_ext U (n - k) q" and "swapidseq_ext
V k r" and "p = q ∘ r" and "U ∪ V = S"
proof -
  from assms
  have "∃q r U V. swapidseq_ext U (n - k) q ∧ swapidseq_ext V k r ∧ p
= q ∘ r ∧ U ∪ V = S"
   (is "∃q r U V. ?split k q r U V")
  proof (induct k rule: inc_induct)
    case base thus ?case
      by (metis diff_self_eq_0 id_o sup_bot.left_neutral empty)
  next
    case (step m)
    then obtain q r U V
      where q: "swapidseq_ext U (n - Suc m) q" and r: "swapidseq_ext
V (Suc m) r"
        and p: "p = q ∘ r" and S: "U ∪ V = S"
      by blast
    obtain a b r' V'
      where "a ≠ b" and r': "V = (insert a (insert b V'))" "swapidseq_ext
V' m r'" "r = (transpose a b) ∘ r'"
      using swapidseq_ext_backwards[OF r] by blast
    have "swapidseq_ext (insert a (insert b U)) (n - m) (q ∘ (transpose
a b))"
      using swapidseq_ext_endswap[OF q ‹a ≠ b›] step(2) by (metis Suc_diff_Suc)
    hence "?split m (q ∘ (transpose a b)) r' (insert a (insert b U)) V'"
      using r' S unfolding p by fastforce
    thus ?case by blast
  qed
  thus ?thesis
    using that by blast
qed
```

## 46.4 Unsolvability of Symmetric Groups

We show that symmetric groups (`sym_group n`) are unsolvable for `(5::'a)` `≤ n`.

```
abbreviation three_cycles :: "nat ⇒ (nat ⇒ nat) set"
  where "three_cycles n ≡
          { cycle_of_list cs | cs. cycle cs ∧ length cs = 3 ∧ set cs
```

```
⊆ {1..n} }"


lemma stupid_lemma:
  assumes "length cs = 3" shows "cs = [ (cs ! 0), (cs ! 1), (cs ! 2)
]"
  using assms by (auto intro!: nth_equalityI)
    (metis Suc_lessI less_2_cases not_less_eq nth_Cons_0
           nth_Cons_Suc numeral_2_eq_2 numeral_3_eq_3)

lemma three_cycles_incl: "three_cycles n ⊆ carrier (alt_group n)"
proof
  fix p assume "p ∈ three_cycles n"
  then obtain cs where cs: "p = cycle_of_list cs" "cycle cs" "length
cs = 3" "set cs ⊆ {1..n}"
    by auto
  obtain a b c where cs_def: "cs = [ a, b, c ]"
    using stupid_lemma[OF cs(3)] by auto
  have "swapidseq (Suc (Suc 0)) ((transpose a b) ∘ (Fun.swap b c id))"
    using comp_Suc[OF comp_Suc[OF id], of b c a b] cs(2) unfolding cs_def
by simp
  hence "evenperm p"
    using cs(1) unfolding cs_def by (simp add: evenperm_unique)
  thus "p ∈ carrier (alt_group n)"
    using permutes_subset[OF cycle_permutes cs(4)]
    unfolding alt_group_carrier cs(1) by simp
qed

lemma alt_group_carrier_as_three_cycles:
  "carrier (alt_group n) = generate (alt_group n) (three_cycles n)"
proof -
  interpret A: group "alt_group n"
    using alt_group_is_group by simp

  show ?thesis
  proof
    show "generate (alt_group n) (three_cycles n) ⊆ carrier (alt_group
n)"
      using A.generate_subgroup_incl[OF three_cycles_incl A.subgroup_self]
.
    show "carrier (alt_group n) ⊆ generate (alt_group n) (three_cycles
n)"
    proof
      have aux_lemma1: "cycle_of_list [a, b, c] ∈ generate (alt_group
n) (three_cycles n)"
        if "a ≠ b" "b ≠ c" "{ a, b, c } ⊆ {1..n}"
        for q :: "nat ⇒ nat" and a b c
      proof (cases)
        assume "c = a"
```

```
      hence "cycle_of_list [ a, b, c ] = id"
        by (simp add: swap_commute)
      thus "cycle_of_list [ a, b, c ] ∈ generate (alt_group n) (three_cycles
n)"
        using one[of "alt_group n"] unfolding alt_group_one by simp
    next
      assume "c ≠ a"
      have "distinct [a, b, c]"
        using ‹a ≠ b› and ‹b ≠ c› and ‹c ≠ a› by auto
      with ‹{ a, b, c } ⊆ {1..n}›
      show "cycle_of_list [ a, b, c ] ∈ generate (alt_group n) (three_cycles
n)"
        by (intro incl) fastforce
    qed

    have aux_lemma2: "q ∈ generate (alt_group n) (three_cycles n)"
      if seq: "swapidseq_ext S (Suc (Suc 0)) q" and S: "S ⊆ {1..n}"
      for S :: "nat set" and q :: "nat ⇒ nat"
    proof -
      obtain a b q' where ab: "a ≠ b" "a ∈ S" "b ∈ S"
        and q': "swapidseq_ext S (Suc 0) q'" "q = (transpose a b) ∘
q'"
        using swapidseq_ext_backwards'[OF seq] by auto
      obtain c d where cd: "c ≠ d" "c ∈ S" "d ∈ S"
        and q: "q = (transpose a b) ∘ (Fun.swap c d id)"
        using swapidseq_ext_backwards'[OF q'(1)]
          swapidseq_ext_zero_imp_id
        unfolding q'(2)
        by fastforce

      consider (eq) "b = c" | (ineq) "b ≠ c" by auto
      thus ?thesis
      proof cases
        case eq
        then have "q = cycle_of_list [ a, b, d ]"
          unfolding q by simp
        moreover have "{ a, b, d } ⊆ {1..n}"
          using ab cd S by blast
        ultimately show ?thesis
          using aux_lemma1[OF ab(1)] cd(1) eq by simp
      next
        case ineq
        hence "q = cycle_of_list [ a, b, c ] ∘ cycle_of_list [ b, c,
d ]"
          unfolding q by (simp add: swap_nilpotent o_assoc)
        moreover have "{ a, b, c } ⊆ {1..n}" and "{ b, c, d } ⊆ {1..n}"
          using ab cd S by blast+
        ultimately show ?thesis
          using eng[OF aux_lemma1[OF ab(1) ineq] aux_lemma1[OF ineq
```

```
cd(1)]]
            unfolding alt_group_mult by simp
        qed
      qed

      fix p assume "p ∈ carrier (alt_group n)" then have p: "p permutes
{1..n}" "evenperm p"
        unfolding alt_group_carrier by auto
      obtain m where m: "swapidseq_ext {1..n} m p"
        using swapidseq_ext_of_permutation[OF p(1)] by auto
      have "even m"
        using swapidseq_ext_imp_swapidseq[OF m] p(2) evenperm_unique by
blast
      then obtain k where k: "m = 2 * k"
        by auto
      show "p ∈ generate (alt_group n) (three_cycles n)"
        using m unfolding k
      proof (induct k arbitrary: p)
        case 0 then have "p = id"
          using swapidseq_ext_zero_imp_id by simp
        show ?case
          using generate.one[of "alt_group n" "three_cycles n"]
          unfolding alt_group_one ‹p = id› .
      next
        case (Suc m)
        have arith: "2 * (Suc m) - (Suc (Suc 0)) = 2 * m" and "Suc (Suc
0) ≤ 2 * Suc m"
          by auto
        then obtain q r U V
          where q: "swapidseq_ext U (2 * m) q" and r: "swapidseq_ext
V (Suc (Suc 0)) r"
            and p: "p = q ∘ r" and UV: "U ∪ V = {1..n}"
          using split_swapidseq_ext[OF _ Suc(2), of "Suc (Suc 0)"] un-
folding arith by metis
        have "swapidseq_ext {1..n} (2 * m) q"
          using UV q swapidseq_ext_finite_expansion[OF swapidseq_ext_finite[OF
r] q] by simp
        thus ?case
          using eng[OF Suc(1) aux_lemma2[OF r], of q] UV unfolding alt_group_mult
p by blast
      qed
    qed
  qed
qed

theorem derived_alt_group_const:
  assumes "n ≥ 5" shows "derived (alt_group n) (carrier (alt_group n))
= carrier (alt_group n)"
proof
```

