

The Isabelle/HOL Algebra Library

Clemens Ballarin (Editor)

With contributions by Jesús Aransay, Clemens Ballarin, Stephan
Hohe, Florian Kammüller and Lawrence C Paulson
April 19, 2009

Contents

1	Objects	6
1.1	Structure with Carrier Set.	6
1.2	Structure with Carrier and Equivalence Relation <code>eq</code>	6
2	Orders and Lattices	13
2.1	Partial Orders	13
2.1.1	The order relation	13
2.1.2	Upper and lower bounds of a set	15
2.1.3	Least and greatest, as predicate	18
2.2	Lattices	21
2.2.1	Supremum	22
2.2.2	Infimum	26
2.3	Total Orders	31
2.4	Complete Lattices	33
2.5	Orders and Lattices where <code>eq</code> is the Equality	35
2.6	Examples	38
2.6.1	The Powerset of a Set is a Complete Lattice	38
3	Monoids and Groups	39
3.1	Definitions	39
3.2	Groups	43
3.3	Cancellation Laws and Basic Properties	45
3.4	Subgroups	47
3.5	Direct Products	48
3.6	Homomorphisms and Isomorphisms	50
3.7	Commutative Structures	51
3.8	The Lattice of Subgroups of a Group	53
3.9	Product Operator for Commutative Monoids	55
3.9.1	Inductive Definition of a Relation for Products over Sets	55

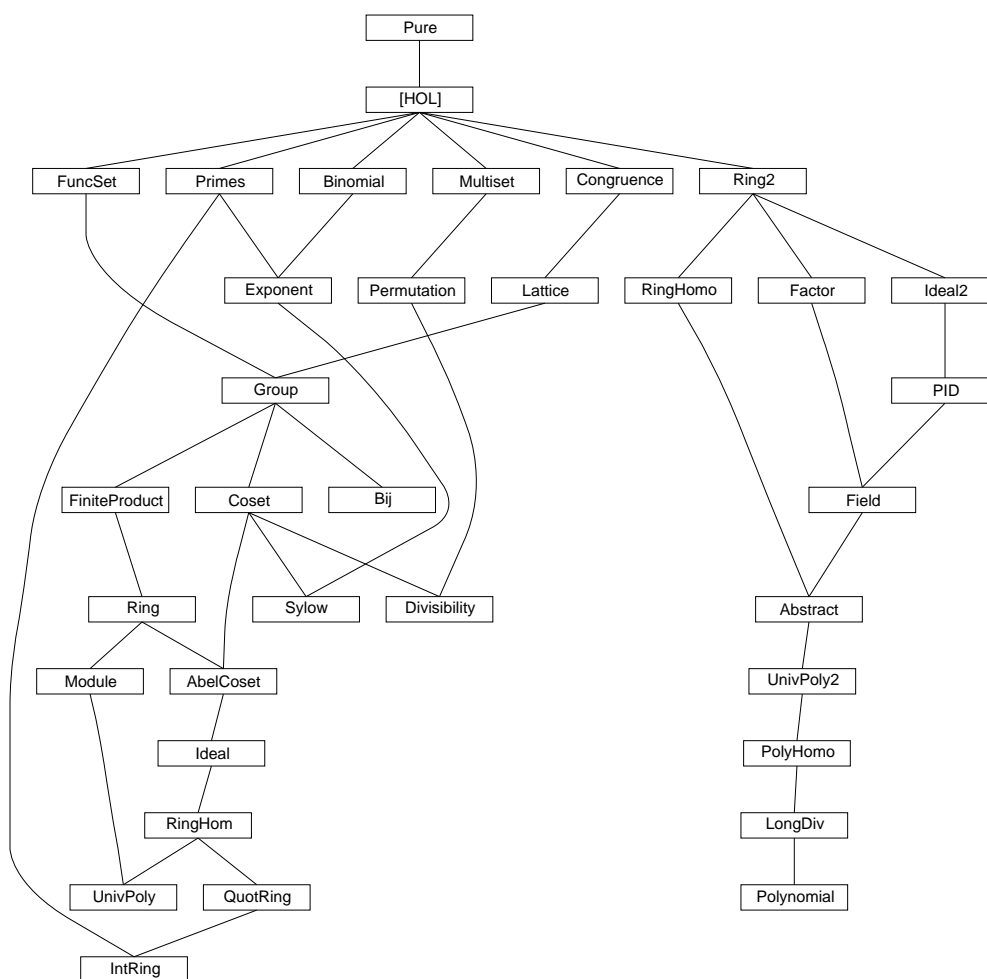
3.9.2	Products over Finite Sets	61
4	Cosets and Quotient Groups	66
4.1	Basic Properties of Cosets	66
4.2	Normal subgroups	74
4.3	More Properties of Cosets	75
4.3.1	Set of Inverses of an <code>r_coset</code>	76
4.3.2	Theorems for <code><#></code> with <code>#></code> or <code><#</code>	77
4.3.3	An Equivalence Relation	78
4.3.4	Two Distinct Right Cosets are Disjoint	79
4.4	Further lemmas for <code>r_congruent</code>	79
4.5	Order of a Group and Lagrange's Theorem	81
4.6	Quotient Groups: Factorization of a Group	82
4.7	The First Isomorphism Theorem	84
5	Sylow's Theorem	87
5.1	The Combinatorial Argument Underlying the First Sylow Theorem	87
5.2	Main Part of the Proof	94
5.3	Discharging the Assumptions of <code>syLOW_central</code>	95
5.3.1	Introduction and Destruct Rules for <code>H</code>	96
5.4	Equal Cardinalities of <code>M</code> and the Set of Cosets	98
5.4.1	The Opposite Injection	98
5.5	Sylow's Theorem	100
6	Bijections of a Set, Permutation and Automorphism Groups	101
6.1	Bijections Form a Group	101
6.2	Automorphisms Form a Group	102
7	Factorial Monoids	103
7.1	Monoids with Cancellation Law	103
7.2	Products of Units in Monoids	105
7.3	Divisibility and Association	107
7.3.1	Function definitions	107
7.3.2	Divisibility	107
7.3.3	Association	110
7.3.4	Division and associativity	113
7.3.5	Multiplication and associativity	114
7.3.6	Units	115
7.3.7	Proper factors	116
7.4	Irreducible Elements and Primes	120
7.4.1	Irreducible elements	120
7.4.2	Prime elements	123
7.5	Factorization and Factorial Monoids	125

7.5.1	Function definitions	125
7.5.2	Comparing lists of elements	125
7.5.3	Properties of lists of elements	129
7.5.4	Factorization in irreducible elements	130
7.5.5	Essentially equal factorizations	134
7.5.6	Factorial monoids and wfactors	141
7.6	Factorizations as Multisets	142
7.6.1	Comparing multisets	143
7.6.2	Interpreting multisets as factorizations	147
7.6.3	Multiplication on multisets	149
7.6.4	Divisibility on multisets	150
7.7	Irreducible Elements are Prime	152
7.8	Greatest Common Divisors and Lowest Common Multiples	158
7.8.1	Definitions	158
7.8.2	Connections to <code>Lattice.thy</code>	158
7.8.3	Existence of gcd and lcm	160
7.9	Conditions for Factoriality	164
7.9.1	Gcd condition	164
7.9.2	Divisor chain condition	172
7.9.3	Primeness condition	174
7.9.4	Application to factorial monoids	180
7.10	Factoriality Theorems	185
8	The Algebraic Hierarchy of Rings	186
8.1	Abelian Groups	186
8.2	Basic Properties	186
8.3	Sums over Finite Sets	189
8.4	Rings: Basic Definitions	192
8.5	Rings	192
8.5.1	Normaliser for Rings	194
8.5.2	Sums over Finite Sets	197
8.6	Integral Domains	197
8.7	Fields	198
8.8	Morphisms	200
8.9	More Lifting from Groups to Abelian Groups	202
8.9.1	Definitions	202
8.9.2	Cosets	204
8.9.3	Subgroups	205
8.9.4	Additive subgroups are normal	206
8.9.5	Congruence Relation	210
8.9.6	Factorization	211
8.9.7	The First Isomorphism Theorem	212
8.9.8	Homomorphisms	213
8.9.9	Cosets	215

8.9.10	Addition of Subgroups	217
9	Ideals	217
9.1	Definitions	217
9.1.1	General definition	217
9.1.2	Ideals Generated by a Subset of <code>carrier R</code>	218
9.1.3	Principal Ideals	218
9.1.4	Maximal Ideals	219
9.1.5	Prime Ideals	219
9.2	Special Ideals	220
9.3	General Ideal Properies	221
9.4	Intersection of Ideals	221
9.5	Addition of Ideals	223
9.6	Ideals generated by a subset of <code>carrier R</code>	224
9.7	Union of Ideals	229
9.8	Properties of Principal Ideals	230
9.9	Prime Ideals	231
9.10	Maximal Ideals	233
9.11	Derived Theorems	235
10	Homomorphisms of Non-Commutative Rings	238
10.1	The Kernel of a Ring Homomorphism	240
10.2	Cosets	241
11	Quotient Rings	243
11.1	Multiplication on Cosets	243
11.2	Quotient Ring Definition	244
11.3	Factorization over General Ideals	245
11.4	Factorization over Prime Ideals	246
11.5	Factorization over Maximal Ideals	247
12	The Ring of Integers	250
12.1	Some properties of <code>int</code>	250
12.2	<code>Z</code> : The Set of Integers as Algebraic Structure	251
12.3	Interpretations	251
12.4	Generated Ideals of <code>Z</code>	255
12.5	Ideals and Divisibility	256
12.6	Ideals and the Modulus	257
12.7	Factorization	259
13	Modules over an Abelian Group	260
13.1	Definitions	260
13.2	Basic Properties of Algebras	262

14 Univariate Polynomials 263

14.1 The Constructor for Univariate Polynomials	263
14.2 Effect of Operations on Coefficients	266
14.3 Polynomials Form a Ring.	268
14.4 Polynomials Form a Commutative Ring.	271
14.5 Polynomials over a commutative ring for a commutative ring	272
14.6 Polynomials Form an Algebra	272
14.7 Further Lemmas Involving Monomials	273
14.8 The Degree Function	278
14.9 Polynomials over Integral Domains	284
14.10 The Evaluation Homomorphism and Universal Property . . .	285
14.11 The long division algorithm: some previous facts.	294
14.12 The long division proof for commutative rings	296
14.13 Sample Application of Evaluation Homomorphism	303



```
theory Congruence imports Main begin
```

1 Objects

1.1 Structure with Carrier Set.

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

1.2 Structure with Carrier and Equivalence Relation eq

```
record 'a eq_object = "'a partial_object" +
  eq :: "'a  $\Rightarrow$  'a  $\Rightarrow$  bool" (infixl ".=" 50)
```

```
constdefs (structure S)
```

```
  elem :: "_  $\Rightarrow$  'a  $\Rightarrow$  'a set  $\Rightarrow$  bool" (infixl ". $\in$ " 50)
  "x . $\in$  A  $\equiv$  ( $\exists$ y  $\in$  A. x .= y)"
```

```
  set_eq :: "_  $\Rightarrow$  'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool" (infixl "{.=}" 50)
  "A {.=} B == (( $\forall$ x  $\in$  A. x . $\in$  B)  $\wedge$  ( $\forall$ x  $\in$  B. x . $\in$  A))"
```

```
  eq_class_of :: "_  $\Rightarrow$  'a  $\Rightarrow$  'a set" ("class'_of" 1)
  "class_of x == {y  $\in$  carrier S. x .= y}"
```

```
  eq_closure_of :: "_  $\Rightarrow$  'a set  $\Rightarrow$  'a set" ("closure'_of" 1)
  "closure_of A == {y  $\in$  carrier S. y . $\in$  A}"
```

```
  eq_is_closed :: "_  $\Rightarrow$  'a set  $\Rightarrow$  bool" ("is'_closed" 1)
  "is_closed A == (A  $\subseteq$  carrier S  $\wedge$  closure_of A = A)"
```

```
syntax
```

```
  not_eq :: "_  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  bool" (infixl ". $\neq$ " 50)
  not_elem :: "_  $\Rightarrow$  'a  $\Rightarrow$  'a set  $\Rightarrow$  bool" (infixl ". $\notin$ " 50)
  set_not_eq :: "_  $\Rightarrow$  'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool" (infixl "{. $\neq$ }" 50)
```

```
translations
```

```
  "x . $\neq$  y" == "~(x .= y)"
  "x . $\notin$  A" == "~(x . $\in$  A)"
  "A {. $\neq$ } B" == "~(A {.=} B)"
```

```
locale equivalence =
```

```
  fixes S (structure)
```

```
  assumes refl [simp, intro]: "x  $\in$  carrier S  $\Longrightarrow$  x .= x"
```

```
  and sym [sym]: "[ x .= y; x  $\in$  carrier S; y  $\in$  carrier S ]  $\Longrightarrow$  y .= x"
```

```
  and trans [trans]: "[ x .= y; y .= z; x  $\in$  carrier S; y  $\in$  carrier S; z  $\in$  carrier S ]  $\Longrightarrow$  x .= z"
```

```

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

lemma (in equivalence) elem_exact:
  assumes "a ∈ carrier S" and "a ∈ A"
  shows "a ∈ A"
using assms
by (fast intro: elemI)

lemma elemE:
  fixes S (structure)
  assumes "a ∈ A"
    and "⋀a'. [[a' ∈ A; a .= a']] ⇒ P"
  shows "P"
using assms
unfolding elem_def
by fast

lemma (in equivalence) elem_cong_1 [trans]:
  assumes cong: "a' .= a"
    and a: "a ∈ A"
    and carr: "a ∈ carrier S" "a' ∈ carrier S"
    and Acarr: "A ⊆ carrier S"
  shows "a' ∈ A"
using a
apply (elim elemE, intro elemI)
proof assumption
  fix b
  assume bA: "b ∈ A"
  note [simp] = carr bA [THEN subsetD [OF Acarr]]
  note cong
  also assume "a .= b"
  finally show "a' .= b" by simp
qed

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

```

```

lemma (in equivalence) mem_imp_elem [simp, intro]:
  "[| x ∈ A; x ∈ carrier S |] ==> x .∈ A"
  unfolding elem_def by blast

```

```

lemma set_eqI:
  fixes R (structure)
  assumes ltr: " $\bigwedge a. a \in A \implies a .\in B$ "
    and rtl: " $\bigwedge b. b \in B \implies b .\in A$ "
  shows " $A \{.\in\} B$ "
  unfolding set_eq_def
  by (fast intro: ltr rtl)

```

```

lemma set_eqI2:
  fixes R (structure)
  assumes ltr: " $\bigwedge a b. a \in A \implies \exists b \in B. a .\in b$ "
    and rtl: " $\bigwedge b. b \in B \implies \exists a \in A. b .\in a$ "
  shows " $A \{.\in\} B$ "
  by (intro set_eqI, unfold elem_def) (fast intro: ltr rtl)+

```

```

lemma set_eqD1:
  fixes R (structure)
  assumes AA': " $A \{.\in\} A'$ "
    and "a ∈ A"
  shows " $\exists a' \in A'. a .\in a'$ "
  using assms
  unfolding set_eq_def elem_def
  by fast

```

```

lemma set_eqD2:
  fixes R (structure)
  assumes AA': " $A \{.\in\} A'$ "
    and "a' ∈ A'"
  shows " $\exists a \in A. a' .\in a$ "
  using assms
  unfolding set_eq_def elem_def
  by fast

```

```

lemma set_eqE:
  fixes R (structure)
  assumes AB: " $A \{.\in\} B$ "
    and r: " $\llbracket \forall a \in A. a .\in B; \forall b \in B. b .\in A \rrbracket \implies P$ "
  shows "P"
  using AB
  unfolding set_eq_def
  by (blast dest: r)

```

```

lemma set_eqE2:
  fixes R (structure)

```



```

    assumes AB: "A {.=} B"
    and r: "[ $\forall a \in A. (\exists b \in B. a . = b); \forall b \in B. (\exists a \in A. b . = a)$ ]"  $\implies$  P"
    shows "P"
using AB
unfolding set_eq_def elem_def
by (blast dest: r)

```

```

lemma set_eqE':
  fixes R (structure)
  assumes AB: "A {.=} B"
    and aA: "a  $\in$  A" and bB: "b  $\in$  B"
    and r: " $\bigwedge a' b'. [a' \in A; b . = a'; b' \in B; a . = b'] \implies P$ "
  shows "P"
proof -
  from AB aA
    have " $\exists b' \in B. a . = b'$ " by (rule set_eqD1)
  from this obtain b'
    where b': "b'  $\in$  B" "a . = b'" by auto

  from AB bB
    have " $\exists a' \in A. b . = a'$ " by (rule set_eqD2)
  from this obtain a'
    where a': "a'  $\in$  A" "b . = a'" by auto

  from a' b'
    show "P" by (rule r)
qed

```

```

lemma (in equivalence) eq_elem_cong_r [trans]:
  assumes a: "a  $\in$  A"
    and cong: "A {.=} A'"
    and carr: "a  $\in$  carrier S"
    and Carr: "A  $\subseteq$  carrier S" "A'  $\subseteq$  carrier S"
  shows "a  $\in$  A'"
using a cong
proof (elim elemE set_eqE)
  fix b
  assume bA: "b  $\in$  A"
    and inA': " $\forall b \in A. b \in A'$ "
  note [simp] = carr Carr Carr[THEN subsetD] bA
  assume "a . = b"
  also from bA inA'
    have "b  $\in$  A'" by fast
  finally
    show "a  $\in$  A'" by simp
qed

```

```

lemma (in equivalence) set_eq_sym [sym]:
  assumes "A {.=} B"

```

```

    and "A  $\subseteq$  carrier S" "B  $\subseteq$  carrier S"
    shows "B  $\{.\} A$ "
using assms
unfolding set_eq_def elem_def
by fast

```

```

lemma (in equivalence) equal_set_eq_trans [trans]:
  assumes AB: "A = B" and BC: "B  $\{.\} C$ "
  shows "A  $\{.\} C$ "
  using AB BC by simp

```

```

lemma (in equivalence) set_eq_equal_trans [trans]:
  assumes AB: "A  $\{.\} B$ " and BC: "B = C"
  shows "A  $\{.\} C$ "
  using AB BC by simp

```

```

lemma (in equivalence) set_eq_trans [trans]:
  assumes AB: "A  $\{.\} B$ " and BC: "B  $\{.\} C$ "
    and carr: "A  $\subseteq$  carrier S" "B  $\subseteq$  carrier S" "C  $\subseteq$  carrier S"
  shows "A  $\{.\} C$ "
proof (intro set_eqI)
  fix a
  assume aA: "a  $\in$  A"
  with carr have "a  $\in$  carrier S" by fast
  note [simp] = carr this

  from aA
    have "a  $\in$  A" by (simp add: elem_exact)
  also note AB
  also note BC
  finally
    show "a  $\in$  C" by simp
next
  fix c
  assume cC: "c  $\in$  C"
  with carr have "c  $\in$  carrier S" by fast
  note [simp] = carr this

  from cC
    have "c  $\in$  C" by (simp add: elem_exact)
  also note BC[symmetric]
  also note AB[symmetric]
  finally
    show "c  $\in$  A" by simp
qed

```

```

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}"
unfolding set_eq_def elem_def
proof safe
  have "x' ∈ {x', y}" by fast
  with xx' show "∃b∈{x', y}. x .= b" by fast
next
  have "y ∈ {x', y}" by fast
  with carr show "∃b∈{x', y}. y .= b" by fast
next
  have "x ∈ {x, y}" by fast
  with xx'[symmetric] carr
  show "∃a∈{x, y}. x' .= a" by fast
next
  have "y ∈ {x, y}" by fast
  with carr show "∃a∈{x, y}. y .= a" by fast
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:
  "[| x .= x'; A ⊆ carrier S; x ∈ closure_of A; x ∈ carrier S; x' ∈ carrier S |] ==> x' ∈ closure_of A"
unfolding eq_closure_of_def elem_def
by (blast intro: trans sym)

lemma (in equivalence) is_closed_eq [dest]:
  "[| x .= x'; x ∈ A; is_closed A; x ∈ carrier S; x' ∈ carrier S |] ==> x' ∈ A"
unfolding eq_is_closed_def
using closure_of_eq [where A = A]
by simp

lemma (in equivalence) is_closed_eq_rev [dest]:
  "[| x .= x'; x' ∈ A; is_closed A; x ∈ carrier S; x' ∈ carrier S |]

```

```

==> x ∈ A"
  by (drule sym) (simp_all add: is_closed_eq)

lemma closure_of_closed [simp, intro]:
  fixes S (structure)
  shows "closure_of A ⊆ carrier S"
unfolding eq_closure_of_def
by fast

lemma closure_of_memI:
  fixes S (structure)
  assumes "a .∈ A"
    and "a ∈ carrier S"
  shows "a ∈ closure_of A"
unfolding eq_closure_of_def
using assms
by fast

lemma closure_ofI2:
  fixes S (structure)
  assumes "a .= a'"
    and "a' ∈ A"
    and "a ∈ carrier S"
  shows "a ∈ closure_of A"
unfolding eq_closure_of_def elem_def
using assms
by fast

lemma closure_of_memE:
  fixes S (structure)
  assumes p: "a ∈ closure_of A"
    and r: "[a ∈ carrier S; a .∈ A] ⇒ P"
  shows "P"
proof -
  from p
    have acarr: "a ∈ carrier S"
    and "a .∈ A"
    by (simp add: eq_closure_of_def)+
  thus "P" by (rule r)
qed

lemma closure_ofE2:
  fixes S (structure)
  assumes p: "a ∈ closure_of A"
    and r: "⋀a'. [a ∈ carrier S; a' ∈ A; a .= a'] ⇒ P"
  shows "P"
proof -
  from p have acarr: "a ∈ carrier S" by (simp add: eq_closure_of_def)

```

```

from p have "∃ a' ∈ A. a .= a'" by (simp add: eq_closure_of_def elem_def)
from this obtain a'
  where "a' ∈ A" and "a .= a'" by auto

from acarr and this
  show "P" by (rule r)
qed

end

```

```

theory Lattice imports Congruence begin

```

2 Orders and Lattices

2.1 Partial Orders

```

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

locale weak_partial_order = equivalence L for L (structure) +
  assumes le_refl [intro, simp]:
    "x ∈ carrier L ==> x ⊆ x"
  and weak_le_anti_sym [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"

constdefs (structure L)
  lless :: "[_, 'a, 'a] => bool" (infixl "⊂" 50)
  "x ⊂ y == x ⊆ y & x ≠ 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  $\sqsubseteq$  y; y .= z; x  $\in$  carrier L; y  $\in$  carrier L; z  $\in$  carrier L ] ]  $\implies$ 
x  $\sqsubseteq$  z"
  by (auto intro: le_cong [THEN iffD1])

lemma weak_refl [intro, simp]: "[[ x .= y; x  $\in$  carrier L; y  $\in$  carrier
L ] ]  $\implies$  x  $\sqsubseteq$  y"
  by (simp add: le_cong_l)

end

lemma weak_llessI:
  fixes R (structure)
  assumes "x  $\sqsubseteq$  y" and "~(x .= y)"
  shows "x  $\sqsubset$  y"
  using assms unfolding lless_def by simp

lemma lless_imp_le:
  fixes R (structure)
  assumes "x  $\sqsubset$  y"
  shows "x  $\sqsubseteq$  y"
  using assms unfolding lless_def by simp

lemma weak_lless_imp_not_eq:
  fixes R (structure)
  assumes "x  $\sqsubset$  y"
  shows " $\neg$  (x .= y)"
  using assms unfolding lless_def by simp

lemma weak_llessE:
  fixes R (structure)
  assumes p: "x  $\sqsubset$  y" and e: "[[x  $\sqsubseteq$  y;  $\neg$  (x .= y)] ]  $\implies$  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'  $\sqsubset$  y"
  and carr: "x  $\in$  carrier L" "x'  $\in$  carrier L" "y  $\in$  carrier L"
  shows "x  $\sqsubset$  y"
  using assms unfolding lless_def by (auto intro: trans sym)

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

```

2.1.2 Upper and lower bounds of a set

```

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

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

lemma Upper_closed [intro!, simp]:
  "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'carr: "a' ∈ carrier L" and Acarr: "A ⊆ carrier L"
    and aa': "a .= a'"
    and aelem: "a ∈ Upper L A"
  shows "a' ∈ Upper L A"
proof (rule Upper_memI[OF _ a'carr])
  fix y
  assume yA: "y ∈ A"
  hence "y ⊆ a" by (intro Upper_memD[OF aelem, THEN conjunct1] Acarr)
  also note aa'
  finally
    show "y ⊆ a'"
    by (simp add: a'carr subsetD[OF Acarr yA] subsetD[OF Upper_closed
aelem])
qed

lemma (in weak_partial_order) Upper_cong:
  assumes Acarr: "A ⊆ carrier L" and A'carr: "A' ⊆ carrier L"
    and AA': "A {.=} A'"
  shows "Upper L A = Upper L A'"
unfolding Upper_def
apply rule
  apply (rule, clarsimp) defer 1
  apply (rule, clarsimp) defer 1
proof -
  fix x a'
  assume carr: "x ∈ carrier L" "a' ∈ carrier L"
    and a'A': "a' ∈ A'"
  assume aLxCond[rule_format]: "∀a. a ∈ A ∧ a ∈ carrier L → a ⊆ x"

  from AA' and a'A' have "∃a∈A. a' .= a" by (rule set_eqD2)
  from this obtain a
    where aA: "a ∈ A"
    and a'a: "a' .= a"
    by auto
  note [simp] = subsetD[OF Acarr aA] carr

  note a'a
  also have "a ⊆ x" by (simp add: aLxCond aA)
  finally show "a' ⊆ x" by simp
next

```



```

fix x a
assume carr: "x ∈ carrier L" "a ∈ carrier L"
  and aA: "a ∈ A"
assume a'LxCond[rule_format]: "∀a'. a' ∈ A' ∧ a' ∈ carrier L ⟶ a'
⊆ x"

from AA' and aA have "∃a'∈A'. a .= a'" by (rule set_eqD1)
from this obtain a'
  where a'A': "a' ∈ A'"
  and aa': "a .= a'"
  by auto
note [simp] = subsetD[OF A'carr a'A'] carr

note aa'
also have "a' ⊆ x" by (simp add: a'LxCond a'A')
finally show "a ⊆ x" by simp
qed

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'carr: "a' ∈ carrier L" and Acarr: "A ⊆ carrier L"
  and aa': "a .= a'"
  and aelem: "a ∈ Lower L A"
  shows "a' ∈ Lower L A"
using assms Lower_closed[of L A]
by (intro Lower_memI) (blast intro: le_cong_1[OF aa'[symmetric]])

```

```

lemma (in weak_partial_order) Lower_cong:
  assumes Acarr: "A  $\subseteq$  carrier L" and A'carr: "A'  $\subseteq$  carrier L"
    and AA': "A  $\{.\} A'$ "
  shows "Lower L A = Lower L A'"
using Lower_memD[of y]
unfolding Lower_def
apply safe
  apply clarsimp defer 1
  apply clarsimp defer 1
proof -
  fix x a'
  assume carr: "x  $\in$  carrier L" "a'  $\in$  carrier L"
    and a'A': "a'  $\in A'$ "
  assume "∀a. a  $\in A \wedge a \in$  carrier L  $\longrightarrow x \sqsubseteq a$ "
  hence aLxCond: " $\bigwedge a. \llbracket a \in A; a \in$  carrier L  $\rrbracket \implies x \sqsubseteq a$ " by fast

  from AA' and a'A' have " $\exists a \in A. a' . = a$ " by (rule set_eqD2)
  from this obtain a
    where aA: "a  $\in A$ "
    and a'a: "a' . = a"
    by auto

  from aA and subsetD[OF Acarr aA]
    have "x  $\sqsubseteq a$ " by (rule aLxCond)
  also note a'a[symmetric]
  finally
    show "x  $\sqsubseteq a'$ " by (simp add: carr subsetD[OF Acarr aA])
next
  fix x a
  assume carr: "x  $\in$  carrier L" "a  $\in$  carrier L"
    and aA: "a  $\in A$ "
  assume "∀a'. a'  $\in A' \wedge a' \in$  carrier L  $\longrightarrow x \sqsubseteq a'$ "
  hence a'LxCond: " $\bigwedge a'. \llbracket a' \in A'; a' \in$  carrier L  $\rrbracket \implies x \sqsubseteq a'$ " by fast+

  from AA' and aA have " $\exists a' \in A'. a . = a'$ " by (rule set_eqD1)
  from this obtain a'
    where a'A': "a'  $\in A'$ "
    and aa': "a . = a'"
    by auto
  from a'A' and subsetD[OF A'carr a'A']
    have "x  $\sqsubseteq a'$ " by (rule a'LxCond)
  also note aa'[symmetric]
  finally show "x  $\sqsubseteq a$ " by (simp add: carr subsetD[OF A'carr a'A'])
qed

```

2.1.3 Least and greatest, as predicate

constdefs (structure L)

```

least :: "[_, 'a, 'a set] => bool"
"least L l A == A ⊆ carrier L & l ∈ A & (ALL x : A. l ⊆ x)"

greatest :: "[_, 'a, 'a set] => bool"
"greatest L g A == A ⊆ carrier L & g ∈ A & (ALL x : A. x ⊆ g)"

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

lemma least_closed [intro, simp]:
  "least L l A ==> 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"
  by (unfold least_def) (auto dest: sym)

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 Acarrs: "A ⊆ carrier L" "A' ⊆ carrier L"
  and AA': "A  $\{.\} A'$ "
  shows "least L x (Upper L A) = least L x (Upper L A')"
  apply (subgoal_tac "Upper L A = Upper L A'", simp)
  by (rule Upper_cong) fact+

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  $\subseteq$  carrier L" "s  $\in$  carrier L"
    shows "least L s (Upper L A)"
  proof -
    have "Upper L A  $\subseteq$  carrier L" by simp
    moreover from above L have "s  $\in$  Upper L A" by (simp add: Upper_def)
    moreover from below have "ALL x : Upper L A. s  $\subseteq$  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  $\in$  A; A  $\subseteq$  carrier L |] ==> x  $\subseteq$  s"
  by (unfold least_def) blast

lemma greatest_closed [intro, simp]:
  "greatest L l A ==> l  $\in$  carrier L"
  by (unfold greatest_def) fast

lemma greatest_mem:
  "greatest L l A ==> l  $\in$  A"
  by (unfold greatest_def) fast

lemma (in weak_partial_order) weak_greatest_unique:
  "[| greatest L x A; greatest L y A |] ==> x  $\dot{=}$  y"
  by (unfold greatest_def) blast

lemma greatest_le:
  fixes L (structure)
  shows "[| greatest L x A; a  $\in$  A |] ==> a  $\subseteq$  x"
  by (unfold greatest_def) fast

lemma (in weak_partial_order) greatest_cong:
  "[| x  $\dot{=}$  x'; x  $\in$  carrier L; x'  $\in$  carrier L; is_closed A |] ==>
  greatest L x A = greatest L x' A"
  by (unfold greatest_def) (auto dest: sym)

greatest is not congruent in the second parameter for A  $\{\dot{=}\}$  A'

lemma (in weak_partial_order) greatest_Lower_cong_l:
  assumes "x  $\dot{=}$  x'"
  and "x  $\in$  carrier L" "x'  $\in$  carrier L"
  and "A  $\subseteq$  carrier L"
  shows "greatest L x (Lower L A) = greatest L x' (Lower L A)"
  apply (rule greatest_cong) using assms by auto

lemma (in weak_partial_order) greatest_Lower_cong_r:
  assumes Acarrs: "A  $\subseteq$  carrier L" "A'  $\subseteq$  carrier L"
  and AA': "A  $\{\dot{=}\}$  A'"
  shows "greatest L x (Lower L A) = greatest L x (Lower L A')"
  apply (subgoal_tac "Lower L A = Lower L A'", simp)

```

by (rule Lower_cong) fact+

```

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 "ALL 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

```

Supremum and infimum

```

constdefs (structure L)
  sup :: "[_, 'a set] => 'a" ("⋒" [90] 90)
  "⋒ A == SOME x. least L x (Upper L A)"

  inf :: "[_, 'a set] => 'a" ("⋓" [90] 90)
  "⋓ A == SOME x. greatest L x (Lower L A)"

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

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

```

2.2 Lattices

```

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

```

```

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

```

```

locale weak_lattice = weak_upper_semilattice + weak_lower_semilattice

```

2.2.1 Supremum

```

lemma (in weak_upper_semilattice) joinI:
  "[| !!l. least L l (Upper L {x, y}) ==> P l; x ∈ carrier L; y ∈ carrier
L |]
  ==> P (x ⊔ y)"
proof (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 ==>  $\bigsqcup$ {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 ==>  $\bigsqcup$ {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 ( $\bigsqcup$ (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"
          apply (rule subsetD [where A = "Upper L (insert x A)"])
          apply (rule Upper_antimono)
          apply blast
          apply (rule y)
          done
        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_singleton_closed 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_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_anti_sym)
    from sup L show "x ⊔ (y ⊔ z) ⊆ s"
    by (fastsimp 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

```

2.2.2 Infimum

```

lemma (in weak_lower_semilattice) meetI:
  "[| !!i. greatest L i (Lower L {x, y}) ==> P i;

```

```

    x ∈ carrier L; y ∈ carrier L |]
    ==> P (x ⊔ y)"
proof (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 greatest_a: "greatest L a (Lower L {x, y})"
  assume "greatest L b (Lower L {x', y})"
  with carr
    have greatest_b: "greatest L b (Lower L {x, y})"
    by (simp add: greatest_Lower_cong_r[OF _ _ seq])

  from greatest_a greatest_b
    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 greatest_a: "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 ==>  $\bigcap$ {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 ==>  $\bigcap$ {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 ( $\bigcap$ (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
    qed
  qed
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"
      apply (rule subsetD [where A = "Lower L (insert x A)"])
      apply (rule Lower_antimono)
      apply blast
      apply (rule y)
      done
    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

```

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  $\sqsubseteq$  i" by (fast elim: greatest_le intro: Lower_memI)
qed

```

```

lemma (in weak_lower_semilattice) weak_meet_assoc_lemma:
  assumes L: "x  $\in$  carrier L" "y  $\in$  carrier L" "z  $\in$  carrier L"
  shows "x  $\sqcap$  (y  $\sqcap$  z) =  $\sqcap$  {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  $\sqcap$  (y  $\sqcap$  z) = i"
  proof (rule weak_le_anti_sym)
    from inf L show "i  $\sqsubseteq$  x  $\sqcap$  (y  $\sqcap$  z)"
      by (fastsimp intro!: meet_le elim: greatest_Lower_below)
  next
    from inf L show "x  $\sqcap$  (y  $\sqcap$  z)  $\sqsubseteq$  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  $\sqcap$  y = y  $\sqcap$  x"
  by (unfold meet_def) (simp add: insert_commute)

```

```

lemma (in weak_lower_semilattice) weak_meet_assoc:
  assumes L: "x  $\in$  carrier L" "y  $\in$  carrier L" "z  $\in$  carrier L"
  shows "(x  $\sqcap$  y)  $\sqcap$  z = x  $\sqcap$  (y  $\sqcap$  z)"
proof -
  have "(x  $\sqcap$  y)  $\sqcap$  z = z  $\sqcap$  (x  $\sqcap$  y)" by (simp only: meet_comm)
  also from L have "... =  $\sqcap$  {z, x, y}" by (simp add: weak_meet_assoc_lemma)
  also from L have "... =  $\sqcap$  {x, y, z}" by (simp add: insert_commute)
  also from L have "... = x  $\sqcap$  (y  $\sqcap$  z)" by (simp add: weak_meet_assoc_lemma
[symmetric])
  finally show ?thesis by (simp add: L)
qed

```

2.3 Total Orders

```

locale weak_total_order = weak_partial_order +
  assumes total: "[| x  $\in$  carrier L; y  $\in$  carrier L |] ==> x  $\sqsubseteq$  y | y  $\sqsubseteq$ 
x"

```

Introduction rule: the usual definition of total order

```

lemma (in weak_partial_order) weak_total_orderI:
  assumes total: "!!x y. [| x ∈ carrier L; y ∈ carrier L |] ==> x ⊆
y | y ⊆ x"
  shows "weak_total_order L"
  proof qed (rule total)

```

Total orders are lattices.

```

sublocale weak_total_order < weak: weak_lattice
proof
  fix x y
  assume L: "x ∈ carrier L" "y ∈ carrier L"
  show "EX 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 "EX 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

```


2.4 Complete Lattices

```

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

```

Introduction rule: the usual definition of complete lattice

```

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

```

```

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

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

```

```

lemma (in weak_complete_lattice) supI:
  "[| !!l. least L l (Upper L A) ==> P l; A  $\subseteq$  carrier L |]
  ==> P ( $\bigsqcup$  A)"
proof (unfold sup_def)
  assume L: "A  $\subseteq$  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  $\subseteq$  carrier L ==>  $\bigsqcup$  A  $\in$  carrier L"
  by (rule supI) simp_all

```

```

lemma (in weak_complete_lattice) top_closed [simp, intro]:
  "⊤  $\in$  carrier L"
  by (unfold top_def) simp

```

```

lemma (in weak_complete_lattice) infI:
  "[| !!i. greatest L i (Lower L A) ==> P i; A  $\subseteq$  carrier L |]
  ==> P ( $\bigsqcap$  A)"
proof (unfold inf_def)
  assume L: "A  $\subseteq$  carrier L"
  and P: "!!i. greatest L i (Lower L A) ==> P i"

```

```

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  $\subseteq$  carrier L ==>  $\bigcap$  A  $\in$  carrier L"
by (rule infI) simp_all

```

```

lemma (in weak_complete_lattice) bottom_closed [simp, intro]:
  " $\perp \in$  carrier L"
by (unfold bottom_def) simp

```

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

```

```

theorem (in weak_partial_order) weak_complete_lattice_criterion1:
  assumes top_exists: "EX g. greatest L g (carrier L)"
  and inf_exists:
    "!!A. [ $\mid$  A  $\subseteq$  carrier L; A  $\sim$  {} ] ==> EX 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  $\subseteq$  carrier L"
  let ?B = "Upper L A"
  from L top have "top  $\in$  ?B" by (fast intro!: Upper_memI intro: greatest_le)
  then have B_non_empty: "?B  $\sim$  {}" by fast
  have B_L: "?B  $\subseteq$  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)" ..
  have "least L b (Upper L A)"
apply (rule least_UpperI)
  apply (rule greatest_le [where A = "Lower L ?B"])
  apply (rule b_inf_B)
  apply (rule Lower_memI)
  apply (erule Upper_memD [THEN conjunct1])
  apply assumption
  apply (rule L)
  apply (fast intro: L [THEN subsetD])
  apply (erule greatest_Lower_below [OF b_inf_B])

```

```

    apply simp
  apply (rule L)
apply (rule greatest_closed [OF b_inf_B])
done
  then show "EX s. least L s (Upper L A)" ..
next
  fix A
  assume L: "A  $\subseteq$  carrier L"
  show "EX 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

```

2.5 Orders and Lattices where eq is the Equality

```

locale partial_order = weak_partial_order +
  assumes eq_is_equal: "op .= = op ="
begin

declare weak_le_anti_sym [rule del]

lemma le_anti_sym [intro]:
  "[| x  $\sqsubseteq$  y; y  $\sqsubseteq$  x; x  $\in$  carrier L; y  $\in$  carrier L |] ==> x = y"
  using weak_le_anti_sym unfolding eq_is_equal .

lemma lless_eq:
  "x  $\sqsubset$  y  $\longleftrightarrow$  x  $\sqsubseteq$  y & x  $\neq$  y"
  unfolding lless_def by (simp add: eq_is_equal)

lemma lless_asym:
  assumes "a  $\in$  carrier L" "b  $\in$  carrier L"
  and "a  $\sqsubset$  b" "b  $\sqsubset$  a"
  shows "P"
  using assms unfolding lless_eq by auto

end

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 .
```

Lattices

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

```
sublocale upper_semilattice < weak: weak_upper_semilattice
  proof qed (rule sup_of_two_exists)
```

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

```
sublocale lower_semilattice < weak: weak_lower_semilattice
  proof qed (rule inf_of_two_exists)
```

```
locale lattice = upper_semilattice + lower_semilattice
```

Supremum

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

```
lemma (in partial_order) sup_of_singleton [simp]:
  "x ∈ carrier L ==>  $\sqcup \{x\} = x$ "
  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  $\sqcup$  (y  $\sqcup$  z) =  $\sqcup \{x, y, z\}$ "
  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  $\sqcup$  y)  $\sqcup$  z = x  $\sqcup$  (y  $\sqcup$  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 ==>  $\sqcap \{x\} = x$ "
  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  $\sqcap$  (y  $\sqcap$  z) =  $\sqcap$ {x, y, z}"
using weak_meet_assoc_lemma L unfolding eq_is_equal .
```

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

Total Orders

```
locale total_order = partial_order +
  assumes total_order_total: "[| x  $\in$  carrier L; y  $\in$  carrier L |] ==>
x  $\sqsubseteq$  y | y  $\sqsubseteq$  x"
```

```
sublocale total_order < weak: weak_total_order
  proof qed (rule total_order_total)
```

Introduction rule: the usual definition of total order

```
lemma (in partial_order) total_orderI:
  assumes total: "!!x y. [| x  $\in$  carrier L; y  $\in$  carrier L |] ==> x  $\sqsubseteq$ 
y | y  $\sqsubseteq$  x"
  shows "total_order L"
  proof qed (rule total)
```

Total orders are lattices.

```
sublocale total_order < weak: lattice
  proof qed (auto intro: sup_of_two_exists inf_of_two_exists)
```

Complete lattices

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

```
sublocale complete_lattice < weak: weak_complete_lattice
  proof qed (auto intro: sup_exists inf_exists)
```

Introduction rule: the usual definition of complete lattice

```
lemma (in partial_order) complete_latticeI:
  assumes sup_exists:
    "!!A. [| A  $\subseteq$  carrier L |] ==> EX s. least L s (Upper L A)"
  and inf_exists:
    "!!A. [| A  $\subseteq$  carrier L |] ==> EX i. greatest L i (Lower L A)"
  shows "complete_lattice L"
  proof qed (auto intro: sup_exists inf_exists)
```

```
theorem (in partial_order) complete_lattice_criterion1:
  assumes top_exists: "EX g. greatest L g (carrier L)"
```

```

    and inf_exists:
      "!!A. [| A  $\subseteq$  carrier L; A  $\sim$  = {} |] ==> EX 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  $\subseteq$  carrier L"
    let ?B = "Upper L A"
    from L top have "top  $\in$  ?B" by (fast intro!: Upper_memI intro: greatest_le)
    then have B_non_empty: "?B  $\sim$  = {}" by fast
    have B_L: "?B  $\subseteq$  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)" ..
    have "least L b (Upper L A)"
  apply (rule least_UpperI)
    apply (rule greatest_le [where A = "Lower L ?B"])
    apply (rule b_inf_B)
    apply (rule Lower_memI)
    apply (erule Upper_memD [THEN conjunct1])
    apply assumption
    apply (rule L)
    apply (fast intro: L [THEN subsetD])
    apply (erule greatest_Lower_below [OF b_inf_B])
    apply simp
    apply (rule L)
  apply (rule greatest_closed [OF b_inf_B])
done
    then show "EX s. least L s (Upper L A)" ..
next
  fix A
  assume L: "A  $\subseteq$  carrier L"
  show "EX 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

```

2.6 Examples

2.6.1 The Powerset of a Set is a Complete Lattice

```

theorem powerset_is_complete_lattice:
  "complete_lattice (| carrier = Pow A, eq = op =, le = op  $\subseteq$  |)"
  (is "complete_lattice ?L")

```

```

proof (rule partial_order.complete_latticeI)
  show "partial_order ?L"
    proof qed auto
next
  fix B
  assume B: "B  $\subseteq$  carrier ?L"
  show "EX s. least ?L s (Upper ?L B)"
  proof
    from B show "least ?L ( $\bigcup$  B) (Upper ?L B)"
      by (fastsimp intro!: least_UpperI simp: Upper_def)
    qed
next
  fix B
  assume B: "B  $\subseteq$  carrier ?L"
  show "EX i. greatest ?L i (Lower ?L B)"
  proof
    from B show "greatest ?L ( $\bigcap$  B  $\cap$  A) (Lower ?L B)"
 $\bigcap$  B is not the infimum of B:  $\bigcap \{\} = \text{UNIV}$  which is in general bigger than A!
    by (fastsimp intro!: greatest_LowerI simp: Lower_def)
    qed
qed

```

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

end

```

theory Group
imports Lattice FuncSet
begin

```

3 Monoids and Groups

3.1 Definitions

Definitions follow [2].

```

record 'a monoid = "'a partial_object" +
  mult    :: "'a, 'a]  $\Rightarrow$  'a" (infixl " $\otimes$ " 70)
  one     :: 'a ("1")

constdefs (structure G)
  m_inv :: "('a, 'b) monoid_scheme  $\Rightarrow$  'a  $\Rightarrow$  'a" ("inv" 80)
  "inv x == (THE y. y  $\in$  carrier G & x  $\otimes$  y = 1 & y  $\otimes$  x = 1)"

Units :: "_  $\Rightarrow$  'a set"
  — The set of invertible elements

```

```

"Units G == {y. y ∈ carrier G & (∃x ∈ carrier G. x ⊗ y = 1 & y ⊗ x
= 1)}"

consts
  pow :: "('a, 'm) monoid_scheme, 'a, 'b::number] => 'a" (infixr "'(^)_"
75)

defs (overloaded)
  nat_pow_def: "pow G a n == nat_rec 1_G (%u b. b ⊗_G a) n"
  int_pow_def: "pow G a z ==
    let p = nat_rec 1_G (%u b. b ⊗_G a)
    in if neg z then inv_G (p (nat (-z))) else p (nat z)"

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) 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 [intro, simp]:
  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
    — Must avoid premature use of hyp_subst_tac.
    apply (rule_tac CollectI)
    apply (rule)
    apply (fast)
    apply (rule bexI [where x = "y' ⊗ x'"])
    apply (auto simp: m_assoc)
    done
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 (unfold Units_def m_inv_def, auto)
  apply (rule theI2, fast)
  apply (fast intro: inv_unique, fast)
  done

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, auto)

```

```

    apply (rule theI2, fast)
    apply (fast intro: inv_unique, fast)
  done

lemma (in monoid) Units_r_inv [simp]:
  "x ∈ Units G ==> x ⊗ inv x = 1"
  apply (unfold Units_def m_inv_def, auto)
  apply (rule theI2, fast)
  apply (fast intro: inv_unique, fast)
  done

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 add: Units_l_inv)
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  $\otimes$  y = 1"
  and G: "x  $\in$  Units G" "y  $\in$  Units G"
  shows "y  $\otimes$  x = 1"
proof -
  from G have "x  $\otimes$  y  $\otimes$  x = x  $\otimes$  1" by (auto simp add: inv Units_closed)
  with G show ?thesis by (simp del: r_one add: m_assoc Units_closed)
qed

```

Power

```

lemma (in monoid) nat_pow_closed [intro, simp]:
  "x  $\in$  carrier G ==> x (^) (n::nat)  $\in$  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  $\otimes$  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  $\in$  carrier G ==> x (^) (n::nat)  $\otimes$  x (^) m = x (^) (n + m)"
  by (induct m) (simp_all add: m_assoc [THEN sym])

lemma (in monoid) nat_pow_pow:
  "x  $\in$  carrier G ==> (x (^) n) (^) m = x (^) (n * m::nat)"
  by (induct m) (simp, simp add: nat_pow_mult add_commute)

```

3.2 Groups

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

```

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

```

```

lemma (in group) is_group: "group G" by (rule group_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 " $\exists y \in \text{carrier } G. y \otimes x = 1 \ \& \ x \otimes y = 1$ "
    by (fast intro: l_inv r_inv)
  qed
  then have carrier_subset_Units: "carrier G  $\leq$  Units G"
    by (unfold Units_def) fast
  show ?thesis proof qed (auto simp: r_one m_assoc carrier_subset_Units)
qed

```

```

lemma (in monoid) group_l_invI:
  assumes l_inv_ex:
    " $\forall x. x \in \text{carrier } G \implies \exists y \in \text{carrier } G. y \otimes x = 1$ "
  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  $\leq$  carrier G" by fast
next
  show "carrier G  $\leq$  Units G" by (rule Units)
qed

```

```

lemma (in group) inv_closed [intro, simp]:
  " $x \in \text{carrier } G \implies \text{inv } x \in \text{carrier } G$ "
  using Units_inv_closed by simp

```

```

lemma (in group) l_inv_ex [simp]:
  " $x \in \text{carrier } G \implies \exists y \in \text{carrier } G. y \otimes x = 1$ "
  using Units_l_inv_ex by simp

```

```

lemma (in group) r_inv_ex [simp]:
  " $x \in \text{carrier } G \implies \exists y \in \text{carrier } G. x \otimes y = 1$ "
  using Units_r_inv_ex by simp

```

```

lemma (in group) l_inv [simp]:
  " $x \in \text{carrier } G \implies \text{inv } x \otimes x = 1$ "
  using Units_l_inv by simp

```

3.3 Cancellation Laws and Basic Properties

```

lemma (in group) l_cancel [simp]:
  " $[x \in \text{carrier } G; y \in \text{carrier } G; z \in \text{carrier } G] \implies$ 
   $(x \otimes y = x \otimes z) = (y = z)$ "
  using Units_l_inv by simp

```

```

lemma (in group) r_inv [simp]:
  " $x \in \text{carrier } G \implies x \otimes \text{inv } x = 1$ "
proof -

```

```

    assume x: "x ∈ carrier G"
    then have "inv x ⊗ (x ⊗ inv x) = inv x ⊗ 1"
      by (simp add: m_assoc [symmetric] l_inv)
    with x show ?thesis by (simp del: r_one)
  qed

lemma (in group) r_cancel [simp]:
  "[| x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |] ==>
   (y ⊗ x = z ⊗ x) = (y = z)"
proof
  assume eq: "y ⊗ x = z ⊗ x"
  and G: "x ∈ carrier G" "y ∈ carrier G" "z ∈ carrier G"
  then have "y ⊗ (x ⊗ inv x) = z ⊗ (x ⊗ inv x)"
    by (simp add: m_assoc [symmetric] del: r_inv Units_r_inv)
  with G show "y = z" by simp
next
  assume eq: "y = z"
  and G: "x ∈ carrier G" "y ∈ carrier G" "z ∈ carrier G"
  then show "y ⊗ x = z ⊗ x" by simp
qed

lemma (in group) inv_one [simp]:
  "inv 1 = 1"
proof -
  have "inv 1 = 1 ⊗ (inv 1)" by (simp del: r_inv Units_r_inv)
  moreover have "... = 1" by simp
  finally show ?thesis .
qed

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 l_inv) (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"
apply (simp add: m_inv_def)
apply (rule the_equality)
  apply (simp add: inv_comm [of y x])
apply (rule r_cancel [THEN iffD1], auto)
done

```

Power

```

lemma (in group) int_pow_def2:
  "a (^) (z::int) = (if neg z then inv (a (^) (nat (-z))) else a (^) (nat z))"
  by (simp add: int_pow_def nat_pow_def Let_def)

lemma (in group) int_pow_0 [simp]:
  "x (^) (0::int) = 1"
  by (simp add: int_pow_def2)

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

```

3.4 Subgroups

```

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

lemma (in subgroup) is_subgroup:
  "subgroup H G" 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 subgroup_imp_subset:
  "subgroup H G ==> H ⊆ carrier G"
  by (rule subgroup.subset)

lemma (in subgroup) subgroup_is_group [intro]:
  assumes "group G"
  shows "group (G⟦carrier := H⟧)"
proof -

```

```

interpret group G by fact
show ?thesis
  apply (rule monoid.group_l_invI)
  apply (unfold_locales) [1]
  apply (auto intro: m_assoc l_inv mem_carrier)
done
qed

```

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

```

lemma (in group) one_in_subset:
  "[| H ⊆ carrier G; H ≠ {}; ∀ a ∈ H. inv a ∈ H; ∀ a ∈ H. ∀ b ∈ H. a ⊗ b
  ∈ H |]
  ==> 1 ∈ H"
by (force simp add: l_inv)

```

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

```

```

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"
proof (rule classical)
  assume "finite (carrier G)" "~ 0 < card H"
  then have "finite H" by (blast intro: finite_subset [OF subset])
  with prems have "subgroup {} G" by simp
  with subgroup_nonempty show ?thesis by contradiction
qed

```

3.5 Direct Products

```

constdefs
  DirProd :: "'a ⇒ 'b ⇒ ('a × 'b) monoid" (infixr "××" 80)
  "G ×× H ≡ (|carrier = carrier G × carrier H,
               mult = (λ(g, h) (g', h')). (g ⊗G g', h ⊗H h')),"

```


$$\text{one} = (1_G, 1_H)$$

```

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 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

```

3.6 Homomorphisms and Isomorphisms

```

constdefs (structure G and H)
  hom :: "_ => _ => ('a => 'b) set"
  "hom G H ==
    {h. h ∈ carrier G -> carrier H &
      (∀ x ∈ carrier G. ∀ y ∈ carrier G. h (x ⊗G y) = h x ⊗H h y)}"

lemma (in group) hom_compose:
  "[|h ∈ hom G H; i ∈ hom H I|] ==> compose (carrier G) i h ∈ hom
G I"
apply (auto simp add: hom_def funcset_compose)
apply (simp add: compose_def funcset_mem)
done

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

lemma iso_refl: "(%x. x) ∈ G ≅ G"
by (simp add: iso_def hom_def inj_on_def bij_betw_def Pi_def)

lemma (in group) iso_sym:
  "h ∈ G ≅ H ==> Inv (carrier G) h ∈ H ≅ G"
apply (simp add: iso_def bij_betw_Inv)
apply (subgoal_tac "Inv (carrier G) h ∈ carrier H → carrier G")
  prefer 2 apply (simp add: bij_betw_imp_funcset [OF bij_betw_Inv])
apply (simp add: hom_def bij_betw_def Inv_f_eq funcset_mem f_Inv_f)
done

lemma (in group) iso_trans:
  "[|h ∈ G ≅ H; i ∈ H ≅ I|] ==> (compose (carrier G) i h) ∈ G ≅ I"
by (auto simp add: iso_def hom_compose bij_betw_compose)

lemma DirProd_commute_iso:
  shows "(λ(x,y). (y,x)) ∈ (G ×× H) ≅ (H ×× G)"
by (auto simp add: iso_def hom_def inj_on_def bij_betw_def Pi_def)

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

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 (struc-
ture) +
  fixes h
  assumes homh: "h ∈ hom G H"

lemma (in group_hom) hom_mult [simp]:

```

```

" [| x ∈ carrier G; y ∈ carrier G | ] ==> h (x ⊗G y) = h x ⊗H h y"
proof -
  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 simp add: funcset_mem)
qed

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

lemma (in group_hom) hom_one [simp]:
  "h 1 = 1H"
proof -
  have "h 1 ⊗H 1H = h 1 ⊗H h 1"
    by (simp add: hom_mult [symmetric] del: hom_mult)
  then show ?thesis by (simp del: r_one)
qed

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

lemma (in group_hom) hom_inv [simp]:
  "x ∈ carrier G ==> h (inv x) = invH (h x)"
proof -
  assume x: "x ∈ carrier G"
  then have "h x ⊗H h (inv x) = 1H"
    by (simp add: hom_mult [symmetric] del: hom_mult)
  also from x have "... = h x ⊗H invH (h x)"
    by (simp add: hom_mult [symmetric] del: hom_mult)
  finally have "h x ⊗H h (inv x) = h x ⊗H invH (h x)" .
  with x show ?thesis by (simp del: H.r_inv H.Units_r_inv)
qed

```

3.7 Commutative Structures

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

```

locale comm_monoid = monoid +
  assumes m_comm: "[| x ∈ carrier G; y ∈ carrier G |] ==> x ⊗ y = y ⊗ x"

lemma (in comm_monoid) m_lcomm:

```

```

"[[x ∈ carrier G; y ∈ carrier G; z ∈ carrier G]] ==>
  x ⊗ (y ⊗ z) = y ⊗ (x ⊗ z)"
proof -
  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) nat_pow_distr:
  "[| x ∈ carrier G; y ∈ carrier G |] ==>
(x ⊗ y) (^) (n::nat) = x (^) n ⊗ y (^) n"
  by (induct n) (simp, simp add: m_ac)

locale comm_group = comm_monoid + group

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

```

```

proof qed (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 (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)

```

3.8 The Lattice of Subgroups of a Group

```

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

lemma (in group) subgroup_inv_equality:
  "[| subgroup H G; x ∈ H |] ==> m_inv (G (| carrier := H |)) x = inv
x"
  apply (rule_tac inv_equality [THEN sym])
  apply (rule group.l_inv [OF subgroup_imp_group, simplified], assumption+)
  apply (rule subsetD [OF subgroup.subset], assumption+)
  apply (rule subsetD [OF subgroup.subset], assumption)
  apply (rule_tac group.inv_closed [OF subgroup_imp_group, simplified],

```

```

assumption+)
done

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

theorem (in group) subgroups_complete_lattice:
  "complete_lattice (| carrier = {H. subgroup H G}, eq = op =, le = op
  ⊆ |)"
  (is "complete_lattice ?L")
proof (rule partial_order.complete_lattice_criterion1)
  show "partial_order ?L" by (rule subgroups_partial_order)
next
  show "∃G. greatest ?L G (carrier ?L)"
  proof
    show "greatest ?L (carrier G) (carrier ?L)"
      by (unfold greatest_def)
      (simp add: subgroup.subset subgroup_self)
  qed
next
  fix A
  assume L: "A ⊆ carrier ?L" and non_empty: "A ~= {}"
  then have Int_subgroup: "subgroup ( $\bigcap$ A) G"
    by (fastsimp intro: subgroups_Inter)
  show "∃I. greatest ?L I (Lower ?L A)"
  proof
    show "greatest ?L ( $\bigcap$ 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  $\subseteq$  H" by fastsimp
    then show "le ?L ?Int H" by simp
  next
    fix H
    assume H: "H  $\in$  Lower ?L A"
    with L Int_subgroup show "le ?L H ?Int"
      by (fastsimp simp: Lower_def intro: Inter_greatest)
  next
    show "A  $\subseteq$  carrier ?L" by (rule L)
  next
    show "?Int  $\in$  carrier ?L" by simp (rule Int_subgroup)
  qed
qed
qed
end

```

```
theory FiniteProduct imports Group begin
```

3.9 Product Operator for Commutative Monoids

3.9.1 Inductive Definition of a Relation for Products over Sets

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

```

inductive_set
  foldSetD :: "[ 'a set, 'b => 'a => 'a, 'a ] => ( 'b set * 'a ) set"
  for D :: " 'a set" and f :: " 'b => 'a => 'a" and e :: 'a
  where
    emptyI [intro]: "e  $\in$  D ==> ({}, e)  $\in$  foldSetD D f e"
    | insertI [intro]: "[| x ~: A; f x y  $\in$  D; (A, y)  $\in$  foldSetD D f e |]
==>
      (insert x A, f x y)  $\in$  foldSetD D f e"

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

constdefs
  foldD :: "[ 'a set, 'b => 'a => 'a, 'a, 'b set ] => 'a"
  "foldD D f e A == THE x. (A, x)  $\in$  foldSetD D f e"

lemma foldSetD_closed:

```

```

" [| (A, z) ∈ foldSetD D f e ; e ∈ D; !!x y. [| x ∈ A; y ∈ D |] ==>
f x y ∈ D
  |] ==> z ∈ D"
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"
apply (erule insert_Diff [THEN subst], rule foldSetD.intros)
  apply auto
done

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 |] ==>
  EX 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

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"
apply (subgoal_tac "x ∈ B")
  prefer 2 apply fast

```



```

    apply (erule insert_Diff [THEN subst], rule foldSetD.intros)
      apply auto
    done

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

lemma (in LCD) finite_imp_foldSetD:
  "[| finite A; A ⊆ B; e ∈ D |] ==> EX 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:
  "e ∈ D ==> ∀A x. A ⊆ B & card A < n --> (A, x) ∈ foldSetD D f e -->
    (∀y. (A, y) ∈ foldSetD D f e --> y = x)"
  apply (induct n)
  apply (auto simp add: less_Suc_eq)
  apply (erule foldSetD.cases)
  apply blast
  apply (erule foldSetD.cases)
  apply blast
  apply clarify

force simplification of card A < card (insert ...).

  apply (erule rev_mp)
  apply (simp add: less_Suc_eq_le)
  apply (rule impI)
  apply (rename_tac xa Aa ya xb Ab yb, case_tac "xa = xb")
  apply (subgoal_tac "Aa = Ab")
  prefer 2 apply (blast elim!: equalityE)
  apply blast

case xa ∉ xb.

  apply (subgoal_tac "Aa - {xb} = Ab - {xa} & xb ∈ Aa & xa ∈ Ab")
  prefer 2 apply (blast elim!: equalityE)
  apply clarify
  apply (subgoal_tac "Aa = insert xb Ab - {xa}")
  prefer 2 apply blast
  apply (subgoal_tac "card Aa ≤ card Ab")
  prefer 2

```

```

    apply (rule Suc_le_mono [THEN subst])
    apply (simp add: card_Suc_Diff1)
  apply (rule_tac A1 = "Aa - {xb}" in finite_imp_foldSetD [THEN exE])
    apply (blast intro: foldSetD_imp_finite finite_Diff)
    apply best
    apply assumption
  apply (frule (1) Diff1_foldSetD)
    apply best
  apply (subgoal_tac "ya = f xb x")
    prefer 2
    apply (subgoal_tac "Aa  $\subseteq$  B")
      prefer 2 apply best
    apply (blast del: equalityCE)
  apply (subgoal_tac "(Ab - {xa}, x)  $\in$  foldSetD D f e")
    prefer 2 apply simp
  apply (subgoal_tac "yb = f xa x")
    prefer 2
    apply (blast del: equalityCE dest: Diff1_foldSetD)
  apply (simp (no_asm_simp))
  apply (rule left_commute)
    apply assumption
    apply best
  apply best
done

lemma (in LCD) foldSetD_determ:
  "[| (A, x)  $\in$  foldSetD D f e; (A, y)  $\in$  foldSetD D f e; e  $\in$  D; A  $\subseteq$  B |]
  ==> y = x"
  by (blast intro: foldSetD_determ_aux [rule_format])

lemma (in LCD) foldD_equality:
  "[| (A, y)  $\in$  foldSetD D f e; e  $\in$  D; A  $\subseteq$  B |] ==> foldD D f e A = y"
  by (unfold foldD_def) (blast intro: foldSetD_determ)

lemma foldD_empty [simp]:
  "e  $\in$  D ==> foldD D f e {} = e"
  by (unfold foldD_def) blast

lemma (in LCD) foldD_insert_aux:
  "[| x ~: A; x  $\in$  B; e  $\in$  D; A  $\subseteq$  B |] ==>
    ((insert x A, v)  $\in$  foldSetD D f e) =
    (EX y. (A, y)  $\in$  foldSetD D f e & v = f x y)"
  apply auto
  apply (rule_tac A1 = A in finite_imp_foldSetD [THEN exE])
    apply (fastsimp dest: foldSetD_imp_finite)
    apply assumption
    apply assumption
  apply (blast intro: foldSetD_determ)

```

done

```
lemma (in LCD) foldD_insert:
  "[| finite A; x ~: A; x ∈ B; e ∈ D; A ⊆ B |] ==>
    foldD D f e (insert x A) = f x (foldD D f e A)"
  apply (unfold foldD_def)
  apply (simp add: foldD_insert_aux)
  apply (rule the_equality)
  apply (auto intro: finite_imp_foldSetD
    cong add: conj_cong simp add: foldD_def [symmetric] foldD_equality)
done
```

```
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 add: foldD_empty)
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"
  apply (induct set: finite)
  apply simp
  apply (auto simp add: left_commute foldD_insert)
done
```

```
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)"
  apply (induct set: finite)
  apply simp
  apply (simp add: foldD_insert foldD_commute Int_insert_left insert_absorb
    Int_mono2 Un_subset_iff)
done
```

```
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 $f :: 'a \Rightarrow 'a \Rightarrow 'a$ instead of $'b \Rightarrow 'a \Rightarrow 'a$.

```

locale ACeD =
  fixes D :: "'a set"
  and f :: "'a => 'a => 'a"    (infixl "." 70)
  and e :: 'a
  assumes ident [simp]: "x ∈ D ==> x · e = x"
  and commute: "[| x ∈ D; y ∈ D |] ==> x · y = y · x"
  and assoc: "[| x ∈ D; y ∈ D; z ∈ D |] ==> (x · y) · z = x · (y · z)"
  and e_closed [simp]: "e ∈ D"
  and f_closed [simp]: "[| x ∈ D; y ∈ D |] ==> x · y ∈ D"

```

```

lemma (in ACeD) left_commute:
  "[| x ∈ D; y ∈ D; z ∈ D |] ==> x · (y · z) = y · (x · z)"
proof -
  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)"
apply (induct set: finite)
apply (simp add: left_commute LCD.foldD_closed [OF LCD.intro [of D]])
apply (simp add: AC insert_absorb Int_insert_left
  LCD.foldD_insert [OF LCD.intro [of D]]
  LCD.foldD_closed [OF LCD.intro [of D]]
  Int_mono2 Un_subset_iff)

```

done

```
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]] Un_subset_iff)
```

3.9.2 Products over Finite Sets

```
constdefs (structure G)
  finprod :: "('b, 'm) monoid_scheme, 'a => 'b, 'a set] => 'b"
  "finprod G f A == if finite A
    then foldD (carrier G) (mult G o f) 1 A
    else undefined"
```

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

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

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

```
translations
  "⊗ i:A. b" == "finprod ⊗ i (%i. b) A"
  — Beware of argument permutation!
```

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

```
declare funcsetI [intro]
  funcset_mem [dest]
```

context comm_monoid begin

```
lemma finprod_insert [simp]:
  "[| finite F; a ∉ F; f ∈ F -> carrier G; f a ∈ carrier G |] ==>
    finprod G f (insert a F) = f a ⊗ finprod G f F"
  apply (rule trans)
  apply (simp add: finprod_def)
  apply (rule trans)
  apply (rule LCD.foldD_insert [OF LCD.intro [of "insert a F"]])
  apply simp
  apply (rule m_lcomm)
  apply fast
  apply fast
```

```

        apply assumption
        apply (fastsimp intro: m_closed)
        apply simp+
        apply fast
        apply (auto simp add: finprod_def)
        done

lemma finprod_one [simp]:
  "finite A ==> ( $\bigotimes$  i:A. 1) = 1"
proof (induct set: finite)
  case empty show ?case by simp
next
  case (insert a A)
  have "(%i. 1)  $\in$  A -> carrier G" by auto
  with insert show ?case by simp
qed

lemma finprod_closed [simp]:
  fixes A
  assumes fin: "finite A" and f: "f  $\in$  A -> carrier G"
  shows "finprod G f A  $\in$  carrier G"
using fin f
proof induct
  case empty show ?case by simp
next
  case (insert a A)
  then have a: "f a  $\in$  carrier G" by fast
  from insert have A: "f  $\in$  A -> carrier G" by fast
  from insert A a show ?case by simp
qed

lemma funcset_Int_left [simp, intro]:
  "[| f  $\in$  A -> C; f  $\in$  B -> C |] ==> f  $\in$  A Int B -> C"
  by fast

lemma funcset_Un_left [iff]:
  "(f  $\in$  A Un B -> C) = (f  $\in$  A -> C & f  $\in$  B -> C)"
  by fast

lemma finprod_Un_Int:
  "[| finite A; finite B; g  $\in$  A -> carrier G; g  $\in$  B -> carrier G |] ==>
    finprod G g (A Un B)  $\otimes$  finprod G g (A Int B) =
    finprod G g A  $\otimes$  finprod G g B"
— The reversed orientation looks more natural, but LOOPS as a simprule!
proof (induct set: finite)
  case empty then show ?case by (simp add: finprod_closed)
next
  case (insert a A)
  then have a: "g a  $\in$  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 finprod_closed
    Int_mono2 Un_subset_iff)
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"
  apply (subst finprod_Un_Int [symmetric])
  apply (auto simp add: finprod_closed)
done

```

```

lemma finprod_multf:
  "[| finite A; f ∈ A -> carrier G; g ∈ A -> carrier G |] ==>
    finprod G (%x. f x ⊗ g x) A = (finprod G f A ⊗ finprod G g A)"
proof (induct set: finite)
  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

```

```

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 fastsimp
    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 fastsimp
    qed
    also from insert have "... = g x ⊗ finprod G g B" by fastsimp
    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 add: finprod_def)
qed
qed

lemma finprod_cong:
  "[| A = B; f ∈ B -> carrier G = True;
    !!i. i ∈ B ==> f i = g i |] ==> finprod G f A = finprod G g B"

  by (rule finprod_cong') force+

Usually, if this rule causes a failed congruence proof error, the reason is that
the premise  $g \in B \rightarrow \text{carrier } G$  cannot be shown. Adding Pi_def to the
simpset is often useful. For this reason, comm_monoid.finprod_cong is not
added to the simpset by default.

end

declare funcsetI [rule del]
  funcset_mem [rule del]

context comm_monoid begin

lemma finprod_0 [simp]:
  "f ∈ {0::nat} -> carrier G ==> finprod G f {...0} = f 0"
by (simp add: Pi_def)

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_mult [simp]:
  "[| f ∈ {..n} -> carrier G; g ∈ {..n} -> carrier G |] ==>
    finprod G (%i. f i ⊗ g i) {..n::nat} =
    finprod G f {..n} ⊗ finprod G g {..n}"
  by (induct n) (simp_all add: m_ac Pi_def)

lemma finprod_reindex:
  assumes fin: "finite A"
  shows "f : (h ' A) → carrier G ==>
    inj_on h A ==> finprod G f (h ' A) = finprod G (%x. f (h x)) A"
  using fin apply induct
  apply (auto simp add: finprod_insert Pi_def)
done

lemma finprod_const:
  assumes fin [simp]: "finite A"
  and a [simp]: "a : carrier G"
  shows "finprod G (%x. a) A = a (^) card A"
  using fin apply induct
  apply force
  apply (subst finprod_insert)
  apply auto
  apply (force simp add: Pi_def)
  apply (subst m_comm)
  apply auto
done

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)
end

end

```

```

theory Coset imports Group begin

```

4 Cosets and Quotient Groups

```

constdefs (structure G)
  r_coset    :: "[_, 'a set, 'a]  $\Rightarrow$  'a set"      (infixl "#>" 60)
  "H #> a  $\equiv \bigcup_{h \in H}. \{h \otimes a\}"$ 

  l_coset    :: "[_, 'a, 'a set]  $\Rightarrow$  'a set"      (infixl "<#" 60)
  "a <# H  $\equiv \bigcup_{h \in H}. \{a \otimes h\}"$ 

  RCOSETS    :: "[_, 'a set]  $\Rightarrow$  ('a set)set"      ("rcosets" 80)
  "rcosets H  $\equiv \bigcup_{a \in \text{carrier } G}. \{H \#> a\}"$ 

  set_mult   :: "[_, 'a set, 'a set]  $\Rightarrow$  'a set" (infixl "<#>" 60)
  "H <#> K  $\equiv \bigcup_{h \in H}. \bigcup_{k \in K}. \{h \otimes k\}"$ 

  SET_INV    :: "[_, 'a set]  $\Rightarrow$  'a set"      ("set'_inv" 80)
  "set_inv H  $\equiv \bigcup_{h \in H}. \{\text{inv } h\}"$ 

locale normal = subgroup + group +
  assumes coset_eq: "( $\forall x \in \text{carrier } G. H \#> x = x <# H$ )"

abbreviation
  normal_rel :: "[ 'a set, ('a, 'b) monoid_scheme]  $\Rightarrow$  bool" (infixl "<" 60) where
    "H < G  $\equiv \text{normal } H \ G"$ 

```

4.1 Basic Properties of Cosets

```

lemma (in group) coset_mult_assoc:
  "[| M  $\subseteq$  carrier G; g  $\in$  carrier G; h  $\in$  carrier G |]
   ==> (M #> g) #> h = M #> (g  $\otimes$  h)"
by (force simp add: r_coset_def m_assoc)

lemma (in group) coset_mult_one [simp]: "M  $\subseteq$  carrier G ==> M #> 1 = M"
by (force simp add: r_coset_def)

lemma (in group) coset_mult_inv1:

```

```

    "[| M #> (x ⊗ (inv y)) = M; x ∈ carrier G ; y ∈ carrier G;
      M ⊆ carrier G |] ==> M #> x = M #> y"
  apply (erule subst [of concl: "%z. M #> x = z #> y"])
  apply (simp add: coset_mult_assoc m_assoc)
done

lemma (in group) coset_mult_inv2:
  "[| M #> x = M #> y; x ∈ carrier G; y ∈ carrier G; M ⊆ carrier
    G |]
    ==> M #> (x ⊗ (inv y)) = M "
  apply (simp add: coset_mult_assoc [symmetric])
  apply (simp add: coset_mult_assoc)
done

lemma (in group) coset_join1:
  "[| H #> x = H; x ∈ carrier G; subgroup H G |] ==> x ∈ H"
  apply (erule subst)
  apply (simp add: r_coset_def)
  apply (blast intro: l_one subgroup.one_closed sym)
done

lemma (in group) solve_equation:
  "[subgroup H G; x ∈ H; y ∈ H] ==> ∃ h ∈ H. y = h ⊗ x"
  apply (rule bexI [of _ "y ⊗ (inv x)"])
  apply (auto simp add: subgroup.m_closed subgroup.m_inv_closed m_assoc
    subgroup.subset [THEN subsetD])
done

lemma (in group) repr_independence:
  "[y ∈ H #> x; x ∈ carrier G; subgroup H G] ==> H #> x = H #> y"
  by (auto simp add: r_coset_def m_assoc [symmetric]
    subgroup.subset [THEN subsetD]
    subgroup.m_closed solve_equation)

lemma (in group) coset_join2:
  "[x ∈ carrier G; subgroup H G; x ∈ H] ==> H #> x = H"
  — Alternative proof is to put x = 1 in repr_independence.
  by (force simp add: subgroup.m_closed r_coset_def solve_equation)

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)

Really needed?

```
lemma (in group) transpose_inv:
  "[| x  $\otimes$  y = z; x  $\in$  carrier G; y  $\in$  carrier G; z  $\in$  carrier G |]
  ==> (inv x)  $\otimes$  z = y"
by (force simp add: m_assoc [symmetric])
```

```
lemma (in group) rcos_self: "[| x  $\in$  carrier G; subgroup H G |] ==> x
 $\in$  H #> x"
apply (simp add: r_coset_def)
apply (blast intro: sym l_one subgroup.subset [THEN subsetD]
        subgroup.one_closed)
done
```

Opposite of "repr_independence"

```
lemma (in group) repr_independenceD:
  assumes "subgroup H G"
  assumes ycarr: "y  $\in$  carrier G"
  and repr: "H #> x = H #> y"
  shows "y  $\in$  H #> x"
proof -
  interpret subgroup H G by fact
  show ?thesis apply (subst repr)
  apply (intro rcos_self)
  apply (rule ycarr)
  apply (rule is_subgroup)
  done
qed
```

Elements of a right coset are in the carrier

```
lemma (in subgroup) elemrcos_carrier:
  assumes "group G"
  assumes acarr: "a  $\in$  carrier G"
  and a': "a'  $\in$  H #> a"
  shows "a'  $\in$  carrier G"
proof -
  interpret group G by fact
  from subset and acarr
  have "H #> a  $\subseteq$  carrier G" by (rule r_coset_subset_G)
  from this and a'
  show "a'  $\in$  carrier G"
  by fast
qed
```

```
lemma (in subgroup) rcos_const:
  assumes "group G"
  assumes hH: "h  $\in$  H"
  shows "H #> h = H"

```

```

proof -
  interpret group G by fact
  show ?thesis apply (unfold r_coset_def)
    apply rule
    apply rule
    apply clarsimp
    apply (intro subgroup.m_closed)
    apply (rule is_subgroup)
    apply assumption
    apply (rule hH)
    apply rule
    apply simp
proof -
  fix h'
  assume h'H: "h' ∈ H"
  note carr = hH[THEN mem_carrier] h'H[THEN mem_carrier]
  from carr
  have a: "h' = (h' ⊗ inv h) ⊗ h" by (simp add: m_assoc)
  from h'H hH
  have "h' ⊗ inv h ∈ H" by simp
  from this and a
  show "∃x∈H. h' = x ⊗ h" by fast
qed
qed

```

Step one for lemma rcos_module

```

lemma (in subgroup) rcos_module_imp:
  assumes "group G"
  assumes xcarr: "x ∈ carrier G"
    and x'cos: "x' ∈ H #> x"
  shows "(x' ⊗ inv x) ∈ H"
proof -
  interpret group G by fact
  from xcarr x'cos
    have x'carr: "x' ∈ carrier G"
    by (rule elemrcos_carrier[OF is_group])
  from xcarr
    have ixcarr: "inv x ∈ carrier G"
    by simp
  from x'cos
    have "∃h∈H. x' = h ⊗ x"
    unfolding r_coset_def
    by fast
  from this
    obtain h
    where hH: "h ∈ H"
    and x': "x' = h ⊗ x"
    by auto
  from hH and subset

```

```

      have hcarr: "h ∈ carrier G" by fast
    note carr = xcarr x'carr hcarr
  from x' and carr
    have "x' ⊗ (inv x) = (h ⊗ x) ⊗ (inv x)" by fast
  also from carr
    have "... = h ⊗ (x ⊗ inv x)" by (simp add: m_assoc)
  also from carr
    have "... = h ⊗ 1" by simp
  also from carr
    have "... = h" by simp
  finally
    have "x' ⊗ (inv x) = h" by simp
  from hH this
    show "x' ⊗ (inv x) ∈ H" by simp
qed

```

Step two for lemma rcos_module

```

lemma (in subgroup) rcos_module_rev:
  assumes "group G"
  assumes carr: "x ∈ carrier G" "x' ∈ carrier G"
    and xixH: "(x' ⊗ inv x) ∈ H"
  shows "x' ∈ H #> x"
proof -
  interpret group G by fact
  from xixH
    have "∃h∈H. x' ⊗ (inv x) = h" by fast
  from this
    obtain h
      where hH: "h ∈ H"
      and hsym: "x' ⊗ (inv x) = h"
    by fast
  from hH subset have hcarr: "h ∈ carrier G" by simp
  note carr = carr hcarr
  from hsym[symmetric] have "h ⊗ x = x' ⊗ (inv x) ⊗ x" by fast
  also from carr
    have "... = x' ⊗ ((inv x) ⊗ x)" by (simp add: m_assoc)
  also from carr
    have "... = x' ⊗ 1" by (simp add: l_inv)
  also from carr
    have "... = x'" by simp
  finally
    have "h ⊗ x = x'" by simp
  from this[symmetric] and hH
    show "x' ∈ H #> x"
    unfolding r_coset_def
    by fast
qed

```

Module property of right cosets

```

lemma (in subgroup) rcos_module:
  assumes "group G"
  assumes carr: "x ∈ carrier G" "x' ∈ carrier G"
  shows "(x' ∈ H #> x) = (x' ⊗ inv x ∈ H)"
proof -
  interpret group G by fact
  show ?thesis proof assume "x' ∈ H #> x"
    from this and carr
    show "x' ⊗ inv x ∈ H"
      by (intro rcos_module_imp[OF is_group])
  next
    assume "x' ⊗ inv x ∈ H"
    from this and carr
    show "x' ∈ H #> x"
      by (intro rcos_module_rev[OF is_group])
  qed
qed

```

Right cosets are subsets of the carrier.