```
  show "derived (alt_group n) (carrier (alt_group n)) ⊆ carrier (alt_group
n)"
    using group.derived_in_carrier[OF alt_group_is_group] by simp
next
  have aux_lemma: "p ∈ derived (alt_group n) (carrier (alt_group n))"
    if "p ∈ three_cycles n" for p
  proof -
    obtain cs where cs: "p = cycle_of_list cs" "cycle cs" "length cs
= 3" "set cs ⊆ {1..n}"
      using ‹p ∈ three_cycles n› by auto
    then obtain a b c where cs_def: "cs = [ a, b, c ]"
      using stupid_lemma[OF cs(3)] by blast
    have "card (set cs) = 3"
      using cs(2-3)
      by (simp add: distinct_card)

    have "set cs ≠ {1..n}"
      using assms cs(3) unfolding sym[OF distinct_card[OF cs(2)]] by auto
    then obtain d where d: "d ∉ set cs" "d ∈ {1..n}"
      using cs(4) by blast

    hence "cycle (d # cs)" and "length (d # cs) = 4" and "card {1..n}
= n"
      using cs(2-3) by auto
    hence "set (d # cs) ≠ {1..n}"
      using assms unfolding sym[OF distinct_card[OF ‹cycle (d # cs)›]]
      by (metis Suc_n_not_le_n eval_nat_numeral(3))
    then obtain e where e: "e ∉ set (d # cs)" "e ∈ {1..n}"
      using d cs(4) by (metis insert_subset list.simps(15) subsetI subset_antisym)


    define q where "q = (Fun.swap d e id) ∘ (Fun.swap b c id)"
    hence "bij q"
      by (simp add: bij_comp)
    moreover have "q b = c" and "q c = b"
      using d(1) e(1) unfolding q_def cs_def by simp+
    moreover have "q a = a"
      using d(1) e(1) cs(2) unfolding q_def cs_def by auto
    ultimately have "q ∘ p ∘ (inv' q) = cycle_of_list [ a, c, b ]"
      using conjugation_of_cycle[OF cs(2), of q]
      unfolding sym[OF cs(1)] unfolding cs_def by simp
    also have " ... = p ∘ p"
      using cs(2) unfolding cs(1) cs_def
      by (simp add: comp_swap swap_commute transpose_comp_triple)
    finally have "q ∘ p ∘ (inv' q) = p ∘ p" .
    moreover have "bij p"
      unfolding cs(1) cs_def by (simp add: bij_comp)
    ultimately have p: "q ∘ p ∘ (inv' q) ∘ (inv' p) = p"
      by (simp add: bijection.intro bijection.inv_comp_right comp_assoc)
```

```
    have "swapidseq (Suc (Suc 0)) q"
      using comp_Suc[OF comp_Suc[OF id], of b c d e] e(1) cs(2)  unfold-
ing q_def cs_def by auto
    hence "evenperm q"
      using even_Suc_Suc_iff evenperm_unique by blast
    moreover have "q permutes { d, e, b, c }"
      unfolding q_def by (simp add: permutes_compose permutes_swap_id)
    hence "q permutes {1..n}"
      using cs(4) d(2) e(2) permutes_subset unfolding cs_def by fastforce
    ultimately have "q ∈ carrier (alt_group n)"
      unfolding alt_group_carrier by simp
    moreover have "p ∈ carrier (alt_group n)"
      using ‹p ∈ three_cycles n› three_cycles_incl by blast
    ultimately have "p ∈ derived_set (alt_group n) (carrier (alt_group
n))"
      using p alt_group_inv_equality unfolding alt_group_mult
      by (metis (no_types, lifting) UN_iff singletonI)
    thus "p ∈ derived (alt_group n) (carrier (alt_group n))"
      unfolding derived_def by (rule incl)
  qed

  interpret A: group "alt_group n"
    using alt_group_is_group .

  have "generate (alt_group n) (three_cycles n) ⊆ derived (alt_group
n) (carrier (alt_group n))"
    using A.generate_subgroup_incl[OF _ A.derived_is_subgroup] aux_lemma
by (meson subsetI)
  thus "carrier (alt_group n) ⊆ derived (alt_group n) (carrier (alt_group
n))"
    using alt_group_carrier_as_three_cycles by simp
qed

corollary alt_group_is_unsolvable:
  assumes "n ≥ 5" shows "¬ solvable (alt_group n)"
proof (rule ccontr)
  assume "¬ ¬ solvable (alt_group n)"
  then obtain m where "(derived (alt_group n) ^^ m) (carrier (alt_group
n)) = { id }"
    using group.solvable_iff_trivial_derived_seq[OF alt_group_is_group]
unfolding alt_group_one by blast
  moreover have "(derived (alt_group n) ^^ m) (carrier (alt_group n))
= carrier (alt_group n)"
    using derived_alt_group_const[OF assms] by (induct m) (auto)
  ultimately have card_eq_1: "card (carrier (alt_group n)) = 1"
    by simp
  have ge_2: "n ≥ 2"
    using assms by simp
```

```
  moreover have "2 = fact n"
    using alt_group_card_carrier[OF ge_2] unfolding card_eq_1
    by (metis fact_2 mult.right_neutral of_nat_fact)
  ultimately have "n = 2"
      by (metis antisym_conv fact_ge_self)
  thus False
    using assms by simp
qed

corollary sym_group_is_unsolvable:
  assumes "n ≥ 5" shows "¬ solvable (sym_group n)"
proof -
  interpret Id: group_hom "alt_group n" "sym_group n" id
    using group.canonical_inj_is_hom[OF sym_group_is_group alt_group_is_subgroup]
alt_group_def by simp
  show ?thesis
    using Id.inj_hom_imp_solvable alt_group_is_unsolvable[OF assms] by
auto
qed

end
```

# 47 Exact Sequences

```
theory Exact_Sequence
  imports Elementary_Groups Solvable_Groups
begin
```

## 47.1 Definitions

```
inductive exact_seq :: "'a monoid list × ('a ⇒ 'a) list ⇒ bool"  where
unity:     " group_hom G1 G2 f ⟹ exact_seq ([G2, G1], [f])" |
extension: "⟦ exact_seq ((G # K # l), (g # q)); group H ; h ∈ hom G H
;
              kernel G H h = image g (carrier K) ⟧ ⟹ exact_seq (H #
G # K # l, h # g # q)"

inductive_simps exact_seq_end_iff [simp]: "exact_seq ([G,H], (g # q))"
inductive_simps exact_seq_cons_iff [simp]: "exact_seq ((G # K # H # l),
(g # h # q))"

abbreviation exact_seq_arrow ::
  "('a ⇒ 'a) ⇒ 'a monoid list × ('a ⇒ 'a) list ⇒  'a monoid ⇒ 'a
monoid list × ('a ⇒ 'a) list"
  (‹(‹indent=3 notation=‹mixfix exact_seq››_ / ⟶ı _)› [1000, 60])
  where "exact_seq_arrow  f t G ≡ (G # (fst t), f # (snd t))"
```

## 47.2 Basic Properties