```

lemma (in subgroup) rcosets_carrier:
  assumes "group G"
  assumes XH: "X ∈ rcosets H"
  shows "X ⊆ carrier G"
proof -
  interpret group G by fact
  from XH have "∃x ∈ carrier G. X = H #> x"
    unfolding RCOSETS_def
    by fast
  from this
  obtain x
    where xcarr: "x ∈ carrier G"
    and X: "X = H #> x"
    by fast
  from subset and xcarr
  show "X ⊆ carrier G"
    unfolding X
    by (rule r_coset_subset_G)
qed

```

Multiplication of general subsets

```

lemma (in monoid) set_mult_closed:
  assumes Acarr: "A ⊆ carrier G"
  and Bcarr: "B ⊆ carrier G"
  shows "A <#> B ⊆ carrier G"
apply rule apply (simp add: set_mult_def, clarsimp)
proof -
  fix a b
  assume "a ∈ A"
  from this and Acarr

```

```

    have acarr: "a ∈ carrier G" by fast

    assume "b ∈ B"
    from this and Bcarr
      have bcarr: "b ∈ carrier G" by fast

    from acarr bcarr
      show "a ⊗ b ∈ carrier G" by (rule m_closed)
qed

lemma (in comm_group) mult_subgroups:
  assumes subH: "subgroup H G"
    and subK: "subgroup K G"
  shows "subgroup (H <#> K) G"
apply (rule subgroup.intro)
  apply (intro set_mult_closed subgroup.subset[OF subH] subgroup.subset[OF
subK])
  apply (simp add: set_mult_def) apply clarsimp defer 1
  apply (simp add: set_mult_def) defer 1
  apply (simp add: set_mult_def, clarsimp) defer 1
proof -
  fix ha hb ka kb
  assume haH: "ha ∈ H" and hbH: "hb ∈ H" and kaK: "ka ∈ K" and kbK:
"kb ∈ K"
  note carr = haH[THEN subgroup.mem_carrier[OF subH]] hbH[THEN subgroup.mem_carrier[OF
subH]]
      kaK[THEN subgroup.mem_carrier[OF subK]] kbK[THEN subgroup.mem_carrier[OF
subK]]
  from carr
    have "(ha ⊗ ka) ⊗ (hb ⊗ kb) = ha ⊗ (ka ⊗ hb) ⊗ kb" by (simp add:
m_assoc)
  also from carr
    have "... = ha ⊗ (hb ⊗ ka) ⊗ kb" by (simp add: m_comm)
  also from carr
    have "... = (ha ⊗ hb) ⊗ (ka ⊗ kb)" by (simp add: m_assoc)
  finally
    have eq: "(ha ⊗ ka) ⊗ (hb ⊗ kb) = (ha ⊗ hb) ⊗ (ka ⊗ kb)" .

  from haH hbH have hH: "ha ⊗ hb ∈ H" by (simp add: subgroup.m_closed[OF
subH])
  from kaK kbK have kK: "ka ⊗ kb ∈ K" by (simp add: subgroup.m_closed[OF
subK])

  from hH and kK and eq
    show "∃ h' ∈ H. ∃ k' ∈ K. (ha ⊗ ka) ⊗ (hb ⊗ kb) = h' ⊗ k'" by fast
next
  have "1 = 1 ⊗ 1" by simp
  from subgroup.one_closed[OF subH] subgroup.one_closed[OF subK] this
    show "∃ h ∈ H. ∃ k ∈ K. 1 = h ⊗ k" by fast

```



```

next
  fix h k
  assume hH: "h ∈ H"
  and kK: "k ∈ K"

  from hH[THEN subgroup.mem_carrier[OF subH]] kK[THEN subgroup.mem_carrier[OF
subK]]
    have "inv (h ⊗ k) = inv h ⊗ inv k" by (simp add: inv_mult_group
m_comm)

  from subgroup.m_inv_closed[OF subH hH] and subgroup.m_inv_closed[OF
subK kK] and this
    show "∃ha∈H. ∃ka∈K. inv (h ⊗ k) = ha ⊗ ka" by fast
qed

lemma (in subgroup) lcos_module_rev:
  assumes "group G"
  assumes carr: "x ∈ carrier G" "x' ∈ carrier G"
  and xixH: "(inv x ⊗ x') ∈ H"
  shows "x' ∈ x <# H"
proof -
  interpret group G by fact
  from xixH
    have "∃h∈H. (inv x) ⊗ x' = h" by fast
  from this
    obtain h
    where hH: "h ∈ H"
    and hsym: "(inv x) ⊗ x' = h"
    by fast

  from hH subset have hcarr: "h ∈ carrier G" by simp
  note carr = carr hcarr
  from hsym[symmetric] have "x ⊗ h = x ⊗ ((inv x) ⊗ x')" by fast
  also from carr
    have "... = (x ⊗ (inv x)) ⊗ x'" by (simp add: m_assoc[symmetric])
  also from carr
    have "... = 1 ⊗ x'" by simp
  also from carr
    have "... = x'" by simp
  finally
    have "x ⊗ h = x'" by simp

  from this[symmetric] and hH
    show "x' ∈ x <# H"
    unfolding l_coset_def
    by fast
qed

```

4.2 Normal subgroups

```
lemma normal_imp_subgroup: "H < G  $\implies$  subgroup H G"
  by (simp add: normal_def subgroup_def)
```

```
lemma (in group) normalI:
  "subgroup H G  $\implies$  ( $\forall x \in \text{carrier } G. H \#> x = x <\# H$ )  $\implies$  H < G"
  by (simp add: normal_def normal_axioms_def prems)
```

```
lemma (in normal) inv_op_closed1:
  "[x  $\in$  carrier G; h  $\in$  H]  $\implies$  (inv x)  $\otimes$  h  $\otimes$  x  $\in$  H"
apply (insert coset_eq)
apply (auto simp add: l_coset_def r_coset_def)
apply (drule bspec, assumption)
apply (drule equalityD1 [THEN subsetD], blast, clarify)
apply (simp add: m_assoc)
apply (simp add: m_assoc [symmetric])
done
```

```
lemma (in normal) inv_op_closed2:
  "[x  $\in$  carrier G; h  $\in$  H]  $\implies$  x  $\otimes$  h  $\otimes$  (inv x)  $\in$  H"
apply (subgoal_tac "inv (inv x)  $\otimes$  h  $\otimes$  (inv x)  $\in$  H")
apply (simp add: )
apply (blast intro: inv_op_closed1)
done
```

Alternative characterization of normal subgroups

```
lemma (in group) normal_inv_iff:
  "(N < G) =
    (subgroup N G & ( $\forall x \in \text{carrier } G. \forall h \in N. x \otimes h \otimes (\text{inv } x) \in N$ ))"
  (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: " $\bigwedge x. x \in \text{carrier } G \implies \forall h \in N. x \otimes h \otimes \text{inv } x \in N$ " by auto
  hence sb: "N  $\subseteq$  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  $\in$  carrier G"
    show "( $\bigcup h \in N. \{h \otimes x\}$ ) = ( $\bigcup h \in N. \{x \otimes h\}$ )"
    proof
      show "( $\bigcup h \in N. \{h \otimes x\}$ )  $\subseteq$  ( $\bigcup h \in N. \{x \otimes h\}$ )"
      proof clarify
        fix n
        assume n: "n  $\in$  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
qed

```

4.3 More Properties of Cosets

```

lemma (in group) lcos_m_assoc:
  "[| M ⊆ carrier G; g ∈ carrier G; h ∈ carrier G |]
  ==> g <# (h <# M) = (g ⊗ h) <# M"
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_swap:
  "[| y ∈ x <# H; x ∈ carrier G; subgroup H G |] ==> x ∈ y <# H"
proof (simp add: l_coset_def)
  assume "∃ h ∈ H. y = x ⊗ h"
  and x: "x ∈ carrier G"
  and sb: "subgroup H G"
  then obtain h' where h': "h' ∈ H & x ⊗ h' = y" by blast
  show "∃ h ∈ H. x = y ⊗ h"
  proof
    show "x = y ⊗ inv h'" using h' x sb
      by (auto simp add: m_assoc subgroup.subset [THEN subsetD])

```

```

    show "inv h' ∈ H" using h' sb
    by (auto simp add: subgroup.subset [THEN subsetD] subgroup.m_inv_closed)
  qed
qed

```

```

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_repr_imp_subset:
  assumes y: "y ∈ x <# H" and x: "x ∈ carrier G" and sb: "subgroup H
G"
  shows "y <# H ⊆ x <# H"
proof -
  from y
  obtain h' where "h' ∈ H" "x ⊗ h' = y" by (auto simp add: l_coset_def)
  thus ?thesis using x sb
  by (auto simp add: l_coset_def m_assoc
    subgroup.subset [THEN subsetD] subgroup.m_closed)
qed

```

```

lemma (in group) l_repr_independence:
  assumes y: "y ∈ x <# H" and x: "x ∈ carrier G" and sb: "subgroup H
G"
  shows "x <# H = y <# H"
proof
  show "x <# H ⊆ y <# H"
  by (rule l_repr_imp_subset,
    (blast intro: l_coset_swap l_coset_carrier y x sb)+)
  show "y <# H ⊆ x <# H" by (rule l_repr_imp_subset [OF y x sb])
qed

```

```

lemma (in group) setmult_subset_G:
  "[| H ⊆ carrier G; K ⊆ carrier G |] ==> H <#> K ⊆ carrier G"
by (auto simp add: set_mult_def subsetD)

```

```

lemma (in group) subgroup_mult_id: "subgroup H G ==> H <#> H = H"
apply (auto simp add: subgroup.m_closed set_mult_def Sigma_def image_def)
apply (rule_tac x = x in bexI)
apply (rule bexI [of _ "1"])
apply (auto simp add: subgroup.m_closed subgroup.one_closed
    r_one subgroup.subset [THEN subsetD])
done

```

4.3.1 Set of Inverses of an r_coset.

```

lemma (in normal) rcos_inv:

```

```

    assumes x:      "x ∈ carrier G"
    shows "set_inv (H #> x) = H #> (inv x)"
  proof (simp add: r_coset_def SET_INV_def x inv_mult_group, safe)
    fix h
    assume "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 prems)
      show "inv x ⊗ inv h ∈ {inv x ⊗ inv h ⊗ x ⊗ inv x}"
      by (simp add: prems m_assoc)
    qed
  next
  fix h
  assume "h ∈ H"
  show "h ⊗ inv x ∈ (⋃ j∈H. {inv x ⊗ inv j})"
  proof
    show "x ⊗ inv h ⊗ inv x ∈ H"
    by (simp add: inv_op_closed2 prems)
    show "h ⊗ inv x ∈ {inv x ⊗ inv (x ⊗ inv h ⊗ inv x)}"
    by (simp add: prems m_assoc [symmetric] inv_mult_group)
  qed
qed

```

4.3.2 Theorems for <#> with #> or <#.

```

lemma (in group) setmult_rcos_assoc:
  "[H ⊆ carrier G; K ⊆ carrier G; x ∈ carrier G]
  ⇒ H <#> (K #> x) = (H <#> K) #> x"
by (force simp add: r_coset_def set_mult_def m_assoc)

lemma (in group) rcos_assoc_lcos:
  "[H ⊆ carrier G; K ⊆ carrier G; x ∈ carrier G]
  ⇒ (H #> x) <#> K = H <#> (x <# K)"
by (force simp add: r_coset_def l_coset_def set_mult_def m_assoc)

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 subset
    r_coset_subset_G 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 prems)

```

```

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 prems)

```

4.3.3 An Equivalence Relation

```

constdefs (structure G)
  r_congruent :: "[('a,'b)monoid_scheme, 'a set] ⇒ ('a*'a)set"
    ("rcong₂ _")
  "rcong H ≡ {(x,y). x ∈ carrier G & y ∈ carrier G & inv x ⊗ 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 equiv.intro)
    show "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 add: m_inv_closed)
        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)
      qed
    qed
  qed

```

```

      thus "inv x  $\otimes$  z  $\in$  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  $\in$  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

```

4.3.4 Two Distinct Right Cosets are Disjoint

```

lemma (in group) rcos_equation:
  assumes "subgroup H G"
  assumes p: "ha  $\otimes$  a = h  $\otimes$  b" "a  $\in$  carrier G" "b  $\in$  carrier G" "h  $\in$  H"
  "ha  $\in$  H" "hb  $\in$  H"
  shows "hb  $\otimes$  a  $\in$  ( $\bigcup_{h \in H}. \{h \otimes b\}$ )"
proof -
  interpret subgroup H G by fact
  from p show ?thesis apply (rule_tac UN_I [of "hb  $\otimes$  ((inv ha)  $\otimes$  h)"])
    apply (simp add: )
    apply (simp add: m_assoc transpose_inv)
    done
qed

```

```

lemma (in group) rcos_disjoint:
  assumes "subgroup H G"
  assumes p: "a  $\in$  rcosets H" "b  $\in$  rcosets H" "a  $\neq$  b"
  shows "a  $\cap$  b = {}"
proof -
  interpret subgroup H G by fact
  from p show ?thesis apply (simp add: RCOSETS_def r_coset_def)
    apply (blast intro: rcos_equation prems sym)
    done
qed

```

4.4 Further lemmas for `r_congruent`

The relation is a congruence

```

lemma (in normal) congruent_rcong:
  shows "congruent2 (rcong H) (rcong H) ( $\lambda$ a b. a  $\otimes$  b <# H)"
proof (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 add: Units_eq)
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  $\otimes$  (inv c  $\otimes$  c)  $\otimes$  b" by simp
also from carr and inv_closed
  have "... = (inv a  $\otimes$  inv c)  $\otimes$  (c  $\otimes$  b)" by (force cong: m_assoc)
also from carr and inv_closed
  have "... = inv (c  $\otimes$  a)  $\otimes$  (c  $\otimes$  b)" by (simp add: inv_mult_group)
finally
  have "inv a  $\otimes$  b = inv (c  $\otimes$  a)  $\otimes$  (c  $\otimes$  b)" .
from abH and this
  have "inv (c  $\otimes$  a)  $\otimes$  (c  $\otimes$  b)  $\in$  H" by simp

from carr and this
  have "(c  $\otimes$  b)  $\in$  (c  $\otimes$  a) <# H"
  by (simp add: lcos_module_rev[OF is_group])
from carr and this and is_subgroup
  show "(c  $\otimes$  a) <# H = (c  $\otimes$  b) <# H" by (intro l_repr_independence,
simp+)
qed

```

4.5 Order of a Group and Lagrange's Theorem

```

constdefs
  order :: "('a, 'b) monoid_scheme  $\Rightarrow$  nat"
  "order S  $\equiv$  card (carrier S)"

lemma (in group) rcosets_part_G:
  assumes "subgroup H G"
  shows " $\bigcup$  (rcosets H) = carrier G"
proof -
  interpret subgroup H G by fact
  show ?thesis
  apply (rule equalityI)
  apply (force simp add: RCOSETS_def r_coset_def)
  apply (auto simp add: RCOSETS_def intro: rcos_self prems)
  done
qed

lemma (in group) cosets_finite:
  " $\llbracket c \in \text{rcosets } H; \ H \subseteq \text{carrier } G; \ \text{finite } (\text{carrier } G) \rrbracket \Longrightarrow \text{finite } c$ "
apply (auto simp add: RCOSETS_def)
apply (simp add: r_coset_subset_G [THEN finite_subset])
done

```

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

```

lemma (in group) inj_on_f:
  " $\llbracket H \subseteq \text{carrier } G; \ a \in \text{carrier } G \rrbracket \Longrightarrow \text{inj\_on } (\lambda y. y \otimes \text{inv } a) \ (\text{H} \#> a)$ "
apply (rule inj_onI)

```

```

apply (subgoal_tac "x ∈ carrier G & y ∈ carrier G")
  prefer 2 apply (blast intro: r_coset_subset_G [THEN subsetD])
apply (simp add: subsetD)
done

```

```

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:
  "[c ∈ rcosets H; H ⊆ carrier G; finite(carrier G)]
  ⇒ card c = card H"
apply (auto simp add: RCOSETS_def)
apply (rule card_bij_eq)
  apply (rule inj_on_f, assumption+)
  apply (force simp add: m_assoc subsetD r_coset_def)
  apply (rule inj_on_g, assumption+)
  apply (force simp add: m_assoc subsetD r_coset_def)

```

The sets $H \#> a$ and H are finite.

```

  apply (simp add: r_coset_subset_G [THEN finite_subset])
apply (blast intro: finite_subset)
done

```

```

lemma (in group) rcosets_subset_PowG:
  "subgroup H G ⇒ rcosets H ⊆ Pow(carrier G)"
apply (simp add: RCOSETS_def)
apply (blast dest: r_coset_subset_G subgroup.subset)
done

```

```

theorem (in group) lagrange:
  "[finite(carrier G); subgroup H G]
  ⇒ card(rcosets H) * card(H) = order(G)"
apply (simp (no_asm_simp) add: order_def rcosets_part_G [symmetric])
apply (subst mult_commute)
apply (rule card_partition)
  apply (simp add: rcosets_subset_PowG [THEN finite_subset])
  apply (simp add: rcosets_part_G)
  apply (simp add: card_cosets_equal subgroup.subset)
apply (simp add: rcos_disjoint)
done

```

4.6 Quotient Groups: Factorization of a Group

constdefs

```

FactGroup :: "('a,'b) monoid_scheme, 'a set] ⇒ ('a set) monoid"
  (infixl "Mod" 65)
  — Actually defined for groups rather than monoids
FactGroup G H ≡

```

```

(carrier = rcosetsG H, mult = set_mult G, one = H)"

lemma (in normal) setmult_closed:
  "[K1 ∈ rcosets H; K2 ∈ rcosets H] ⇒ K1 <#> K2 ∈ rcosets H"
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 (auto simp add: RCOSETS_def rcos_sum m_assoc)

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
prems)

theorem (in normal) factorgroup_is_group:
  "group (G Mod H)"
apply (simp add: FactGroup_def)
apply (rule groupI)
  apply (simp add: setmult_closed)
  apply (simp add: normal_imp_subgroup subgroup_in_rcosets [OF is_group])
  apply (simp add: restrictI setmult_closed rcosets_assoc)
  apply (simp add: normal_imp_subgroup
    subgroup_in_rcosets rcosets_mult_eq)
apply (auto dest: rcosets_inv_mult_group_eq simp add: setinv_closed)
done

lemma mult_FactGroup [simp]: "X ⊗(G Mod H) X' = X <#>G X'"
by (simp add: FactGroup_def)

lemma (in normal) inv_FactGroup:
  "X ∈ carrier (G Mod H) ⇒ invG Mod H X = set_inv X"
apply (rule group.inv_equality [OF factorgroup_is_group])

```

```

apply (simp_all add: FactGroup_def setinv_closed rcosets_inv_mult_group_eq)
done

```

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

```

lemma (in normal) r_coset_hom_Mod:
  "(\a. H #> a) \in hom G (G Mod H)"
  by (auto simp add: FactGroup_def RCOSETS_def Pi_def hom_def rcos_sum)

```

4.7 The First Isomorphism Theorem

The quotient by the kernel of a homomorphism is isomorphic to the range of that homomorphism.

```

constdefs
  kernel :: "('a, 'm) monoid_scheme \Rightarrow ('b, 'n) monoid_scheme \Rightarrow
    ('a \Rightarrow 'b) \Rightarrow 'a set"
  — the kernel of a homomorphism
  "kernel G H h \equiv \{x. x \in carrier G & h x = 1_H\}"

```

```

lemma (in group_hom) subgroup_kernel: "subgroup (kernel G H h) G"
apply (rule subgroup.intro)
apply (auto simp add: kernel_def group.intro prems)
done

```

The kernel of a homomorphism is a normal subgroup

```

lemma (in group_hom) normal_kernel: "(kernel G H h) <\ G"
apply (simp add: G.normal_inv_iff subgroup_kernel)
apply (simp add: kernel_def)
done

```

```

lemma (in group_hom) FactGroup_nonempty:
  assumes X: "X \in carrier (G Mod kernel G H h)"
  shows "X \neq \{\}"
proof -
  from X
  obtain g where "g \in carrier G"
    and "X = kernel G H h #> g"
    by (auto simp add: FactGroup_def RCOSETS_def)
  thus ?thesis
    by (auto simp add: kernel_def r_coset_def image_def intro: hom_one)
qed

```

```

lemma (in group_hom) FactGroup_contents_mem:
  assumes X: "X \in carrier (G Mod (kernel G H h))"
  shows "contents (h`X) \in carrier H"
proof -
  from X
  obtain g where g: "g \in 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 image_def
g)
  thus ?thesis by (auto simp add: g)
qed

```

```

lemma (in group_hom) FactGroup_hom:
  "(\X. contents (h'X)) \in hom (G Mod (kernel G H h)) H"
apply (simp add: hom_def FactGroup_contents_mem normal.factorgroup_is_group
[OF normal_kernel] group.axioms monoid.m_closed)
proof (simp add: hom_def funcsetI FactGroup_contents_mem, intro ballI)

```

```

  fix X and X'
  assume X: "X \in carrier (G Mod kernel G H h)"
  and X': "X' \in carrier (G Mod kernel G H h)"
  then
  obtain g and g'
    where "g \in carrier G" and "g' \in carrier G"
    and "X = kernel G H h #> g" and "X' = kernel G H h #> g'"
  by (auto simp add: FactGroup_def RCOSETS_def)
  hence all: "\x\in X. h x = h g" "\x\in X'. h x = h g'"
  and Xsub: "X \subseteq carrier G" and X'sub: "X' \subseteq carrier G"
  by (force simp add: kernel_def r_coset_def image_def)+
  hence "h ' (X <#> X') = {h g \otimes_H h g'}" using X X'
  by (auto dest!: FactGroup_nonempty
    simp add: set_mult_def image_eq_UN
    subsetD [OF Xsub] subsetD [OF X'sub])
  thus "contents (h ' (X <#> X')) = contents (h ' X) \otimes_H contents (h '
X')"
  by (simp add: all image_eq_UN FactGroup_nonempty X X')
qed

```

Lemma for the following injectivity result

```

lemma (in group_hom) FactGroup_subset:
  "[[g \in carrier G; g' \in carrier G; h g = h g']]
  \implies kernel G H h #> g \subseteq kernel G H h #> g'"
apply (clarsimp simp add: kernel_def r_coset_def image_def)
apply (rename_tac y)
apply (rule_tac x="y \otimes g \otimes inv g'" in exI)
apply (simp add: G.m_assoc)
done

```

```

lemma (in group_hom) FactGroup_inj_on:
  "inj_on (\X. contents (h'X)) (carrier (G Mod kernel G H h))"
proof (simp add: inj_on_def, clarify)
  fix X and X'
  assume X: "X \in carrier (G Mod kernel G H h)"
  and X': "X' \in 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 "contents (h ' X) = contents (h ' X')"
hence h: "h g = h g'"
  by (simp add: image_eq_UN all FactGroup_nonempty X X')
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. contents (h ' X)) ' carrier (G Mod kernel G H h) = carrier H"
proof
  show "(λX. contents (h ' X)) ' carrier (G Mod kernel G H h) ⊆ carrier H"
  by (auto simp add: FactGroup_contents_mem)
  show "carrier H ⊆ (λX. contents (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. contents (h ' X)) ' carrier (G Mod kernel G H h)"
  by (auto intro!: bexI simp add: FactGroup_def RCOSETS_def image_eq_UN)
qed
qed

```

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

```

theorem (in group_hom) FactGroup_iso:
  "h ' carrier G = carrier H
  ⇒ (λX. contents (h ' X)) ∈ (G Mod (kernel G H h)) ≅ H"
by (simp add: iso_def FactGroup_hom FactGroup_inj_on bij_betw_def
    FactGroup_onto)

```

end

```

theory Exponent
imports Main Primes Binomial
begin

```

5 Sylow's Theorem

5.1 The Combinatorial Argument Underlying the First Sylow Theorem

```

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

```

Prime Theorems

```

lemma prime_imp_one_less: "prime p ==> Suc 0 < p"
by (unfold prime_def, force)

```

```

lemma prime_iff:
  "(prime p) = (Suc 0 < p & (∀ a b. p dvd a*b --> (p dvd a) | (p dvd b)))"
apply (auto simp add: prime_imp_one_less)
apply (blast dest!: prime_dvd_mult)
apply (auto simp add: prime_def)
apply (erule dvdE)
apply (case_tac "k=0", simp)
apply (drule_tac x = m in spec)
apply (drule_tac x = k in spec)
apply (simp add: dvd_mult_cancel1 dvd_mult_cancel2)
done

```

```

lemma zero_less_prime_power: "prime p ==> 0 < p^a"
by (force simp add: prime_iff)

```

```

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

```

```

lemma prime_dvd_cases:
  "[| p*k dvd m*n; prime p |]
  ==> (∃ x. k dvd x*n & m = p*x) | (∃ y. k dvd m*y & n = p*y)"
apply (simp add: prime_iff)
apply (frule dvd_mult_left)
apply (subgoal_tac "p dvd m | p dvd n")
  prefer 2 apply blast
apply (erule disjE)
apply (rule disjI1)
apply (rule_tac [2] disjI2)

```

```

apply (auto elim!: dvdE)
done

```

```

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

```

apply simp
apply clarify
apply (erule prime_dvd_cases [THEN disjE], assumption, auto)

```

```

  apply (case_tac "a")
    apply simp
    apply clarify
    apply (drule spec, drule spec, erule (1) notE impE)
    apply (drule_tac x = nat in spec)
    apply (drule_tac x = b in spec)
    apply simp

```

```

  apply (case_tac "b")
    apply simp
    apply clarify
    apply (drule spec, drule spec, erule (1) notE impE)
    apply (drule_tac x = a in spec)
    apply (drule_tac x = nat in spec, simp)
  done

```

```

lemma div_combine:
  "[| prime p;  $\sim (p \text{ } ^ ( \text{Suc } r) \text{ dvd } n) ; p^{(a+r)} \text{ dvd } n * k$  |]
  ==>  $p^a \text{ dvd } k$ "
by (drule_tac a = "Suc r" and b = a in prime_power_dvd_cases, assumption,
auto)

```

```

lemma Suc_le_power: "Suc 0 < p ==> Suc n <= p^n"
apply (induct n)
apply (simp (no_asm_simp))
apply simp
apply (subgoal_tac "2 * n + 2 <= p * p^n", simp)
apply (subgoal_tac "2 * p^n <= p * p^n")
apply arith
apply (drule_tac k = 2 in mult_le_mono2, simp)

```


done

```
lemma power_dvd_bound: "[|p^n dvd a; Suc 0 < p; a > 0|] ==> n < a"
apply (drule dvd_imp_le)
apply (drule_tac [2] n = n in Suc_le_power, auto)
done
```

Exponent Theorems

```
lemma exponent_ge [rule_format]:
  "[|p^k dvd n; prime p; 0 < n|] ==> k <= exponent p n"
apply (simp add: exponent_def)
apply (erule Greatest_le)
apply (blast dest: prime_imp_one_less power_dvd_bound)
done
```

```
lemma power_exponent_dvd: "s>0 ==> (p ^ exponent p s) dvd s"
apply (simp add: exponent_def)
apply clarify
apply (rule_tac k = 0 in GreatestI)
prefer 2 apply (blast dest: prime_imp_one_less power_dvd_bound, simp)
done
```

```
lemma power_Suc_exponent_Not_dvd:
  "[|(p * p ^ exponent p s) dvd s; prime p |] ==> s=0"
apply (subgoal_tac "p ^ Suc (exponent p s) dvd s")
  prefer 2 apply simp
apply (rule ccontr)
apply (drule exponent_ge, auto)
done
```

```
lemma exponent_power_eq [simp]: "prime p ==> exponent p (p^a) = a"
apply (simp (no_asm_simp) add: exponent_def)
apply (rule Greatest_equality, simp)
apply (simp (no_asm_simp) add: prime_imp_one_less power_dvd_imp_le)
done
```

```
lemma exponent_equalityI:
  "!r::nat. (p^r dvd a) = (p^r dvd b) ==> exponent p a = exponent p b"
by (simp (no_asm_simp) add: exponent_def)
```

```
lemma exponent_eq_0 [simp]: "¬ prime p ==> exponent p s = 0"
by (simp (no_asm_simp) add: exponent_def)
```

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

```

apply (rule exponent_ge)
apply (auto simp add: power_add)
apply (blast intro: prime_imp_one_less power_exponent_dvd mult_dvd_mono)
done

```

```

lemma exponent_mult_add2: "[| a > 0; b > 0 |]
  ==> exponent p (a * b) <= (exponent p a) + (exponent p b)"
apply (case_tac "prime p")
apply (rule leI, clarify)
apply (cut_tac p = p and s = "a*b" in power_exponent_dvd, auto)
apply (subgoal_tac "p ^ (Suc (exponent p a + exponent p b)) dvd a * b")
apply (rule_tac [2] le_imp_power_dvd [THEN dvd_trans])
  prefer 3 apply assumption
  prefer 2 apply simp
apply (frule_tac a = "Suc (exponent p a) " and b = "Suc (exponent p b)
" in prime_power_dvd_cases)
  apply (assumption, force, simp)
apply (blast dest: power_Suc_exponent_Not_dvd)
done

```

```

lemma exponent_mult_add: "[| a > 0; b > 0 |]
  ==> exponent p (a * b) = (exponent p a) + (exponent p b)"
by (blast intro: exponent_mult_add1 exponent_mult_add2 order_antisym)

```

```

lemma not_divides_exponent_0: "~ (p dvd n) ==> exponent p n = 0"
apply (case_tac "exponent p n", simp)
apply (case_tac "n", simp)
apply (cut_tac s = n and p = p in power_exponent_dvd)
apply (auto dest: dvd_mult_left)
done

```

```

lemma exponent_1_eq_0 [simp]: "exponent p (Suc 0) = 0"
apply (case_tac "prime p")
apply (auto simp add: prime_iff not_divides_exponent_0)
done

```

Main Combinatorial Argument

```

lemma le_extend_mult: "[| c > 0; a <= b |] ==> a <= b * (c::nat)"
apply (rule_tac P = "%x. x <= b * c" in subst)
apply (rule mult_1_right)
apply (rule mult_le_mono, auto)
done

```

```

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

```

```

apply (rule notI)
apply (drule contrapos_nn [OF _ leI, THEN notnotD], assumption)
apply (drule less_imp_le [of a])
apply (drule le_imp_power_dvd)
apply (drule_tac b = "p ^ r" in dvd_trans, assumption)
apply (metis dvd_diffD1 dvd_triv_right le_extend_mult linorder_linear linorder_not_less
mult_commute nat_dvd_not_less neq0_conv)
done

```

```

lemma p_fac_forw: "[| (m::nat) > 0; k>0; k < p^a; (p^r) dvd (p^a)* m
- k |]
==> (p^r) dvd (p^a) - k"
apply (frule p_fac_forw_lemma [THEN le_imp_power_dvd, of _ k p], auto)
apply (subgoal_tac "p^r dvd p^a*m")
  prefer 2 apply (blast intro: dvd_mult2)
apply (drule dvd_diffD1)
  apply assumption
  prefer 2 apply (blast intro: nat_dvd_diff)
apply (drule gr0_implies_Suc, auto)
done

```

```

lemma r_le_a_forw:
  "[| (k::nat) > 0; k < p^a; p>0; (p^r) dvd (p^a) - k |] ==> r <= a"
by (rule_tac m = "Suc 0" in p_fac_forw_lemma, auto)

```

```

lemma p_fac_backw: "[| m>0; k>0; (p::nat)≠0; k < p^a; (p^r) dvd p^a
- k |]
==> (p^r) dvd (p^a)*m - k"
apply (frule_tac k1 = k and p1 = p in r_le_a_forw [THEN le_imp_power_dvd],
auto)
apply (subgoal_tac "p^r dvd p^a*m")
  prefer 2 apply (blast intro: dvd_mult2)
apply (drule dvd_diffD1)
  apply assumption
  prefer 2 apply (blast intro: nat_dvd_diff)
apply (drule less_imp_Suc_add, auto)
done

```

```

lemma exponent_p_a_m_k_equation: "[| m>0; k>0; (p::nat)≠0; k < p^a
|]
==> exponent p (p^a * m - k) = exponent p (p^a - k)"
apply (blast intro: exponent_equalityI p_fac_forw p_fac_backw)
done

```

Suc rules that we have to delete from the simpset

```

lemmas bad_Sucs = binomial_Suc_Suc mult_Suc mult_Suc_right

```

```

lemma p_not_div_choose_lemma [rule_format]:
  "[|  $\forall i. \text{Suc } i < K \rightarrow \text{exponent } p (\text{Suc } i) = \text{exponent } p (\text{Suc}(j+i))$  |]
     $\Rightarrow k < K \rightarrow \text{exponent } p ((j+k) \text{ choose } k) = 0$ "
apply (cases "prime p")
  prefer 2 apply simp
apply (induct k)
apply (simp (no_asm))

apply (subgoal_tac "(Suc (j+k) choose Suc k) > 0")
  prefer 2 apply (simp add: zero_less_binomial_iff, clarify)
apply (subgoal_tac "exponent p ((Suc (j+k) choose Suc k) * Suc k) =
  exponent p (Suc k)")

```

First, use the assumed equation. We simplify the LHS to $\text{exponent } p (\text{Suc } (j + k) \text{ choose Suc } k) + \text{exponent } p (\text{Suc } k)$ the common terms cancel, proving the conclusion.

```

  apply (simp del: bad_Sucs add: exponent_mult_add)

```

Establishing the equation requires first applying `Suc_times_binomial_eq ...`

```

  apply (simp del: bad_Sucs add: Suc_times_binomial_eq [symmetric])

```

...then `exponent_mult_add` and the quantified premise.

```

  apply (simp del: bad_Sucs add: zero_less_binomial_iff exponent_mult_add)
done

```

```

lemma p_not_div_choose:
  "[|  $k < K; k \leq n;$ 
     $\forall j. 0 < j \ \& \ j < K \rightarrow \text{exponent } p (n - k + (K - j)) = \text{exponent } p (K - j)$  |]
     $\Rightarrow \text{exponent } p (n \text{ choose } k) = 0$ "
  apply (cut_tac j = "n-k" and k = k and p = p in p_not_div_choose_lemma)
    prefer 3 apply simp
    prefer 2 apply assumption
  apply (drule_tac x = "K - Suc i" in spec)
  apply (simp add: Suc_diff_le)
done

```

```

lemma const_p_fac_right:
  " $m > 0 \Rightarrow \text{exponent } p ((p^a * m - \text{Suc } 0) \text{ choose } (p^a - \text{Suc } 0)) = 0$ "
  apply (case_tac "prime p")
    prefer 2 apply simp
  apply (frule_tac a = a in zero_less_prime_power)
  apply (rule_tac K = "p^a" in p_not_div_choose)
    apply simp
    apply simp
  apply (case_tac "m")
    apply (case_tac [2] "p^a")

```

```

    apply auto

  apply (subgoal_tac "0 < p")
    prefer 2 apply (force dest!: prime_imp_one_less)
  apply (subst exponent_p_a_m_k_equation, auto)
  done

lemma const_p_fac:
  "m > 0 ==> exponent p (((p^a) * m) choose p^a) = exponent p m"
  apply (case_tac "prime p")
    prefer 2 apply simp
  apply (subgoal_tac "0 < p^a * m & p^a <= p^a * m")
    prefer 2 apply (force simp add: prime_iff)

  A similar trick to the one used in p_not_div_choose_lemma: insert an equation;
  use exponent_mult_add on the LHS; on the RHS, first transform the binomial
  coefficient, then use exponent_mult_add.

  apply (subgoal_tac "exponent p (((p^a) * m) choose p^a) * p^a =
    a + exponent p m")
    apply (simp del: bad_Sucs add: zero_less_binomial_iff exponent_mult_add
      prime_iff)

  one subgoal left!

  apply (subst times_binomial_minus1_eq, simp, simp)
  apply (subst exponent_mult_add, simp)
  apply (simp (no_asm_simp) add: zero_less_binomial_iff)
  apply arith
  apply (simp del: bad_Sucs add: exponent_mult_add const_p_fac_right)
  done

end

```

theory Sylow imports Coset Exponent begin

See also [3].

The combinatorial argument is in theory Exponent

```

locale sylow = group +
  fixes p and a and m and calM and RelM
  assumes prime_p: "prime p"
    and order_G: "order(G) = (p^a) * m"
    and finite_G [iff]: "finite (carrier G)"
  defines "calM == {s. s ⊆ carrier(G) & card(s) = p^a}"
    and "RelM == {(N1, N2). N1 ∈ calM & N2 ∈ calM &
      (∃ g ∈ carrier(G). N1 = (N2 #> g) )}"

```

```

lemma (in sylow) RelM_refl_on: "refl_on calM RelM"
apply (auto simp add: refl_on_def RelM_def calM_def)
apply (blast intro!: coset_mult_one [symmetric])
done

lemma (in sylow) RelM_sym: "sym RelM"
proof (unfold sym_def RelM_def, clarify)
  fix y g
  assume "y ∈ calM"
  and g: "g ∈ carrier G"
  hence "y = y #> g #> (inv g)" by (simp add: coset_mult_assoc calM_def)
  thus "∃ g' ∈ carrier G. y = y #> g #> g'"
  by (blast intro: g inv_closed)
qed

lemma (in sylow) RelM_trans: "trans RelM"
by (auto simp add: trans_def RelM_def calM_def coset_mult_assoc)

lemma (in sylow) RelM_equiv: "equiv calM RelM"
apply (unfold equiv_def)
apply (blast intro: RelM_refl_on RelM_sym RelM_trans)
done

lemma (in sylow) M_subset_calM_prep: "M' ∈ calM // RelM ==> M' ⊆ calM"
apply (unfold RelM_def)
apply (blast elim!: quotientE)
done

```

5.2 Main Part of the Proof

```

locale sylow_central = sylow +
  fixes H and M1 and M
  assumes M_in_quot: "M ∈ calM // RelM"
  and not_dvd_M: "~(p ^ Suc(exponent p m) dvd card(M))"
  and M1_in_M: "M1 ∈ M"
  defines "H == {g. g ∈ carrier G & M1 #> g = M1}"

lemma (in sylow_central) M_subset_calM: "M ⊆ calM"
by (rule M_in_quot [THEN M_subset_calM_prep])

lemma (in sylow_central) card_M1: "card(M1) = p^a"
apply (cut_tac M_subset_calM M1_in_M)
apply (simp add: calM_def, blast)
done

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

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

```

```

apply (subgoal_tac "0 < card M1")
  apply (blast dest: card_nonempty)
apply (cut_tac prime_p [THEN prime_imp_one_less])
apply (simp (no_asm_simp) add: card_M1)
done

lemma (in sylow_central) M1_subset_G [simp]: "M1  $\subseteq$  carrier G"
  apply (rule subsetD [THEN PowD])
  apply (rule_tac [2] M1_in_M)
  apply (rule M_subset_calM [THEN subset_trans])
  apply (auto simp add: calM_def)
done

lemma (in sylow_central) M1_inj_H: " $\exists f \in H \rightarrow M1. \text{inj\_on } f \ H$ "
  proof -
    from exists_x_in_M1 obtain m1 where m1M: "m1  $\in$  M1"..
    have m1G: "m1  $\in$  carrier G" by (simp add: m1M M1_subset_G [THEN subsetD])
    show ?thesis
      proof
        show "inj_on ( $\lambda z \in H. m1 \otimes z$ ) H"
          by (simp add: inj_on_def l_cancel [of m1 x y, THEN iffD1] H_def
m1G)
        show "restrict (op  $\otimes$  m1) H  $\in$  H  $\rightarrow$  M1"
          proof (rule restrictI)
            fix z assume zH: "z  $\in$  H"
            show "m1  $\otimes$  z  $\in$  M1"
              proof -
                from zH
                have zG: "z  $\in$  carrier G" and M1zeq: "M1 #> z = M1"
                  by (auto simp add: H_def)
                show ?thesis
                  by (rule subst [OF M1zeq], simp add: m1M zG rcosI)
              qed
            qed
          qed
        qed
      qed

```

5.3 Discharging the Assumptions of sylow_central

```

lemma (in sylow) EmptyNotInEquivSet: "{ }  $\notin$  calM // RelM"
  by (blast elim!: quotientE dest: RelM_equiv [THEN equiv_class_self])

lemma (in sylow) existsM1inM: "M  $\in$  calM // RelM  $\impl \exists M1. M1 \in M$ "
  apply (subgoal_tac "M  $\neq$  { }")
  apply blast
  apply (cut_tac EmptyNotInEquivSet, blast)
  done

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

```

```

apply (unfold order_def)
apply (blast intro: one_closed zero_less_card_empty)
done

lemma (in sylow) zero_less_m: "m > 0"
apply (cut_tac zero_less_o_G)
apply (simp add: order_G)
done

lemma (in sylow) card_calM: "card(calM) = (p^a) * m choose p^a"
by (simp add: calM_def n_subsets order_G [symmetric] order_def)

lemma (in sylow) zero_less_card_calM: "card calM > 0"
by (simp add: card_calM zero_less_binomial le_extend_mult zero_less_m)

lemma (in sylow) max_p_div_calM:
  "~ (p ^ Suc(exponent p m) dvd card(calM))"
apply (subgoal_tac "exponent p m = exponent p (card calM) ")
  apply (cut_tac zero_less_card_calM prime_p)
  apply (force dest: power_Suc_exponent_Not_dvd)
apply (simp add: card_calM zero_less_m [THEN const_p_fac])
done

lemma (in sylow) finite_calM: "finite calM"
apply (unfold calM_def)
apply (rule_tac B = "Pow (carrier G) " in finite_subset)
apply auto
done

lemma (in sylow) lemma_A1:
  "∃M ∈ calM // RelM. ~ (p ^ Suc(exponent p m) dvd card(M))"
apply (rule max_p_div_calM [THEN contrapos_np])
apply (simp add: finite_calM equiv_imp_dvd_card [OF _ RelM_equiv])
done

```

5.3.1 Introduction and Destruct Rules for H

```

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

lemma (in sylow_central) H_into_carrier_G: "x ∈ H ==> x ∈ carrier G"
by (simp add: H_def)

lemma (in sylow_central) in_H_imp_eq: "g : H ==> M1 #> g = M1"
by (simp add: H_def)

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

```



```

apply (simp add: coset_mult_assoc [symmetric] m_closed)
done

```

```

lemma (in sylow_central) H_not_empty: "H ≠ {}"
apply (simp add: H_def)
apply (rule exI [of _ 1], simp)
done

```

```

lemma (in sylow_central) H_is_subgroup: "subgroup H G"
apply (rule subgroupI)
apply (rule subsetI)
apply (erule H_into_carrier_G)
apply (rule H_not_empty)
apply (simp add: H_def, clarify)
apply (erule_tac P = "%z. ?lhs(z) = M1" in subst)
apply (simp add: coset_mult_assoc)
apply (blast intro: H_m_closed)
done

```

```

lemma (in sylow_central) 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 (in sylow_central) finite_M1: "finite M1"
by (rule finite_subset [OF M1_subset_G finite_G])

```

```

lemma (in sylow_central) finite_rcosetGM1g: "g ∈ carrier G ==> finite (M1
#> g)"
apply (rule finite_subset)
apply (rule subsetI)
apply (erule rcosetGM1g_subset_G, assumption)
apply (rule finite_G)
done

```

```

lemma (in sylow_central) M1_cardeq_rcosetGM1g:
  "g ∈ carrier G ==> card(M1 #> g) = card(M1)"
by (simp (no_asm_simp) add: M1_subset_G card_cosets_equal rcosetsI)