```
lemma exact_seq_length1: "exact_seq t ⟹ length (fst t) = Suc (length
(snd t))"
  by (induct t rule: exact_seq.induct) auto

lemma exact_seq_length2: "exact_seq t ⟹ length (snd t) ≥ Suc 0"
  by (induct t rule: exact_seq.induct) auto

lemma dropped_seq_is_exact_seq:
  assumes "exact_seq (G, F)" and "(i :: nat) < length F"
  shows "exact_seq (drop i G, drop i F)"
proof-
  have "exact_seq (drop i (fst t), drop i (snd t))" if "exact_seq t" "i
< length (snd t)" for t i
    using that
  proof (induction arbitrary: i)
    case (unity G1 G2 f) thus ?case
      by (simp add: exact_seq.unity)
  next
    case (extension G K l g q H h) show ?case
    proof (cases)
      assume "i = 0" thus ?case
        using exact_seq.extension[OF extension.hyps] by simp
    next
      assume "i ≠ 0" hence "i ≥ Suc 0" by simp
      then obtain k where "k < length (snd (G # K # l, g # q))" "i =
Suc k"
        using Suc_le_D extension.prems by auto
      thus ?thesis using extension.IH by simp
    qed
  qed
  thus ?thesis using assms by auto
qed

lemma truncated_seq_is_exact_seq:
  assumes "exact_seq (l, q)" and "length l ≥ 3"
  shows "exact_seq (tl l, tl q)"
  using exact_seq_length1[OF assms(1)] dropped_seq_is_exact_seq[OF assms(1),
of "Suc 0"]
        exact_seq_length2[OF assms(1)] assms(2) by (simp add: drop_Suc)

lemma exact_seq_imp_exact_hom:
   assumes "exact_seq (G1 # l,q) ⟶g1 G2 ⟶g2 G3"
   shows "g1 ‘ (carrier G1) = kernel G2 G3 g2"
proof -
  have "(hd (tl (snd t))) ‘ (carrier (hd (tl (tl (fst t))))) =
          kernel (hd (tl (fst t))) (hd (fst t)) (hd (snd t))"
    if "exact_seq t" and "length (fst t) ≥ 3 ∧ length (snd t) ≥ 2" for
t
```

```
    using that
  proof (induction)
    case (unity G1 G2 f)
    then show ?case by auto
  next
    case (extension G l g q H h)
    then show ?case by auto
  qed
  with assms show ?thesis by fastforce
qed

lemma exact_seq_imp_exact_hom_arbitrary:
   assumes "exact_seq (G, F)"
     and "Suc i < length F"
   shows "(F ! (Suc i)) ` (carrier (G ! (Suc (Suc i)))) = kernel (G !
(Suc i)) (G ! i) (F ! i)"
proof -
  have "length (drop i F) ≥ 2" "length (drop i G) ≥ 3"
    using assms(2) exact_seq_length1[OF assms(1)] by auto
  then obtain l q
    where "drop i G = (G ! i) # (G ! (Suc i)) # (G ! (Suc (Suc i))) #
l"
     and  "drop i F = (F ! i) # (F ! (Suc i)) # q"
    by (metis Cons_nth_drop_Suc Suc_less_eq assms exact_seq_length1 fst_conv
        le_eq_less_or_eq le_imp_less_Suc prod.sel(2))
  thus ?thesis
  using dropped_seq_is_exact_seq[OF assms(1), of i] assms(2)
        exact_seq_imp_exact_hom[of "G ! i" "G ! (Suc i)" "G ! (Suc (Suc
i))" l q] by auto
qed

lemma exact_seq_imp_group_hom :
  assumes "exact_seq ((G # l, q)) ⟶g H"
  shows "group_hom G H g"
proof-
  have aux_lemma: "group_hom (hd (tl (fst t))) (hd (fst t)) (hd(snd t))"
if "exact_seq t" for t
    using that
  proof (induction)
    case (unity G1 G2 f)
    then show ?case by auto
  next
    case (extension G l g q H h)
    then show ?case unfolding group_hom_def group_hom_axioms_def by auto
  qed
  show ?thesis using aux_lemma[OF assms]
    by simp
qed
```

```
lemma exact_seq_imp_group_hom_arbitrary:
  assumes "exact_seq (G, F)" and "(i :: nat) < length F"
  shows "group_hom (G ! (Suc i)) (G ! i) (F ! i)"
proof -
  have "length (drop i F) ≥ 1" "length (drop i G) ≥ 2"
    using assms(2) exact_seq_length1[OF assms(1)] by auto
  then obtain l q
    where "drop i G = (G ! i) # (G ! (Suc i)) # l"
     and  "drop i F = (F ! i) # q"
    by (metis Cons_nth_drop_Suc Suc_leI assms exact_seq_length1 fst_conv
        le_eq_less_or_eq le_imp_less_Suc prod.sel(2))
  thus ?thesis
  using dropped_seq_is_exact_seq[OF assms(1), of i] assms(2)
        exact_seq_imp_group_hom[of "G ! i" "G ! (Suc i)" l q "F ! i"]
by simp
qed
```

## 47.3 Link Between Exact Sequences and Solvable Conditions

```
lemma exact_seq_solvable_imp :
  assumes "exact_seq ([G1],[]) ⟶g1 G2 ⟶g2 G3"
    and "inj_on g1 (carrier G1)"
    and "g2 ` (carrier G2) = carrier G3"
  shows "solvable G2 ⟹ (solvable G1) ∧ (solvable G3)"
proof -
  assume G2: "solvable G2"
  have "group_hom G1 G2 g1"
    using exact_seq_imp_group_hom_arbitrary[OF assms(1), of "Suc 0"] by
simp
  hence "solvable G1"
    using group_hom.inj_hom_imp_solvable[of G1 G2 g1] assms(2) G2 by simp
  moreover have "group_hom G2 G3 g2"
    using exact_seq_imp_group_hom_arbitrary[OF assms(1), of 0] by simp
  hence "solvable G3"
    using group_hom.surj_hom_imp_solvable[of G2 G3 g2] assms(3) G2 by
simp
  ultimately show ?thesis by simp
qed

lemma exact_seq_solvable_recip :
  assumes "exact_seq ([G1],[]) ⟶g1 G2 ⟶g2 G3"
    and "inj_on g1 (carrier G1)"
    and "g2 ` (carrier G2) = carrier G3"
  shows "(solvable G1) ∧ (solvable G3) ⟹ solvable G2"
proof -
  assume "(solvable G1) ∧ (solvable G3)"
  hence G1: "solvable G1" and G3: "solvable G3" by auto
  have g1: "group_hom G1 G2 g1" and g2: "group_hom G2 G3 g2"
    using exact_seq_imp_group_hom_arbitrary[OF assms(1), of "Suc 0"]
```

```
          exact_seq_imp_group_hom_arbitrary[OF assms(1), of 0] by auto
  show ?thesis
    using solvable_condition[OF g1 g2 assms(3)]
          exact_seq_imp_exact_hom[OF assms(1)] G1 G3 by auto
qed

proposition exact_seq_solvable_iff :
  assumes "exact_seq ([G1],[]) ⟶g1 G2 ⟶g2 G3"
    and "inj_on g1 (carrier G1)"
    and "g2 ` (carrier G2) = carrier G3"
  shows "(solvable G1) ∧ (solvable G3) ⟷  solvable G2"
  using exact_seq_solvable_recip exact_seq_solvable_imp assms by blast


lemma exact_seq_eq_triviality:
  assumes "exact_seq ([E,D,C,B,A], [k,h,g,f])"
  shows "trivial_group C ⟷ f ` carrier A = carrier B ∧ inj_on k (carrier
D)" (is "_ = ?rhs")
proof
  assume C: "trivial_group C"
  with assms have "inj_on k (carrier D)"
    apply (auto simp: group_hom.image_from_trivial_group trivial_group_def
hom_one)
    apply (simp add: group_hom_def group_hom_axioms_def group_hom.inj_iff_trivial_ker)
    done
  with assms C show ?rhs
    apply (auto simp: group_hom.image_from_trivial_group trivial_group_def
hom_one)
     apply (auto simp: group_hom_def group_hom_axioms_def hom_def kernel_def)
    done
next
  assume ?rhs
  with assms show "trivial_group C"
    apply (simp add: trivial_group_def)
    by (metis group_hom.inj_iff_trivial_ker group_hom.trivial_hom_iff
group_hom_axioms.intro group_hom_def)
qed

lemma exact_seq_imp_triviality:
   "⟦exact_seq ([E,D,C,B,A], [k,h,g,f]); f ∈ iso A B; k ∈ iso D E⟧ ⟹
trivial_group C"
  by (metis (no_types, lifting) Group.iso_def bij_betw_def exact_seq_eq_triviality
mem_Collect_eq)

lemma exact_seq_epi_eq_triviality:
   "exact_seq ([D,C,B,A], [h,g,f]) ⟹ (f ` carrier A = carrier B) ⟷
trivial_homomorphism B C g"
  by (auto simp: trivial_homomorphism_def kernel_def)
```