```

```

lemma (in sylow_central) M1_RelM_rcosetGM1g:
  "g ∈ carrier G ==> (M1, M1 #> g) ∈ RelM"
apply (simp (no_asm) add: RelM_def calM_def card_M1 M1_subset_G)
apply (rule conjI)
  apply (blast intro: rcosetGM1g_subset_G)
  apply (simp (no_asm_simp) add: card_M1 M1_cardeq_rcosetGM1g)
  apply (rule bexI [of _ "inv g"])
  apply (simp_all add: coset_mult_assoc M1_subset_G)
done

```

5.4 Equal Cardinalities of M and the Set of Cosets

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

```

lemma ElemClassEquiv:
  "[| equiv A r; C ∈ A // r |] ==> ∀x ∈ C. ∀y ∈ C. (x,y)∈r"
by (unfold equiv_def quotient_def sym_def trans_def, blast)

lemma (in sylow_central) M_elem_map:
  "M2 ∈ M ==> ∃g. g ∈ carrier G & M1 #> g = M2"
apply (cut_tac M1_in_M M_in_quot [THEN RelM_equiv [THEN ElemClassEquiv]])
apply (simp add: RelM_def)
apply (blast dest!: bspec)
done

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

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

lemma (in sylow_central) M_funcset_rcosets_H:
  "(%x:M. H #> (SOME g. g ∈ carrier G & M1 #> g = x)) ∈ M → rcosets
H"
apply (rule rcosetsI [THEN restrictI])
apply (rule H_is_subgroup [THEN subgroup.subset])
apply (erule M_elem_map_carrier)
done

lemma (in sylow_central) inj_M_GmodH: "∃f ∈ M→rcosets H. inj_on f M"
apply (rule bexI)
apply (rule_tac [2] M_funcset_rcosets_H)
apply (rule inj_onI, simp)
apply (rule trans [OF _ M_elem_map_eq])
prefer 2 apply assumption
apply (rule M_elem_map_eq [symmetric, THEN trans], assumption)
apply (rule coset_mult_inv1)
apply (erule_tac [2] M_elem_map_carrier)+
apply (rule_tac [2] M1_subset_G)
apply (rule coset_join1 [THEN in_H_imp_eq])
apply (rule_tac [3] H_is_subgroup)
prefer 2 apply (blast intro: m_closed M_elem_map_carrier inv_closed)
apply (simp add: coset_mult_inv2 H_def M_elem_map_carrier subset_eq)
done

```

5.4.1 The Opposite Injection

```

lemma (in sylow_central) H_elem_map:
  "H1 ∈ rcosets H ==> ∃g. g ∈ carrier G & H #> g = H1"
by (auto simp add: RCOSETS_def)

```

```

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

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

lemma EquivElemClass:
  "[|equiv A r; M ∈ A//r; M1∈M; (M1,M2) ∈ r |] ==> M2 ∈ M"
by (unfold equiv_def quotient_def sym_def trans_def, blast)

lemma (in sylow_central) rcosets_H_funcset_M:
  "(λC ∈ rcosets H. M1 #> (@g. g ∈ carrier G ∧ H #> g = C)) ∈ rcosets
H → M"
apply (simp add: RCOSETS_def)
apply (fast intro: someI2
  intro!: restrictI M1_in_M
    EquivElemClass [OF RelM_equiv M_in_quot _ M1_RelM_rcosetGM1g])
done

close to a duplicate of inj_M_GmodH

lemma (in sylow_central) inj_GmodH_M:
  "∃g ∈ rcosets H→M. inj_on g (rcosets H)"
apply (rule bexI)
apply (rule_tac [2] rcosets_H_funcset_M)
apply (rule inj_onI)
apply (simp)
apply (rule trans [OF _ H_elem_map_eq])
prefer 2 apply assumption
apply (rule H_elem_map_eq [symmetric, THEN trans], assumption)
apply (rule coset_mult_inv1)
apply (erule_tac [2] H_elem_map_carrier)+
apply (rule_tac [2] H_is_subgroup [THEN subgroup.subset])
apply (rule coset_join2)
apply (blast intro: m_closed inv_closed H_elem_map_carrier)
apply (rule H_is_subgroup)
apply (simp add: H_I coset_mult_inv2 M1_subset_G H_elem_map_carrier)
done

lemma (in sylow_central) calM_subset_PowG: "calM ⊆ Pow(carrier G)"
by (auto simp add: calM_def)

lemma (in sylow_central) finite_M: "finite M"
apply (rule finite_subset)
apply (rule M_subset_calM [THEN subset_trans])
apply (rule calM_subset_PowG, blast)

```

done

```
lemma (in sylow_central) cardMeqIndexH: "card(M) = card(rcosets H)"
apply (insert inj_M_GmodH inj_GmodH_M)
apply (blast intro: card_bij finite_M H_is_subgroup
      rcosets_subset_PowG [THEN finite_subset]
      finite_Pow_iff [THEN iffD2])
```

done

```
lemma (in sylow_central) index_lem: "card(M) * card(H) = order(G)"
by (simp add: cardMeqIndexH lagrange H_is_subgroup)
```

```
lemma (in sylow_central) lemma_leq1: "p^a ≤ card(H)"
apply (rule dvd_imp_le)
  apply (rule div_combine [OF prime_p not_dvd_M])
  prefer 2 apply (blast intro: subgroup.finite_imp_card_positive H_is_subgroup)
apply (simp add: index_lem order_G power_add mult_dvd_mono power_exponent_dvd
      zero_less_m)
```

done

```
lemma (in sylow_central) lemma_leq2: "card(H) ≤ p^a"
apply (subst card_M1 [symmetric])
apply (cut_tac M1_inj_H)
apply (blast intro!: M1_subset_G intro:
      card_inj_H_into_carrier_G finite_subset [OF _ finite_G])
```

done

```
lemma (in sylow_central) card_H_eq: "card(H) = p^a"
by (blast intro: le_anti_sym lemma_leq1 lemma_leq2)
```

```
lemma (in sylow) sylow_thm: "∃H. subgroup H G & card(H) = p^a"
apply (cut_tac lemma_A1, clarify)
apply (frule existsM1inM, clarify)
apply (subgoal_tac "sylow_central G p a m M1 M")
  apply (blast dest: sylow_central.H_is_subgroup sylow_central.card_H_eq)
apply (simp add: sylow_central_def sylow_central_axioms_def prems)
done
```

Needed because the locale's automatic definition refers to `semigroup G` and `group_axioms G` rather than simply to `group G`.

```
lemma sylow_eq: "sylow G p a m = (group G & sylow_axioms G p a m)"
by (simp add: sylow_def group_def)
```

5.5 Sylow's Theorem

```
theorem sylow_thm:
  "[| prime p; group(G); order(G) = (p^a) * m; finite (carrier G)|]
  ==> ∃H. subgroup H G & card(H) = p^a"
apply (rule sylow.sylow_thm [of G p a m])
```

```

apply (simp add: sylow_eq sylow_axioms_def)
done

end

```

```

theory Bij imports Group begin

```

6 Bijections of a Set, Permutation and Automorphism Groups

```

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

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

declare Id_compose [simp] compose_Id [simp]

lemma Bij_imp_extensional: "f  $\in$  Bij S  $\implies$  f  $\in$  extensional S"
  by (simp add: Bij_def)

lemma Bij_imp_funcset: "f  $\in$  Bij S  $\implies$  f  $\in$  S  $\rightarrow$  S"
  by (auto simp add: Bij_def bij_betw_imp_funcset)



### 6.1 Bijections Form a Group



lemma restrict_Inv_Bij: "f  $\in$  Bij S  $\implies$  ( $\lambda x \in S. (\text{Inv } S \ f) \ x$ )  $\in$  Bij S"
  by (simp add: Bij_def bij_betw_Inv)

lemma id_Bij: " $(\lambda x \in S. x) \in \text{Bij } S$ "
  by (auto simp add: Bij_def bij_betw_def inj_on_def)

lemma compose_Bij: " $\llbracket x \in \text{Bij } S; y \in \text{Bij } S \rrbracket \implies \text{compose } S \ x \ y \in \text{Bij } S$ "
  by (auto simp add: Bij_def bij_betw_compose)

lemma Bij_compose_restrict_eq:
  "f  $\in$  Bij S  $\implies$  compose S (restrict (Inv S f) S) f = ( $\lambda x \in S. x$ )"
  by (simp add: Bij_def compose_Inv_id)

```

```

theorem group_BijGroup: "group (BijGroup S)"
apply (simp add: BijGroup_def)
apply (rule groupI)
  apply (simp add: compose_Bij)
  apply (simp add: id_Bij)
  apply (simp add: compose_Bij)
  apply (blast intro: compose_assoc [symmetric] Bij_imp_funcset)
  apply (simp add: id_Bij Bij_imp_funcset Bij_imp_extensional, simp)
apply (blast intro: Bij_compose_restrict_eq restrict_Inv_Bij)
done

```

6.2 Automorphisms Form a Group

```

lemma Bij_Inv_mem: "[[ f ∈ Bij S; x ∈ S ] ⇒ Inv S f x ∈ S"
by (simp add: Bij_def bij_betw_def Inv_mem)

```

```

lemma Bij_Inv_lemma:
  assumes eq: "⋀x y. [x ∈ S; y ∈ S] ⇒ h(g x y) = g (h x) (h y)"
  shows "[[h ∈ Bij S; g ∈ S → S → S; x ∈ S; y ∈ S]
    ⇒ Inv S h (g x y) = g (Inv S h x) (Inv S h y)]"
apply (simp add: Bij_def bij_betw_def)
apply (subgoal_tac "∃x'∈S. ∃y'∈S. x = h x' & y = h y'", clarify)
  apply (simp add: eq [symmetric] Inv_f_f funcset_mem [THEN funcset_mem],
    blast)
done

```

constdefs

```

auto :: "('a, 'b) monoid_scheme ⇒ ('a ⇒ 'a) set"
"auto G ≡ hom G G ∩ Bij (carrier G)"

```

```

AutoGroup :: "('a, 'c) monoid_scheme ⇒ ('a ⇒ 'a) monoid"
"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_hom:
  "[[h ∈ hom G G; h ∈ Bij (carrier G)]
    ⇒ restrict (Inv (carrier G) h) (carrier G) ∈ hom G G"
by (simp add: hom_def Bij_Inv_mem restrictI mult_funcset
  group.axioms Bij_Inv_lemma)

```

```

lemma inv_BijGroup:
  "f ∈ Bij S ⇒ m_inv (BijGroup S) f = (λx ∈ S. (Inv S f) x)"

```

```

apply (rule group.inv_equality)
apply (rule group.BijGroup)
apply (simp_all add: BijGroup_def restrict_Inv_Bij Bij_compose_restrict_eq)
done

lemma (in group) subgroup_auto:
  "subgroup (auto G) (BijGroup (carrier G))"
proof (rule subgroup.intro)
  show "auto G  $\subseteq$  carrier (BijGroup (carrier G))"
    by (force simp add: auto_def BijGroup_def)
next
  fix x y
  assume "x  $\in$  auto G" "y  $\in$  auto G"
  thus "x  $\otimes_{\text{BijGroup (carrier G)}}$  y  $\in$  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)  $\in$  auto G" by (simp add: BijGroup_def id_in_auto)
next
  fix x
  assume "x  $\in$  auto G"
  thus "invBijGroup (carrier G) x  $\in$  auto G"
    by (simp del: restrict_apply
            add: inv_BijGroup auto_def restrict_Inv_Bij restrict_Inv_hom)
qed

theorem (in group) AutoGroup: "group (AutoGroup G)"
by (simp add: AutoGroup_def subgroup.subgroup_is_group subgroup_auto
            group_BijGroup)

end

```

```

theory Divisibility
imports Permutation Coset Group
begin

```

7 Factorial Monoids

7.1 Monoids with Cancellation Law

```

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

```

$G \rceil \implies a = b$ "

```
lemma (in monoid) monoid_cancelI:
  assumes l_cancel:
    " $\bigwedge a\ b\ c. \llbracket c \otimes a = c \otimes b; a \in \text{carrier } G; b \in \text{carrier } G; c \in \text{carrier } G \rrbracket \implies a = b$ "
  and r_cancel:
    " $\bigwedge a\ b\ c. \llbracket a \otimes c = b \otimes c; a \in \text{carrier } G; b \in \text{carrier } G; c \in \text{carrier } G \rrbracket \implies a = b$ "
  shows "monoid_cancel G"
  proof qed fact+
```

```
lemma (in monoid_cancel) is_monoid_cancel:
  "monoid_cancel G"
..
```

```
sublocale group  $\subseteq$  monoid_cancel
proof qed simp+
```

```
locale comm_monoid_cancel = monoid_cancel + comm_monoid
```

```
lemma comm_monoid_cancelI:
  fixes G (structure)
  assumes "comm_monoid G"
  assumes cancel:
    " $\bigwedge a\ b\ c. \llbracket a \otimes c = b \otimes c; a \in \text{carrier } G; b \in \text{carrier } G; c \in \text{carrier } G \rrbracket \implies a = b$ "
  shows "comm_monoid_cancel G"
proof -
  interpret comm_monoid G by fact
  show "comm_monoid_cancel G"
  apply unfold_locales
  apply (subgoal_tac " $a \otimes c = b \otimes c$ ")
  apply (iprover intro: cancel)
  apply (simp add: m_comm)
  apply (iprover intro: cancel)
  done
qed
```

```
lemma (in comm_monoid_cancel) is_comm_monoid_cancel:
  "comm_monoid_cancel G"
  by intro_locales
```

```
sublocale comm_group  $\subseteq$  comm_monoid_cancel
..
```


7.2 Products of Units in Monoids

```

lemma (in monoid) Units_m_closed[simp, intro]:
  assumes h1unit: "h1 ∈ Units G" and h2unit: "h2 ∈ Units G"
  shows "h1 ⊗ h2 ∈ Units G"
unfolding Units_def
using assms
apply safe
apply fast
apply (intro bexI[of _ "inv h2 ⊗ inv h1"], safe)
  apply (simp add: m_assoc Units_closed)
  apply (simp add: m_assoc[symmetric] Units_closed Units_l_inv)
  apply (simp add: m_assoc Units_closed)
  apply (simp add: m_assoc[symmetric] Units_closed Units_r_inv)
apply fast
done

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 add: Units_l_inv)
  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 add: Units_l_inv)
  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 (simp add: Units_def, fast)
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  $\otimes$  inv (a  $\otimes$  b)  $\in$  carrier G" by simp

have "a  $\otimes$  (b  $\otimes$  inv (a  $\otimes$  b)) = (a  $\otimes$  b)  $\otimes$  inv (a  $\otimes$  b)"
  by (simp add: m_assoc del: Units_r_inv)
also have "... = 1" by simp
finally have li: "a  $\otimes$  (b  $\otimes$  inv (a  $\otimes$  b)) = 1" .

have "1 = b  $\otimes$  inv b" by (simp add: Units_r_inv[symmetric])
also have "... = b  $\otimes$  1  $\otimes$  inv b" by simp
also have "... = b  $\otimes$  (inv (a  $\otimes$  b)  $\otimes$  (a  $\otimes$  b))  $\otimes$  inv b"
  by (simp add: Units_l_inv[OF abunit, symmetric] del: Units_l_inv)
also have "... = (b  $\otimes$  inv (a  $\otimes$  b)  $\otimes$  a)  $\otimes$  (b  $\otimes$  inv b)"
  by (simp add: m_assoc del: Units_l_inv)
also have "... = b  $\otimes$  inv (a  $\otimes$  b)  $\otimes$  a" by simp
finally have ri: "(b  $\otimes$  inv (a  $\otimes$  b))  $\otimes$  a = 1" by simp

from c li ri
  show "a  $\in$  Units G" by (simp add: Units_def, fast)
qed

lemma (in comm_monoid) unit_factor:
  assumes abunit: "a  $\otimes$  b  $\in$  Units G"
  and [simp]: "a  $\in$  carrier G" "b  $\in$  carrier G"
  shows "a  $\in$  Units G"
using abunit[simplified Units_def]
proof clarsimp
  fix i
  assume [simp]: "i  $\in$  carrier G"
  and li: "i  $\otimes$  (a  $\otimes$  b) = 1"
  and ri: "a  $\otimes$  b  $\otimes$  i = 1"

  have carr': "b  $\otimes$  i  $\in$  carrier G" by simp

  have "(b  $\otimes$  i)  $\otimes$  a = (i  $\otimes$  b)  $\otimes$  a" by (simp add: m_comm)
  also have "... = i  $\otimes$  (b  $\otimes$  a)" by (simp add: m_assoc)
  also have "... = i  $\otimes$  (a  $\otimes$  b)" by (simp add: m_comm)
  also note li
  finally have li': "(b  $\otimes$  i)  $\otimes$  a = 1" .

  have "a  $\otimes$  (b  $\otimes$  i) = a  $\otimes$  b  $\otimes$  i" by (simp add: m_assoc)
  also note ri
  finally have ri': "a  $\otimes$  (b  $\otimes$  i) = 1" .

  from carr' li' ri'
    show "a  $\in$  Units G" by (simp add: Units_def, fast)
qed

```

7.3 Divisibility and Association

7.3.1 Function definitions

```

constdefs (structure G)
  factor :: "[_, 'a, 'a]  $\Rightarrow$  bool" (infix "divides1" 65)
  "a divides b ==  $\exists c \in \text{carrier } G. b = a \otimes c$ "

constdefs (structure G)
  associated :: "[_, 'a, 'a]  $\Rightarrow$  bool" (infix " $\sim$ " 55)
  "a  $\sim$  b == a divides b  $\wedge$  b divides a"

abbreviation
  "division_rel G == ( $\lambda$ carrier = carrier G, eq = op  $\sim_G$ , le = op dividesG)"

constdefs (structure G)
  properfactor :: "[_, 'a, 'a]  $\Rightarrow$  bool"
  "properfactor G a b == a divides b  $\wedge$   $\neg$ (b divides a)"

constdefs (structure G)
  irreducible :: "[_, 'a]  $\Rightarrow$  bool"
  "irreducible G a == a  $\notin$  Units G  $\wedge$  ( $\forall b \in \text{carrier } G. \text{properfactor } G b a \longrightarrow b \in \text{Units } G$ )"

constdefs (structure G)
  prime :: "[_, 'a]  $\Rightarrow$  bool"
  "prime G p == p  $\notin$  Units G  $\wedge$ 
    ( $\forall a \in \text{carrier } G. \forall b \in \text{carrier } G. p \text{ divides } (a \otimes b) \longrightarrow p \text{ divides } a \vee p \text{ divides } b$ )"

```

7.3.2 Divisibility

```

lemma dividesI:
  fixes G (structure)
  assumes carr: "c  $\in$  carrier G"
  and p: "b = a  $\otimes$  c"
  shows "a divides b"
unfolding factor_def
using assms by fast

```

```

lemma dividesI' [intro]:
  fixes G (structure)
  assumes p: "b = a  $\otimes$  c"
  and carr: "c  $\in$  carrier G"
  shows "a divides b"
using assms
by (fast intro: dividesI)

```

```

lemma dividesD:
  fixes G (structure)

```

```

    assumes "a divides b"
    shows " $\exists c \in \text{carrier } G. b = a \otimes c$ "
using assms
unfolding factor_def
by fast

lemma dividesE [elim]:
  fixes G (structure)
  assumes d: "a divides b"
    and elim: " $\bigwedge c. [b = a \otimes c; c \in \text{carrier } G] \implies P$ "
  shows "P"
proof -
  from dividesD[OF d]
    obtain c
    where "c  $\in$  carrier G"
    and "b = a  $\otimes$  c"
    by auto
  thus "P" by (elim elim)
qed

lemma (in monoid) divides_refl[simp, intro!]:
  assumes carr: "a  $\in$  carrier G"
  shows "a divides a"
apply (intro dividesI[of "1"])
apply (simp, simp add: carr)
done

lemma (in monoid) divides_trans [trans]:
  assumes dvds: "a divides b" "b divides c"
    and acarr: "a  $\in$  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 ab: "a divides b"
    and carr: "a  $\in$  carrier G" "b  $\in$  carrier G" "c  $\in$  carrier G"
  shows "(c  $\otimes$  a) divides (c  $\otimes$  b)"
using ab
apply (elim dividesE, simp add: m_assoc[symmetric] carr)
apply (fast intro: dividesI)
done

lemma (in monoid_cancel) divides_mult_l [simp]:
  assumes carr: "a  $\in$  carrier G" "b  $\in$  carrier G" "c  $\in$  carrier G"
  shows "(c  $\otimes$  a) divides (c  $\otimes$  b) = a divides b"
apply safe
apply (elim dividesE, intro dividesI, assumption)
apply (rule l_cancel[of c])

```

```

    apply (simp add: m_assoc carr)+
  apply (fast intro: divides_mult_lI carr)
done

```

```

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
apply (simp add: m_comm[of a c] m_comm[of b c])
apply (rule divides_mult_lI, assumption+)
done

```

```

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" "b ∈ 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 carr[intro]: "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier
G"
    and ab: "a divides b"
  shows "a divides (c ⊗ b)"
using ab carr
apply (simp add: m_comm[of c b])
apply (fast intro: divides_prod_r)
done

```

```

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  $\otimes$  (inv u  $\otimes$  a)" ..
qed

lemma (in comm_monoid) divides_unit:
  assumes udvd: "a divides u"
    and carr: "a  $\in$  carrier G" "u  $\in$  Units G"
  shows "a  $\in$  Units G"
using udvd carr
by (blast intro: unit_factor)

```

```

lemma (in comm_monoid) Unit_eq_dividesone:
  assumes ucarr: "u  $\in$  carrier G"
  shows "u  $\in$  Units G = u divides 1"
using ucarr
by (fast dest: divides_unit intro: unit_divides)

```

7.3.3 Association

```

lemma associatedI:
  fixes G (structure)
  assumes "a divides b" "b divides a"
  shows "a  $\sim$  b"
using assms
by (simp add: associated_def)

lemma (in monoid) associatedI2:
  assumes uunit[simp]: "u  $\in$  Units G"
    and a: "a = b  $\otimes$  u"
    and bcarr[simp]: "b  $\in$  carrier G"
  shows "a  $\sim$  b"
using uunit bcarr
unfolding a
apply (intro associatedI)
  apply (rule dividesI[of "inv u"], simp)
  apply (simp add: m_assoc Units_closed Units_r_inv)
apply fast
done

```

```

lemma (in monoid) associatedI2':
  assumes a: "a = b  $\otimes$  u"
    and uunit: "u  $\in$  Units G"
    and bcarr: "b  $\in$  carrier G"
  shows "a  $\sim$  b"
using assms by (intro associatedI2)

```

```

lemma associatedD:
  fixes G (structure)
  assumes "a  $\sim$  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"
  hence "∃u∈carrier G. a = b ⊗ u" by (rule dividesD)
  from this obtain u
    where ucarr: "u ∈ carrier G" and a: "a = b ⊗ u"
    by auto

  assume "a divides b"
  hence "∃u'∈carrier G. b = a ⊗ u'" by (rule dividesD)
  from this obtain u'
    where u'carr: "u' ∈ carrier G" and b: "b = a ⊗ u'"
    by auto
  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
  hence "u ∈ Units G" by (simp add: Units_def ucarr)

```

```

    from ucarr this 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 add: associated_def)+
  thus "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)
  from this obtain u
    where "u ∈ Units G" "a = b ⊗ u"
    by auto
  thus "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"
    and "a ∈ carrier G" "b ∈ carrier G"
  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" "b ∈ 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 (rule associated_sym, assumption+)
  apply (iprover intro: associated_trans)
  done

```

7.3.4 Division and associativity

```

lemma divides_antisym:
  fixes G (structure)
  assumes "a divides b" "b divides a"
    and "a ∈ carrier G" "b ∈ carrier G"
  shows "a ~ b"
using assms
by (fast intro: associatedI)

lemma (in monoid) divides_cong_l [trans]:
  assumes xx': "x ~ x'"
    and xvdvy: "x' divides y"
    and carr [simp]: "x ∈ carrier G" "x' ∈ carrier G" "y ∈ carrier
G"
  shows "x divides y"
proof -
  from xx'
    have "x divides x'" by (simp add: associatedD)
  also note xvdvy
  finally
    show "x divides y" by simp
qed

lemma (in monoid) divides_cong_r [trans]:
  assumes xvdvy: "x divides y"
    and yy': "y ~ y'"
    and carr[simp]: "x ∈ carrier G" "y ∈ carrier G" "y' ∈ carrier G"
  shows "x divides y'"
proof -
  note xvdvy
  also from yy'
    have "y divides y'" by (simp add: associatedD)
  finally
    show "x divides y'" by simp
qed

lemma (in monoid) division_weak_partial_order [simp, intro!]:
  "weak_partial_order (division_rel G)"
  apply unfold_locales

```

```

apply simp_all
apply (simp add: associated_sym)
apply (blast intro: associated_trans)
apply (simp add: divides_antisym)
apply (blast intro: divides_trans)
apply (blast intro: divides_cong_l divides_cong_r associated_sym)
done

```

7.3.5 Multiplication and associativity

```

lemma (in monoid_cancel) mult_cong_r:
  assumes "b ~ b'"
  and carr: "a ∈ carrier G" "b ∈ carrier G" "b' ∈ carrier G"
  shows "a ⊗ b ~ a ⊗ b'"
using assms
apply (elim associatedE2, intro associatedI2)
apply (auto intro: m_assoc[symmetric])
done

```

```

lemma (in comm_monoid_cancel) mult_cong_l:
  assumes "a ~ a'"
  and carr: "a ∈ carrier G" "a' ∈ carrier G" "b ∈ carrier G"
  shows "a ⊗ b ~ a' ⊗ b"
using assms
apply (elim associatedE2, intro associatedI2)
  apply assumption
  apply (simp add: m_assoc Units_closed)
  apply (simp add: m_comm Units_closed)
  apply simp+
done

```

```

lemma (in monoid_cancel) assoc_l_cancel:
  assumes carr: "a ∈ carrier G" "b ∈ carrier G" "b' ∈ carrier G"
  and "a ⊗ b ~ a ⊗ b'"
  shows "b ~ b'"
using assms
apply (elim associatedE2, intro associatedI2)
  apply assumption
  apply (rule l_cancel[of a])
  apply (simp add: m_assoc Units_closed)
  apply fast+
done

```

```

lemma (in comm_monoid_cancel) assoc_r_cancel:
  assumes "a ⊗ b ~ a' ⊗ b"
  and carr: "a ∈ carrier G" "a' ∈ carrier G" "b ∈ carrier G"
  shows "a ~ a'"
using assms
apply (elim associatedE2, intro associatedI2)

```

```

    apply assumption
  apply (rule r_cancel[of a b])
    apply (simp add: m_assoc Units_closed)
    apply (simp add: m_comm Units_closed)
  apply fast+
done

```

7.3.6 Units

```

lemma (in monoid_cancel) assoc_unit_l [trans]:
  assumes asc: "a ~ b" and bunit: "b ∈ Units G"
  and carr: "a ∈ carrier G"
  shows "a ∈ Units G"
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}"
apply (simp add: set_eq_def elem_def, rule, simp_all)
proof clarsimp
  fix a
  assume aunit: "a ∈ Units G"
  show "a ~ 1"
  apply (rule associatedI)
  apply (fast intro: dividesI[of "inv a"] aunit Units_r_inv[symmetric])
  apply (fast intro: dividesI[of "a"] l_one[symmetric] Units_closed[OF
aunit])
  done
next

```

```

    have "1 ∈ Units G" by simp
    moreover have "1 ~ 1" by simp
    ultimately show "∃ a ∈ Units G. 1 ~ a" by fast
qed

lemma (in comm_monoid) Units_Lower:
  "Units G = Lower (division_rel G) (carrier G)"
apply (simp add: Units_def Lower_def)
apply (rule, rule)
  apply clarsimp
  apply (rule unit_divides)
  apply (unfold Units_def, fast)
  apply assumption
apply clarsimp
proof -
  fix x
  assume xcarr: "x ∈ carrier G"
  assume r[rule_format]: "∀ y. y ∈ carrier G ⟶ x divides y"
  have "1 ∈ carrier G" by simp
  hence "x divides 1" by (rule r)
  hence "∃ x' ∈ carrier G. 1 = x ⊗ x'" by (rule dividesE, fast)
  from this obtain x'
    where x'carr: "x' ∈ carrier G"
    and xx': "1 = x ⊗ x'"
    by auto

  note xx'
  also with xcarr x'carr
    have "... = x' ⊗ x" by (simp add: m_comm)
  finally
    have "1 = x' ⊗ x" .

  from x'carr xx'[symmetric] this[symmetric]
    show "∃ y ∈ carrier G. y ⊗ x = 1 ∧ x ⊗ y = 1" by fast
qed

```

7.3.7 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: " $\neg(a \sim b)$ "
    shows "properfactor G a b"
  apply (rule properfactorI, rule advdb)
  proof (rule ccontr, simp)
    assume "b divides a"
    with advdb have " $a \sim b$ " by (rule associatedI)
    with neq show "False" by fast
  qed

lemma (in comm_monoid_cancel) properfactorI3:
  assumes p: " $p = a \otimes b$ "
  and nunit: " $b \notin \text{Units } G$ "
  and carr: " $a \in \text{carrier } G$ " " $b \in \text{carrier } G$ " " $p \in \text{carrier } G$ "
  shows "properfactor G a p"
  unfolding p
  using carr
  apply (intro properfactorI, fast)
  proof (clarsimp, elim dividesE)
    fix c
    assume ccarr: " $c \in \text{carrier } G$ "
    note [simp] = carr ccarr

    have " $a \otimes 1 = a$ " by simp
    also assume " $a = a \otimes b \otimes c$ "
    also have " $\dots = a \otimes (b \otimes c)$ " by (simp add: m_assoc)
    finally have " $a \otimes 1 = a \otimes (b \otimes c)$ " .

    hence rinvc: " $1 = b \otimes c$ " by (intro l_cancel[of "a" "1" "b  $\otimes$  c"], simp+)
    also have " $\dots = c \otimes b$ " by (simp add: m_comm)
    finally have linvc: " $1 = c \otimes b$ " .

    from ccarr linvc[symmetric] rinvc[symmetric]
    have " $b \in \text{Units } G$ " unfolding Units_def by fastsimp
    with nunit
    show "False" ..
  qed

lemma properfactorE:
  fixes G (structure)
  assumes pf: "properfactor G a b"
  and r: " $\llbracket a \text{ divides } b; \neg(b \text{ divides } a) \rrbracket \implies P$ "
  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;  $\neg(a \sim b)$ ]  $\implies$  P"
    shows "P"
using pf
unfolding properfactor_def
by (fast elim: elim associatedE)

lemma (in monoid) properfactor_unitE:
  assumes uunit: "u  $\in$  Units G"
  and pf: "properfactor G a u"
  and acarr: "a  $\in$  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 dvds: "a divides b" "properfactor G b c"
  and carr: "a  $\in$  carrier G" "b  $\in$  carrier G" "c  $\in$  carrier G"
  shows "properfactor G a c"
using dvds carr
apply (elim properfactorE, intro properfactorI)
  apply (iprover intro: divides_trans)+
done

lemma (in monoid) properfactor_trans2 [trans]:
  assumes dvds: "properfactor G a b" "b divides c"
  and carr: "a  $\in$  carrier G" "b  $\in$  carrier G" "c  $\in$  carrier G"
  shows "properfactor G a c"
using dvds carr
apply (elim properfactorE, intro properfactorI)
  apply (iprover intro: divides_trans)+
done

lemma properfactor_lless:
  fixes G (structure)
  shows "properfactor G = lless (division_rel G)"
apply (rule ext) apply (rule ext) apply rule
  apply (fastsimp elim: properfactorE2 intro: weak_llessI)
  apply (fastsimp elim: weak_llessE intro: properfactorI2)
done

lemma (in monoid) properfactor_cong_1 [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" "b ∈ carrier G" "c ∈ carrier G"
  shows "properfactor G (c ⊗ a) (c ⊗ b)"
  using ab carr
  by (fastsimp 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 (fastsimp elim: properfactorE intro: properfactorI)

lemma (in comm_monoid_cancel) properfactor_mult_rI [intro]:
  assumes ab: "properfactor G a b"
    and carr: "a ∈ carrier G" "b ∈ carrier G" "c ∈ carrier G"
  shows "properfactor G (a ⊗ c) (b ⊗ c)"
  using ab carr

```

```

by (fastsimp 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 (fastsimp 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+)

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+)

```

7.4 Irreducible Elements and Primes

7.4.1 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 irred: "irreducible G a"
    and aa': "a ~ a'"
    and carr[simp]: "a ∈ carrier G" "a' ∈ carrier G"
  shows "irreducible G a'"
using assms
apply (elim irreducibleE, intro irreducibleI)
apply simp_all
proof clarify
  assume "a' ∈ Units G"
  also note aa'[symmetric]
  finally have aunit: "a ∈ Units G" by simp

  assume "a ∉ Units G"
  with aunit
    show "False" by fast
next
fix b
  assume r[rule_format]: "∀b. b ∈ carrier G ∧ properfactor G b a →
b ∈ Units G"
  and bcarr[simp]: "b ∈ carrier G"
  assume "properfactor G b a'"
  also note aa'[symmetric]
  finally
    have "properfactor G b a" by simp

  with bcarr
    show "b ∈ Units G" by (fast intro: r)
qed

lemma (in monoid) irreducible_prod_rI:
  assumes airr: "irreducible G a"
    and bunit: "b ∈ Units G"
    and carr[simp]: "a ∈ carrier G" "b ∈ carrier G"
  shows "irreducible G (a ⊗ b)"
using airr carr bunit
apply (elim irreducibleE, intro irreducibleI, clarify)
apply (subgoal_tac "a ∈ Units G", simp)
apply (intro prod_unit_r[of a b] carr bunit, assumption)
proof -
  fix c
  assume [simp]: "c ∈ carrier G"
  and r[rule_format]: "∀b. b ∈ carrier G ∧ properfactor G b a → b
∈ Units G"
  assume "properfactor G c (a ⊗ b)"
  also have "a ⊗ b ~ a" by (intro associatedI2[OF bunit], simp+)

```

```

    finally
      have pfa: "properfactor G c a" by simp
    show "c ∈ Units G" by (rule r, simp add: pfa)
  qed

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)"
apply (subst m_comm, simp+)
apply (intro irreducible_prod_rI assms)
done

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")
  assume aunit: "a ∈ Units G"

  have "irreducible G b"
  apply (rule irreducibleI)
  proof (rule ccontr, simp)
    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"
    hence "properfactor G c (a ⊗ b)" by (simp add: properfactor_prod_l[of
c b a])
    from ccarr this show "c ∈ Units G" by (fast intro: isunit)
  qed

  from aunit this show "P" by (rule e2)
next
  assume anunit: "a ∉ Units G"
  with carr have "properfactor G b (b ⊗ a)" by (fast intro: properfactorI3)

```

```

hence bf: "properfactor G b (a  $\otimes$  b)" by (subst m_comm[of a b], simp+)
hence bunit: "b  $\in$  Units G" by (intro isunit, simp)

have "irreducible G a"
apply (rule irreducibleI)
proof (rule ccontr, simp)
  assume "a  $\in$  Units G"
  with bunit have "(a  $\otimes$  b)  $\in$  Units G" by fast
  with abnunit show "False" ..
next
fix c
assume ccarr: "c  $\in$  carrier G"
and "properfactor G c a"
hence "properfactor G c (a  $\otimes$  b)" by (simp add: properfactor_prod_r[of
c a b])
from ccarr this show "c  $\in$  Units G" by (fast intro: isunit)
qed

from this bunit show "P" by (rule e1)
qed
qed

```

7.4.2 Prime elements

```

lemma primeI:
  fixes G (structure)
  assumes "p  $\notin$  Units G"
  and " $\bigwedge a\ b. \llbracket a \in \text{carrier } G; b \in \text{carrier } G; p \text{ divides } (a \otimes b) \rrbracket \implies$ 
p divides a  $\vee$  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: " $\llbracket p \notin \text{Units } G; \forall a \in \text{carrier } G. \forall b \in \text{carrier } G.
p \text{ divides } a \otimes b \longrightarrow p \text{ divides } a \vee p \text{ divides }
b \rrbracket \implies P$ "
  shows "P"
using pprime
unfolding prime_def
by (blast dest: e)

```

```

lemma (in comm_monoid_cancel) prime_divides:
  assumes carr: "a  $\in$  carrier G" "b  $\in$  carrier G"
  and pprime: "prime G p"
  and pdvd: "p divides a  $\otimes$  b"

```

```

    shows "p divides a  $\vee$  p divides b"
using assms
by (blast elim: primeE)

lemma (in monoid_cancel) prime_cong [trans]:
  assumes pprime: "prime G p"
    and pp': "p  $\sim$  p'"
    and carr[simp]: "p  $\in$  carrier G" "p'  $\in$  carrier G"
  shows "prime G p'"
using pprime
apply (elim primeE, intro primeI)
proof clarify
  assume pnunit: "p  $\notin$  Units G"
  assume "p'  $\in$  Units G"
  also note pp'[symmetric]
  finally
    have "p  $\in$  Units G" by simp
  with pnunit
  show False ..
next
  fix a b
  assume r[rule_format]:
    " $\forall a \in \text{carrier } G. \forall b \in \text{carrier } G. p \text{ divides } a \otimes b \longrightarrow p \text{ divides } a \vee p \text{ divides } b$ "
  assume p'dvd: "p' divides a  $\otimes$  b"
    and carr'[simp]: "a  $\in$  carrier G" "b  $\in$  carrier G"

  note pp'
  also note p'dvd
  finally
    have "p divides a  $\otimes$  b" by simp
  hence "p divides a  $\vee$  p divides b" by (intro r, simp+)
  moreover {
    note pp'[symmetric]
    also assume "p divides a"
    finally
      have "p' divides a" by simp
    hence "p' divides a  $\vee$  p' divides b" by simp
  }
  moreover {
    note pp'[symmetric]
    also assume "p divides b"
    finally
      have "p' divides b" by simp
    hence "p' divides a  $\vee$  p' divides b" by simp
  }
  ultimately
  show "p' divides a  $\vee$  p' divides b" by fast
qed

```

7.5 Factorization and Factorial Monoids

7.5.1 Function definitions

```

constdefs (structure G)
  factors :: "[_, 'a list, 'a]  $\Rightarrow$  bool"
  "factors G fs a == ( $\forall x \in (\text{set fs}). \text{irreducible } G \ x) \wedge \text{foldr } (\text{op } \otimes) \text{ fs } 1 = a$ "

  wfactors :: "[_, 'a list, 'a]  $\Rightarrow$  bool"
  "wfactors G fs a == ( $\forall x \in (\text{set fs}). \text{irreducible } G \ x) \wedge \text{foldr } (\text{op } \otimes) \text{ fs } 1 \sim a$ "

abbreviation
  list_assoc :: "('a,_) monoid_scheme  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool" (infix "[~]" 44) where
    "list_assoc G == list_all2 (op  $\sim_G$ )"

constdefs (structure G)
  essentially_equal :: "[_, 'a list, 'a list]  $\Rightarrow$  bool"
  "essentially_equal G fs1 fs2 == ( $\exists fs1'. fs1 <\sim\sim> fs1' \wedge fs1' [\sim] fs2$ )"

locale factorial_monoid = comm_monoid_cancel +
  assumes factors_exist:
    "[a  $\in$  carrier G; a  $\notin$  Units G]  $\Longrightarrow \exists fs. \text{set } fs \subseteq \text{carrier } G \wedge$ 
factors G fs a"
  and factors_unique:
    "[factors G fs a; factors G fs' a; a  $\in$  carrier G; a  $\notin$  Units
G;
    set fs  $\subseteq$  carrier G; set fs'  $\subseteq$  carrier G]  $\Longrightarrow$  essentially_equal
G fs fs'"

```

7.5.2 Comparing lists of elements

Association on lists

```

lemma (in monoid) listassoc_refl [simp, intro]:
  assumes "set as  $\subseteq$  carrier G"
  shows "as [~] as"
using assms
by (induct as) simp+

lemma (in monoid) listassoc_sym [sym]:
  assumes "as [~] bs"
  and "set as  $\subseteq$  carrier G" and "set bs  $\subseteq$  carrier G"
  shows "bs [~] as"
using assms
proof (induct as arbitrary: bs, simp)
  case Cons
  thus ?case