```
lemma exact_seq_mon_eq_triviality:
   "exact_seq ([D,C,B,A], [h,g,f]) ⟹ inj_on h (carrier C) ⟷ trivial_homomorphism
B C g"
  by (auto simp: trivial_homomorphism_def kernel_def group.is_monoid inj_on_one_iff'
image_def) blast

lemma exact_sequence_sum_lemma:
  assumes "comm_group G" and h: "h ∈ iso A C" and k: "k ∈ iso B D"
    and ex: "exact_seq ([D,G,A], [g,i])" "exact_seq ([C,G,B], [f,j])"
    and fih: "⋀x. x ∈ carrier A ⟹ f(i x) = h x"
    and gjk: "⋀x. x ∈ carrier B ⟹ g(j x) = k x"
  shows "(λ(x, y). i x ⊗G j y) ∈ Group.iso (A ×× B) G ∧ (λz. (f z,
g z)) ∈ Group.iso G (C ×× D)"
    (is "?ij ∈ _ ∧ ?gf ∈ _")
proof (rule epi_iso_compose_rev)
  interpret comm_group G
    by (rule assms)
  interpret f: group_hom G C f
    using ex by (simp add: group_hom_def group_hom_axioms_def)
  interpret g: group_hom G D g
    using ex by (simp add: group_hom_def group_hom_axioms_def)
  interpret i: group_hom A G i
    using ex by (simp add: group_hom_def group_hom_axioms_def)
  interpret j: group_hom B G j
    using ex by (simp add: group_hom_def group_hom_axioms_def)
  have kerf: "kernel G C f = j ` carrier B" and "group A" "group B" "i
∈ hom A G"
    using ex by (auto simp: group_hom_def group_hom_axioms_def)
  then obtain h' where "h' ∈ hom C A" "(∀x ∈ carrier A. h'(h x) = x)"
    and hh': "(∀y ∈ carrier C. h(h' y) = y)" and "group_isomorphisms
A C h h'"
    using h by (auto simp: group.iso_iff_group_isomorphisms group_isomorphisms_def)
  have homij: "?ij ∈ hom (A ×× B) G"
    unfolding case_prod_unfold
    apply (rule hom_group_mult)
    using ex by (simp_all add: group_hom_def hom_of_fst [unfolded o_def]
hom_of_snd [unfolded o_def])
  show homgf: "?gf ∈ hom G (C ×× D)"
    using ex by (simp add: hom_paired)
  show "?ij ∈ epi (A ×× B) G"
  proof (clarsimp simp add: epi_iff_subset homij)
    fix x
    assume x: "x ∈ carrier G"
    with ‹i ∈ hom A G› ‹h' ∈ hom C A› have "x ⊗G invG(i(h'(f x))) ∈
kernel G C f"
      by (simp add: kernel_def hom_in_carrier hh' fih)
    with kerf obtain y where y: "y ∈ carrier B" "j y = x ⊗G invG(i(h'(f
x)))"
      by auto
```

```
    have "i (h' (f x)) ⊗G (x ⊗G invG i (h' (f x))) = x ⊗G (i (h' (f
x)) ⊗G invG i (h' (f x)))"
      by (meson ‹h' ∈ hom C A› x f.hom_closed hom_in_carrier i.hom_closed
inv_closed m_lcomm)
    also have "... = x"
      using ‹h' ∈ hom C A› hom_in_carrier x by fastforce
    finally show "x ∈ (λ(x, y). i x ⊗G j y) ‘ (carrier A × carrier B)"
      using x y apply (clarsimp simp: image_def)
      apply (rule_tac x="h'(f x)" in bexI)
       apply (rule_tac x=y in bexI, auto)
      by (meson ‹h' ∈ hom C A› f.hom_closed hom_in_carrier)
  qed
  show "(λz. (f z, g z)) ∘ (λ(x, y). i x ⊗G j y) ∈ Group.iso (A ××
B) (C ×× D)"
    apply (rule group.iso_eq [where f = "λ(x,y). (h x,k y)"])
    using ex
    apply (auto simp: group_hom_def group_hom_axioms_def DirProd_group
iso_paired2 h k fih gjk kernel_def set_eq_iff)
     apply (metis f.hom_closed f.r_one fih imageI)
    apply (metis g.hom_closed g.l_one gjk imageI)
    done
qed
```

## 47.4 Splitting lemmas and Short exact sequences

Ported from HOL Light by LCP

```
definition short_exact_sequence
  where "short_exact_sequence A B C f g ≡ ∃T1 T2 e1 e2. exact_seq ([T1,A,B,C,T2],
[e1,f,g,e2]) ∧ trivial_group T1 ∧ trivial_group T2"

lemma short_exact_sequenceD:
  assumes "short_exact_sequence A B C f g" shows "exact_seq ([A,B,C],
[f,g]) ∧ f ∈ epi B A ∧ g ∈ mon C B"
  using assms
  apply (auto simp: short_exact_sequence_def group_hom_def group_hom_axioms_def)
  apply (simp add: epi_iff_subset group_hom.intro group_hom.kernel_to_trivial_group
group_hom_axioms.intro)
  by (metis (no_types, lifting) group_hom.inj_iff_trivial_ker group_hom.intro
group_hom_axioms.intro
      hom_one image_empty image_insert mem_Collect_eq mon_def trivial_group_def)

lemma short_exact_sequence_iff:
  "short_exact_sequence A B C f g ⟷ exact_seq ([A,B,C], [f,g]) ∧ f
∈ epi B A ∧ g ∈ mon C B"
proof -
  have "short_exact_sequence A B C f g"
    if "exact_seq ([A, B, C], [f, g])" and "f ∈ epi B A" and "g ∈ mon
C B"
  proof -
```