```

```

    apply (induct bs, simp)
    apply clarsimp
    apply (iprover intro: associated_sym)
  done
qed

lemma (in monoid) listassoc_trans [trans]:
  assumes "as [~] bs" and "bs [~] cs"
    and "set as  $\subseteq$  carrier G" and "set bs  $\subseteq$  carrier G" and "set cs  $\subseteq$ 
carrier G"
  shows "as [~] cs"
using assms
apply (simp add: list_all2_conv_all_nth set_conv_nth, safe)
apply (rule associated_trans)
  apply (subgoal_tac "as ! i ~ bs ! i", assumption)
  apply (simp, simp)
  apply blast+
done

lemma (in monoid_cancel) irrlist_listassoc_cong:
  assumes " $\forall a \in \text{set as. irreducible G } a$ "
    and "as [~] bs"
    and "set as  $\subseteq$  carrier G" and "set bs  $\subseteq$  carrier G"
  shows " $\forall a \in \text{set bs. irreducible G } a$ "
using assms
apply (clarsimp simp add: list_all2_conv_all_nth set_conv_nth)
apply (blast intro: irreducible_cong)
done

Permutations

lemma perm_map [intro]:
  assumes p: "a <~~> b"
  shows "map f a <~~> map f b"
using p
by induct auto

lemma perm_map_switch:
  assumes m: "map f a = map f b" and p: "b <~~> c"
  shows " $\exists d. a <~~> d \wedge \text{map f } d = \text{map f } c$ "
using p m
by (induct arbitrary: a) (simp, force, force, blast)

lemma (in monoid) perm_assoc_switch:
  assumes a: "as [~] bs" and p: "bs <~~> cs"
  shows " $\exists bs'. as <~~> bs' \wedge bs' [~] cs$ "
using p a
apply (induct bs cs arbitrary: as, simp)
  apply (clarsimp simp add: list_all2_Cons2, blast)
  apply (clarsimp simp add: list_all2_Cons2)

```

```

    apply blast
  apply blast
done

```

```

lemma (in monoid) perm_assoc_switch_r:
  assumes p: "as <~~> bs" and a: "bs [~] cs"
  shows "∃bs'. as [~] bs' ∧ bs' <~~> cs"
using p a
apply (induct as bs arbitrary: cs, simp)
  apply (clarsimp simp add: list_all2_Cons1, blast)
  apply (clarsimp simp add: list_all2_Cons1)
  apply blast
apply blast
done

```

```

declare perm_sym [sym]

```

```

lemma perm_setP:
  assumes perm: "as <~~> bs"
  and as: "P (set as)"
  shows "P (set bs)"
proof -
  from perm
    have "multiset_of as = multiset_of bs"
    by (simp add: multiset_of_eq_perm)
  hence "set as = set bs" by (rule multiset_of_eq_setD)
  with as
    show "P (set bs)" by simp
qed

```

```

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  $\subseteq$  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  $\subseteq$  carrier G" "set bs  $\subseteq$  carrier G"
  shows "essentially_equal G bs as"
using ee
proof (elim essentially_equalE)
  fix fs
  assume "as  $\sim\sim$  fs" "fs  $[\sim]$  bs"
  hence " $\exists$  fs'. as  $[\sim]$  fs'  $\wedge$  fs'  $\sim\sim$  bs" by (rule perm_assoc_switch_r)
  from this obtain fs'
    where a: "as  $[\sim]$  fs'" and p: "fs'  $\sim\sim$  bs"
    by auto
  from p have "bs  $\sim\sim$  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  $\subseteq$  carrier G"
  and bscarr: "set bs  $\subseteq$  carrier G"
  and cscarr: "set cs  $\subseteq$  carrier G"
  shows "essentially_equal G as cs"
using ab bc
proof (elim essentially_equalE)
  fix abs bcs
  assume "abs  $[\sim]$  bs" and pb: "bs  $\sim\sim$  bcs"
  hence " $\exists$  bs'. abs  $\sim\sim$  bs'  $\wedge$  bs'  $[\sim]$  bcs" by (rule perm_assoc_switch)
  from this obtain bs'
    where p: "abs  $\sim\sim$  bs'" and a: "bs'  $[\sim]$  bcs"
    by auto

  assume "as  $\sim\sim$  abs"
  with p
    have pp: "as  $\sim\sim$  bs'" by fast

  from pp ascarr have c1: "set bs'  $\subseteq$  carrier G" by (rule perm_closed)
  from pb bscarr have c2: "set bcs  $\subseteq$  carrier G" by (rule perm_closed)

```



```

note a
also assume "bcs [~] cs"
finally (listassoc_trans) have "bs' [~] cs" by (simp add: c1 c2 cscarr)

with pp
  show ?thesis
  by (rule essentially_equalI)
qed

```

7.5.3 Properties of lists of elements

Multiplication of factors in a list

```

lemma (in monoid) multlist_closed [simp, intro]:
  assumes ascarr: "set fs  $\subseteq$  carrier G"
  shows "foldr (op  $\otimes$ ) fs 1  $\in$  carrier G"
by (insert ascarr, induct fs, simp+)

```

```

lemma (in comm_monoid) multlist_dividesI :
  assumes "f  $\in$  set fs" and "f  $\in$  carrier G" and "set fs  $\subseteq$  carrier G"
  shows "f divides (foldr (op  $\otimes$ ) fs 1)"
using assms
apply (induct fs)
  apply simp
  apply (case_tac "f = a", simp)
  apply (fast intro: dividesI)
  apply clarsimp
  apply (elim dividesE, intro dividesI)
  defer 1
  apply (simp add: m_comm)
  apply (simp add: m_assoc[symmetric])
  apply (simp add: m_comm)
  apply simp
done

```

```

lemma (in comm_monoid_cancel) multlist_listassoc_cong:
  assumes "fs [~] fs'"
  and "set fs  $\subseteq$  carrier G" and "set fs'  $\subseteq$  carrier G"
  shows "foldr (op  $\otimes$ ) fs 1  $\sim$  foldr (op  $\otimes$ ) fs' 1"
using assms
proof (induct fs arbitrary: fs', simp)
  case (Cons a as fs')
  thus ?case
  apply (induct fs', simp)
  proof clarsimp
    fix b bs
    assume "a  $\sim$  b"
    and acarr: "a  $\in$  carrier G" and bcarr: "b  $\in$  carrier G"
    and ascarr: "set as  $\subseteq$  carrier G"
    hence p: "a  $\otimes$  foldr op  $\otimes$  as 1  $\sim$  b  $\otimes$  foldr op  $\otimes$  as 1"

```

```

      by (fast intro: mult_cong_l)
    also
      assume "as [~] bs"
      and bscarr: "set bs  $\subseteq$  carrier G"
      and " $\bigwedge fs'. \llbracket as [~] fs'; set fs' \subseteq carrier G \rrbracket \implies foldr op \otimes$ "
as 1  $\sim foldr op \otimes fs' 1$ "
      hence "foldr op  $\otimes$  as 1  $\sim foldr op \otimes bs 1$ " by simp
      with ascarr bscarr bcarr
      have "b  $\otimes foldr op \otimes as 1 \sim b \otimes foldr op \otimes bs 1$ "
      by (fast intro: mult_cong_r)
    finally
      show "a  $\otimes foldr op \otimes as 1 \sim b \otimes foldr op \otimes bs 1$ "
      by (simp add: ascarr bscarr acarr bcarr)
  qed
qed

lemma (in comm_monoid) multlist_perm_cong:
  assumes prm: "as <~~> bs"
  and ascarr: "set as  $\subseteq$  carrier G"
  shows "foldr (op  $\otimes$ ) as 1 = foldr (op  $\otimes$ ) bs 1"
using prm ascarr
apply (induct, simp, clarsimp simp add: m_ac, clarsimp)
proof clarsimp
  fix xs ys zs
  assume "xs <~~> ys" "set xs  $\subseteq$  carrier G"
  hence "set ys  $\subseteq$  carrier G" by (rule perm_closed)
  moreover assume "set ys  $\subseteq$  carrier G  $\implies foldr op \otimes ys 1 = foldr op$ 
 $\otimes zs 1$ "
  ultimately show "foldr op  $\otimes ys 1 = foldr op \otimes zs 1$ " by simp
qed

lemma (in comm_monoid_cancel) multlist_ee_cong:
  assumes "essentially_equal G fs fs'"
  and "set fs  $\subseteq$  carrier G" and "set fs'  $\subseteq$  carrier G"
  shows "foldr (op  $\otimes$ ) fs 1  $\sim foldr (op \otimes) fs' 1$ "
using assms
apply (elim essentially_equalE)
apply (simp add: multlist_perm_cong multlist_listassoc_cong perm_closed)
done

```

7.5.4 Factorization in irreducible elements

```

lemma wfactorsI:
  fixes G (structure)
  assumes " $\forall f \in set fs. irreducible G f$ "
  and "foldr (op  $\otimes$ ) fs 1  $\sim a$ "
  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: "[ $\forall f \in \text{set } fs. \text{irreducible } G f; \text{foldr } (\text{op } \otimes) \text{ fs } 1 \sim a$ ]]  $\implies$ 
P"
  shows "P"
using wf
unfolding wfactors_def
by (fast dest: e)
```

```
lemma (in monoid) factorsI:
  assumes " $\forall f \in \text{set } fs. \text{irreducible } G f$ "
  and " $\text{foldr } (\text{op } \otimes) \text{ fs } 1 = a$ "
  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: "[ $\forall f \in \text{set } fs. \text{irreducible } G f; \text{foldr } (\text{op } \otimes) \text{ fs } 1 = a$ ]]  $\implies$  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  $\subseteq$  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  $\subseteq$  carrier G"
  shows " $\exists a'. \text{factors } G \text{ as } a' \wedge a' \sim a$ "
using assms
by (blast elim: wfactorsE intro: factorsI)
```

```
lemma (in monoid) factors_closed [dest]:
  assumes "factors G fs a" and "set fs  $\subseteq$  carrier G"
  shows " $a \in \text{carrier } G$ "
using assms
by (elim factorsE, clarsimp)
```

```
lemma (in monoid) nunit_factors:
  assumes anunit: " $a \notin \text{Units } G$ "
```

```

    and fs: "factors G as a"
    shows "length as > 0"
  apply (insert fs, elim factorsE)
  proof (cases "length as = 0")
    assume "length as = 0"
    hence fold: "foldr op  $\otimes$  as 1 = 1" by force

    assume "foldr op  $\otimes$  as 1 = a"
    with fold
      have "a = 1" by simp
    then have "a  $\in$  Units G" by fast
    with anunit
      have "False" by simp
    thus ?thesis ..
  qed simp

lemma (in monoid) unit_wfactors [simp]:
  assumes aunit: "a  $\in$  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  $\in$  Units G"
  and wf: "wfactors G fs a"
  and carr[simp]: "set fs  $\subseteq$  carrier G"
  shows "fs = []"
proof (rule ccontr, cases fs, simp)
  fix f fs'
  assume fs: "fs = f # fs'"

  from carr
    have fcarr[simp]: "f  $\in$  carrier G"
    and carr'[simp]: "set fs'  $\subseteq$  carrier G"
    by (simp add: fs)+

  from fs wf
    have "irreducible G f" by (simp add: wfactors_def)
  hence fnunit: "f  $\notin$  Units G" by (fast elim: irreducibleE)

  from fs wf
    have a: "f  $\otimes$  foldr (op  $\otimes$ ) fs' 1  $\sim$  a" by (simp add: wfactors_def)

  note aunit
  also from fs wf
    have a: "f  $\otimes$  foldr (op  $\otimes$ ) fs' 1  $\sim$  a" by (simp add: wfactors_def)
    have "a  $\sim$  f  $\otimes$  foldr (op  $\otimes$ ) fs' 1"
    by (simp add: Units_closed[OF aunit] a[symmetric])
  finally

```

```

      have "f  $\otimes$  foldr (op  $\otimes$ ) fs' 1  $\in$  Units G" by simp
      hence "f  $\in$  Units G" by (intro unit_factor[of f], simp+)

    with fnunit show "False" by simp
  qed

Comparing wfactors

lemma (in comm_monoid_cancel) wfactors_listassoc_cong_1:
  assumes fact: "wfactors G fs a"
    and asc: "fs  $[\sim]$  fs'"
    and carr: "a  $\in$  carrier G" "set fs  $\subseteq$  carrier G" "set fs'  $\subseteq$  carrier
G"
  shows "wfactors G fs' a"
using fact
apply (elim wfactorsE, intro wfactorsI)
proof -
  assume " $\forall f \in \text{set fs. irreducible G } f$ "
  also note asc
  finally (irrlist_listassoc_cong)
    show " $\forall f \in \text{set fs'. irreducible G } f$ " by (simp add: carr)
next
  from asc[symmetric]
    have "foldr op  $\otimes$  fs' 1  $\sim$  foldr op  $\otimes$  fs 1"
    by (simp add: multlist_listassoc_cong carr)
  also assume "foldr op  $\otimes$  fs 1  $\sim$  a"
  finally
    show "foldr op  $\otimes$  fs' 1  $\sim$  a" by (simp add: carr)
qed

lemma (in comm_monoid) wfactors_perm_cong_1:
  assumes "wfactors G fs a"
    and "fs  $\langle \sim \rangle$  fs'"
    and "set fs  $\subseteq$  carrier G"
  shows "wfactors G fs' a"
using assms
apply (elim wfactorsE, intro wfactorsI)
  apply (rule irrlist_perm_cong, assumption+)
  apply (simp add: multlist_perm_cong[symmetric])
done

lemma (in comm_monoid_cancel) wfactors_ee_cong_1 [trans]:
  assumes ee: "essentially_equal G as bs"
    and bfs: "wfactors G bs b"
    and carr: "b  $\in$  carrier G" "set as  $\subseteq$  carrier G" "set bs  $\subseteq$  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  $\subseteq$  carrier G" by (simp add: perm_closed)

note bfs
also assume [symmetric]: "fs [~] bs"
also (wfactors_listassoc_cong_1)
  note prm[symmetric]
finally (wfactors_perm_cong_1)
  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  $\sim$  a'"
  and carr[simp]: "a  $\in$  carrier G" "a'  $\in$  carrier G" "set fs  $\subseteq$  carrier
  G"
  shows "wfactors G fs a'"
using fac
proof (elim wfactorsE, intro wfactorsI)
  assume "foldr op  $\otimes$  fs 1  $\sim$  a" also note aa'
  finally show "foldr op  $\otimes$  fs 1  $\sim$  a'" by simp
qed

```

7.5.5 Essentially equal factorizations

```

lemma (in comm_monoid_cancel) unitfactor_ee:
  assumes uunit: "u  $\in$  Units G"
  and carr: "set as  $\subseteq$  carrier G"
  shows "essentially_equal G (as[0 := (as!0  $\otimes$  u)]) as" (is "essentially_equal
  G ?as' as")
using assms
apply (intro essentially_equalI[of _ ?as'], simp)
apply (cases as, simp)
apply (clarsimp, fast intro: associatedI2[of u])
done

lemma (in comm_monoid_cancel) factors_cong_unit:
  assumes uunit: "u  $\in$  Units G" and anunit: "a  $\notin$  Units G"
  and afs: "factors G as a"
  and ascarr: "set as  $\subseteq$  carrier G"
  shows "factors G (as[0 := (as!0  $\otimes$  u)]) (a  $\otimes$  u)" (is "factors G ?as'
  ?a'")
using assms
apply (elim factorsE, clarify)
apply (cases as)
apply (simp add: nunit_factors)
apply clarsimp
apply (elim factorsE, intro factorsI)
apply (clarsimp, fast intro: irreducible_prod_rI)

```

```

apply (simp add: m_ac Units_closed)
done

```

```

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 op ⊗ as 1 ~ a"
  hence "a ~ foldr op ⊗ as 1" by (rule associated_sym, simp+)
  also from prm
    have "foldr op ⊗ as 1 = foldr op ⊗ bs 1" by (rule multlist_perm_cong,
simp)
  also assume "foldr op ⊗ 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 op ⊗ as 1 ~ a"
  hence "a ~ foldr op ⊗ as 1" by (rule associated_sym, simp+)
  also from assoc
    have "foldr op ⊗ as 1 ~ foldr op ⊗ bs 1" by (rule multlist_listassoc_cong,
simp+)
  also assume "foldr op ⊗ 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"
hence as'carr[simp]: "set fs  $\subseteq$  carrier G" by (simp add: perm_closed)
from afs prm
  have afs': "wfactors G fs a" by (rule wfactors_perm_cong_1, simp)
assume "fs [~] bs"
from this afs' bfs
  show "a ~ b" by (rule listassoc_wfactorsD, simp+)
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  $\subseteq$  carrier G" "set bs  $\subseteq$  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  $\notin$  Units G"
  and bfs: "factors G bs b" and bnunit: "b  $\notin$  Units G"
  and ascarr: "set as  $\subseteq$  carrier G" and bscarr: "set bs  $\subseteq$  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
    have " $\exists u \in \text{Units } G. a = b \otimes u$ " by (fast elim: associatedE2)
  from this obtain u
    where uunit: "u  $\in$  Units G"
    and a: "a = b  $\otimes$  u" by auto

  from uunit bscarr
    have ee: "essentially_equal G (bs[0 := (bs!0  $\otimes$  u)]) bs"
    (is "essentially_equal G ?bs' bs")
    by (rule unitfactor_ee)

  from bscarr uunit
    have bs'carr: "set ?bs'  $\subseteq$  carrier G"
    by (cases bs) (simp add: Units_closed)+

  from uunit bnunit bfs bscarr
    have fac: "factors G ?bs' (b  $\otimes$  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")
  assume aunit: "a ∈ Units G"
  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)
  thus ?thesis by (simp add: e e')
next
  assume anunit: "a ∉ Units G"
  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

  have "∃ a'. factors G as a' ∧ a' ~ a" by (rule wfactors_factors[OF
asf ascarr])
  from this obtain a'
    where fa': "factors G as a'"
    and a': "a' ~ a"
    by auto
  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

  have "∃b'. factors G bs b' ∧ b' ~ b" by (rule wfactors_factors[OF
bsf bscarr])
  from this obtain b'
    where fb': "factors G bs b'"
    and b': "b' ~ b"
    by auto
  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")

```

```

    assume "a ∈ Units G"
    hence "wfactors G [] a" by (rule unit_wfactors)
    thus ?thesis by (intro exI) force
next
    assume "a ∉ Units G"
    hence "∃fs. set fs ⊆ carrier G ∧ factors G fs a" by (intro factors_exist
acarr)
    from this obtain fs
      where fscarr: "set fs ⊆ carrier G"
      and f: "factors G fs a"
      by auto
    from f have "wfactors G fs a" by (rule factors_wfactors) fact
    from fscarr this
      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)"
using assms
unfolding factors_def
apply (safe, force)
apply (induct fa)
  apply simp
apply (simp add: m_assoc)
done

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)"
apply (insert wfactors_factors[OF asf ascarr])
apply (insert 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+
  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  $\subseteq$  carrier G" and acarr: "a  $\in$  carrier G"
    and f: "f  $\in$  set fs"
  shows "f divides a"
apply (insert wfactors_factors[OF p fscarr], clarsimp)
proof -
  fix a'
  assume fsa': "factors G fs a'"
  and a'a: "a'  $\sim$  a"
  with fscarr
    have a'carr: "a'  $\in$  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

```

7.5.6 Factorial monoids and wfactors

```

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

  from wfactors_exists[OF acarr]
  obtain as
    where ascarr: "set as  $\subseteq$  carrier G"
    and afs: "wfactors G as a"
    by auto
  from afs ascarr
    have " $\exists a'. \text{factors } G \text{ as } a' \wedge a' \sim a$ " by (rule wfactors_factors)
  from this obtain a'
    where afs': "factors G as a'"
    and a'a: "a'  $\sim$  a"
    by auto
  from afs' ascarr
    have a'carr: "a'  $\in$  carrier G" by fast
  have a'nunit: "a'  $\notin$  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
  have "∃u. u ∈ Units G ∧ a' = a ⊗ u" by (blast elim: associatedE2)
from this obtain u
  where uunit: "u ∈ Units G"
  and a': "a' = a ⊗ u"
  by auto

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: Units_r_inv 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, clarsimp+)

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)

```

7.6 Factorizations as Multisets

Gives useful operations like intersection

abbreviation

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

constdefs (structure G)

```
"fmset G as ≡ multiset_of (map (λa. assocs G a) as)"
```

Helper lemmas

lemma (in monoid) assocs_repr_independence:

```
assumes "y ∈ assocs G x"
```

```
and "x ∈ carrier G"
```

```
shows "assocs G x = assocs G y"
```

```

using assms
apply safe
  apply (elim closure_ofE2, intro closure_ofI2[of _ _ y])
    apply (clarsimp, iprover intro: associated_trans associated_sym, simp+)
  apply (elim closure_ofE2, intro closure_ofI2[of _ _ x])
    apply (clarsimp, iprover intro: associated_trans, simp+)
done

```

```

lemma (in monoid) assocs_self:
  assumes "x ∈ carrier G"
  shows "x ∈ assocs G x"
using assms
by (fastsimp 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"
  and "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]

```

7.6.1 Comparing multisets

```

lemma (in monoid) fmset_perm_cong:
  assumes prm: "as <~~> bs"
  shows "fmset G as = fmset G bs"
using perm_map[OF prm]
by (simp add: multiset_of_eq_perm fmset_def)

```

```

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
apply (induct as arbitrary: bs, simp)
apply (clarsimp simp add: Cons_eq_map_conv list_all2_Cons1, safe)
  apply (clarsimp elim!: closure_ofE2) defer 1
  apply (clarsimp elim!: closure_ofE2) defer 1

```

```

proof -
  fix a x z
  assume carr[simp]: "a ∈ carrier G" "x ∈ carrier G" "z ∈ carrier
G"
  assume "x ~ a"
  also assume "a ~ z"
  finally have "x ~ z" by simp
  with carr
    show "x ∈ assocs G z"
    by (intro closure_ofI2) simp+
next
  fix a x z
  assume carr[simp]: "a ∈ carrier G" "x ∈ carrier G" "z ∈ carrier
G"
  assume "x ~ z"
  also assume [symmetric]: "a ~ z"
  finally have "x ~ a" by simp
  with carr
    show "x ∈ assocs G a"
    by (intro closure_ofI2) simp+
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
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 monoid_cancel) fmset_ee__hlp_induct:
  assumes prm: "cas <~~> cbs"
  and cdef: "cas = map (assocs G) as" "cbs = map (assocs G) bs"
  shows "∀as bs. (cas <~~> cbs ∧ cas = map (assocs G) as ∧
    cbs = map (assocs G) bs) ⟶ (∃as'. as <~~> as' ∧ map
  (assocs G) as' = cbs)"
  apply (rule perm.induct[of cas cbs], rule prm)
  apply safe apply simp_all
  apply (simp add: map_eq_Cons_conv, blast)
  apply force
  proof -
    fix ys as bs
    assume p1: "map (assocs G) as <~~> ys"
    and r1[rule_format]:
      "∀asa bs. map (assocs G) as = map (assocs G) asa ∧
        ys = map (assocs G) bs
        ⟶ (∃as'. asa <~~> as' ∧ map (assocs G) as' = map
  (assocs G) bs)"
    and p2: "ys <~~> map (assocs G) bs"
    and r2[rule_format]:
      "∀as bsa. ys = map (assocs G) as ∧
        map (assocs G) bs = map (assocs G) bsa
        ⟶ (∃as'. as <~~> as' ∧ map (assocs G) as' = map (assocs
  G) bsa)"
    and p3: "map (assocs G) as <~~> map (assocs G) bs"

    from p1
      have "multiset_of (map (assocs G) as) = multiset_of ys"
      by (simp add: multiset_of_eq_perm)
    hence setys: "set (map (assocs G) as) = set ys" by (rule multiset_of_eq_setD)

    have "set (map (assocs G) as) = { assocs G x | x. x ∈ set as}" by clarsimp
  fast
    with setys have "set ys ⊆ { assocs G x | x. x ∈ set as}" by simp
    hence "∃yy. ys = map (assocs G) yy"
    apply (induct ys, simp, clarsimp)
  proof -
    fix yy x
    show "∃yya. (assocs G x) # map (assocs G) yy =
      map (assocs G) yya"
    by (rule exI[of _ "x#yy"], simp)
  qed
    from this obtain yy
      where ys: "ys = map (assocs G) yy"
      by auto

    from p1 ys
      have "∃as'. as <~~> as' ∧ map (assocs G) as' = map (assocs G) yy"

```

```

    by (intro r1, simp)
  from this obtain as'
    where asas': "as <~~> as'"
    and as'yy: "map (assocs G) as' = map (assocs G) yy"
    by auto

  from p2 ys
    have "∃ as'. yy <~~> as' ∧ map (assocs G) as' = map (assocs G) bs"
    by (intro r2, simp)
  from this obtain as''
    where yyas'': "yy <~~> as''"
    and as''bs: "map (assocs G) as'' = map (assocs G) bs"
    by auto

  from as'yy and yyas''
    have "∃ cs. as' <~~> cs ∧ map (assocs G) cs = map (assocs G) as''"
    by (rule perm_map_switch)
  from this obtain cs
    where as'cs: "as' <~~> cs"
    and csas'': "map (assocs G) cs = map (assocs G) as''"
    by auto

  from asas' and as'cs
    have ascs: "as <~~> cs" by fast
  from csas'' and as''bs
    have "map (assocs G) cs = map (assocs G) bs" by simp
  from ascs and this
    show "∃ as'. as <~~> as' ∧ map (assocs G) as' = map (assocs G) bs" by
fast
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 mpp: "map (assocs G) as <~~> map (assocs G) bs"
    by (simp add: fmset_def multiset_of_eq_perm)

  have "∃ cas. cas = map (assocs G) as" by simp
  from this obtain cas where cas: "cas = map (assocs G) as" by simp

  have "∃ cbs. cbs = map (assocs G) bs" by simp
  from this obtain cbs where cbs: "cbs = map (assocs G) bs" by simp

  from cas cbs mpp
    have [rule_format]:
      "∀ as bs. (cas <~~> cbs ∧ cas = map (assocs G) as ∧

```

```

      cbs = map (assoc G) bs
      → (∃ as'. as <~~> as' ∧ map (assoc G) as' = cbs)"
    by (intro fmset_ee_hlp_induct, simp+)
  with mpp cas cbs
    have "∃ as'. as <~~> as' ∧ map (assoc G) as' = map (assoc G) bs"
    by simp

  from this obtain as'
    where tp: "as <~~> as'"
    and tm: "map (assoc G) as' = map (assoc G) bs"
    by auto
  from tm have lene: "length as' = length bs" by (rule map_eq_imp_length_eq)
  from tp have "set as = set as'" by (simp add: multiset_of_eq_perm multiset_of_eq_setD)
  with ascarr
    have as'_carr: "set as' ⊆ carrier G" by simp

  from tm as'_carr[THEN subsetD] bscarr[THEN subsetD]
    have "as' [~] bs"
    by (induct as' arbitrary: bs) (simp, fastsimp dest: assoc_eqD[THEN associated_sym])

  from tp and this
    show "essentially_equal G as bs" by (fast intro: essentially_equalI)
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)

```

7.6.2 Interpreting multisets as factorizations

```

lemma (in monoid) mset_fmsetEx:
  assumes elems: "∧ X. X ∈ set_of Cs ⇒ ∃ x. P x ∧ X = assoc G x"
  shows "∃ cs. (∀ c ∈ set cs. P c) ∧ fmset G cs = Cs"
proof -
  have "∃ Cs'. Cs = multiset_of Cs'"
    by (rule surjE[OF surj_multiset_of], fast)
  from this obtain Cs'
    where Cs: "Cs = multiset_of Cs'"
    by auto

  have "∃ cs. (∀ c ∈ set cs. P c) ∧ multiset_of (map (assoc G) cs) = Cs"
  using elems
  unfolding Cs
  apply (induct Cs', simp)
  apply clarsimp

```

```

    apply (subgoal_tac "∃ cs. (∀ x ∈ set cs. P x) ∧
                                multiset_of (map (assocs G) cs) = multiset_of
Cs'")
  proof clarsimp
    fix a Cs' cs
    assume ih: "∧ X. X = a ∨ X ∈ set Cs' ⇒ ∃ x. P x ∧ X = assocs G
x"
    and csP: "∀ x ∈ set cs. P x"
    and mset: "multiset_of (map (assocs G) cs) = multiset_of Cs'"
    from ih
      have "∃ x. P x ∧ a = assocs G x" by fast
    from this obtain c
      where cP: "P c"
      and a: "a = assocs G c"
      by auto
    from cP csP
      have tP: "∀ x ∈ set (c#cs). P x" by simp
    from mset a
      have "multiset_of (map (assocs G) (c#cs)) = multiset_of Cs' + {#a#}"
by simp
    from tP this
      show "∃ cs. (∀ x ∈ set cs. P x) ∧
                                multiset_of (map (assocs G) cs) =
                                multiset_of Cs' + {#a#}" by fast
  qed simp
  thus ?thesis by (simp add: fmset_def)
qed

lemma (in monoid) mset_wfactorsEx:
  assumes elems: "∧ X. X ∈ set_of 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)
    from this 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) fast+
  from this obtain c
    where ccarr: "c ∈ carrier G"
    and cfs: "wfactors G cs c"
    by auto

  with cscarr Cs
    show ?thesis by fast
qed

```

7.6.3 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+)
  with carr
    have "fmset G cs = fmset G (as@bs)" by (intro ee_fmset, simp+)
    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
add: carr)+
  then show "c ~ a ⊗ b" by (rule ee_wfactorsD[of "cs" "as@bs"]) (simp
add: assms m)+
qed

```

7.6.4 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"
  hence "∃ cs. set cs ⊆ carrier G ∧ wfactors G cs c" by (rule wfactors_exist)
  from this obtain cs
    where cscarr: "set cs ⊆ carrier G"
    and cfs: "wfactors G cs c" by auto
  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+

  thus ?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 "count (fmset G as) X < count (fmset G bs) X"
  hence "0 < count (fmset G bs) X" by simp
  hence "X ∈ set_of (fmset G bs)" by simp
  hence "X ∈ set (map (assocs G) bs)" by (simp add: fmset_def)
  hence "∃ x. x ∈ set bs ∧ X = assocs G x" by (induct bs) auto
  from this 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
from this 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_conv_count_eq mset_le_def)
hence basic: "b ~ a ⊗ c"
  by (rule fmset_wfactors_mult) fact+

thus ?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 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"
apply (rule properfactorI)
apply (rule fmsubset_divides[of as bs], fact+)
proof
  assume "b divides a"
  hence "fmset G bs ≤# fmset G as"
    by (rule divides_fmsubset) fact+
  with asubb
    have "fmset G as = fmset G bs" by (simp add: mset_le_antisym)
  with anb
    show "False" ..
qed

```

```

lemma (in factorial_monoid) properfactor_fmset:
  assumes pf: "properfactor G a b"
    and "wfactors G as a" and "wfactors G bs b"
    and "a ∈ carrier G" and "b ∈ carrier G"
    and "set as ⊆ carrier G" and "set bs ⊆ carrier G"
  shows "fmset G as ≤# fmset G bs ∧ fmset G as ≠ fmset G bs"
using pf
apply (elim properfactorE)
apply rule
  apply (intro divides_fmsubset, assumption)
  apply (rule assms)+
proof
  assume bna: "¬ b divides a"
  assume "fmset G as = fmset G bs"
  then have "essentially_equal G as bs" by (rule fmset_ee) fact+
  hence "a ~ b" by (rule ee_wfactorsD[of as bs]) fact+
  hence "b divides a" by (elim associatedE)
  with bna
    show "False" ..
qed

```

7.7 Irreducible Elements are Prime

```

lemma (in factorial_monoid) irreducible_is_prime:
  assumes pirr: "irreducible G p"
    and pcarr: "p ∈ carrier G"
  shows "prime G p"
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
  have "∃c∈carrier G. a ⊗ b = p ⊗ c" by (rule dividesD)
from this obtain c
  where ccarr: "c ∈ carrier G"
  and abpc: "a ⊗ b = p ⊗ c"
  by auto

  from acarr have "∃fs. set fs ⊆ carrier G ∧ wfactors G fs a" by (rule
wfactors_exist)
  from this obtain as where ascarr: "set as ⊆ carrier G" and afs: "wfactors
G as a" by auto

  from bcarr have "∃fs. set fs ⊆ carrier G ∧ wfactors G fs b" by (rule
wfactors_exist)
  from this obtain bs where bscarr: "set bs ⊆ carrier G" and bfs: "wfactors
G bs b" by auto

  from ccarr have "∃fs. set fs ⊆ carrier G ∧ wfactors G fs c" by (rule
wfactors_exist)
  from this obtain cs where cscarr: "set cs ⊆ carrier G" and cfs: "wfactors
G cs c" by auto

  note carr[simp] = pcarr acarr bcarr ccarr ascarr bscarr cscarr

  from afs and bfs
    have abfs: "wfactors G (as @ bs) (a ⊗ b)" by (rule wfactors_mult)
fact+

  from pirr cfs
    have pcfs: "wfactors G (p # cs) (p ⊗ c)" by (rule wfactors_mult_single)
fact+
  with abpc
    have abfs': "wfactors G (p # cs) (a ⊗ b)" by simp

  from abfs' abfs
    have "essentially_equal G (p # cs) (as @ bs)"
    by (rule wfactors_unique) simp+

  hence "∃ds. p # cs <~~> ds ∧ ds [~] (as @ bs)"
    by (fast elim: essentially_equalE)
  from this obtain ds
    where "p # cs <~~> ds"
    and dsassoc: "ds [~] (as @ bs)"

```

```

    by auto

then have "p ∈ set ds"
  by (simp add: perm_set_eq[symmetric])
with dsassoc
  have "∃p'. p' ∈ set (as@bs) ∧ p ~ p'"
  unfolding list_all2_conv_all_nth set_conv_nth
  by force

from this obtain p'
  where "p' ∈ set (as@bs)"
  and pp': "p ~ p'"
  by auto

hence "p' ∈ set as ∨ p' ∈ set bs" by simp
moreover
{
  assume p'elem: "p' ∈ set as"
  with ascarr have [simp]: "p' ∈ carrier G" by fast

  note pp'
  also from afs
    have "p' divides a" by (rule wfactors_dividesI) fact+
  finally
    have "p divides a" by simp
}
moreover
{
  assume p'elem: "p' ∈ set bs"
  with bscarr have [simp]: "p' ∈ carrier G" by fast

  note pp'
  also from bfs
    have "p' divides b" by (rule wfactors_dividesI) fact+
  finally
    have "p divides b" by simp
}
ultimately
  show "p divides a ∨ p divides b" by fast
qed

```

— A version using factors, more complicated

```

lemma (in factorial_monoid) factors_irreducible_is_prime:
  assumes pirr: "irreducible G p"
  and pcarr: "p ∈ carrier G"
  shows "prime G p"
using pirr
apply (elim irreducibleE, intro primeI)

```

```

apply assumption
proof -
  fix a b
  assume acarr: "a ∈ carrier G"
    and bcarr: "b ∈ carrier G"
    and pdvdab: "p divides (a ⊗ b)"
  assume irreduc[rule_format]:
    "∀b. b ∈ carrier G ∧ properfactor G b p ⟶ b ∈ Units G"
  from pdvdab
    have "∃c∈carrier G. a ⊗ b = p ⊗ c" by (rule dividesD)
  from this obtain c
    where ccarr: "c ∈ carrier G"
    and abpc: "a ⊗ b = p ⊗ c"
    by auto
  note [simp] = pcarr acarr bcarr ccarr

  show "p divides a ∨ p divides b"
  proof (cases "a ∈ Units G")
    assume aunite: "a ∈ Units G"

    note pdvdab
    also have "a ⊗ b = b ⊗ a" by (simp add: m_comm)
    also from aunite
      have bab: "b ⊗ a ∼ b"
      by (intro associatedI2[of "a"], simp+)
    finally
      have "p divides b" by simp
    thus "p divides a ∨ p divides b" ..
  next
    assume anunit: "a ∉ Units G"

    show "p divides a ∨ p divides b"
    proof (cases "b ∈ Units G")
      assume bunit: "b ∈ Units G"

      note pdvdab
      also from bunit
        have baa: "a ⊗ b ∼ a"
        by (intro associatedI2[of "b"], simp+)
      finally
        have "p divides a" by simp
      thus "p divides a ∨ p divides b" ..
    next
      assume bnunit: "b ∉ Units G"

      have cnunit: "c ∉ Units G"
      proof (rule ccontr, simp)
        assume cunit: "c ∈ Units G"
        from bnunit

```

```

        have "properfactor G a (a  $\otimes$  b)"
        by (intro properfactorI3[of _ _ b], simp+)
    also note abpc
    also from cunit
        have "p  $\otimes$  c  $\sim$  p"
        by (intro associatedI2[of c], simp+)
    finally
        have "properfactor G a p" by simp

    with acarr
        have "a  $\in$  Units G" by (fast intro: irreduc)
    with anunit
        show "False" ..
qed

have abnunit: "a  $\otimes$  b  $\notin$  Units G"
proof clarsimp
    assume abunit: "a  $\otimes$  b  $\in$  Units G"
    hence "a  $\in$  Units G" by (rule unit_factor) fact+
    with anunit
        show "False" ..
qed

    from acarr anunit have " $\exists$ fs. set fs  $\subseteq$  carrier G  $\wedge$  factors G fs"
a" by (rule factors_exist)
    then obtain as where ascarr: "set as  $\subseteq$  carrier G" and afac: "factors
G as a" by auto

    from bcarr bnunit have " $\exists$ fs. set fs  $\subseteq$  carrier G  $\wedge$  factors G fs"
b" by (rule factors_exist)
    then obtain bs where bscarr: "set bs  $\subseteq$  carrier G" and bfac: "factors
G bs b" by auto

    from ccarr cnunit have " $\exists$ fs. set fs  $\subseteq$  carrier G  $\wedge$  factors G fs"
c" by (rule factors_exist)
    then obtain cs where cscarr: "set cs  $\subseteq$  carrier G" and cfac: "factors
G cs c" by auto

    note [simp] = ascarr bscarr cscarr

    from afac and bfac
        have abfac: "factors G (as @ bs) (a  $\otimes$  b)" by (rule factors_mult)
fact+

    from pirr cfac
        have pcfac: "factors G (p # cs) (p  $\otimes$  c)" by (rule factors_mult_single)
fact+
    with abpc
        have abfac': "factors G (p # cs) (a  $\otimes$  b)" by simp

```

```

from abfac' abfac
  have "essentially_equal G (p # cs) (as @ bs)"
  by (rule factors_unique) (fact | simp)+

hence "∃ ds. p # cs <~~> ds ∧ ds [~] (as @ bs)"
  by (fast elim: essentially_equalE)
from this obtain ds
  where "p # cs <~~> ds"
  and dsassoc: "ds [~] (as @ bs)"
  by auto

then have "p ∈ set ds"
  by (simp add: perm_set_eq[symmetric])
with dsassoc
  have "∃ p'. p' ∈ set (as@bs) ∧ p ~ p'"
  unfolding list_all2_conv_all_nth set_conv_nth
  by force

from this obtain p'
  where "p' ∈ set (as@bs)"
  and pp': "p ~ p'" by auto

hence "p' ∈ set as ∨ p' ∈ set bs" by simp
moreover
{
  assume p'elem: "p' ∈ set as"
  with ascarr have [simp]: "p' ∈ carrier G" by fast

  note pp'
  also from afac p'elem
    have "p' divides a" by (rule factors_dividesI) fact+
  finally
    have "p divides a" by simp
}
moreover
{
  assume p'elem: "p' ∈ set bs"
  with bscarr have [simp]: "p' ∈ carrier G" by fast

  note pp'
  also from bfac
    have "p' divides b" by (rule factors_dividesI) fact+
  finally have "p divides b" by simp
}
ultimately
  show "p divides a ∨ p divides b" by fast
qed
qed

```

qed

7.8 Greatest Common Divisors and Lowest Common Multiples

7.8.1 Definitions

```

constdefs (structure G)
  isgcd :: "('a,_) monoid_scheme, 'a, 'a, 'a]  $\Rightarrow$  bool"  ("(_ gcdofz _ _)" [81,81,81] 80)
  "x gcdof a b  $\equiv$  x divides a  $\wedge$  x divides b  $\wedge$ 
    ( $\forall y \in \text{carrier } G. (y \text{ divides } a \wedge y \text{ divides } b \longrightarrow y \text{ divides } x)$ )"

  islcm :: "[_ , 'a, 'a, 'a]  $\Rightarrow$  bool"  ("(_ lcmofz _ _)" [81,81,81] 80)
  "x lcmof a b  $\equiv$  a divides x  $\wedge$  b divides x  $\wedge$ 
    ( $\forall y \in \text{carrier } G. (a \text{ divides } y \wedge b \text{ divides } y \longrightarrow x \text{ divides } y)$ )"

constdefs (structure G)
  somegcd :: "('a,_) monoid_scheme  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  'a"
  "somegcd G a b == SOME x. x  $\in$  carrier G  $\wedge$  x gcdof a b"

  somelcm :: "('a,_) monoid_scheme  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  'a"
  "somelcm G a b == SOME x. x  $\in$  carrier G  $\wedge$  x lcmof a b"

constdefs (structure G)
  "SomeGcd G A == inf (division_rel G) A"

locale gcd_condition_monoid = comm_monoid_cancel +
  assumes gcdof_exists:
    "[a  $\in$  carrier G; b  $\in$  carrier G]  $\Longrightarrow \exists c. c \in$  carrier G  $\wedge$  c gcdof a b"

locale primeness_condition_monoid = comm_monoid_cancel +
  assumes irreducible_prime:
    "[a  $\in$  carrier G; irreducible G a]  $\Longrightarrow$  prime G a"

locale divisor_chain_condition_monoid = comm_monoid_cancel +
  assumes division_wellfounded:
    "wf {(x, y). x  $\in$  carrier G  $\wedge$  y  $\in$  carrier G  $\wedge$  properfactor G x y}"