```
    show ?thesis
      unfolding short_exact_sequence_def
    proof (intro exI conjI)
      have "kernel A (singleton_group 1_A) (λx. 1_A) = f ` carrier B"
        using that by (simp add: kernel_def singleton_group_def epi_def)
      moreover have "kernel C B g = {1_C}"
        using that group_hom.inj_iff_trivial_ker mon_def by fastforce
      ultimately show "exact_seq ([singleton_group (one A), A, B, C, singleton_group
(one C)], [λx. 1_A, f, g, id])"
        using that
        by (simp add: group_hom_def group_hom_axioms_def group.id_hom_singleton)
    qed auto
qed
  then show ?thesis
    using short_exact_sequenceD by blast
qed

lemma very_short_exact_sequence:
  assumes "exact_seq ([D,C,B,A], [h,g,f])" "trivial_group A" "trivial_group
D"
  shows "g ∈ iso B C"
  using assms
  apply simp
  by (metis (no_types, lifting) group_hom.image_from_trivial_group group_hom.iso_iff
      group_hom.kernel_to_trivial_group group_hom.trivial_ker_imp_inj
group_hom_axioms.intro group_hom_def hom_carrier inj_on_one_iff')

lemma splitting_sublemma_gen:
  assumes ex: "exact_seq ([C,B,A], [g,f])" and fim: "f ` carrier A =
H"
      and "subgroup K B" and 1: "H ∩ K ⊆ {one B}" and eq: "set_mult
B H K = carrier B"
  shows "g ∈ iso (subgroup_generated B K) (subgroup_generated C(g ` carrier
B))"
proof -
  interpret KB: subgroup K B
    by (rule assms)
  interpret fAB: group_hom A B f
    using ex by simp
  interpret gBC: group_hom B C g
    using ex by (simp add: group_hom_def group_hom_axioms_def)
  have "group A" "group B" "group C" and kerg: "kernel B C g = f ` carrier
A"
      using ex by (auto simp: group_hom_def group_hom_axioms_def)
  have ker_eq: "kernel B C g = H"
    using ex by (simp add: fim)
  then have "subgroup H B"
    using ex by (simp add: group_hom.img_is_subgroup)
  show ?thesis
```

```
    unfolding iso_iff
  proof (intro conjI)
    show "g ∈ hom (subgroup_generated B K) (subgroup_generated C(g ‘
carrier B))"
      by (metis ker_eq ‹subgroup K B› eq gBC.hom_between_subgroups gBC.set_mult_ker_hom(2)
order_refl subgroup.subset)
    show "g ‘ carrier (subgroup_generated B K) = carrier (subgroup_generated
C(g ‘ carrier B))"
      by (metis assms(3) eq fAB.H.subgroupE(1) gBC.img_is_subgroup gBC.set_mult_ker_hom(2)
ker_eq subgroup.carrier_subgroup_generated_subgroup)
    interpret gKBC: group_hom "subgroup_generated B K" C g
      apply (auto simp: group_hom_def group_hom_axioms_def ‹group C›)
      by (simp add: fAB.H.hom_from_subgroup_generated gBC.homh)
    have *: "x = 1_B"
      if x: "x ∈ carrier (subgroup_generated B K)" and "g x = 1_C" for
x
    proof -
      have x': "x ∈ carrier B"
        using that fAB.H.carrier_subgroup_generated_subset by blast
      moreover have "x ∈ H"
        using kerg fim x' that by (auto simp: kernel_def set_eq_iff)
      ultimately show ?thesis
        by (metis "1" x Int_iff singletonD KB.carrier_subgroup_generated_subgroup
subsetCE)
    qed
    show "inj_on g (carrier (subgroup_generated B K))"
      using "*" gKBC.inj_on_one_iff by auto
  qed
qed

lemma splitting_sublemma:
  assumes ex: "short_exact_sequence C B A g f" and fim: "f ‘ carrier
A = H"
      and "subgroup K B" and 1: "H ∩ K ⊆ {one B}" and eq: "set_mult
B H K = carrier B"
    shows "f ∈ iso A (subgroup_generated B H)" (is ?f)
          "g ∈ iso (subgroup_generated B K) C" (is ?g)
proof -
  show ?f
    using short_exact_sequenceD [OF ex]
    apply (clarsimp simp add: group_hom_def group.iso_onto_image)
    using fim group.iso_onto_image by blast
  have "C = subgroup_generated C(g ‘ carrier B)"
    using short_exact_sequenceD [OF ex]
    apply simp
    by (metis epi_iff_subset group.subgroup_generated_group_carrier hom_carrier
subset_antisym)
  then show ?g
    using short_exact_sequenceD [OF ex]
```

```
    by (metis "1" ‹subgroup K B› eq fim splitting_sublemma_gen)
qed

lemma splitting_lemma_left_gen:
  assumes ex: "exact_seq ([C,B,A], [g,f])" and f': "f' ∈ hom B A" and
iso: "(f' ∘ f) ∈ iso A A"
    and injf: "inj_on f (carrier A)" and surj: "g ` carrier B = carrier
C"
 obtains H K where "H ◁ B" "K ◁ B" "H ∩ K ⊆ {one B}" "set_mult B H K
= carrier B"
                   "f ∈ iso A (subgroup_generated B H)" "g ∈ iso (subgroup_generated
B K) C"
proof -
  interpret gBC: group_hom B C g
    using ex by (simp add: group_hom_def group_hom_axioms_def)
  have "group A" "group B" "group C" and kerg: "kernel B C g = f ` carrier
A"
    using ex by (auto simp: group_hom_def group_hom_axioms_def)
  then have *: "f ` carrier A ∩ kernel B A f' = {1B} ∧ f ` carrier A
<#>B kernel B A f' = carrier B"
    using group_semidirect_sum_image_ker [of f A B f' A] assms by auto
  interpret f'AB: group_hom B A f'
    using assms by (auto simp: group_hom_def group_hom_axioms_def)
  let ?H = "f ` carrier A"
  let ?K = "kernel B A f'"
  show thesis
  proof
    show "?H ◁ B"
      by (simp add: gBC.normal_kernel flip: kerg)
    show "?K ◁ B"
      by (rule f'AB.normal_kernel)
    show "?H ∩ ?K ⊆ {1B}" "?H <#>B ?K = carrier B"
      using * by auto
    show "f ∈ Group.iso A (subgroup_generated B ?H)"
      using ex by (simp add: injf iso_onto_image group_hom_def group_hom_axioms_def)
    have C: "C = subgroup_generated C(g ` carrier B)"
      using surj by (simp add: gBC.subgroup_generated_group_carrier)
    show "g ∈ Group.iso (subgroup_generated B ?K) C"
      apply (subst C)
      apply (rule splitting_sublemma_gen [OF ex refl])
      using * by (auto simp: f'AB.subgroup_kernel)
  qed
qed

lemma splitting_lemma_left:
  assumes ex: "exact_seq ([C,B,A], [g,f])" and f': "f' ∈ hom B A"
    and inv: "(⋀x. x ∈ carrier A ⟹ f'(f x) = x)"
    and injf: "inj_on f (carrier A)" and surj: "g ` carrier B = carrier
C"
```

```
 obtains H K where "H ◁ B" "K ◁ B" "H ∩ K ⊆ {one B}" "set_mult B H K
= carrier B"
                  "f ∈ iso A (subgroup_generated B H)" "g ∈ iso (subgroup_generated
B K) C"
proof -
  interpret fAB: group_hom A B f
    using ex by simp
  interpret gBC: group_hom B C g
    using ex by (simp add: group_hom_def group_hom_axioms_def)
  have "group A" "group B" "group C" and kerg: "kernel B C g = f ‘ carrier
A"
      using ex by (auto simp: group_hom_def group_hom_axioms_def)
  have iso: "f’ ∘ f ∈ Group.iso A A"
    using ex by (auto simp: inv intro:  group.iso_eq [OF ‹group A› id_iso])
  show thesis
    by (metis that splitting_lemma_left_gen [OF ex f’ iso injf surj])
qed