```

7.8.2 Connections to Lattice.thy

```

lemma gcdof_greatestLower:
  fixes G (structure)
  assumes carr[simp]: "a  $\in$  carrier G" "b  $\in$  carrier G"
  shows "(x  $\in$  carrier G  $\wedge$  x gcdof a b) =

```

```

      greatest (division_rel G) x (Lower (division_rel G) {a, b})"
unfolding isgcd_def greatest_def Lower_def elem_def
proof (simp, safe)
  fix xa
  assume r1[rule_format]: " $\forall x. (x = a \vee x = b) \wedge x \in \text{carrier } G \longrightarrow xa \text{ divides } x$ "
  assume r2[rule_format]: " $\forall y \in \text{carrier } G. y \text{ divides } a \wedge y \text{ divides } b \longrightarrow y \text{ divides } x$ "

  assume "xa  $\in$  carrier G" "x divides a" "x divides b"
  with carr
  show "xa divides x"
    by (fast intro: r1 r2)
next
fix a' y
assume r1[rule_format]:
  " $\forall xa \in \{1. \forall x. (x = a \vee x = b) \wedge x \in \text{carrier } G \longrightarrow 1 \text{ divides } x\} \cap \text{carrier } G.$ "
  xa divides x"
assume "y  $\in$  carrier G" "y divides a" "y divides b"
with carr
  show "y divides x"
    by (fast intro: r1)
qed (simp, simp)

lemma lcmof_leastUpper:
  fixes G (structure)
  assumes carr[simp]: "a  $\in$  carrier G" "b  $\in$  carrier G"
  shows "(x  $\in$  carrier G  $\wedge$  x lcmof a b) =
    least (division_rel G) x (Upper (division_rel G) {a, b})"
unfolding islcm_def least_def Upper_def elem_def
proof (simp, safe)
  fix xa
  assume r1[rule_format]: " $\forall x. (x = a \vee x = b) \wedge x \in \text{carrier } G \longrightarrow x \text{ divides } xa$ "
  assume r2[rule_format]: " $\forall y \in \text{carrier } G. a \text{ divides } y \wedge b \text{ divides } y \longrightarrow x \text{ divides } y$ "

  assume "xa  $\in$  carrier G" "a divides x" "b divides x"
  with carr
  show "x divides xa"
    by (fast intro: r1 r2)
next
fix a' y
assume r1[rule_format]:
  " $\forall xa \in \{1. \forall x. (x = a \vee x = b) \wedge x \in \text{carrier } G \longrightarrow x \text{ divides } 1\} \cap \text{carrier } G.$ "
  x divides xa"
assume "y  $\in$  carrier G" "a divides y" "b divides y"

```

```

with carr
  show "x divides y"
  by (fast intro: r1)
qed (simp, simp)

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"
unfolding somegcd_def meet_def inf_def
by (simp add: 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

```

7.8.3 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 acarr have "∃as. set as ⊆ carrier G ∧ wfactors G as a" by (rule
wfactors_exist)
  from this obtain as
    where ascarr: "set as ⊆ carrier G"
    and afs: "wfactors G as a"
    by auto
  from afs have airr: "∀a ∈ set as. irreducible G a" by (fast elim:
wfactorsE)

  from bcarr have "∃bs. set bs ⊆ carrier G ∧ wfactors G bs b" by (rule
wfactors_exist)
  from this obtain bs
    where bscarr: "set bs ⊆ carrier G"
    and bfs: "wfactors G bs b"
    by auto

```



```

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 ∈ set_of (fmset G as #∩ fmset G bs)"
  hence "X ∈ set_of (fmset G as)" by (simp add: multiset_inter_def)
  hence "X ∈ set (map (assocs G) as)" by (simp add: fmset_def)
  hence "∃x. X = assocs G x ∧ x ∈ set as" by (induct as) auto
  from this obtain x
    where X: "X = assocs G x"
    and xas: "x ∈ set as"
    by auto
  with xcarr have xcarr: "x ∈ carrier G" by fast
  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 fast
qed

from this 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 add: multiset_inter_def mset_le_def)
  thus "c divides a" by (rule fmsubset_divides) fact+
next
  from csmset
    have "fmset G cs ≤# fmset G bs"
    by (simp add: multiset_inter_def mset_le_def, force)
  thus "c divides b" by (rule fmsubset_divides) fact+
next
  fix y
  assume ycarr: "y ∈ carrier G"
  hence "∃ys. set ys ⊆ carrier G ∧ wfactors G ys y" by (rule wfactors_exist)
  from this obtain ys
    where yscarr: "set ys ⊆ carrier G"
    and yfs: "wfactors G ys y"
    by auto

```

```

    assume "y divides a"
    hence ya: "fmset G ys ≤# fmset G as" by (rule divides_fmsubset) fact+

    assume "y divides b"
    hence 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: mset_le_def multiset_inter_count)
    thus "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 acar: "a ∈ carrier G" and bcarr: "b ∈ carrier G"
  shows "∃c. c ∈ carrier G ∧ c lcmof a b"
proof -
  from acar have "∃as. set as ⊆ carrier G ∧ wfactors G as a" by (rule
wfactors_exist)
  from this obtain as
    where ascarr: "set as ⊆ carrier G"
    and afs: "wfactors G as a"
    by auto
  from afs have airr: "∀a ∈ set as. irreducible G a" by (fast elim:
wfactorsE)

  from bcarr have "∃bs. set bs ⊆ carrier G ∧ wfactors G bs b" by (rule
wfactors_exist)
  from this obtain bs
    where bscarr: "set bs ⊆ carrier G"
    and bfs: "wfactors G bs b"
    by auto
  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 ∈ set_of ((fmset G as - fmset G bs) + fmset G bs)"
  hence "X ∈ set_of (fmset G as) ∨ X ∈ set_of (fmset G bs)"
    by (cases "X :# fmset G bs", simp, simp)
  moreover
  {
    assume "X ∈ set_of (fmset G as)"

```

```

    hence "X ∈ set (map (assocs G) as)" by (simp add: fmset_def)
    hence "∃x. x ∈ set as ∧ X = assocs G x" by (induct as) auto
    from this 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
      have "∃x. (x ∈ carrier G ∧ irreducible G x) ∧ X = assocs G
x" by fast
  }
  moreover
  {
    assume "X ∈ set_of (fmset G bs)"
    hence "X ∈ set (map (assocs G) bs)" by (simp add: fmset_def)
    hence "∃x. x ∈ set bs ∧ X = assocs G x" by (induct as) auto
    from this 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
      have "∃x. (x ∈ carrier G ∧ irreducible G x) ∧ X = assocs G
x" by fast
  }
  ultimately
  show "∃x. (x ∈ carrier G ∧ irreducible G x) ∧ X = assocs G x" by
fast
qed

    from this 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: mset_le_def,
force)
      thus "a divides c" by (rule fmsubset_divides) fact+
    next
      from csmset have "fmset G bs ≤# fmset G cs" by (simp add: mset_le_def)
      thus "b divides c" by (rule fmsubset_divides) fact+

```

```

next
  fix y
  assume ycarr: "y ∈ carrier G"
  hence "∃ys. set ys ⊆ carrier G ∧ wfactors G ys y" by (rule wfactors_exist)
  from this obtain ys
    where yscarr: "set ys ⊆ carrier G"
    and yfs: "wfactors G ys y"
    by auto

  assume "a divides y"
  hence ya: "fmset G as ≤# fmset G ys" by (rule divides_fmsubset) fact+

  assume "b divides y"
  hence yb: "fmset G bs ≤# fmset G ys" by (rule divides_fmsubset) fact+

  from ya yb csmset
  have "fmset G cs ≤# fmset G ys"
    apply (simp add: mset_le_def, clarify)
    apply (case_tac "count (fmset G as) a < count (fmset G bs) a")
    apply simp
    apply simp
  done
  thus "c divides y" by (rule fmsubset_divides) fact+
qed

with ccarr
  show "∃c. c ∈ carrier G ∧ c lcmof a b" by fast
qed

```

7.9 Conditions for Factoriality

7.9.1 Gcd condition

```

lemma (in gcd_condition_monoid) division_weak_lower_semilattice [simp]:
  shows "weak_lower_semilattice (division_rel G)"
proof -
  interpret weak_partial_order "division_rel G" ..
  show ?thesis
  apply (unfold_locales, simp_all)
  proof -
    fix x y
    assume carr: "x ∈ carrier G" "y ∈ carrier G"
    hence "∃z. z ∈ carrier G ∧ z gcdof x y" by (rule gcdof_exists)
    from this obtain z
      where zcarr: "z ∈ carrier G"
      and isgcd: "z gcdof x y"
      by auto
    with carr
    have "greatest (division_rel G) z (Lower (division_rel G) {x, y})"
      by (subst gcdof_greatestLower[symmetric], simp+)
  qed

```

```

    thus "∃z. greatest (division_rel G) z (Lower (division_rel G) {x,
y})" by fast
  qed
qed

lemma (in gcd_condition_monoid) gcdof_cong_1:
  assumes a'a: "a' ~ a"
    and agcd: "a gcdof b c"
    and a'carr: "a' ∈ carrier G" and carr': "a ∈ carrier G" "b ∈ carrier
G" "c ∈ carrier G"
  shows "a' gcdof b c"
proof -
  note carr = a'carr carr'
  interpret weak_lower_semilattice "division_rel G" by simp
  have "a' ∈ carrier G ∧ a' gcdof b c"
    apply (simp add: gcdof_greatestLower carr')
    apply (subst greatest_Lower_cong_1[of _ a])
    apply (simp add: a'a)
    apply (simp add: carr)
    apply (simp add: carr)
    apply (simp add: carr)
    apply (simp add: gcdof_greatestLower[symmetric] agcd carr)
  done
  thus ?thesis ..
qed

lemma (in gcd_condition_monoid) gcd_closed [simp]:
  assumes carr: "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
    apply (simp add: somegcd_meet[OF carr])
    apply (rule meet_closed[simplified], fact+)
  done
qed

lemma (in gcd_condition_monoid) gcd_isgcd:
  assumes carr: "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 carr
  have "somegcd G a b ∈ carrier G ∧ (somegcd G a b) gcdof a b"
    apply (subst gcdof_greatestLower, simp, simp)
    apply (simp add: somegcd_meet[OF carr] meet_def)
    apply (rule inf_of_two_greatest[simplified], assumption+)
  done
  thus "(somegcd G a b) gcdof a b" by simp

```

qed

```

lemma (in gcd_condition_monoid) gcd_exists:
  assumes carr: "a ∈ carrier G" "b ∈ carrier G"
  shows "∃x∈carrier G. x = somegcd G a b"
proof -
  interpret weak_lower_semilattice "division_rel G" by simp
  show ?thesis
    apply (simp add: somegcd_meet[OF carr])
    apply (rule meet_closed[simplified], fact+)
  done
qed

```

```

lemma (in gcd_condition_monoid) gcd_divides_l:
  assumes carr: "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
    apply (simp add: somegcd_meet[OF carr])
    apply (rule meet_left[simplified], fact+)
  done
qed

```

```

lemma (in gcd_condition_monoid) gcd_divides_r:
  assumes carr: "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
    apply (simp add: somegcd_meet[OF carr])
    apply (rule meet_right[simplified], fact+)
  done
qed

```

```

lemma (in gcd_condition_monoid) gcd_divides:
  assumes sub: "z divides x" "z divides y"
  and L: "x ∈ carrier G" "y ∈ carrier G" "z ∈ carrier G"
  shows "z divides (somegcd G x y)"
proof -
  interpret weak_lower_semilattice "division_rel G" by simp
  show ?thesis
    apply (simp add: somegcd_meet L)
    apply (rule meet_le[simplified], fact+)
  done
qed

```

```

lemma (in gcd_condition_monoid) gcd_cong_l:
  assumes xx': "x ∼ x'"

```

```

    and carr: "x ∈ carrier G" "x' ∈ carrier G" "y ∈ carrier G"
  shows "somegcd G x y ~ somegcd G x' y"
proof -
  interpret weak_lower_semilattice "division_rel G" by simp
  show ?thesis
    apply (simp add: somegcd_meet carr)
    apply (rule meet_cong_l[simplified], fact+)
  done
qed

```

```

lemma (in gcd_condition_monoid) gcd_cong_r:
  assumes carr: "x ∈ carrier G" "y ∈ carrier G" "y' ∈ carrier G"
    and yy': "y ~ y'"
  shows "somegcd G x y ~ somegcd G x y'"
proof -
  interpret weak_lower_semilattice "division_rel G" by simp
  show ?thesis
    apply (simp add: somegcd_meet carr)
    apply (rule meet_cong_r[simplified], fact+)
  done
qed

```

```

lemma (in gcd_condition_monoid) gcdI:
  assumes dvd: "a divides b" "a divides c"
    and others: "∀y∈carrier G. y divides b ∧ y divides c ⟶ y divides a"
    and acarr: "a ∈ carrier G" and bcarr: "b ∈ carrier G" and ccarr:
      "c ∈ carrier G"
  shows "a ~ somegcd G b c"
  apply (simp add: somegcd_def)
  apply (rule someI2_ex)
  apply (rule exI[of _ a], simp add: isgcd_def)
  apply (simp add: assms)
  apply (simp add: isgcd_def assms, clarify)
  apply (insert assms, blast intro: associatedI)
  done

```

```

lemma (in gcd_condition_monoid) gcdI2:
  assumes "a gcdof b c"
    and "a ∈ carrier G" and bcarr: "b ∈ carrier G" and ccarr: "c ∈ carrier G"
  shows "a ~ somegcd G b c"
  using assms
  unfolding isgcd_def
  by (blast intro: gcdI)

```

```

lemma (in gcd_condition_monoid) SomeGcd_ex:

```

```

    assumes "finite A" "A  $\subseteq$  carrier G" "A  $\neq$  {}"
    shows " $\exists x \in \text{carrier } G. x = \text{SomeGcd } G \ A$ "
  proof -
    interpret weak_lower_semilattice "division_rel G" by simp
    show ?thesis
      apply (simp add: SomeGcd_def)
      apply (rule finite_inf_closed[simplified], fact+)
    done
  qed

lemma (in gcd_condition_monoid) gcd_assoc:
  assumes carr: "a  $\in$  carrier G" "b  $\in$  carrier G" "c  $\in$  carrier G"
  shows "somegcd G (somegcd G a b) c  $\sim$  somegcd G a (somegcd G b c)"
  proof -
    interpret weak_lower_semilattice "division_rel G" by simp
    show ?thesis
      apply (subst (2 3) somegcd_meet, (simp add: carr)+)
      apply (simp add: somegcd_meet carr)
      apply (rule weak_meet_assoc[simplified], fact+)
    done
  qed

lemma (in gcd_condition_monoid) gcd_mult:
  assumes acarr: "a  $\in$  carrier G" and bcarr: "b  $\in$  carrier G" and ccarr:
    "c  $\in$  carrier G"
  shows "c  $\otimes$  somegcd G a b  $\sim$  somegcd G (c  $\otimes$  a) (c  $\otimes$  b)"
  proof -
    let ?d = "somegcd G a b"
    let ?e = "somegcd G (c  $\otimes$  a) (c  $\otimes$  b)"
    note carr[simp] = acarr bcarr ccarr
    have dcarr: "?d  $\in$  carrier G" by simp
    have ecarr: "?e  $\in$  carrier G" by simp
    note carr = carr dcarr ecarr

    have "?d divides a" by (simp add: gcd_divides_l)
    hence cd'ca: "c  $\otimes$  ?d divides (c  $\otimes$  a)" by (simp add: divides_mult_lI)

    have "?d divides b" by (simp add: gcd_divides_r)
    hence cd'cb: "c  $\otimes$  ?d divides (c  $\otimes$  b)" by (simp add: divides_mult_lI)

    from cd'ca cd'cb
      have cd'e: "c  $\otimes$  ?d divides ?e"
      by (rule gcd_divides) simp+

    hence " $\exists u. u \in \text{carrier } G \wedge ?e = c \otimes ?d \otimes u$ "
      by (elim dividesE, fast)
    from this obtain u
      where ucarr[simp]: "u  $\in$  carrier G"
      and e_cdu: "?e = c  $\otimes$  ?d  $\otimes$  u"

```



```

    by auto

note carr = carr ucarr

have "?e divides c ⊗ a" by (rule gcd_divides_l) simp+
hence "∃x. x ∈ carrier G ∧ c ⊗ a = ?e ⊗ x"
  by (elim dividesE, fast)
from this obtain x
  where xcarr: "x ∈ carrier G"
  and ca_ex: "c ⊗ a = ?e ⊗ x"
  by auto
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)+
hence du'a: "?d ⊗ u divides a" by (rule dividesI[OF xcarr])

have "?e divides c ⊗ b" by (intro gcd_divides_r, simp+)
hence "∃x. x ∈ carrier G ∧ c ⊗ b = ?e ⊗ x"
  by (elim dividesE, fast)
from this obtain x
  where xcarr: "x ∈ carrier G"
  and cb_ex: "c ⊗ b = ?e ⊗ x"
  by auto
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+)
hence 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+)
hence 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)" .
  hence i2: "1 = u ⊗ v" by (rule l_cancel) simp+
  hence i1: "1 = v ⊗ u" by (simp add: m_comm)
  from vcarr i1[symmetric] i2[symmetric]

```

```

      show "u ∈ Units G"
      by (unfold Units_def, simp, fast)
qed

from e_cdu uunit
  have "somegcd G (c ⊗ a) (c ⊗ b) ∼ c ⊗ somegcd G a b"
  by (intro associatedI2[of u], simp+)
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: "ALL a b. a : carrier G & b : carrier G & a ∼ b
    --> f a : carrier G & f b : carrier G & f a ∼ f b"
  and carr: "a ∈ carrier G" "b ∈ carrier G"
  shows "f a ∼ f b"
  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"
apply unfold_locales
apply (rule primeI)
  apply (elim irreducibleE, assumption)
proof -
  fix p a b
  assume pcarr: "p ∈ carrier G" and acarr: "a ∈ carrier G" and bcarr:
    "b ∈ carrier G"
  and pirr: "irreducible G p"
  and pdvdab: "p divides a ⊗ b"
  from pirr
    have pnunit: "p ∉ Units G"
    and r[rule_format]: "∀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"
  with pcarr acarr
  have "1 ~ somegcd G p a"
  apply (intro gcdI, simp, simp, simp)
  apply (fast intro: unit_divides)
  apply (fast intro: unit_divides)
  apply (clarsimp simp add: Unit_eq_dividesone[symmetric])
  apply (rule r, rule, assumption)
  apply (rule properfactorI, assumption)
proof (rule ccontr, simp)
  fix y
  assume ycarr: "y ∈ carrier G"
  assume "p divides y"
  also assume "y divides a"
  finally
    have "p divides a" by (simp add: pcarr ycarr acarr)
  with npdvda
  show "False" ..
qed simp+
with pcarr acarr
  have pa: "somegcd G p a ~ 1" by (fast intro: associated_sym[of
    "1"] gcd_closed)

  assume npdvdb: "¬ p divides b"
  with pcarr bcarr
  have "1 ~ somegcd G p b"
  apply (intro gcdI, simp, simp, simp)

```

```

    apply (fast intro: unit_divides)
    apply (fast intro: unit_divides)
    apply (clarsimp simp add: Unit_eq_dividesone[symmetric])
    apply (rule r, rule, assumption)
    apply (rule properfactorI, assumption)
    proof (rule ccontr, simp)
      fix y
      assume ycarr: "y ∈ carrier G"
      assume "p divides y"
      also assume "y divides b"
      finally have "p divides b" by (simp add: pcarr ycarr bcarr)
      with npdvdb
        show "False" ..
    qed simp+
    with pcarr bcarr
      have pb: "somegcd G p b ~ 1" by (fast intro: associated_sym[of
"1"] gcd_closed)

    from pcarr acarr bcarr pdvdab
      have "p gcdof p (a ⊗ b)" by (fast intro: isgcd_divides_1)

    with pcarr acarr bcarr
      have "p ~ somegcd G p (a ⊗ b)" by (fast intro: gcdI2)
    also from pa pb pcarr acarr bcarr
      have "somegcd G p (a ⊗ b) ~ 1" by (rule relprime_mult)
    finally have "p ~ 1" by (simp add: pcarr acarr bcarr)

    with pcarr
      have "p ∈ Units G" by (fast intro: assoc_unit_1)
    with pnunit
      show "False" ..
  qed
qed

sublocale gcd_condition_monoid ⊆ primeness_condition_monoid
  by (rule primeness_condition)

```

7.9.2 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[rule_format]: "a ∈ carrier G ⟶ (∃ as. set as ⊆ carrier G ∧
wfactors G as a)"
    apply (rule wf_induct[OF division_wellfounded])
  proof -
    fix x
    assume ih: "∀ y. (y, x) ∈ {(x, y). x ∈ carrier G ∧ y ∈ carrier G

```

```

 $\wedge$  properfactor  $G$   $x$   $y$ }
 $\longrightarrow y \in \text{carrier } G \longrightarrow (\exists \text{as. set as} \subseteq \text{carrier } G \wedge$ 
wfactores  $G$  as  $y)$ "

show " $x \in \text{carrier } G \longrightarrow (\exists \text{as. set as} \subseteq \text{carrier } G \wedge \text{wfactores } G \text{ as}$ 
 $x)$ "
  apply clarify
  apply (cases "x  $\in$  Units  $G$ ")
  apply (rule exI[of _ "[]"], simp)
  apply (cases "irreducible  $G$   $x$ ")
  apply (rule exI[of _ "[x]"], simp add: wfactores_def)
  proof -
    assume xcarr: " $x \in \text{carrier } G$ "
    and xnunit: " $x \notin \text{Units } G$ "
    and xnirr: " $\neg \text{irreducible } G$   $x$ "
    hence " $\exists y. y \in \text{carrier } G \wedge \text{properfactor } G$   $y$   $x \wedge y \notin \text{Units } G$ "
    apply - apply (rule ccontr, simp)
    apply (subgoal_tac "irreducible  $G$   $x$ ", simp)
    apply (rule irreducibleI, simp, simp)
  done
  from this obtain y
    where ycarr: " $y \in \text{carrier } G$ "
    and ynunit: " $y \notin \text{Units } G$ "
    and pfyx: "properfactor  $G$   $y$   $x$ "
    by auto

  have ih':
    " $\wedge y. [y \in \text{carrier } G; \text{properfactor } G$   $y$   $x]$ 
 $\implies \exists \text{as. set as} \subseteq \text{carrier } G \wedge \text{wfactores } G \text{ as } y$ "
    by (rule ih[rule_format, simplified]) (simp add: xcarr)+

  from ycarr pfyx
    have " $\exists \text{as. set as} \subseteq \text{carrier } G \wedge \text{wfactores } G \text{ as } y$ "
    by (rule ih')
  from this obtain ys
    where yscarr: "set  $ys \subseteq \text{carrier } G$ "
    and yfs: "wfactores  $G$   $ys$   $y$ "
    by auto

  from pfyx
    have "y divides x"
    and nyx: " $\neg y \sim x$ "
    by - (fast elim: properfactorE2)+
  hence " $\exists z. z \in \text{carrier } G \wedge x = y \otimes z$ "
    by (fast elim: dividesE)

  from this obtain z
    where zcarr: " $z \in \text{carrier } G$ "
    and x: " $x = y \otimes z$ "

```

```

      by auto

    from zcarr ycarr
    have "properfactor G z x"
      apply (subst x)
      apply (intro properfactorI3[of _ _ y])
      apply (simp add: m_comm)
      apply (simp add: yunit)+
    done
  with zcarr
    have "∃ as. set as ⊆ carrier G ∧ wfactors G as z"
      by (rule ih')
  from this obtain zs
    where zscarr: "set zs ⊆ carrier G"
    and zfs: "wfactors G zs z"
    by auto

  from yscarr zscarr
    have xscarr: "set (ys@zs) ⊆ carrier G" by simp
  from yfs zfs ycarr zcarr yscarr zscarr
    have "wfactors G (ys@zs) (y⊗z)" by (rule wfactors_mult)
  hence "wfactors G (ys@zs) x" by (simp add: x)

  from xscarr this
    show "∃ xs. set xs ⊆ carrier G ∧ wfactors G xs x" by fast
qed
qed

  from acarr
    show ?thesis by (rule r)
qed

```

7.9.3 Primeness condition

```

lemma (in comm_monoid_cancel) multlist_prime_pos:
  assumes carr: "a ∈ carrier G" "set as ⊆ carrier G"
  and aprime: "prime G a"
  and "a divides (foldr (op ⊗) as 1)"
  shows "∃ i < length as. a divides (as!i)"
proof -
  have r[rule_format]:
    "set as ⊆ carrier G ∧ a divides (foldr (op ⊗) as 1)
    → (∃ i. i < length as ∧ a divides (as!i))"
  apply (induct as)
  apply clarsimp defer 1
  apply clarsimp defer 1
proof -
  assume "a divides 1"
  with carr

```

```

      have "a ∈ Units G"
      by (fast intro: divides_unit[of a 1])
    with aprime
      show "False" by (elim primeE, simp)
  next
    fix aa as
    assume ih[rule_format]: "a divides foldr op ⊗ as 1 → (∃ i < length
as. a divides as ! i)"
      and carr': "aa ∈ carrier G" "set as ⊆ carrier G"
      and "a divides aa ⊗ foldr op ⊗ as 1"
    with carr aprime
      have "a divides aa ∨ a divides foldr op ⊗ as 1"
      by (intro prime_divides) simp+
    moreover {
      assume "a divides aa"
      hence p1: "a divides (aa#as)!0" by simp
      have "0 < Suc (length as)" by simp
      with p1 have "∃ i < Suc (length as). a divides (aa # as) ! i" by fast
    }
    moreover {
      assume "a divides foldr op ⊗ as 1"
      hence "∃ i. i < length as ∧ a divides as ! i" by (rule ih)
      from this obtain i where "a divides as ! i" and len: "i < length
as" by auto
      hence p1: "a divides (aa#as) ! (Suc i)" by simp
      from len have "Suc i < Suc (length as)" by simp
      with p1 have "∃ i < Suc (length as). a divides (aa # as) ! i" by force
    }
    ultimately
      show "∃ i < Suc (length as). a divides (aa # as) ! i" by fast
  qed

  from assms
    show ?thesis
    by (intro r, safe)
qed

lemma (in primeness_condition_monoid) wfactors_unique__hlp_induct:
  "∀ a as'. a ∈ carrier G ∧ set as ⊆ carrier G ∧ set as' ⊆ carrier G
  ∧
    wfactors G as a ∧ wfactors G as' a → essentially_equal G
  as as'"
  apply (induct as)
  apply clarsimp defer 1
  apply clarsimp defer 1
  proof -
    fix a as'
    assume acarr: "a ∈ carrier G"
    and "wfactors G [] a"

```

```

hence aunit: "a ∈ Units G"
  apply (elim wfactorsE)
  apply (simp, rule assoc_unit_r[of "1"], simp+)
done

assume "set as' ⊆ carrier G" "wfactors G as' a"
with aunit
  have "as' = []"
  by (intro unit_wfactors_empty[of a])
thus "essentially_equal G [] as'" by simp
next
fix a as ah as'
assume ih[rule_format]:
  "∀ a as'. a ∈ carrier G ∧ set as' ⊆ carrier G ∧
    wfactors G as a ∧ wfactors G as' a → essentially_equal
G as as'"
  and acarr: "a ∈ carrier G" and ahcarr: "ah ∈ carrier G"
  and ascarr: "set as ⊆ carrier G" and as'carr: "set as' ⊆ carrier
G"
  and afs: "wfactors G (ah # as) a"
  and afs': "wfactors G as' a"
hence ahdvda: "ah divides a"
  by (intro wfactors_dividesI[of "ah#as" "a"], simp+)
hence "∃ a' ∈ carrier G. a = ah ⊗ a'" by (fast elim: dividesE)
from this obtain a'
  where a'carr: "a' ∈ carrier G"
  and a: "a = ah ⊗ a'"
  by auto
have a'fs: "wfactors G as a'"
  apply (rule wfactorsE[OF afs], rule wfactorsI, simp)
  apply (simp add: a, insert ascarr a'carr)
  apply (intro assoc_l_cancel[of ah _ a'] multlist_closed ahcarr, assumption+)
done

from afs have ahirr: "irreducible G ah" by (elim wfactorsE, simp)
with ascarr have ahprime: "prime G ah" by (intro irreducible_prime
ahcarr)

note carr [simp] = acarr ahcarr ascarr as'carr a'carr

note ahdvda
also from afs'
  have "a divides (foldr (op ⊗) as' 1)"
  by (elim wfactorsE associatedE, simp)
finally have "ah divides (foldr (op ⊗) as' 1)" by simp

with ahprime
  have "∃ i < length as'. ah divides as'!i"
  by (intro multlist_prime_pos, simp+)

```



```

from this obtain i
  where len: "i < length as'" and ahdvd: "ah divides as'!i"
  by auto
from afs' carr have irrasi: "irreducible G (as'!i)"
  by (fast intro: nth_mem[OF len] elim: wfactorsE)
from len carr
  have asicarr[simp]: "as'!i ∈ carrier G" by (unfold set_conv_nth,
force)
note carr = carr asicarr

from ahdvd have "∃ x ∈ carrier G. as'!i = ah ⊗ x" by (fast elim: dividesE)
from this obtain x where "x ∈ carrier G" and asi: "as'!i = ah ⊗ x"
by auto

with carr irrasi[simplified asi]
  have asiah: "as'!i ~ ah" apply -
  apply (elim irreducible_prodE[of "ah" "x"], assumption+)
  apply (rule associatedI2[of x], assumption+)
  apply (rule irreducibleE[OF ahirr], simp)
done

note setparts = set_take_subset[of i as'] set_drop_subset[of "Suc i"
as']
note partscarr [simp] = setparts[THEN subset_trans[OF _ as'carr]]
note carr = carr partscarr

have "∃ aa_1. aa_1 ∈ carrier G ∧ wfactors G (take i as') aa_1"
  apply (intro wfactors_prod_exists)
  using setparts afs' by (fast elim: wfactorsE, simp)
from this obtain aa_1
  where aa1carr: "aa_1 ∈ carrier G"
  and aa1fs: "wfactors G (take i as') aa_1"
  by auto

have "∃ aa_2. aa_2 ∈ carrier G ∧ wfactors G (drop (Suc i) as') aa_2"
  apply (intro wfactors_prod_exists)
  using setparts afs' by (fast elim: wfactorsE, simp)
from this obtain aa_2
  where aa2carr: "aa_2 ∈ carrier G"
  and aa2fs: "wfactors G (drop (Suc i) as') aa_2"
  by auto

note carr = carr aa1carr[simp] aa2carr[simp]

from aa1fs aa2fs
  have v1: "wfactors G (take i as' @ drop (Suc i) as') (aa_1 ⊗ aa_2)"
  by (intro wfactors_mult, simp+)
hence v1': "wfactors G (as'!i # take i as' @ drop (Suc i) as') (as'!i
⊗ (aa_1 ⊗ aa_2))"

```

```

    apply (intro wfactors_mult_single)
    using setparts afs'
    by (fast intro: nth_mem[OF len] elim: wfactorsE, simp+)

from aa2carr carr aa1fs aa2fs
  have "wfactors G (as'!i # drop (Suc i) as') (as'!i  $\otimes$  aa_2)"
  apply (intro wfactors_mult_single)
    apply (rule wfactorsE[OF afs'], fast intro: nth_mem[OF len])
    apply (fast intro: nth_mem[OF len])
    apply fast
    apply fast
  apply assumption
done
with len carr aa1carr aa2carr aa1fs
  have v2: "wfactors G (take i as' @ as'!i # drop (Suc i) as') (aa_1
 $\otimes$  (as'!i  $\otimes$  aa_2))"
  apply (intro wfactors_mult)
    apply fast
    apply (simp, (fast intro: nth_mem[OF len]))?)
done

from len
  have as': "as' = (take i as' @ as'!i # drop (Suc i) as')"
  by (simp add: drop_Suc_conv_tl)
with carr
  have eer: "essentially_equal G (take i as' @ as'!i # drop (Suc i)
as') as'"
  by simp

with v2 afs' carr aa1carr aa2carr nth_mem[OF len]
  have "aa_1  $\otimes$  (as'!i  $\otimes$  aa_2)  $\sim$  a"
  apply (intro ee_wfactorsD[of "take i as' @ as'!i # drop (Suc i) as'"
"as'"])
    apply fast+
    apply (simp, fast)
done
then
have t1: "as'!i  $\otimes$  (aa_1  $\otimes$  aa_2)  $\sim$  a"
  apply (simp add: m_assoc[symmetric])
  apply (simp add: m_comm)
done
from carr asiah
have "ah  $\otimes$  (aa_1  $\otimes$  aa_2)  $\sim$  as'!i  $\otimes$  (aa_1  $\otimes$  aa_2)"
  apply (intro mult_cong_1)
  apply (fast intro: associated_sym, simp+)
done
also note t1
finally
  have "ah  $\otimes$  (aa_1  $\otimes$  aa_2)  $\sim$  a" by simp

```

```

with carr aa1carr aa2carr a'carr nth_mem[OF len]
  have a': "aa_1  $\otimes$  aa_2  $\sim$  a'"
  by (simp add: a, fast intro: assoc_l_cancel[of ah _ a'])

note v1
also note a'
finally have "wfactors G (take i as' @ drop (Suc i) as') a'" by simp

from a'fs this carr
  have "essentially_equal G as (take i as' @ drop (Suc i) as')"
  by (intro ih[of a']) simp

hence ee1: "essentially_equal G (ah # as) (ah # take i as' @ drop (Suc
i) as')"
  apply (elim essentially_equalE) apply (fastsimp intro: essentially_equalI)
  done

from carr
  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'"
      apply (simp add: list_all2_append)
      apply (simp add: asiah[symmetric] ahcarr asicarr)
      done
  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')"
  apply (intro essentially_equalI)
  apply (subgoal_tac "as' ! i # take i as' @ drop (Suc i) as' <~~>
    take i as' @ as' ! i # drop (Suc i) as'")
  apply simp
  apply (rule perm_append_Cons)
  apply simp
  done
finally
  have "essentially_equal G (ah # as)
    (take i as' @ as' ! i # drop (Suc i) as')"

```

```

by simp

  thus "essentially_equal G (ah # as) as'" by (subst as', assumption)
qed

lemma (in primeness_condition_monoid) wfactors_unique:
  assumes "wfactors G as a" "wfactors G as' a"
    and "a ∈ carrier G" "set as ⊆ carrier G" "set as' ⊆ carrier G"
  shows "essentially_equal G as as'"
apply (rule wfactors_unique__hlp_induct[rule_format, of a])
apply (simp add: assms)
done

```

7.9.4 Application to factorial monoids

Number of factors for wellfoundedness

```

constdefs
  factorcount :: "_ ⇒ 'a ⇒ nat"
  "factorcount G a == THE c. (ALL as. set as ⊆ carrier G ∧
                                wfactors G as a ⟶ c = length as)"

lemma (in monoid) ee_length:
  assumes ee: "essentially_equal G as bs"
  shows "length as = length bs"
apply (rule essentially_equalE[OF ee])
apply (subgoal_tac "length as = length fs1'")
  apply (simp add: list_all2_lengthD)
  apply (simp add: perm_length)
done

lemma (in factorial_monoid) factorcount_exists:
  assumes carr[simp]: "a ∈ carrier G"
  shows "EX c. ALL as. set as ⊆ carrier G ∧ wfactors G as a ⟶ c = length
as"
proof -
  have "∃ as. set as ⊆ carrier G ∧ wfactors G as a" by (intro wfactors_exist,
simp)
  from this obtain as
    where ascarr[simp]: "set as ⊆ carrier G"
    and afs: "wfactors G as a"
    by (auto simp del: carr)

  have "ALL as'. set as' ⊆ carrier G ∧ wfactors G as' a ⟶ length as
= length as'"
  proof clarify
    fix as'
    assume [simp]: "set as' ⊆ carrier G"
    and bfs: "wfactors G as' a"
    from afs bfs

```

```

      have "essentially_equal G as as'"
      by (intro ee_wfactorsI[of a a as as'], simp+)
      thus "length as = length as'" by (rule ee_length)
    qed
    thus "EX c. ALL as'. set as'  $\subseteq$  carrier G  $\wedge$  wfactors G as' a  $\longrightarrow$  c =
length as'" ..
  qed

```

```

lemma (in factorial_monoid) factorcount_unique:
  assumes afs: "wfactors G as a"
  and acarr[simp]: "a  $\in$  carrier G" and ascarr[simp]: "set as  $\subseteq$  carrier
G"
  shows "factorcount G a = length as"
proof -
  have "EX ac. ALL as. set as  $\subseteq$  carrier G  $\wedge$  wfactors G as a  $\longrightarrow$  ac =
length as" by (rule factorcount_exists, simp)
  from this obtain ac where
    alen: "ALL as. set as  $\subseteq$  carrier G  $\wedge$  wfactors G as a  $\longrightarrow$  ac = length
as"
  by auto
  have ac: "ac = factorcount G a"
  apply (simp add: factorcount_def)
  apply (rule theI2)
  apply (rule alen)
  apply (elim allE[of _ "as"], rule allE[OF alen, of "as"], simp add:
ascarr afs)
  apply (elim allE[of _ "as"], rule allE[OF alen, of "as"], simp add:
ascarr afs)
  done

  from ascarr afs have "ac = length as" by (iprover intro: alen[rule_format])
  with ac show ?thesis by simp
qed

```

```

lemma (in factorial_monoid) divides_fcount:
  assumes dvd: "a divides b"
  and acarr: "a  $\in$  carrier G" and bcarr:"b  $\in$  carrier G"
  shows "factorcount G a  $\leq$  factorcount G b"
apply (rule dividesE[OF dvd])
proof -
  fix c
  from assms
    have " $\exists$  as. set as  $\subseteq$  carrier G  $\wedge$  wfactors G as a" by fast
  from this obtain as
    where ascarr: "set as  $\subseteq$  carrier G"
    and afs: "wfactors G as a"
    by auto
  with acarr have fca: "factorcount G a = length as" by (intro factorcount_unique)

```

```

assume ccarr: "c ∈ carrier G"
hence "∃ cs. set cs ⊆ carrier G ∧ wfactors G cs c" by fast
from this obtain cs
  where cscarr: "set cs ⊆ carrier G"
  and cfs: "wfactors G cs c"
  by auto

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+)
  with b have "wfactors G (as@cs) b" by simp
  hence "factorcount G b = length (as@cs)" by (intro factorcount_unique,
simp+)
  hence "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"
apply (rule associatedE[OF asc])
apply (drule divides_fcount[OF _ acarr bcarr])
apply (drule divides_fcount[OF _ bcarr acarr])
apply simp
done

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"
apply (rule properfactorE[OF pf], elim dividesE)
proof -
  fix c
  from assms
  have "∃ as. set as ⊆ carrier G ∧ wfactors G as a" by fast
  from this obtain as
    where ascarr: "set as ⊆ carrier G"
    and afs: "wfactors G as a"
    by auto
  with acarr have fca: "factorcount G a = length as" by (intro factorcount_unique)

  assume ccarr: "c ∈ carrier G"
  hence "∃ cs. set cs ⊆ carrier G ∧ wfactors G cs c" by fast
  from this obtain cs
    where cscarr: "set cs ⊆ carrier G"
    and cfs: "wfactors G cs c"

```

```

    by auto

  assume b: "b = a  $\otimes$  c"

  have "wfactors G (as@cs) (a  $\otimes$  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)
  hence fcb: "factorcount G b = length as + length cs" by simp

  assume nbdvda: " $\neg$  b divides a"
  have "c  $\notin$  Units G"
  proof (rule ccontr, simp)
    assume cunit: "c  $\in$  Units G"

    have "b  $\otimes$  inv c = a  $\otimes$  c  $\otimes$  inv c" by (simp add: b)
    also with ccarr acarr cunit
      have "... = a  $\otimes$  (c  $\otimes$  inv c)" by (fast intro: m_assoc)
    also with ccarr cunit
      have "... = a  $\otimes$  1" by (simp add: Units_r_inv)
    also with acarr
      have "... = a" by simp
    finally have "a = b  $\otimes$  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"
  apply -
  apply (rule ccontr, simp)
  proof -
    assume "wfactors G [] c"
    hence "1  $\sim$  c" by (elim wfactorsE, simp)
    with ccarr
      have cunit: "c  $\in$  Units G" by (intro assoc_unit_r[of "1" "c"],
    simp+)
    assume "c  $\notin$  Units G"
    with cunit show "False" by simp
  qed

  with fca fcb show ?thesis by simp
qed

sublocale factorial_monoid  $\subseteq$  divisor_chain_condition_monoid
apply unfold_locales
apply (rule wfUNIVI)
apply (rule measure_induct[of "factorcount G"])

```

```

apply simp
proof -
  fix P x
  assume r1[rule_format]:
    " $\forall y. (\forall z. z \in \text{carrier } G \wedge y \in \text{carrier } G \wedge \text{properfactor } G \ z$ 
 $y \longrightarrow P \ z) \longrightarrow P \ y$ "
    and r2[rule_format]: " $\forall y. \text{factorcount } G \ y < \text{factorcount } G \ x \longrightarrow P$ 
 $y$ "
  show "P x"
    apply (rule r1)
    apply (rule r2)
    apply (rule properfactor_fcount, simp+)
  done
qed

sublocale factorial_monoid  $\subseteq$  primeness_condition_monoid
proof qed (rule irreducible_is_prime)

lemma (in factorial_monoid) primeness_condition:
  shows "primeness_condition_monoid G"
  ..

lemma (in factorial_monoid) gcd_condition [simp]:
  shows "gcd_condition_monoid G"
  proof qed (rule gcdof_exists)

sublocale factorial_monoid  $\subseteq$  gcd_condition_monoid
proof qed (rule gcdof_exists)

lemma (in factorial_monoid) division_weak_lattice [simp]:
  shows "weak_lattice (division_rel G)"
proof -
  interpret weak_lower_semilattice "division_rel G" by simp

  show "weak_lattice (division_rel G)"
  apply (unfold_locales, simp_all)
  proof -
    fix x y
    assume carr: "x  $\in$  carrier G" "y  $\in$  carrier G"

    hence " $\exists z. z \in \text{carrier } G \wedge z \text{ lcmof } x \ y$ " by (rule lcmof_exists)
    from this obtain z
      where zcarr: "z  $\in$  carrier G"
      and isgcd: "z lcmof x y"
      by auto
    with carr
    have "least (division_rel G) z (Upper (division_rel G) {x, y})"
      by (simp add: lcmof_leastUpper[symmetric])
  qed

```



```

      thus "∃z. least (division_rel G) z (Upper (division_rel G) {x, y})"
    by fast
      qed
    qed

```

7.10 Factoriality Theorems

```

theorem factorial_condition_one:
  shows "(divisor_chain_condition_monoid G ∧ primeness_condition_monoid
G) =
      factorial_monoid G"
apply rule
proof 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 fm: "factorial_monoid G"
  interpret factorial_monoid "G" by (rule fm)
  show "divisor_chain_condition_monoid G ∧ primeness_condition_monoid
G"
    by rule unfold_locales
qed

theorem factorial_condition_two:
  shows "(divisor_chain_condition_monoid G ∧ gcd_condition_monoid G)
= factorial_monoid G"
apply rule
proof 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 fm: "factorial_monoid G"
  interpret factorial_monoid "G" by (rule fm)
  show "divisor_chain_condition_monoid G ∧ gcd_condition_monoid G"
    by rule unfold_locales
qed
end

```

```

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

```

8 The Algebraic Hierarchy of Rings

8.1 Abelian Groups

```

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

```

Derived operations.

```

constdefs (structure R)
  a_inv :: "['a, 'm) ring_scheme, 'a ] => 'a" ("⊖" [81] 80)
  "a_inv R == m_inv (| carrier = carrier R, mult = add R, one = zero R
  |)"

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

locale abelian_monoid =
  fixes G (structure)
  assumes a_comm_monoid:
    "comm_monoid (| carrier = carrier G, mult = add G, one = zero G |)"

```

The following definition is redundant but simple to use.

```

locale abelian_group = abelian_monoid +
  assumes a_comm_group:
    "comm_group (| carrier = carrier G, mult = add G, one = zero G |)"

```

8.2 Basic Properties

```

lemma abelian_monoidI:
  fixes R (structure)
  assumes a_closed:
    "!!x y. [| x ∈ carrier R; y ∈ carrier R |] ==> x ⊕ y ∈ carrier
  R"
  and zero_closed: "0 ∈ carrier R"
  and a_assoc:
    "!!x y z. [| x ∈ carrier R; y ∈ carrier R; z ∈ carrier R |] ==>
    (x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)"
  and l_zero: "!!x. x ∈ carrier R ==> 0 ⊕ x = x"
  and a_comm:
    "!!x y. [| x ∈ carrier R; y ∈ carrier R |] ==> x ⊕ y = y ⊕ x"
  shows "abelian_monoid R"
  by (auto intro!: abelian_monoid.intro comm_monoidI intro: assms)

```

```

lemma abelian_groupI:
  fixes R (structure)
  assumes a_closed:
    "!!x y. [| x ∈ carrier R; y ∈ carrier R |] ==> x ⊕ y ∈ carrier
R"
    and zero_closed: "zero R ∈ carrier R"
    and a_assoc:
      "!!x y z. [| x ∈ carrier R; y ∈ carrier R; z ∈ carrier R |] ==>
(x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)"
    and a_comm:
      "!!x y. [| x ∈ carrier R; y ∈ carrier R |] ==> x ⊕ y = y ⊕ x"
    and l_zero: "!!x. x ∈ carrier R ==> 0 ⊕ x = x"
    and l_inv_ex: "!!x. x ∈ carrier R ==> EX y : carrier R. y ⊕ x = 0"
  shows "abelian_group R"
  by (auto intro!: abelian_group.intro abelian_monoidI
    abelian_group_axioms.intro comm_monoidI comm_groupI
    intro: assms)

lemma (in abelian_monoid) a_monoid:
  "monoid (| carrier = carrier G, mult = add G, one = zero G |)"
  by (rule comm_monoid.axioms, rule a_comm_monoid)

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

lemma (in abelian_monoid) a_closed [intro, simp]:
  "[| x ∈ carrier G; y ∈ carrier G |] ==> x ⊕ y ∈ carrier G"
  by (rule monoid.m_closed [OF a_monoid, simplified monoid_record_simps])

lemma (in abelian_monoid) zero_closed [intro, simp]:
  "0 ∈ carrier G"
  by (rule monoid.one_closed [OF a_monoid, simplified monoid_record_simps])

lemma (in abelian_group) a_inv_closed [intro, simp]:
  "x ∈ carrier G ==> ⊖ x ∈ carrier G"
  by (simp add: a_inv_def
    group.inv_closed [OF a_group, simplified monoid_record_simps])

lemma (in abelian_group) minus_closed [intro, simp]:
  "[| x ∈ carrier G; y ∈ carrier G |] ==> x ⊖ y ∈ carrier G"
  by (simp add: a_minus_def)

lemma (in abelian_group) a_l_cancel [simp]:
  "[| x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |] ==>

```

```

    (x ⊕ y = x ⊕ z) = (y = z)"
  by (rule group.l_cancel [OF a_group, simplified monoid_record_simps])

lemma (in abelian_group) a_r_cancel [simp]:
  "[| x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |] ==>
    (y ⊕ x = z ⊕ x) = (y = z)"
  by (rule group.r_cancel [OF a_group, simplified monoid_record_simps])

lemma (in abelian_monoid) a_assoc:
  "[| x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |] ==>
    (x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)"
  by (rule monoid.m_assoc [OF a_monoid, simplified monoid_record_simps])

lemma (in abelian_monoid) l_zero [simp]:
  "x ∈ carrier G ==> 0 ⊕ x = x"
  by (rule monoid.l_one [OF a_monoid, simplified monoid_record_simps])

lemma (in abelian_group) l_neg:
  "x ∈ carrier G ==> ⊖ x ⊕ x = 0"
  by (simp add: a_inv_def
    group.l_inv [OF a_group, simplified monoid_record_simps])

lemma (in abelian_monoid) a_comm:
  "[| x ∈ carrier G; y ∈ carrier G |] ==> x ⊕ y = y ⊕ x"
  by (rule comm_monoid.m_comm [OF a_comm_monoid,
    simplified monoid_record_simps])

lemma (in abelian_monoid) a_lcomm:
  "[| x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |] ==>
    x ⊕ (y ⊕ z) = y ⊕ (x ⊕ z)"
  by (rule comm_monoid.m_lcomm [OF a_comm_monoid,
    simplified monoid_record_simps])

lemma (in abelian_monoid) r_zero [simp]:
  "x ∈ carrier G ==> x ⊕ 0 = x"
  using monoid.r_one [OF a_monoid]
  by simp

lemma (in abelian_group) r_neg:
  "x ∈ carrier G ==> x ⊕ (⊖ x) = 0"
  using group.r_inv [OF a_group]
  by (simp add: a_inv_def)

lemma (in abelian_group) minus_zero [simp]:
  "⊖ 0 = 0"
  by (simp add: a_inv_def
    group.inv_one [OF a_group, simplified monoid_record_simps])

lemma (in abelian_group) minus_minus [simp]:

```

```

"x ∈ carrier G ==> ⊖ (⊖ x) = x"
using group.inv_inv [OF a_group, simplified monoid_record_simps]
by (simp add: a_inv_def)

lemma (in abelian_group) a_inv_inj:
  "inj_on (a_inv G) (carrier G)"
  using group.inv_inj [OF a_group, simplified monoid_record_simps]
  by (simp add: a_inv_def)

lemma (in abelian_group) minus_add:
  "[| x ∈ carrier G; y ∈ carrier G |] ==> ⊖ (x ⊕ y) = ⊖ x ⊕ ⊖ y"
  using comm_group.inv_mult [OF a_comm_group]
  by (simp add: a_inv_def)

lemma (in abelian_group) minus_equality:
  "[| x ∈ carrier G; y ∈ carrier G; y ⊕ x = 0 |] ==> ⊖ x = y"
  using group.inv_equality [OF a_group]
  by (auto simp add: a_inv_def)

lemma (in abelian_monoid) minus_unique:
  "[| x ∈ carrier G; y ∈ carrier G; y' ∈ carrier G;
    y ⊕ x = 0; x ⊕ y' = 0 |] ==> y = y'"
  using monoid.inv_unique [OF a_monoid]
  by (simp add: a_inv_def)

lemmas (in abelian_monoid) a_ac = a_assoc a_comm a_lcomm

Derive an abelian_group from a comm_group

lemma comm_group_abelian_groupI:
  fixes G (structure)
  assumes cg: "comm_group (|carrier = carrier G, mult = add G, one = zero
G|)"
  shows "abelian_group G"
proof -
  interpret comm_group "(|carrier = carrier G, mult = add G, one = zero
G|)"
  by (rule cg)
  show "abelian_group G" ..
qed

```

8.3 Sums over Finite Sets

This definition makes it easy to lift lemmas from `finprod`.

```

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

syntax

```

```

    "_finsum" :: "index => idt => 'a set => 'b => 'b"
      ("( $3\oplus_{--:_.}$  _)" [1000, 0, 51, 10] 10)
syntax (xsymbols)
    "_finsum" :: "index => idt => 'a set => 'b => 'b"
      ("( $3\oplus_{--\in_.}$  _)" [1000, 0, 51, 10] 10)
syntax (HTML output)
    "_finsum" :: "index => idt => 'a set => 'b => 'b"
      ("( $3\oplus_{--\in_.}$  _)" [1000, 0, 51, 10] 10)
translations
    " $\oplus_{i:A.}$  b" == "finsum  $\diamond_{i.}$  b) A"
    — Beware of argument permutation!

context abelian_monoid begin

lemma finsum_empty [simp]:
  "finsum G f {} = 0"
  by (rule comm_monoid.finprod_empty [OF a_comm_monoid,
    folded finsum_def, simplified monoid_record_simps])

lemma finsum_insert [simp]:
  "[| finite F; a  $\notin$  F; f  $\in$  F -> carrier G; f a  $\in$  carrier G |]
  ==> finsum G f (insert a F) = f a  $\oplus$  finsum G f F"
  by (rule comm_monoid.finprod_insert [OF a_comm_monoid,
    folded finsum_def, simplified monoid_record_simps])

lemma finsum_zero [simp]:
  "finite A ==> ( $\oplus_{i\in A.}$  0) = 0"
  by (rule comm_monoid.finprod_one [OF a_comm_monoid, folded finsum_def,
    simplified monoid_record_simps])

lemma finsum_closed [simp]:
  fixes A
  assumes fin: "finite A" and f: "f  $\in$  A -> carrier G"
  shows "finsum G f A  $\in$  carrier G"
  apply (rule comm_monoid.finprod_closed [OF a_comm_monoid,
    folded finsum_def, simplified monoid_record_simps])
  apply (rule fin)
  apply (rule f)
  done

lemma finsum_Un_Int:
  "[| finite A; finite B; g  $\in$  A -> carrier G; g  $\in$  B -> carrier G |] ==>
    finsum G g (A Un B)  $\oplus$  finsum G g (A Int B) =
    finsum G g A  $\oplus$  finsum G g B"
  by (rule comm_monoid.finprod_Un_Int [OF a_comm_monoid,
    folded finsum_def, simplified monoid_record_simps])

```

```

lemma finsum_Un_disjoint:
  "[| finite A; finite B; A Int B = {};
    g ∈ A -> carrier G; g ∈ B -> carrier G |]
  ==> finsum G g (A Un B) = finsum G g A ⊕ finsum G g B"
  by (rule comm_monoid.finprod_Un_disjoint [OF a_comm_monoid,
    folded finsum_def, simplified monoid_record_simps])

lemma finsum_addf:
  "[| finite A; f ∈ A -> carrier G; g ∈ A -> carrier G |] ==>
  finsum G (%x. f x ⊕ g x) A = (finsum G f A ⊕ finsum G g A)"
  by (rule comm_monoid.finprod_multf [OF a_comm_monoid,
    folded finsum_def, simplified monoid_record_simps])

lemma finsum_cong':
  "[| A = B; g : B -> carrier G;
    !!i. i : B ==> f i = g i |] ==> finsum G f A = finsum G g B"
  by (rule comm_monoid.finprod_cong' [OF a_comm_monoid,
    folded finsum_def, simplified monoid_record_simps]) auto

lemma finsum_0 [simp]:
  "f : {0::nat} -> carrier G ==> finsum G f {..0} = f 0"
  by (rule comm_monoid.finprod_0 [OF a_comm_monoid, folded finsum_def,
    simplified monoid_record_simps])

lemma finsum_Suc [simp]:
  "f : {..Suc n} -> carrier G ==>
  finsum G f {..Suc n} = (f (Suc n) ⊕ finsum G f {..n})"
  by (rule comm_monoid.finprod_Suc [OF a_comm_monoid, folded finsum_def,
    simplified monoid_record_simps])

lemma finsum_Suc2:
  "f : {..Suc n} -> carrier G ==>
  finsum G f {..Suc n} = (finsum G (%i. f (Suc i)) {..n} ⊕ f 0)"
  by (rule comm_monoid.finprod_Suc2 [OF a_comm_monoid, folded finsum_def,
    simplified monoid_record_simps])

lemma finsum_add [simp]:
  "[| f : {..n} -> carrier G; g : {..n} -> carrier G |] ==>
  finsum G (%i. f i ⊕ g i) {..n::nat} =
  finsum G f {..n} ⊕ finsum G g {..n}"
  by (rule comm_monoid.finprod_mult [OF a_comm_monoid, folded finsum_def,
    simplified monoid_record_simps])

lemma finsum_cong:
  "[| A = B; f : B -> carrier G;
    !!i. i : B ==simp=> f i = g i |] ==> finsum G f A = finsum G g B"
  by (rule comm_monoid.finprod_cong [OF a_comm_monoid, folded finsum_def,
    simplified monoid_record_simps]) (auto simp add: simp_implies_def)

```

Usually, if this rule causes a failed congruence proof error, the reason is that

the premise $g \in B \rightarrow \text{carrier } G$ cannot be shown. Adding Pi_def to the simpset is often useful.

```
lemma finsum_reindex:
  assumes fin: "finite A"
  shows "f : (h ' A) → carrier G ==>
    inj_on h A ==> finsum G f (h ' A) = finsum G (%x. f (h x)) A"
  using fin apply induct
  apply (auto simp add: finsum_insert Pi_def)
done
```

```
lemma finsum_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 0) = f i"
  using i_in_A finsum_insert [of "A - {i}" i "(λj. if i = j then f j else
  0)"]
  fin_A f_Pi finsum_zero [of "A - {i}"]
  finsum_cong [of "A - {i}" "A - {i}" "(λj. if i = j then f j else 0)"
  "(λi. 0)"]
  unfolding Pi_def simp_implies_def by (force simp add: insert_absorb)
end
```

8.4 Rings: Basic Definitions

```
locale ring = abelian_group R + monoid R for R (structure) +
  assumes l_distr: "[| x ∈ carrier R; y ∈ carrier R; z ∈ carrier R |]
    ==> (x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
  and r_distr: "[| x ∈ carrier R; y ∈ carrier R; z ∈ carrier R |]
    ==> z ⊗ (x ⊕ y) = z ⊗ x ⊕ z ⊗ y"

locale cring = ring + comm_monoid R

locale "domain" = cring +
  assumes one_not_zero [simp]: "1 ~= 0"
  and integral: "[| a ⊗ b = 0; a ∈ carrier R; b ∈ carrier R |] ==>
    a = 0 | b = 0"

locale field = "domain" +
  assumes field_Units: "Units R = carrier R - {0}"
```

8.5 Rings

```
lemma ringI:
  fixes R (structure)
```



```

    assumes abelian_group: "abelian_group R"
    and monoid: "monoid R"
    and l_distr: "!!x y z. [| x ∈ carrier R; y ∈ carrier R; z ∈ carrier
R |]
    ==> (x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
    and r_distr: "!!x y z. [| x ∈ carrier R; y ∈ carrier R; z ∈ carrier
R |]
    ==> z ⊗ (x ⊕ y) = z ⊗ x ⊕ z ⊗ y"
  shows "ring R"
  by (auto intro: ring.intro
    abelian_group.axioms ring_axioms.intro assms)

lemma (in ring) is_abelian_group:
  "abelian_group R"
..

lemma (in ring) is_monoid:
  "monoid R"
  by (auto intro!: monoidI m_assoc)

lemma (in ring) is_ring:
  "ring R"
  by (rule ring_axioms)

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  $\otimes$  (x  $\oplus$  y) = z  $\otimes$  x  $\oplus$  z  $\otimes$  y" .
  qed (rule l_distr)
qed (auto intro: cring.intro
    abelian_group.axioms comm_monoid.axioms ring_axioms.intro assms)

```

```

lemma (in cring) is_cring:
  "cring R" by (rule cring_axioms)

```

8.5.1 Normaliser for Rings

```

lemma (in abelian_group) r_neg2:
  "[| x  $\in$  carrier G; y  $\in$  carrier G |] ==> x  $\oplus$  ( $\ominus$  x  $\oplus$  y) = y"
proof -
  assume G: "x  $\in$  carrier G" "y  $\in$  carrier G"
  then have "(x  $\oplus$   $\ominus$  x)  $\oplus$  y = y"
    by (simp only: r_neg l_zero)
  with G show ?thesis
    by (simp add: a_ac)
qed

```

```

lemma (in abelian_group) r_neg1:
  "[| x  $\in$  carrier G; y  $\in$  carrier G |] ==>  $\ominus$  x  $\oplus$  (x  $\oplus$  y) = y"
proof -
  assume G: "x  $\in$  carrier G" "y  $\in$  carrier G"
  then have "( $\ominus$  x  $\oplus$  x)  $\oplus$  y = y"
    by (simp only: l_neg l_zero)
  with G show ?thesis by (simp add: a_ac)
qed

```

The following proofs are from Jacobson, Basic Algebra I, pp. 88–89

```

lemma (in ring) l_null [simp]:
  "x  $\in$  carrier R ==> 0  $\otimes$  x = 0"
proof -
  assume R: "x  $\in$  carrier R"
  then have "0  $\otimes$  x  $\oplus$  0  $\otimes$  x = (0  $\oplus$  0)  $\otimes$  x"
    by (simp add: l_distr del: l_zero r_zero)
  also from R have "... = 0  $\otimes$  x  $\oplus$  0" by simp
  finally have "0  $\otimes$  x  $\oplus$  0  $\otimes$  x = 0  $\otimes$  x  $\oplus$  0" .
  with R show ?thesis by (simp del: r_zero)
qed

```

```

lemma (in ring) r_null [simp]:
  "x  $\in$  carrier R ==> x  $\otimes$  0 = 0"
proof -
  assume R: "x  $\in$  carrier R"
  then have "x  $\otimes$  0  $\oplus$  x  $\otimes$  0 = x  $\otimes$  (0  $\oplus$  0)"
    by (simp add: r_distr 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 ?thesis by (simp del: r_zero)
qed

lemma (in ring) 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 l_null)
  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 (in ring) 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 r_null)
  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 (in abelian_group) minus_eq:
  "[| x ∈ carrier G; y ∈ carrier G |] ==> x ⊖ y = x ⊕ ⊖ y"
  by (simp only: a_minus_def)

Setup algebra method: compute distributive normal form in locale contexts
use "ringsimp.ML"

setup Algebra.setup

lemmas (in ring) ring_simplrules
  [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 cring) nat_pow_zero:
  "(n::nat) ~= 0 ==> 0 (^) n = 0"
  by (induct n) simp_all

lemma (in ring) 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
  from this and onezero
    have "x = x ⊗ 0" by simp
  from this and xcarr
    have "x = 0" by simp
  thus "x ∈ {0}" by fast
qed fast

lemma (in ring) one_zeroI:
  assumes carrzero: "carrier R = {0}"
  shows "1 = 0"
proof -
  from one_closed and carrzero
    show "1 = 0" by simp
qed

lemma (in ring) carrier_one_zero:
  shows "(carrier R = {0}) = (1 = 0)"
  by (rule, erule one_zeroI, erule one_zeroD)

lemma (in ring) carrier_one_not_zero:
  shows "(carrier R ≠ {0}) = (1 ≠ 0)"
  by (simp add: carrier_one_zero)

```

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
ML_val {* Algebra.print_structures @{context} *}
  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

```

8.5.2 Sums over Finite Sets

```

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

```

8.6 Integral Domains

```

lemma (in "domain") zero_not_one [simp]:
  "0 ≠ 1"
  by (rule not_sym) simp

lemma (in "domain") 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 (in "domain") 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 (in "domain") 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

```

8.7 Fields

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

```

lemma (in cring) cring_fieldI:
  assumes field_Units: "Units R = carrier R - {0}"
  shows "field R"
proof
  from field_Units
  have a: "0 ∉ Units R" by fast
  have "1 ∈ Units R" by fast
  from this and a
  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"
  from this and 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 add: Units_l_inv)
  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" .
  thus "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"
apply (rule cring_fieldI, simp add: Units_def)
apply (rule, clarsimp)
apply (simp add: notzero)
proof (clarsimp)
  fix x
  assume xcarr: "x ∈ carrier R"
  and "x ≠ 0"
  from this
  have "∃y∈carrier R. x ⊗ y = 1" by (rule invex)
  from this
  obtain y
  where ycarr: "y ∈ carrier R"
  and xy: "x ⊗ y = 1"
  by fast
  from xy xcarr ycarr have "y ⊗ x = 1" by (simp add: m_comm)
  from ycarr and this and xy
  show "∃y∈carrier R. y ⊗ x = 1 ∧ x ⊗ y = 1" by fast
qed

```

8.8 Morphisms

```

constdefs (structure R S)
  ring_hom :: "[('a, 'm) ring_scheme, ('b, 'n) ring_scheme] => ('a =>
'b) set"
  "ring_hom R S == {h. h ∈ carrier R -> carrier S &
    (ALL x y. x ∈ carrier R & y ∈ carrier R -->
      h (x ⊗ y) = h x ⊗S h y & h (x ⊕ y) = h x ⊕S h y) &
    h 1 = 1S}"

lemma ring_hom_memI:
  fixes R (structure) and S (structure)
  assumes hom_closed: "!!x. x ∈ carrier R ==> h x ∈ carrier S"
  and hom_mult: "!!x y. [| x ∈ carrier R; y ∈ carrier R |] ==>
    h (x ⊗ y) = h x ⊗S h y"
  and hom_add: "!!x y. [| x ∈ carrier R; y ∈ carrier R |] ==>
    h (x ⊕ y) = h x ⊕S h y"
  and hom_one: "h 1 = 1S"
  shows "h ∈ ring_hom R S"
  by (auto simp add: ring_hom_def assms Pi_def)

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)

locale ring_hom_cring = R: cring R + S: cring S
  for R (structure) and S (structure) +
  fixes h
  assumes homh [simp, intro]: "h ∈ ring_hom R S"
  notes hom_closed [simp, intro] = ring_hom_closed [OF homh]
  and hom_mult [simp] = ring_hom_mult [OF homh]

```



```

    and hom_add [simp] = ring_hom_add [OF homh]
    and hom_one [simp] = ring_hom_one [OF homh]

lemma (in ring_hom_cring) hom_zero [simp]:
  "h 0 = 0S"
proof -
  have "h 0 ⊕S h 0 = h 0 ⊕S 0S"
    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]:
  "[| finite A; f ∈ A -> carrier R |] ==>
   h (finsum R f A) = finsum S (h o f) A"
proof (induct set: finite)
  case empty then show ?case by simp
next
  case insert then show ?case by (simp add: Pi_def)
qed

lemma (in ring_hom_cring) hom_finprod:
  "[| finite A; f ∈ A -> carrier R |] ==>
   h (finprod R f A) = finprod S (h o f) A"
proof (induct set: finite)
  case empty then show ?case by simp
next
  case insert then show ?case by (simp add: Pi_def)
qed

declare ring_hom_cring.hom_finprod [simp]

lemma id_ring_hom [simp]:
  "id ∈ ring_hom R R"
  by (auto intro!: ring_hom_memI)

end

theory AbelCoset

```

```
imports Coset Ring
begin
```

8.9 More Lifting from Groups to Abelian Groups

8.9.1 Definitions

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

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

```
constdefs (structure G)
  a_r_coset    :: "[_, 'a set, 'a]  $\Rightarrow$  'a set"    (infixl "<+>_r" 60)
  "a_r_coset G  $\equiv$  r_coset ( $\downarrow$ carrier = carrier G, mult = add G, one = zero
  G)"

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

  A_RCOSETS    :: "[_, 'a set]  $\Rightarrow$  ('a set)set"    ("a'_rcosets" [81] 80)
  "A_RCOSETS G H  $\equiv$  RCOSETS ( $\downarrow$ carrier = carrier G, mult = add G, one = zero
  G) H"

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

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

constdefs (structure G)
  a_r_congruent :: "[('a,'b) ring_scheme, 'a set]  $\Rightarrow$  ('a*'a)set"
                  ("racong" [81] 80)
  "a_r_congruent G  $\equiv$  r_congruent ( $\downarrow$ carrier = carrier G, mult = add G,
  one = zero G)"

constdefs
  A_FactGroup :: "[('a,'b) ring_scheme, 'a set]  $\Rightarrow$  ('a set) monoid"
                (infixl "A'_Mod" 65)
  — Actually defined for groups rather than monoids
  "A_FactGroup G H  $\equiv$  FactGroup ( $\downarrow$ carrier = carrier G, mult = add G, one
  = zero G) H"

constdefs
  a_kernel :: "('a, 'm) ring_scheme  $\Rightarrow$  ('b, 'n) ring_scheme  $\Rightarrow$ 
              ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a set"
  — the kernel of a homomorphism (additive)
  "a_kernel G H h  $\equiv$  kernel ( $\downarrow$ carrier = carrier G, mult = add G, one = zero
```

```

G|)
                                     (| carrier = carrier H, mult = add H, one
= zero H|) h"

locale abelian_group_hom = G: abelian_group G + H: abelian_group H
  for G (structure) and H (structure) +
  fixes h
  assumes a_group_hom: "group_hom (| carrier = carrier G, mult = add
G, one = zero G |)
                                     (| carrier = carrier H, mult = add H,
one = zero H |) h"

lemmas a_r_coset_defs =
  a_r_coset_def r_coset_def

lemma a_r_coset_def':
  fixes G (structure)
  shows "H >+ a  $\equiv \bigcup_{h \in H}. \{h \oplus 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  $\equiv \bigcup_{h \in H}. \{a \oplus 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  $\equiv \bigcup_{a \in \text{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  $\equiv \bigcup_{h \in H}. \bigcup_{k \in K}. \{h \oplus 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  $\equiv \bigcup_{h \in H}. \{\ominus h\}$ "
  unfolding A_SET_INV_defs
  by (fold a_inv_def)
```

8.9.2 Cosets

```
lemma (in abelian_group) a_coset_add_assoc:
  "[| M  $\subseteq$  carrier G; g  $\in$  carrier G; h  $\in$  carrier G |]
   ==> (M +> g) +> h = M +> (g  $\oplus$  h)"
  by (rule group.coset_mult_assoc [OF a_group,
    folded a_r_coset_def, simplified monoid_record_simps])
```

```
lemma (in abelian_group) a_coset_add_zero [simp]:
  "M  $\subseteq$  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  $\oplus$  ( $\ominus$  y)) = M; x  $\in$  carrier G ; y  $\in$  carrier G;
   M  $\subseteq$  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  $\in$  carrier G; y  $\in$  carrier G; M  $\subseteq$  carrier
   G |]
   ==> M +> (x  $\oplus$  ( $\ominus$  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  $\in$  carrier G; subgroup H (|carrier = carrier G,
   mult = add G, one = zero G|) |] ==> x  $\in$  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 (|carrier = carrier G, mult = add G, one = zero G|);
   x  $\in$  H; y  $\in$  H|] ==>  $\exists h \in H. y = h \oplus x$ "
  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  $\in$  H +> x; x  $\in$  carrier G; subgroup H (|carrier = carrier G, mult
   = add G, one = zero G|) |] ==> H +> x = H +> y"
```

```

by (rule group.repr_independence [OF a_group,
    folded a_r_coset_def, simplified monoid_record_simps])

lemma (in abelian_group) a_coset_join2:
  "[x ∈ carrier G; subgroup H (carrier = carrier G, mult = add G,
  one = zero G); x ∈ H] ==> H +> x = H"
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_RCOSSETS_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"
by (rule group.transpose_inv [OF a_group,
    folded a_r_coset_def a_inv_def, simplified monoid_record_simps])

```

8.9.3 Subgroups

```

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

```

```

lemma (in additive_subgroup) is_additive_subgroup:
  shows "additive_subgroup H G"
by (rule additive_subgroup_axioms)

```

```

lemma additive_subgroupI:
  fixes G (structure)
  assumes a_subgroup: "subgroup H (carrier = carrier G, mult = add G,
  one = zero G)"
  shows "additive_subgroup H G"
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])

```

8.9.4 Additive subgroups are normal

Every subgroup of an `abelian_group` is normal

```

locale abelian_subgroup = additive_subgroup + abelian_group G +
  assumes a_normal: "normal H (|carrier = carrier G, mult = add G, one
= zero G)"

```

```

lemma (in abelian_subgroup) is_abelian_subgroup:
  shows "abelian_subgroup H G"
by (rule abelian_subgroup_axioms)

```

```

lemma abelian_subgroupI:
  assumes a_normal: "normal H (|carrier = carrier G, mult = add G, one
= zero G)"
  and a_comm: "!!x y. [| x ∈ carrier G; y ∈ carrier G |] ==> x ⊕G
y = y ⊕G x"
  shows "abelian_subgroup H G"
proof -
  interpret normal "H" "(|carrier = carrier G, mult = add G, one = zero
G)"
  by (rule a_normal)

  show "abelian_subgroup H G"
  proof qed (simp add: a_comm)
qed

```

```

lemma abelian_subgroupI2:
  fixes G (structure)
  assumes a_comm_group: "comm_group (|carrier = carrier G, mult = add

```

```

G, one = zero G)
    and a_subgroup: "subgroup H (carrier = carrier G, mult = add G,
one = zero G)"
    shows "abelian_subgroup H G"
proof -
    interpret comm_group "(carrier = carrier G, mult = add G, one = zero
G)"
    by (rule a_comm_group)
    interpret subgroup "H" "(carrier = carrier G, mult = add G, one = zero
G)"
    by (rule a_subgroup)

    show "abelian_subgroup H G"
    apply unfold_locales
    proof (simp add: r_coset_def l_coset_def, clarsimp)
        fix x
        assume xcarr: "x ∈ carrier G"
        from a_subgroup
        have Hcarr: "H ⊆ carrier G" by (unfold subgroup_def, simp)
        from xcarr Hcarr
        show "(⋃ h ∈ H. {h ⊕G x}) = (⋃ h ∈ H. {x ⊕G h})"
        using m_comm[simplified]
        by fast
    qed
qed

lemma abelian_subgroupI3:
  fixes G (structure)
  assumes asg: "additive_subgroup H G"
  and ag: "abelian_group G"
  shows "abelian_subgroup H G"
apply (rule abelian_subgroupI2)
apply (rule abelian_group.a_comm_group[OF ag])
apply (rule additive_subgroup.a_subgroup[OF asg])
done

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]])

Alternative characterization of normal subgroups

```
lemma (in abelian_group) a_normal_inv_iff:
  "(N <| carrier = carrier G, mult = add G, one = zero G |) =
    (subgroup N (carrier = carrier G, mult = add G, one = zero G) &
    (∀ x ∈ carrier G. ∀ h ∈ N. x ⊕ h ⊕ (⊖ x) ∈ N))"
  (is "_ = ?rhs")
```

```
by (rule group.normal_inv_iff [OF a_group,
  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 (carrier = carrier G, mult
  = add G, one = zero 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 (carrier = carrier G,
  mult = add G, one = zero 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: "y ∈ x <+ H" and x: "x ∈ carrier G" and sb: "subgroup H
  (carrier = carrier G, mult = add G, one = zero G)"
  shows "y <+ H ⊆ x <+ H"
apply (rule group.l_repr_imp_subset [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) a_l_repr_independence:
  assumes y: "y ∈ x <+ H" and x: "x ∈ carrier G" and sb: "subgroup H
  (carrier = carrier G, mult = add G, one = zero G)"
  shows "x <+ H = y <+ H"
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 (carrier = carrier
G, mult = add G, one = zero 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_RCOSSETS_def, simplified monoid_record_simps])

```

8.9.5 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:
  shows "[a ∈ a_rcosets H; b ∈ a_rcosets H; a ≠ b] ⇒ a ∩ b = {}"
by (rule group.rcos_disjoint [OF a_group a_subgroup,
  folded A_RCOSSETS_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_RCOSSETS_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_RCOSSETS_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 (rule group.card_cosets_equal [OF a_group,
  folded A_RCOSSETS_def, simplified monoid_record_simps])

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_RCOSSETS_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_RCOSSETS_def, simplified monoid_record_simps order_def, folded
order_def])
  (fast intro!: additive_subgroup.a_subgroup)+

```

8.9.6 Factorization

```

lemmas A_FactGroup_defs = A_FactGroup_def FactGroup_def

```

```

lemma A_FactGroup_def':
  fixes G (structure)
  shows "G A_Mod H ≡ (carrier = a_rcosetsg H, mult = set_add G, one =
H)"
unfolding A_FactGroup_defs
by (fold A_RCOSSETS_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_RCOSSETS_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_RCOSSETS_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_RCOSSETS_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_RCOSSETS_def, simplified monoid_record_simps])

lemma (in abelian_subgroup) a_rcosets_inv_mult_group_eq:
  "M ∈ a_rcosets H  $\implies$  a_set_inv M <+> M = H"
  by (rule normal.rcosets_inv_mult_group_eq [OF a_normal,
    folded A_RCOSSETS_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, it results in a commutative group

```

theorem (in abelian_subgroup) a_factorgroup_is_comm_group:
  "comm_group (G A_Mod H)"
  apply (intro comm_group.intro comm_monoid.intro) prefer 3
    apply (rule a_factorgroup_is_group)
    apply (rule group.axioms[OF a_factorgroup_is_group])
  apply (rule comm_monoid_axioms.intro)
  apply (unfold A_FactGroup_def FactGroup_def RCOSSETS_def, fold set_add_def
    a_r_coset_def, clarsimp)
  apply (simp add: a_rcos_sum a_comm)
  done

```

```

lemma add_A_FactGroup [simp]: "X  $\otimes_{(G \text{ A\_Mod } H)}$  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)  $\implies$  inv_G 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 $G \text{ Mod } H$

```

lemma (in abelian_subgroup) a_r_coset_hom_A_Mod:
  "( $\lambda a. H +> a$ ) ∈ hom ((carrier = carrier G, mult = add G, one = zero 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

8.9.7 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  $\equiv$  {x  $\in$  carrier R. h x = 0S}"
by (rule a_kernel_def[unfolded kernel_def, simplified ring_record_simps])

```

8.9.8 Homomorphisms

```

lemma abelian_group_homI:
  assumes "abelian_group G"
  assumes "abelian_group H"
  assumes a_group_hom: "group_hom (| carrier = carrier G, mult = add
G, one = zero G |)
                                (| carrier = carrier H, mult = add H,
one = zero H |) h"
  shows "abelian_group_hom G H h"
proof -
  interpret G: abelian_group G by fact
  interpret H: abelian_group H by fact
  show ?thesis apply (intro abelian_group_hom.intro abelian_group_hom_axioms.intro)
    apply fact
    apply fact
    apply (rule a_group_hom)
    done
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  $\oplus_G$  y) = h x  $\oplus_H$  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  $\in$  carrier G  $\implies$  h x  $\in$  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  $\in$  carrier H"
by (rule group_hom.one_closed[OF a_group_hom,
  simplified ring_record_simps])

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 (rule group_hom.inv_closed[OF a_group_hom,
    folded a_inv_def, simplified ring_record_simps])
```

```
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"
apply (rule additive_subgroup.intro)
apply (rule group_hom.subgroup_kernel[OF a_group_hom,
    folded a_kernel_def, simplified ring_record_simps])
done
```

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 (rule group_hom.normal_kernel[OF a_group_hom,
    folded a_kernel_def, simplified ring_record_simps])
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_contents_mem:
  assumes X: "X ∈ carrier (G A_Mod (a_kernel G H h))"
  shows "contents (h'X) ∈ carrier H"
by (rule group_hom.FactGroup_contents_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. contents (h'X)) ∈ hom (G A_Mod (a_kernel G H h))
    (carrier = carrier H, mult = add H, one = zero 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 ( $\lambda X.$  contents ( $h \text{ ' } X$ )) ( $\text{carrier } (G \text{ A\_Mod } a\_kernel \text{ } G \text{ } H \text{ } h)$ )"
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: " $h \text{ ' } \text{carrier } G = \text{carrier } H$ "
  shows " $(\lambda X.$  contents ( $h \text{ ' } X$ )) '  $\text{carrier } (G \text{ A\_Mod } a\_kernel \text{ } G \text{ } H \text{ } h) =$ 
 $\text{carrier } H$ "
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 $G \text{ Mod } \text{kernel } G \text{ } H \text{ } h$ is isomorphic to H .

```

theorem (in abelian_group_hom) A_FactGroup_iso:
  " $h \text{ ' } \text{carrier } G = \text{carrier } H$ "
 $\implies (\lambda X.$  contents ( $h \text{ ' } X$ ))  $\in (G \text{ A\_Mod } (a\_kernel \text{ } G \text{ } H \text{ } h)) \cong$ 
  ( $| \text{carrier} = \text{carrier } H, \text{mult} = \text{add } H, \text{one} = \text{zero } H \text{ } |$ )"
by (rule group_hom.FactGroup_iso[OF a_group_hom,
  folded a_kernel_def A_FactGroup_def, simplified ring_record_simps])

```

8.9.9 Cosets

Not everything from `CosetExt.thy` is lifted here.