lemma splitting_lemma_right_gen:
  assumes ex: "short_exact_sequence C B A g f" and g’: "g’ ∈ hom C B"
and iso: "(g ∘ g’) ∈ iso C C"
 obtains H K where "H ◁ B" "subgroup K B" "H ∩ K ⊆ {one B}" "set_mult
B H K = carrier B"
                  "f ∈ iso A (subgroup_generated B H)" "g ∈ iso (subgroup_generated
B K) C"
proof
  interpret fAB: group_hom A B f
    using short_exact_sequenceD [OF ex] by (simp add: group_hom_def group_hom_axioms_def)
  interpret gBC: group_hom B C g
    using short_exact_sequenceD [OF ex] by (simp add: group_hom_def group_hom_axioms_def)
  have *: "f ‘ carrier A ∩ g’ ‘ carrier C = {1_B}"
          "f ‘ carrier A <#>_B g’ ‘ carrier C = carrier B"
          "group A" "group B" "group C"
          "kernel B C g = f ‘ carrier A"
    using group_semidirect_sum_ker_image [of g g’ C C B] short_exact_sequenceD
[OF ex]
    by (simp_all add: g’ iso group_hom_def)
  show "kernel B C g ◁ B"
    by (simp add: gBC.normal_kernel)
  show "(kernel B C g) ∩ (g’ ‘ carrier C) ⊆ {1_B}" "(kernel B C g) <#>_B
(g’ ‘ carrier C) = carrier B"
    by (auto simp: *)
  show "f ∈ Group.iso A (subgroup_generated B (kernel B C g))"
    by (metis "*"(6) fAB.group_hom_axioms group.iso_onto_image group_hom_def
short_exact_sequenceD [OF ex])
  show "subgroup (g’ ‘ carrier C) B"
    using splitting_sublemma
    by (simp add: fAB.H.is_group g’ gBC.is_group group_hom.img_is_subgroup
group_hom_axioms_def group_hom_def)
```

```
  then show "g ∈ Group.iso (subgroup_generated B (g' ` carrier C)) C"
    by (metis (no_types, lifting) iso_iff fAB.H.hom_from_subgroup_generated
gBC.homh image_comp inj_on_imageI iso subgroup.carrier_subgroup_generated_subgroup)
qed

lemma splitting_lemma_right:
  assumes ex: "short_exact_sequence C B A g f" and g': "g' ∈ hom C B"
and gg': "⋀z. z ∈ carrier C ⟹ g(g' z) = z"
 obtains H K where "H ◁ B" "subgroup K B" "H ∩ K ⊆ {one B}" "set_mult
B H K = carrier B"
                   "f ∈ iso A (subgroup_generated B H)" "g ∈ iso (subgroup_generated
B K) C"
proof -
  have *: "group A" "group B" "group C"
    using group_semidirect_sum_ker_image [of g g' C C B] short_exact_sequenceD
[OF ex]
    by (simp_all add: g'  group_hom_def)
  show thesis
    apply (rule splitting_lemma_right_gen [OF ex g' group.iso_eq [OF _
id_iso]])
    using * apply (auto simp: gg' intro: that)
    done
qed


end
theory Left_Coset
imports Coset




begin

definition
  LCOSETS  :: "[_, 'a set] ⇒ ('a set)set"
    (‹(‹open_block notation=‹prefix lcosets››lcosetsı _)› [81] 80)
  where "lcosets_G H = (⋃a∈carrier G. {a <#_G H})"

definition
  LFactGroup :: "[('a,'b) monoid_scheme, 'a set] ⇒ ('a set) monoid" (infixl
‹LMod› 65)
    — Actually defined for groups rather than monoids
   where "LFactGroup G H = ⦇carrier = lcosets_G H, mult = set_mult G,
one = H⦈"

lemma (in group) lcos_self: "[| x ∈ carrier G; subgroup H G |] ==> x
∈ x <# H"
  by (simp add: group_l_invI subgroup.lcos_module_rev subgroup.one_closed)
```

Elements of a left coset are in the carrier

```
lemma (in subgroup) elemlcos_carrier:
  assumes "group G" "a ∈ carrier G" "a' ∈ a <# H"
  shows "a' ∈ carrier G"
  by (meson assms group.l_coset_carrier subgroup_axioms)
```

Step one for lemma `rcos_module`

```
lemma (in subgroup) lcos_module_imp:
  assumes "group G"
  assumes xcarr: "x ∈ carrier G"
      and x'cos: "x' ∈ x <# H"
  shows "(inv x ⊗ x') ∈ H"
proof -
  interpret group G by fact
  obtain h
    where hH: "h ∈ H" and x': "x' = x ⊗ h" and hcarr: "h ∈ carrier
G"
    using assms by (auto simp: l_coset_def)
  have "(inv x) ⊗ x' = (inv x) ⊗ (x ⊗ h)"
    by (simp add: x')
  have "... = (x ⊗ inv x) ⊗ h"
    by (metis hcarr inv_closed inv_inv l_inv m_assoc xcarr)
  also have "... = h"
    by (simp add: hcarr xcarr)
  finally have "(inv x) ⊗ x' = h"
    using x' by metis
  then show "(inv x) ⊗ x' ∈ H"
    using hH by force
qed
```

Left cosets are subsets of the carrier.

```
lemma (in subgroup) lcosets_carrier:
  assumes "group G"
  assumes XH: "X ∈ lcosets H"
  shows "X ⊆ carrier G"
proof -
  interpret group G by fact
  show "X ⊆ carrier G"
    using XH l_coset_subset_G subset by (auto simp: LCOSETS_def)
qed

lemma (in group) lcosets_part_G:
  assumes "subgroup H G"
  shows "⋃(lcosets H) = carrier G"
proof -
  interpret subgroup H G by fact
  show ?thesis
  proof
    show "⋃ (lcosets H) ⊆ carrier G"
      by (force simp add: LCOSETS_def l_coset_def)
```

```
    show "carrier G ⊆ ⋃ (lcosets H)"
      by (auto simp add: LCOSETS_def intro: lcos_self assms)
  qed
qed

lemma (in group) lcosets_subset_PowG:
     "subgroup H G  ⟹ lcosets H ⊆ Pow(carrier G)"
  using lcosets_part_G subset_Pow_Union by blast

lemma (in group) lcos_disjoint:
  assumes "subgroup H G"
  assumes p: "a ∈ lcosets H" "b ∈ lcosets H" "a≠b"
  shows "a ∩ b = {}"
proof -
  interpret subgroup H G by fact
  show ?thesis
    using p l_repr_independence subgroup_axioms unfolding LCOSETS_def
disjoint_iff
    by blast
qed
```

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

```
lemma (in group) inj_on_f':
    "⟦H ⊆ carrier G;  a ∈ carrier G⟧ ⟹ inj_on (λy. y ⊗ inv a) (a <#
H)"
  by (simp add: inj_on_g l_coset_subset_G)

lemma (in group) inj_on_f'':
    "⟦H ⊆ carrier G;  a ∈ carrier G⟧ ⟹ inj_on (λy. inv a ⊗ y) (a <#
H)"
  by (meson inj_on_cmult inv_closed l_coset_subset_G subset_inj_on)

lemma (in group) inj_on_g':
    "⟦H ⊆ carrier G;  a ∈ carrier G⟧ ⟹ inj_on (λy. a ⊗ y) H"
  using inj_on_cmult inj_on_subset by blast

lemma (in group) l_card_cosets_equal:
  assumes "c ∈ lcosets H" and H: "H ⊆ carrier G" and fin: "finite(carrier
G)"
  shows "card H = card c"
proof -
  obtain x where x: "x ∈ carrier G" and c: "c = x <# H"
    using assms by (auto simp add: LCOSETS_def)
  have "inj_on ((⊗) x) H"
    by (simp add: H group.inj_on_g' x)
  moreover
  have "(⊗) x ` H ⊆ x <# H"
    by (force simp add: m_assoc subsetD l_coset_def)
  moreover
```

```
  have "inj_on ((⊗) (inv x)) (x <# H)"
    by (simp add: H group.inj_on_f'' x)
  moreover
  have "⋀h. h ∈ H ⟹ inv x ⊗ (x ⊗ h) ∈ H"
    by (metis H in_mono inv_solve_left m_closed x)
  then have "(⊗) (inv x) ` (x <# H) ⊆ H"
    by (auto simp: l_coset_def)
  ultimately show ?thesis
    by (metis H fin c card_bij_eq finite_imageD finite_subset)
qed

theorem (in group) l_lagrange:
  assumes "finite(carrier G)" "subgroup H G"
  shows "card(lcosets H) * card(H) = order(G)"
proof -
  have "card H * card (lcosets H) = card (⋃ (lcosets H))"
    using card_partition
    by (metis (no_types, lifting) assms finite_UnionD l_card_cosets_equal
lcos_disjoint lcosets_part_G subgroup.subset)
  then show ?thesis
    by (simp add: assms(2) lcosets_part_G mult.commute order_def)
qed

end


theory SimpleGroups
imports Coset "HOL-Computational_Algebra.Primes"
begin
```

## 48 Simple Groups

```
locale simple_group = group +
  assumes order_gt_one: "order G > 1"
  assumes no_real_normal_subgroup: "⋀H. H ◁ G ⟹ (H = carrier G ∨ H
= {1})"

lemma (in simple_group) is_simple_group: "simple_group G"
  by (rule simple_group_axioms)
```

Simple groups are non-trivial.

```
lemma (in simple_group) simple_not_triv: "carrier G ≠ {1}"
  using order_gt_one unfolding order_def by auto
```

Every group of prime order is simple

```
lemma (in group) prime_order_simple:
  assumes prime: "prime (order G)"
  shows "simple_group G"
```

```
proof
  from prime show "1 < order G"
    unfolding prime_nat_iff by auto
next
  fix H
  assume "H ◁ G"
  hence HG: "subgroup H G" unfolding normal_def by simp
  hence "card H dvd order G"
    by (metis dvd_triv_right lagrange)
  with prime have "card H = 1 ∨ card H = order G"
    unfolding prime_nat_iff by simp
  thus "H = carrier G ∨ H = {1}"
  proof
    assume "card H = 1"
    moreover from HG have "1 ∈ H" by (metis subgroup.one_closed)
    ultimately show ?thesis by (auto simp: card_Suc_eq)
  next
    assume "card H = order G"
    moreover from HG have "H ⊆ carrier G" unfolding subgroup_def by
simp
    moreover from prime have "finite (carrier G)"
      using order_gt_0_iff_finite by force
    ultimately show ?thesis
      unfolding order_def by (metis card_subset_eq)
  qed
qed
```

Being simple is a property that is preserved by isomorphisms.

```
lemma (in simple_group) iso_simple:
  assumes H: "group H"
  assumes iso: "φ ∈ iso G H"
  shows "simple_group H"
unfolding simple_group_def simple_group_axioms_def
proof (intro conjI strip H)
  from iso have "order G = order H" unfolding iso_def order_def using
bij_betw_same_card by auto
  with order_gt_one show "1 < order H" by simp
next
  have inv_iso: "(inv_into (carrier G) φ) ∈ iso H G" using iso
    by (simp add: iso_set_sym)
  fix N
  assume NH: "N ◁ H"
  then interpret Nnormal: normal N H by simp
  define M where "M = (inv_into (carrier G) φ) ` N"
  hence MG: "M ◁ G"
    using inv_iso NH H by (metis is_group iso_normal_subgroup)
  have surj: "φ ` carrier G = carrier H"
    using iso unfolding iso_def bij_betw_def by simp
  hence MN: "φ ` M = N"
```

```
    unfolding M_def using Nnormal.subset image_inv_into_cancel by metis
  then have "N = {1H}" if "M = {1}"
    using Nnormal.subgroup_axioms subgroup.one_closed that by force
  then show "N = carrier H ∨ N = {1H}"
    by (metis MG MN no_real_normal_subgroup surj)
qed
```

As a corollary of this: Factorizing a group by itself does not result in a simple group!

```
lemma (in group) self_factor_not_simple: "¬ simple_group (G Mod (carrier
G))"
proof
  assume assm: "simple_group (G Mod (carrier G))"
  with self_factor_iso simple_group.iso_simple have "simple_group (G⦇carrier
:= {1}⦈)"
    using subgroup_imp_group triv_subgroup by blast
  thus False
    using simple_group.simple_not_triv by force
qed

end


theory SndIsomorphismGrp
imports Coset
begin
```

# 49 The Second Isomorphism Theorem for Groups

This theory provides a proof of the second isomorphism theorems for groups. The theorems consist of several facts about normal subgroups.

The first lemma states that whenever we have a subgroup `S` and a normal subgroup `H` of a group `G`, their intersection is normal in `G`

```
locale second_isomorphism_grp = normal +
  fixes S:: "'a set"
  assumes subgrpS: "subgroup S G"

context second_isomorphism_grp
begin

interpretation groupS: group "G⦇carrier := S⦈"
  using subgrpS
  by (metis subgroup_imp_group)

lemma normal_subgrp_intersection_normal:
  shows "S ∩ H ⊲ (G⦇carrier := S⦈)"
proof(auto simp: groupS.normal_inv_iff)
```

```
  from subgrpS is_subgroup have "⋀x. x ∈ {S, H} ⟹ subgroup x G" by
auto
  hence "subgroup (⋂ {S, H}) G" using subgroups_Inter by blast
  hence "subgroup (S ∩ H) G" by auto
  moreover have "S ∩ H ⊆ S" by simp
  ultimately show "subgroup (S ∩ H) (G⦇carrier := S⦈)"
    by (simp add: subgroup_incl subgrpS)
next
  fix g h
  assume g: "g ∈ S" and hH: "h ∈ H" and hS: "h ∈ S"
  from g hH subgrpS show "g ⊗ h ⊗ inv_G⦇carrier := S⦈ g ∈ H"
    by (metis inv_op_closed2 subgroup.mem_carrier m_inv_consistent)
  from g hS subgrpS show "g ⊗ h ⊗ inv_G⦇carrier := S⦈ g ∈ S"
    by (metis subgroup.m_closed subgroup.m_inv_closed m_inv_consistent)
qed

lemma normal_set_mult_subgroup:
  shows "subgroup (H <#> S) G"
proof(rule subgroupI)
  show "H <#> S ⊆ carrier G"
    by (metis setmult_subset_G subgroup.subset subgrpS subset)
next
  have "1 ∈ H" "1 ∈ S"
    using is_subgroup subgrpS subgroup.one_closed by auto
  hence "1 ⊗ 1 ∈ H <#> S"
    unfolding set_mult_def by blast
  thus "H <#> S ≠ {}" by auto
next
  fix g
  assume g: "g ∈ H <#> S"
  then obtain h s where h: "h ∈ H" and s: "s ∈ S" and ghs: "g = h ⊗
s" unfolding set_mult_def
    by auto
  hence "s ∈ carrier G" by (metis subgroup.mem_carrier subgrpS)
  with h ghs obtain h' where h': "h' ∈ H" and "g = s ⊗ h'"
    using coset_eq unfolding r_coset_def l_coset_def by auto
  with s have "inv g = (inv h') ⊗ (inv s)"
    by (metis inv_mult_group mem_carrier subgroup.mem_carrier subgrpS)
  moreover from h' s subgrpS have "inv h' ∈ H" "inv s ∈ S"
    using subgroup.m_inv_closed m_inv_closed by auto
  ultimately show "inv g ∈ H <#> S"
    unfolding set_mult_def by auto
next
  fix g g'
  assume g: "g ∈ H <#> S" and h: "g' ∈ H <#> S"
  then obtain h h' s s' where hh'ss': "h ∈ H" "h' ∈ H" "s ∈ S" "s' ∈
S" and "g = h ⊗ s" and "g' = h' ⊗ s'"
    unfolding set_mult_def by auto
  hence "g ⊗ g' = (h ⊗ s) ⊗ (h' ⊗ s')" by metis
```

```
  also from hh'ss' have inG: "h ∈ carrier G" "h' ∈ carrier G" "s ∈ carrier
G" "s' ∈ carrier G"
    using subgrpS mem_carrier subgroup.mem_carrier by force+
  hence "(h ⊗ s) ⊗ (h' ⊗ s') = h ⊗ (s ⊗ h') ⊗ s'"
    using m_assoc by auto
  also from hh'ss' inG obtain h'' where h'': "h'' ∈ H" and "s ⊗ h' =
h'' ⊗ s"
    using coset_eq unfolding r_coset_def l_coset_def
    by fastforce
  hence "h ⊗ (s ⊗ h') ⊗ s' = h ⊗ (h'' ⊗ s) ⊗ s'"
    by simp
  also from h'' inG have "... = (h ⊗ h'') ⊗ (s ⊗ s')"
    using m_assoc mem_carrier by auto
  finally have "g ⊗ g' = h ⊗ h'' ⊗ (s ⊗ s')".
  moreover have "... ∈ H <#> S"
    unfolding set_mult_def using h'' hh'ss' subgrpS subgroup.m_closed
by fastforce
  ultimately show "g ⊗ g' ∈ H <#> S"
    by simp
qed

lemma H_contained_in_set_mult:
  shows "H ⊆ H <#> S"
proof
  fix x
  assume x: "x ∈ H"
  have "x ⊗ 1 ∈ H <#> S" unfolding set_mult_def
    using second_isomorphism_grp.subgrpS second_isomorphism_grp_axioms
subgroup.one_closed x by force
  with x show "x ∈ H <#> S" by (metis mem_carrier r_one)
qed

lemma S_contained_in_set_mult:
  shows "S ⊆ H <#> S"
proof
  fix s
  assume s: "s ∈ S"
  then have "1 ⊗ s ∈ H <#> S" unfolding set_mult_def by force
  with s show "s ∈ H <#> S" using subgrpS subgroup.mem_carrier l_one
by force
qed

lemma normal_intersection_hom:
  shows "group_hom (G(|carrier := S|)) ((G(|carrier := H <#> S|)) Mod H) (λg.
H #> g)"
proof -
  have "group ((G(|carrier := H <#> S|)) Mod H)"
    by (simp add: H_contained_in_set_mult normal.factorgroup_is_group
normal_axioms
```

```
      normal_restrict_supergroup normal_set_mult_subgroup)
  moreover have "H #> g ∈ carrier ((G⦇carrier := H <#> S⦈) Mod H)" if
g: "g ∈ S" for g
  proof -
    from g that have "g ∈ H <#> S"
      using S_contained_in_set_mult by blast
    thus "H #> g ∈ carrier ((G⦇carrier := H <#> S⦈) Mod H)"
      unfolding FactGroup_def RCOSETS_def r_coset_def by auto
  qed
  moreover have "⋀x y. ⟦x ∈ S; y ∈ S⟧ ⟹ H #> x ⊗ y = H #> x <#> (H
#> y)"
    using normal.rcos_sum normal_axioms subgroup.mem_carrier subgrpS by
fastforce
  ultimately show ?thesis
    by (auto simp: group_hom_def group_hom_axioms_def hom_def)
qed

lemma normal_intersection_hom_kernel:
  shows "kernel (G⦇carrier := S⦈) ((G⦇carrier := H <#> S⦈) Mod H) (λg.
H #> g) = H ∩ S"
proof -
  have "kernel (G⦇carrier := S⦈) ((G⦇carrier := H <#> S⦈) Mod H) (λg.
H #> g)
      = {g ∈ S. H #> g = 𝟭(G⦇carrier := H <#> S⦈) Mod H}"
    unfolding kernel_def by auto
  also have "... = {g ∈ S. H #> g = H}"
    unfolding FactGroup_def by auto
  also have "... = {g ∈ S. g ∈ H}"
    by (meson coset_join1 is_group rcos_const subgroupE(1) subgroup_axioms
subgrpS subset_eq)
  also have "... = H ∩ S" by auto
  finally show ?thesis.
qed

lemma normal_intersection_hom_surj:
  shows "(λg. H #> g) ` carrier (G⦇carrier := S⦈) = carrier ((G⦇carrier
:= H <#> S⦈) Mod H)"
proof auto
  fix g
  assume "g ∈ S"
  hence "g ∈ H <#> S"
    using S_contained_in_set_mult by auto
  thus "H #> g ∈ carrier ((G⦇carrier := H <#> S⦈) Mod H)"
    unfolding FactGroup_def RCOSETS_def r_coset_def by auto
next
  fix x
  assume "x ∈ carrier (G⦇carrier := H <#> S⦈ Mod H)"
  then obtain h s where h: "h ∈ H" and s: "s ∈ S" and "x = H #> (h
⊗ s)"
```

```
    unfolding FactGroup_def RCOSETS_def r_coset_def set_mult_def by auto
  hence "x = (H #> h) #> s"
    by (metis h s coset_mult_assoc mem_carrier subgroup.mem_carrier subgrpS
subset)
  also have "... = H #> s"
    by (metis h is_group rcos_const)
  finally have "x = H #> s".
  with s show "x ∈ (#>) H ‘ S"
    by simp
qed
```

Finally we can prove the actual isomorphism theorem:

```
theorem normal_intersection_quotient_isom:
  shows "(λX. the_elem ((λg. H #> g) ‘ X)) ∈ iso ((G(|carrier := S|)) Mod
(H ∩ S)) (((G(|carrier := H <#> S|))) Mod H)"
using normal_intersection_hom_kernel[symmetric] normal_intersection_hom
normal_intersection_hom_surj
by (metis group_hom.FactGroup_iso_set)

end


corollary (in group) normal_subgroup_set_mult_closed:
  assumes "M ◁ G" and "N ◁ G"
  shows "M <#> N ◁ G"
proof (rule normalI)
  from assms show "subgroup (M <#> N) G"
    using second_isomorphism_grp.normal_set_mult_subgroup normal_imp_subgroup
    unfolding second_isomorphism_grp_def second_isomorphism_grp_axioms_def
by force
next
  show "∀x∈carrier G. M <#> N #> x = x <# (M <#> N)"
  proof
    fix x
    assume x: "x ∈ carrier G"
    have "M <#> N #> x = M <#> (N #> x)"
      by (metis assms normal_inv_iff setmult_rcos_assoc subgroup.subset
x)
    also have "... = M <#> (x <# N)"
      by (metis assms(2) normal.coset_eq x)
    also have "... = (M #> x) <#> N"
      by (metis assms normal_imp_subgroup rcos_assoc_lcos subgroup.subset
x)
    also have "... = x <# (M <#> N)"
      by (simp add: assms normal.coset_eq normal_imp_subgroup setmult_lcos_assoc
subgroup.subset x)
    finally show "M <#> N #> x = x <# (M <#> N)" .
  qed
qed
```

**end**

```
theory Algebra
  imports Sylow Chinese_Remainder Zassenhaus Galois_Connection Generated_Fields
Free_Abelian_Groups
     Divisibility Embedded_Algebras IntRing Sym_Groups Exact_Sequence
Polynomials Algebraic_Closure
     Left_Coset SimpleGroups SndIsomorphismGrp
begin

end
```

# References


[1] C. Ballarin. *Computer Algebra and Theorem Proving.* PhD thesis, University of Cambridge, 1999. Also Computer Laboratory Technical Report number 473.

[2] K. Conrad. Cyclicity of $(Z/(p))^x$. Expository paper from the author's website. http://www.math.uconn.edu/~kconrad/blurbs/grouptheory/cyclicFp.pdf.

[3] N. Jacobson. *Basic Algebra I.* Freeman, 1985.

[4] F. Kammüller and L. C. Paulson. A formal proof of sylow's theorem: An experiment in abstract algebra with Isabelle HOL. *J. Automated Reasoning*, 23:235–264, 1999.