```

lemma (in additive_subgroup) a_Hcarr [simp]:
  assumes hH: " $h \in H$ "
  shows " $h \in \text{carrier } G$ "
by (rule subgroup.mem_carrier [OF a_subgroup,
  simplified monoid_record_simps]) (rule hH)

```

```

lemma (in abelian_subgroup) a_elemtcos_carrier:
  assumes acarr: " $a \in \text{carrier } G$ "
  and a': " $a' \in H \implies a$ "
  shows " $a' \in \text{carrier } G$ "
by (rule subgroup.elemtcos_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: " $h \in H$ "
  shows " $H \implies h = H$ "
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: "y ∈ H +> x"
    and xcarr: "x ∈ carrier G"
  shows "H +> x = H +> y"
  apply (rule a_repr_independence)
  apply (rule y)
  apply (rule xcarr)
  apply (rule a_subgroup)
  done

```

```

lemma (in abelian_subgroup) a_repr_independenceD:
  assumes ycarr: "y ∈ carrier G"

```



```

    and repr: "H +> x = H +> y"
  shows "y ∈ H +> x"
by (rule group.repr_independenceD [OF a_group a_subgroup,
    folded a_r_coset_def, simplified monoid_record_simps]) (rule ycarr,
rule repr)

```

```

lemma (in abelian_subgroup) a_rcosets_carrier:
  "X ∈ a_rcosets H ⇒ X ⊆ carrier G"
by (rule subgroup.rcosets_carrier [OF a_subgroup a_group,
    folded A_RCOSSETS_def, simplified monoid_record_simps])

```

8.9.10 Addition of Subgroups

```

lemma (in abelian_monoid) set_add_closed:
  assumes Acarr: "A ⊆ carrier G"
  and Bcarr: "B ⊆ carrier G"
  shows "A <+> B ⊆ carrier G"
by (rule monoid.set_mult_closed [OF a_monoid,
    folded set_add_def, simplified monoid_record_simps]) (rule Acarr,
rule Bcarr)

```

```

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"
apply (rule additive_subgroup.intro)
apply (unfold set_add_def)
apply (intro comm_group.mult_subgroups)
  apply (rule a_comm_group)
  apply (rule additive_subgroup.a_subgroup[OF subH])
  apply (rule additive_subgroup.a_subgroup[OF subK])
done

end

```

```

theory Ideal
imports Ring AbelCoset
begin

```

9 Ideals

9.1 Definitions

9.1.1 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
apply (intro abelian_subgroupI3 abelian_group.intro)
  apply (rule ideal.axioms, rule ideal_axioms)
  apply (rule abelian_group.axioms, rule ring.axioms, rule ideal.axioms,
rule ideal_axioms)
apply (rule abelian_group.axioms, rule ring.axioms, rule ideal.axioms,
rule ideal_axioms)
done

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 (carrier = carrier R, mult = add R,
one = zero 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 apply (intro ideal.intro ideal_axioms.intro additive_subgroupI)
    apply (rule a_subgroup)
    apply (rule is_ring)
    apply (erule (1) I_l_closed)
    apply (erule (1) I_r_closed)
  done
qed

```

9.1.2 Ideals Generated by a Subset of carrier R

```

constdefs (structure R)
  genideal :: "('a, 'b) ring_scheme ⇒ 'a set ⇒ 'a set" ("Idl %_" [80]
79)
  "genideal R S ≡ Inter {I. ideal I R ∧ S ⊆ I}"

```

9.1.3 Principal Ideals

```

locale principalideal = ideal +
  assumes generate: "∃i ∈ carrier R. I = Idl {i}"

lemma (in principalideal) is_principalideal:
  shows "principalideal I R"
by (rule principalideal_axioms)

lemma principalidealI:

```

```

fixes R (structure)
assumes "ideal I R"
assumes generate: " $\exists i \in \text{carrier } R. I = \text{Idl } \{i\}$ "
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

```

9.1.4 Maximal Ideals

```

locale maximalideal = ideal +
  assumes I_notcarr: "carrier R  $\neq$  I"
  and I_maximal: " $\llbracket \text{ideal } J \text{ R}; I \subseteq J; J \subseteq \text{carrier } R \rrbracket \implies J = I \vee J = \text{carrier } R$ "

```

```

lemma (in maximalideal) is_maximalideal:
  shows "maximalideal I R"
by (rule maximalideal_axioms)

```

```

lemma maximalidealI:
  fixes R
  assumes "ideal I R"
  assumes I_notcarr: "carrier R  $\neq$  I"
  and I_maximal: " $\bigwedge J. \llbracket \text{ideal } J \text{ R}; I \subseteq J; J \subseteq \text{carrier } R \rrbracket \implies J = I \vee J = \text{carrier } R$ "
  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

```

9.1.5 Prime Ideals

```

locale primeideal = ideal + cring +
  assumes I_notcarr: "carrier R  $\neq$  I"
  and I_prime: " $\llbracket a \in \text{carrier } R; b \in \text{carrier } R; a \otimes b \in I \rrbracket \implies a \in I \vee b \in I$ "

```

```

lemma (in primeideal) is_primeideal:
  shows "primeideal I R"
by (rule primeideal_axioms)

```

```

lemma primeidealI:
  fixes R (structure)
  assumes "ideal I R"
  assumes "cring R"
  assumes I_notcarr: "carrier R  $\neq$  I"

```

```

      and I_prime: " $\bigwedge a \ b. \llbracket a \in \text{carrier } R; b \in \text{carrier } R; a \otimes b \in I \rrbracket$ "
 $\Rightarrow a \in I \vee b \in I$ "
    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"
  assumes "cring R"
  assumes I_l_closed: " $\bigwedge a \ x. \llbracket a \in I; x \in \text{carrier } R \rrbracket \Rightarrow x \otimes a \in I$ "
    and I_r_closed: " $\bigwedge a \ x. \llbracket a \in I; x \in \text{carrier } R \rrbracket \Rightarrow a \otimes x \in I$ "
    and I_notcarr: "carrier R  $\neq$  I"
    and I_prime: " $\bigwedge a \ b. \llbracket a \in \text{carrier } R; b \in \text{carrier } R; a \otimes b \in I \rrbracket$ "
 $\Rightarrow a \in I \vee b \in I$ "
  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)
      apply (intro primeideal_axioms.intro)
      apply (rule I_notcarr)
      apply (erule (2) I_prime)
    done
  qed

```

9.2 Special Ideals

```

lemma (in ring) zeroideal:
  shows "ideal {0} R"
  apply (intro idealI subgroup.intro)
    apply (rule is_ring)
    apply simp+
    apply (fold a_inv_def, simp)
  apply simp+
done

```

```

lemma (in ring) oneideal:
  shows "ideal (carrier R) R"
  apply (intro idealI subgroup.intro)
    apply (rule is_ring)
    apply simp+

```

```

    apply (fold a_inv_def, simp)
  apply simp+
done

lemma (in "domain") zeroprimeideal:
  shows "primeideal {0} R"
apply (intro primeidealI)
  apply (rule zeroideal)
  apply (rule domain.axioms, rule domain_axioms)
defer 1
  apply (simp add: integral)
proof (rule ccontr, simp)
  assume "carrier R = {0}"
  from this have "1 = 0" by (rule one_zeroI)
  from this and one_not_zero
    show "False" by simp
qed

```

9.3 General Ideal Properties

```

lemma (in ideal) one_imp_carrier:
  assumes I_one_closed: "1 ∈ I"
  shows "I = carrier R"
apply (rule)
apply (rule)
apply (rule a_Hcarr, simp)
proof
  fix x
  assume xcarr: "x ∈ carrier R"
  from I_one_closed and this
    have "x ⊗ 1 ∈ I" by (intro I_l_closed)
  from this and xcarr
    show "x ∈ I" by simp
qed

```

```

lemma (in ideal) Icarr:
  assumes iI: "i ∈ I"
  shows "i ∈ carrier R"
using iI by (rule a_Hcarr)

```

9.4 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
show ?thesis
apply (intro idealI subgroup.intro)
  apply (rule is_ring)
  apply (force simp add: a_subset)
  apply (simp add: a_inv_def[symmetric])
  apply simp
  apply (simp add: a_inv_def[symmetric])
  apply (clarsimp, rule)
  apply (fast intro: ideal.I_l_closed ideal.intro assms)+
  apply (clarsimp, rule)
  apply (fast intro: ideal.I_r_closed ideal.intro assms)+
done
qed

```

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

```

lemma (in ring) i_Intersect:
  assumes Sideals: " $\bigwedge I. I \in S \implies \text{ideal } I \text{ } R$ "
  and notempty: " $S \neq \{\}$ "
  shows "ideal (Inter S) R"
apply (unfold_locales)
apply (simp_all add: Inter_def INTER_def)
  apply (rule, simp) defer 1
  apply rule defer 1
  apply rule defer 1
  apply (fold a_inv_def, rule) defer 1
  apply rule defer 1
  apply rule defer 1
proof -
  fix x
  assume " $\forall I \in S. x \in I$ "
  hence xI: " $\bigwedge I. I \in S \implies x \in I$ " by simp

  from notempty have " $\exists I0. I0 \in S$ " by blast
  from this obtain I0 where IOS: " $I0 \in S$ " by auto

  interpret ideal I0 R by (rule Sideals[OF IOS])

  from xI[OF IOS] have " $x \in I0$ " .
  from this and a_subset show " $x \in \text{carrier } R$ " by fast
next
  fix x y
  assume " $\forall I \in S. x \in I$ "
  hence xI: " $\bigwedge I. I \in S \implies x \in I$ " by simp
  assume " $\forall I \in S. y \in I$ "
  hence yI: " $\bigwedge I. I \in S \implies y \in I$ " by simp

  fix J

```

```

    assume JS: "J ∈ S"
    interpret ideal J R by (rule Sideals[OF JS])
    from xI[OF JS] and yI[OF JS]
      show "x ⊕ y ∈ J" by (rule a_closed)
next
  fix J
  assume JS: "J ∈ S"
  interpret ideal J R by (rule Sideals[OF JS])
  show "0 ∈ J" by simp
next
  fix x
  assume "∀ I ∈ S. x ∈ I"
  hence xI: "⋀ I. I ∈ S ⇒ x ∈ I" by simp

  fix J
  assume JS: "J ∈ S"
  interpret ideal J R by (rule Sideals[OF JS])

  from xI[OF JS]
    show "⊖ x ∈ J" by (rule a_inv_closed)
next
  fix x y
  assume "∀ I ∈ S. x ∈ I"
  hence xI: "⋀ I. I ∈ S ⇒ x ∈ I" by simp
  assume ycarr: "y ∈ carrier R"

  fix J
  assume JS: "J ∈ S"
  interpret ideal J R by (rule Sideals[OF JS])

  from xI[OF JS] and ycarr
    show "y ⊗ x ∈ J" by (rule I_l_closed)
next
  fix x y
  assume "∀ I ∈ S. x ∈ I"
  hence xI: "⋀ I. I ∈ S ⇒ x ∈ I" by simp
  assume ycarr: "y ∈ carrier R"

  fix J
  assume JS: "J ∈ S"
  interpret ideal J R by (rule Sideals[OF JS])

  from xI[OF JS] and ycarr
    show "x ⊗ y ∈ J" by (rule I_r_closed)
qed

```

9.5 Addition of Ideals

lemma (in ring) add_ideals:

```

    assumes idealI: "ideal I R"
    and idealJ: "ideal J R"
    shows "ideal (I <+> J) R"
  apply (rule ideal.intro)
  apply (rule add_additive_subgroups)
  apply (intro ideal.axioms[OF idealI])
  apply (intro ideal.axioms[OF idealJ])
  apply (rule is_ring)
  apply (rule ideal_axioms.intro)
  apply (simp add: set_add_defs, clarsimp) defer 1
  apply (simp add: set_add_defs, clarsimp) defer 1
  proof -
    fix x i j
    assume xcarr: "x ∈ carrier R"
    and iI: "i ∈ I"
    and jJ: "j ∈ J"
    from xcarr ideal.Icarr[OF idealI iI] ideal.Icarr[OF idealJ jJ]
    have c: "(i ⊕ j) ⊗ x = (i ⊗ x) ⊕ (j ⊗ x)" by algebra
    from xcarr and iI
    have a: "i ⊗ x ∈ I" by (simp add: ideal.I_r_closed[OF idealI])
    from xcarr and jJ
    have b: "j ⊗ x ∈ J" by (simp add: ideal.I_r_closed[OF idealJ])
    from a b c
    show "∃ha∈I. ∃ka∈J. (i ⊕ j) ⊗ x = ha ⊕ ka" by fast
  next
    fix x i j
    assume xcarr: "x ∈ carrier R"
    and iI: "i ∈ I"
    and jJ: "j ∈ J"
    from xcarr ideal.Icarr[OF idealI iI] ideal.Icarr[OF idealJ jJ]
    have c: "x ⊗ (i ⊕ j) = (x ⊗ i) ⊕ (x ⊗ j)" by algebra
    from xcarr and iI
    have a: "x ⊗ i ∈ I" by (simp add: ideal.I_l_closed[OF idealI])
    from xcarr and jJ
    have b: "x ⊗ j ∈ J" by (simp add: ideal.I_l_closed[OF idealJ])
    from a b c
    show "∃ha∈I. ∃ka∈J. x ⊗ (i ⊕ j) = ha ⊕ ka" by fast
  qed

```

9.6 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  $\subseteq$  carrier R"
  shows "S  $\subseteq$  Idl S"
unfolding genideal_def
by fast
```

```
lemma (in ring) genideal_self':
  assumes carr: "i  $\in$  carrier R"
  shows "i  $\in$  Idl {i}"
proof -
  from carr
    have "{i}  $\subseteq$  Idl {i}" by (fast intro!: genideal_self)
  thus "i  $\in$  Idl {i}" by fast
qed
```

genideal generates the minimal ideal

```
lemma (in ring) genideal_minimal:
  assumes a: "ideal I R"
  and b: "S  $\subseteq$  I"
  shows "Idl S  $\subseteq$  I"
unfolding genideal_def
by (rule, elim InterD, simp add: a b)
```

Generated ideals and subsets

```
lemma (in ring) Idl_subset_ideal:
  assumes Ideal: "ideal I R"
  and Hcarr: "H  $\subseteq$  carrier R"
  shows "(Idl H  $\subseteq$  I) = (H  $\subseteq$  I)"
proof
  assume a: "Idl H  $\subseteq$  I"
  from Hcarr have "H  $\subseteq$  Idl H" by (rule genideal_self)
  from this and a
    show "H  $\subseteq$  I" by simp
next
  fix x
  assume HI: "H  $\subseteq$  I"

  from Ideal and HI
    have "I  $\in$  {I. ideal I R  $\wedge$  H  $\subseteq$  I}" by fast
  from this
    show "Idl H  $\subseteq$  I"
    unfolding genideal_def
    by fast
qed
```

```
lemma (in ring) subset_Idl_subset:
  assumes Icarr: "I  $\subseteq$  carrier R"
```

```

    and HI: "H  $\subseteq$  I"
  shows "Idl H  $\subseteq$  Idl I"
proof -
  from HI and genideal_self[OF Icarr]
    have HIdlI: "H  $\subseteq$  Idl I" by fast

  from Icarr
    have Iideal: "ideal (Idl I) R" by (rule genideal_ideal)
  from HI and Icarr
    have "H  $\subseteq$  carrier R" by fast
  from Iideal and this
    have "(H  $\subseteq$  Idl I) = (Idl H  $\subseteq$  Idl I)"
    by (rule Idl_subset_ideal[symmetric])

  from HIdlI and this
    show "Idl H  $\subseteq$  Idl I" by simp
qed

lemma (in ring) Idl_subset_ideal':
  assumes acar: "a  $\in$  carrier R" and bcarr: "b  $\in$  carrier R"
  shows "(Idl {a}  $\subseteq$  Idl {b}) = (a  $\in$  Idl {b})"
apply (subst Idl_subset_ideal[OF genideal_ideal[of "{b}"], of "{a}"])
  apply (fast intro: bcarr, fast intro: acar)
apply fast
done

lemma (in ring) genideal_zero:
  "Idl {0} = {0}"
apply rule
  apply (rule genideal_minimal[OF zeroideal], simp)
apply (simp add: genideal_self')
done

lemma (in ring) genideal_one:
  "Idl {1} = carrier R"
proof -
  interpret ideal "Idl {1}" "R" by (rule genideal_ideal, fast intro: one_closed)
  show "Idl {1} = carrier R"
  apply (rule, rule a_subset)
  apply (simp add: one_imp_carrier genideal_self')
  done
qed

```

Generation of Principal Ideals in Commutative Rings

```

constdefs (structure R)
  cgenideal :: "('a, 'b) monoid_scheme  $\Rightarrow$  'a  $\Rightarrow$  'a set" ("PIdlz _" [80]
79)
  "cgenideal R a  $\equiv$  { x  $\otimes$  a | x. x  $\in$  carrier R }"

```

genhideal (?) really generates an ideal

```

lemma (in cring) cgenideal_ideal:
  assumes acarr: "a ∈ carrier R"
  shows "ideal (PIdl a) R"
apply (unfold cgenideal_def)
apply (rule idealI[OF is_ring])
  apply (rule subgroup.intro)
    apply (simp_all add: monoid_record_simps)
    apply (blast intro: acarr m_closed)
    apply clarsimp defer 1
  defer 1
  apply (fold a_inv_def, clarsimp) defer 1
  apply clarsimp defer 1
  apply clarsimp defer 1
proof -
  fix x y
  assume xcarr: "x ∈ carrier R"
  and ycarr: "y ∈ carrier R"
  note carr = acarr xcarr ycarr

  from carr
    have "x ⊗ a ⊕ y ⊗ a = (x ⊕ y) ⊗ a" by (simp add: l_distr)
  from this and carr
    show "∃z. x ⊗ a ⊕ y ⊗ a = z ⊗ a ∧ z ∈ carrier R" by fast
next
  from l_null[OF acarr, symmetric] and zero_closed
    show "∃x. 0 = x ⊗ a ∧ x ∈ carrier R" by fast
next
  fix x
  assume xcarr: "x ∈ carrier R"
  note carr = acarr xcarr

  from carr
    have "⊖ (x ⊗ a) = (⊖ x) ⊗ a" by (simp add: l_minus)
  from this and carr
    show "∃z. ⊖ (x ⊗ a) = z ⊗ a ∧ z ∈ carrier R" by fast
next
  fix x y
  assume xcarr: "x ∈ carrier R"
  and ycarr: "y ∈ carrier R"
  note carr = acarr xcarr ycarr

  from carr
    have "y ⊗ a ⊗ x = (y ⊗ x) ⊗ a" by (simp add: m_assoc, simp add:
m_comm)
  from this and carr
    show "∃z. y ⊗ a ⊗ x = z ⊗ a ∧ z ∈ carrier R" by fast
next
  fix x y
  assume xcarr: "x ∈ carrier R"

```

```

    and ycarr: "y ∈ carrier R"
    note carr = acarr xcarr ycarr

    from carr
    have "x ⊗ (y ⊗ a) = (x ⊗ y) ⊗ a" by (simp add: m_assoc)
    from this and carr
    show "∃z. x ⊗ (y ⊗ a) = z ⊗ a ∧ z ∈ carrier R" by fast
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
  from this and 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
    apply rule
    apply clarify
    using aJ
    apply (erule I_1_closed)
    done
qed

lemma (in cring) cgenideal_eq_genideal:
  assumes icarr: "i ∈ carrier R"
  shows "PIDl i = Idl {i}"
apply rule
apply (intro cgenideal_minimal)
apply (rule genideal_ideal, fast intro: icarr)
apply (rule genideal_self', fast intro: icarr)
apply (intro genideal_minimal)
apply (rule cgenideal_ideal [OF icarr])
apply (simp, rule cgenideal_self [OF icarr])
done

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 icarr: "i ∈ carrier R"
  shows "principalideal (PIdl i) R"
apply (rule principalidealI)
apply (rule cgenideal_ideal [OF icarr])
proof -
  from icarr
    have "PIdl i = Idl {i}" by (rule cgenideal_eq_genideal)
  from icarr and this
    show "∃ i' ∈ carrier R. PIdl i = Idl {i'}" by fast
qed

```

9.7 Union of Ideals

```

lemma (in ring) union_genideal:
  assumes idealI: "ideal I R"
  and idealJ: "ideal J R"
  shows "Idl (I ∪ J) = I <+> J"
apply rule
apply (rule ring.genideal_minimal)
  apply (rule is_ring)
  apply (rule add_ideals[OF idealI idealJ])
  apply (rule)
  apply (simp add: set_add_defs) apply (elim disjE) defer 1 defer 1
  apply (rule) apply (simp add: set_add_defs genideal_def) apply clarsimp
defer 1
proof -
  fix x
  assume xI: "x ∈ I"
  have ZJ: "0 ∈ J"
    by (intro additive_subgroup.zero_closed, rule ideal.axioms[OF idealJ])
  from ideal.Icarr[OF idealI xI]
    have "x = x ⊕ 0" by algebra
  from xI and ZJ and this
    show "∃ h ∈ I. ∃ k ∈ J. x = h ⊕ k" by fast
next
  fix x
  assume xJ: "x ∈ J"
  have ZI: "0 ∈ I"
    by (intro additive_subgroup.zero_closed, rule ideal.axioms[OF idealI])
  from ideal.Icarr[OF idealJ xJ]
    have "x = 0 ⊕ x" by algebra
  from ZI and xJ and this
    show "∃ h ∈ I. ∃ k ∈ J. x = h ⊕ k" by fast
next
  fix i j K

```

```

assume iI: "i ∈ I"
  and jJ: "j ∈ J"
  and idealK: "ideal K R"
  and IK: "I ⊆ K"
  and JK: "J ⊆ K"
from iI and IK
  have iK: "i ∈ K" by fast
from jJ and JK
  have jK: "j ∈ K" by fast
from iK and jK
  show "i ⊕ j ∈ K" by (intro additive_subgroup.a_closed) (rule ideal.axioms[OF
idealK])
qed

```

9.8 Properties of Principal Ideals

0 generates the zero ideal

```

lemma (in ring) zero_genideal:
  shows "Idl {0} = {0}"
apply rule
apply (simp add: genideal_minimal zeroideal)
apply (fast intro!: genideal_self)
done

```

1 generates the unit ideal

```

lemma (in ring) one_genideal:
  shows "Idl {1} = carrier R"
proof -
  have "1 ∈ Idl {1}" by (simp add: genideal_self')
  thus "Idl {1} = carrier R" by (intro ideal.one_imp_carrier, fast intro:
genideal_ideal)
qed

```

The zero ideal is a principal ideal

```

corollary (in ring) zeropideal:
  shows "principalideal {0} R"
apply (rule principalidealI)
apply (rule zeroideal)
apply (blast intro!: zero_closed zero_genideal[symmetric])
done

```

The unit ideal is a principal ideal

```

corollary (in ring) onepideal:
  shows "principalideal (carrier R) R"
apply (rule principalidealI)
apply (rule oneideal)
apply (blast intro!: one_closed one_genideal[symmetric])
done

```

Every principal ideal is a right coset of the carrier

```

lemma (in principalideal) rcos_generate:
  assumes "cring R"
  shows " $\exists x \in I. I = \text{carrier } R \#> x$ "
proof -
  interpret cring R by fact
  from generate
    obtain i
      where icarr: " $i \in \text{carrier } R$ "
      and I1: " $I = \text{Idl } \{i\}$ "
    by fast+

  from icarr and genideal_self[of "{i}"]
    have " $i \in \text{Idl } \{i\}$ " by fast
  hence iI: " $i \in I$ " by (simp add: I1)

  from I1 icarr
    have I2: " $I = \text{PIdl } i$ " by (simp add: cgenideal_eq_genideal)

  have "PIdl i = carrier R #> i"
    unfolding cgenideal_def r_coset_def
    by fast

  from I2 and this
    have " $I = \text{carrier } R \#> i$ " by simp

  from iI and this
    show " $\exists x \in I. I = \text{carrier } R \#> x$ " by fast
qed

```

9.9 Prime Ideals

```

lemma (in ideal) primeidealCD:
  assumes "cring R"
  assumes notprime: " $\neg \text{primeideal } I \text{ } R$ "
  shows " $\text{carrier } R = I \vee (\exists a \ b. a \in \text{carrier } R \wedge b \in \text{carrier } R \wedge a \otimes b \in I \wedge a \notin I \wedge b \notin I)$ "
proof (rule ccontr, clarsimp)
  interpret cring R by fact
  assume InR: " $\text{carrier } R \neq I$ "
  and " $\forall a. a \in \text{carrier } R \longrightarrow (\forall b. a \otimes b \in I \longrightarrow b \in \text{carrier } R \longrightarrow a \in I \vee b \in I)$ "
  hence I_prime: " $\bigwedge a \ b. [a \in \text{carrier } R; b \in \text{carrier } R; a \otimes b \in I] \implies a \in I \vee b \in I$ " by simp
  have "primeideal I R"
    apply (rule primeideal.intro [OF is_ideal is_cring])
    apply (rule primeideal_axioms.intro)
    apply (rule InR)
    apply (erule (2) I_prime)

```

```

    done
  from this and 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"
apply (rule domain.intro, rule is_cring)
apply (rule domain_axioms.intro)
proof (rule ccontr, simp)
  interpret primeideal "{0}" "R" by (rule pi)
  assume "1 = 0"
  hence "carrier R = {0}" by (rule one_zeroD)
  from this[symmetric] and I_notcarr
    show "False" by simp
next
  interpret primeideal "{0}" "R" by (rule pi)
  fix a b
  assume ab: "a ⊗ b = 0"
  and carr: "a ∈ carrier R" "b ∈ carrier R"
  from ab
    have abI: "a ⊗ b ∈ {0}" by fast
  from carr and this
    have "a ∈ {0} ∨ b ∈ {0}" by (rule I_prime)
  thus "a = 0 ∨ b = 0" by simp
qed

corollary (in cring) domain_eq_zeroprimeideal:
  shows "domain R = primeideal {0} R"
apply rule
  apply (erule domain.zeroprimeideal)
  apply (erule zeroprimeideal_domainI)
done

```


9.10 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 apply (rule idealI)
    apply (rule cring.axioms[OF is_cring])
    apply (rule subgroup.intro)
    apply (simp, fast)
    apply clarsimp apply (simp add: r_distr acarr)
    apply (simp add: acarr)
    apply (simp add: a_inv_def[symmetric], clarify) defer 1
    apply clarsimp defer 1
    apply (fast intro!: helper_I_closed acarr)
  proof -
    fix x
    assume xcarr: "x ∈ carrier R"
      and ax: "a ⊗ x ∈ I"
    from ax and acarr xcarr
      have "⊖(a ⊗ x) ∈ I" by simp
    also from acarr xcarr
      have "⊖(a ⊗ x) = a ⊗ (⊖x)" by algebra
    finally
      show "a ⊗ (⊖x) ∈ I" .
    from acarr
      have "a ⊗ 0 = 0" by simp
  next
    fix x y
    assume xcarr: "x ∈ carrier R"
      and ycarr: "y ∈ carrier R"
      and ayI: "a ⊗ y ∈ I"
    from ayI and acarr xcarr ycarr
      have "a ⊗ (y ⊗ x) ∈ I" by (simp add: helper_I_closed)
    moreover from xcarr ycarr

```

```

      have "y  $\otimes$  x = x  $\otimes$  y" by (simp add: m_comm)
      ultimately
      show "a  $\otimes$  (x  $\otimes$  y)  $\in$  I" by simp
    qed
  qed

```

In a cring every maximal ideal is prime

```

lemma (in cring) maximalideal_is_prime:
  assumes "maximalideal I R"
  shows "primeideal I R"
proof -
  interpret maximalideal I R by fact
  show ?thesis apply (rule ccontr)
    apply (rule primeidealCE)
    apply (rule is_cring)
    apply assumption
    apply (simp add: I_notcarr)
  proof -
    assume " $\exists$  a b. a  $\in$  carrier R  $\wedge$  b  $\in$  carrier R  $\wedge$  a  $\otimes$  b  $\in$  I  $\wedge$  a  $\notin$ 
I  $\wedge$  b  $\notin$  I"
    from this
    obtain a b
      where acarr: "a  $\in$  carrier R"
      and bcarr: "b  $\in$  carrier R"
      and abI: "a  $\otimes$  b  $\in$  I"
      and anI: "a  $\notin$  I"
      and bnI: "b  $\notin$  I"
      by fast
    def J  $\equiv$  "{x $\in$ carrier R. a  $\otimes$  x  $\in$  I}"

    from is_cring and acarr
    have idealJ: "ideal J R" unfolding J_def by (rule helper_max_prime)

    have IsubJ: "I  $\subseteq$  J"
    proof
      fix x
      assume xI: "x  $\in$  I"
      from this and acarr
      have "a  $\otimes$  x  $\in$  I" by (intro I_1_closed)
      from xI[THEN a_Hcarr] this
      show "x  $\in$  J" unfolding J_def by fast
    qed

    from abI and acarr bcarr
    have "b  $\in$  J" unfolding J_def by fast
    from bnI and this
    have JnI: "J  $\neq$  I" by fast
    from acarr
    have "a = a  $\otimes$  1" by algebra

```

```

from this and anI
have "a  $\otimes$  1  $\notin$  I" by simp
from one_closed and this
have "1  $\notin$  J" unfolding J_def by fast
hence Jncarr: "J  $\neq$  carrier R" by fast

interpret ideal J R by (rule idealJ)

have "J = I  $\vee$  J = carrier R"
  apply (intro I_maximal)
  apply (rule idealJ)
  apply (rule IsubJ)
  apply (rule a_subset)
  done

from this and JnI and Jncarr
show "False" by simp
qed
qed

```

9.11 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  $\neq$  {0}"
    and haveideals: "{I. ideal I R} = {{0}, carrier R}"
  shows "field R"
  apply (rule cring_fieldI)
  apply (rule, rule, rule)
  apply (erule Units_closed)
  defer 1
    apply rule
  defer 1
  proof (rule ccontr, simp)
    assume zUnit: "0  $\in$  Units R"
    hence a: "0  $\otimes$  inv 0 = 1" by (rule Units_r_inv)
    from zUnit
      have "0  $\otimes$  inv 0 = 0" by (intro l_null, rule Units_inv_closed)
    from a[symmetric] and this
      have "1 = 0" by simp
    hence "carrier R = {0}" by (rule one_zeroD)
    from this and carrnzero
      show "False" by simp
  next
    fix x
    assume xcarr': "x  $\in$  carrier R - {0}"
    hence xcarr: "x  $\in$  carrier R" by fast
    from xcarr'
      have xnZ: "x  $\neq$  0" by fast

```

```

from xcarr
  have xIdl: "ideal (PIdl x) R" by (intro cgenideal_ideal, fast)

from xcarr
  have "x ∈ PIdl x" by (intro cgenideal_self, fast)
from this and xnZ
  have "PIdl x ≠ {0}" by fast
from haveideals and this
  have "PIdl x = carrier R"
  by (blast intro!: xIdl)
hence "1 ∈ PIdl x" by simp
hence "∃ y. 1 = y ⊗ x ∧ y ∈ carrier R" unfolding cgenideal_def by blast
from this
  obtain y
    where ycarr: "y ∈ carrier R"
    and ylinv: "1 = y ⊗ x"
  by fast+
from ylinv and xcarr ycarr
  have yrinv: "1 = x ⊗ y" by (simp add: m_comm)
from ycarr and ylinv[symmetric] and yrinv[symmetric]
  have "∃ y ∈ carrier R. y ⊗ x = 1 ∧ x ⊗ y = 1" by fast
from this and xcarr
  show "x ∈ Units R"
  unfolding Units_def
  by fast
qed

lemma (in field) all_ideals:
  shows "{I. ideal I R} = {{0}, carrier R}"
apply (rule, rule)
proof -
  fix I
  assume a: "I ∈ {I. ideal I R}"
  with this
    interpret ideal I R by simp

  show "I ∈ {{0}, carrier R}"
  proof (cases "∃ a. a ∈ I - {0}")
    assume "∃ a. a ∈ I - {0}"
    from this
      obtain a
        where aI: "a ∈ I"
        and anZ: "a ≠ 0"
      by fast+
    from aI[THEN a_Hcarr] anZ
      have aUnit: "a ∈ Units R" by (simp add: field_Units)
    hence 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)
  end
end

```

```

hence oneI: "1 ∈ I" by (simp add: a[symmetric])

have "carrier R ⊆ I"
proof
  fix x
  assume xcarr: "x ∈ carrier R"
  from oneI and this
    have "1 ⊗ x ∈ I" by (rule I_r_closed)
  from this and xcarr
    show "x ∈ I" by simp
qed
from this and a_subset
  have "I = carrier R" by fast
thus "I ∈ {{0}, carrier R}" by fast
next
assume "¬ (∃ a. a ∈ I - {0})"
hence IZ: "∧ a. a ∈ I ⇒ a = 0" by simp

have a: "I ⊆ {0}"
proof
  fix x
  assume "x ∈ I"
  hence "x = 0" by (rule IZ)
  thus "x ∈ {0}" by fast
qed

have "0 ∈ I" by simp
hence "{0} ⊆ I" by fast

from this and a
  have "I = {0}" by fast
thus "I ∈ {{0}, carrier R}" by fast
qed
qed (simp add: 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"
apply (rule maximalidealI)
apply (rule zeroideal)
proof-
  from one_not_zero
  have "1 ∉ {0}" by simp

```

```

    from this and one_closed
      show "carrier R  $\neq$  {0}" by fast
next
  fix J
  assume Jideal: "ideal J R"
  hence "J  $\in$  {I. ideal I R}"
    by fast

  from this and all_ideals
    show "J = {0}  $\vee$  J = carrier R" by simp
qed

lemma (in cring) zeromaximalideal_fieldI:
  assumes zeromax: "maximalideal {0} R"
  shows "field R"
apply (rule trivialideals_fieldI, rule maximalideal.I_notcarr[OF zeromax])
apply rule apply clarsimp defer 1
  apply (simp add: zeroideal oneideal)
proof -
  fix J
  assume Jn0: "J  $\neq$  {0}"
    and idealJ: "ideal J R"
  interpret ideal J R by (rule idealJ)
  have "{0}  $\subseteq$  J" by (rule ccontr, simp)
  from zeromax and idealJ and this and a_subset
    have "J = {0}  $\vee$  J = carrier R" by (rule maximalideal.I_maximal)
  from this and Jn0
    show "J = carrier R" by simp
qed

lemma (in cring) zeromaximalideal_eq_field:
  "maximalideal {0} R = field R"
apply rule
  apply (erule zeromaximalideal_fieldI)
apply (erule field.zeromaximalideal)
done

end

theory RingHom
imports Ideal
begin

```

10 Homomorphisms of Non-Commutative Rings

Lifting existing lemmas in a `ring_hom_ring` locale

```

locale ring_hom_ring = R: ring R + S: ring S
  for R (structure) and S (structure) +
  fixes h
  assumes homh: "h ∈ ring_hom R S"
  notes hom_mult [simp] = ring_hom_mult [OF homh]
    and hom_one [simp] = ring_hom_one [OF homh]

sublocale ring_hom_cring ⊆ ring: ring_hom_ring
  proof qed (rule homh)

sublocale ring_hom_ring ⊆ abelian_group: abelian_group_hom R S
apply (rule abelian_group_homI)
  apply (rule R.is_abelian_group)
  apply (rule S.is_abelian_group)
apply (intro group_hom.intro group_hom_axioms.intro)
  apply (rule R.a_group)
  apply (rule S.a_group)
apply (insert homh, unfold hom_def ring_hom_def)
apply simp
done

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 = 1_S"
  shows "ring_hom_ring R S h"
proof -
  interpret ring R by fact
  interpret ring S by fact
  show ?thesis apply unfold_locales
apply (unfold ring_hom_def, safe)
  apply (simp add: hom_closed Pi_def)
  apply (erule (1) compatible_mult)
  apply (erule (1) compatible_add)
apply (rule compatible_one)
done
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 apply (intro ring_hom_ring.intro ring_hom_ring_axioms.intro)
      apply (rule R.is_ring)
      apply (rule S.is_ring)
      apply (rule h)
    done
  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 = 1_S"
  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 apply (intro ring_hom_ring.intro ring_hom_ring_axioms.intro,
      rule R.is_ring, rule S.is_ring)
      apply (insert group_hom.homh[OF a_group_hom])
      apply (unfold hom_def ring_hom_def, simp)
      apply safe
      apply (erule (1) compatible_mult)
      apply (rule compatible_one)
    done
  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 by (intro ring_hom_cring.intro ring_hom_cring_axioms.intro)
      (rule R.is_cring, rule S.is_cring, rule homh)
  qed

```

10.1 The Kernel of a Ring Homomorphism

— the kernel of a ring homomorphism is an ideal

```
lemma (in ring_hom_ring) kernel_is_ideal:
```



```

    shows "ideal (a_kernel R S h) R"
  apply (rule idealI)
    apply (rule R.is_ring)
    apply (rule additive_subgroup.a_subgroup[OF additive_subgroup_a_kernel])
    apply (unfold a_kernel_def', simp+)
  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 = 0_S"
by (simp add: a_kernel_defs)

```

10.2 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 "... = 0_S ⊕_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 = acar xcar
  note hcarr = acar[THEN hom_closed] xcar[THEN hom_closed]

  from hx and hcarr
    have a: "h x  $\oplus_S$   $\ominus_S$  h a = 0S" by algebra
  from carr
    have "h x  $\oplus_S$   $\ominus_S$  h a = h (x  $\oplus$   $\ominus$  a)" by simp
  from a and this
    have b: "h (x  $\oplus$   $\ominus$  a) = 0S" by simp

  from carr have "x  $\oplus$   $\ominus$  a  $\in$  carrier R" by simp
  from this and b
    have "x  $\oplus$   $\ominus$  a  $\in$  a_kernel R S h"
    unfolding a_kernel_def'
    by fast

  from this and carr
    show "x  $\in$  a_kernel R S h  $\rightarrow$  a" by (simp add: a_rcos_module_rev)
qed

corollary (in ring_hom_ring) rcos_eq_homeq:
  assumes acar: "a  $\in$  carrier R"
  shows "(a_kernel R S h)  $\rightarrow$  a = {x  $\in$  carrier R. h x = h a}"
apply rule defer 1
apply clarsimp defer 1
proof
  interpret ideal "a_kernel R S h" "R" by (rule kernel_is_ideal)

  fix x
  assume xrcos: "x  $\in$  a_kernel R S h  $\rightarrow$  a"
  from acar and this
    have xcar: "x  $\in$  carrier R"
    by (rule a_elemrcos_carrier)

  from xrcos
    have "h x = h a" by (rule rcos_imp_homeq[OF acar])
  from xcar and this
    show "x  $\in$  {x  $\in$  carrier R. h x = h a}" by fast
next
  interpret ideal "a_kernel R S h" "R" by (rule kernel_is_ideal)

  fix x
  assume xcar: "x  $\in$  carrier R"
  and hx: "h x = h a"
  from acar xcar hx
    show "x  $\in$  a_kernel R S h  $\rightarrow$  a" by (rule homeq_imp_rcos)

```

qed

end

```
theory QuotRing
imports RingHom
begin
```

11 Quotient Rings

11.1 Multiplication on Cosets

```
constdefs (structure R)
  rcoset_mult :: "[('a, _) ring_scheme, 'a set, 'a set, 'a set]  $\Rightarrow$  'a set"
  ("[mod _:] _  $\otimes$  _" [81,81,81] 80)
  "rcoset_mult R I A B  $\equiv \bigcup_{a \in A}. \bigcup_{b \in B}. I +> (a \otimes b)"$ 
```

rcoset_mult fulfils the properties required by congruences

```
lemma (in ideal) rcoset_mult_add:
  "[x  $\in$  carrier R; y  $\in$  carrier R]  $\implies$  [mod I:] (I +> x)  $\otimes$  (I +> y) =
  I +> (x  $\otimes$  y)"
apply rule
apply (rule, simp add: rcoset_mult_def, clarsimp)
defer 1
apply (rule, simp add: rcoset_mult_def)
defer 1
proof -
  fix z x' y'
  assume carr: "x  $\in$  carrier R" "y  $\in$  carrier R"
  and x'rcos: "x'  $\in$  I +> x"
  and y'rcos: "y'  $\in$  I +> y"
  and zrcos: "z  $\in$  I +> x'  $\otimes$  y'"

  from x'rcos
  have " $\exists h \in I. x' = h \oplus x$ " by (simp add: a_r_coset_def r_coset_def)
  from this obtain hx
  where hxI: "hx  $\in$  I"
  and x': "x' = hx  $\oplus$  x"
  by fast+

  from y'rcos
  have " $\exists h \in I. y' = h \oplus y$ " by (simp add: a_r_coset_def r_coset_def)
  from this
  obtain hy
  where hyI: "hy  $\in$  I"
  and y': "y' = hy  $\oplus$  y"
```

```

    by fast+

from zrcos
  have "∃h∈I. z = h ⊕ (x' ⊗ y')" by (simp add: a_r_coset_def r_coset_def)
from this
  obtain hz
  where hzI: "hz ∈ I"
  and z: "z = hz ⊕ (x' ⊗ y')"
  by fast+

note carr = carr hxI[THEN a_Hcarr] hyI[THEN a_Hcarr] hzI[THEN a_Hcarr]

from z have "z = hz ⊕ (x' ⊗ y'" .
also from x' y'
  have "... = 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)

from this and z2
  have "∃h∈I. z = h ⊕ x ⊗ y" by fast
thus "z ∈ I +> x ⊗ y" by (simp add: a_r_coset_def r_coset_def)
next
fix z
assume xcarr: "x ∈ carrier R"
and ycarr: "y ∈ carrier R"
and zrcos: "z ∈ I +> x ⊗ y"
from xcarr
  have xself: "x ∈ I +> x" by (intro a_rcos_self)
from ycarr
  have yself: "y ∈ I +> y" by (intro a_rcos_self)

from xself and yself and zrcos
  show "∃a∈I +> x. ∃b∈I +> y. z ∈ I +> a ⊗ b" by fast
qed

```

11.2 Quotient Ring Definition

```

constdefs (structure R)
  FactRing :: "[('a,'b) ring_scheme, 'a set] ⇒ ('a set) ring"
    (infixl "Quot" 65)
  "FactRing R I ≡
    (carrier = a_rcosets I, mult = rcset_mult R I, one = (I +> 1), zero
= I, add = set_add R)"

```

11.3 Factorization over General Ideals

The quotient is a ring

```

lemma (in ideal) quotient_is_ring:
  shows "ring (R Quot I)"
apply (rule ringI)
  — abelian group
  apply (rule comm_group_abelian_groupI)
  apply (simp add: FactRing_def)
  apply (rule a_factorgroup_is_comm_group[unfolded A_FactGroup_def'])
  — mult monoid
  apply (rule monoidI)
    apply (simp_all add: FactRing_def A_RCOSSETS_def RCOSSETS_def
      a_r_coset_def[symmetric])
    — mult closed
    apply (clarify)
    apply (simp add: rcoset_mult_add, fast)
    — mult one_closed
    apply (force intro: one_closed)
    — mult assoc
    apply clarify
    apply (simp add: rcoset_mult_add m_assoc)
    — mult one
    apply clarify
    apply (simp add: rcoset_mult_add l_one)
  apply clarify
  apply (simp add: rcoset_mult_add r_one)
  — distr
  apply clarify
  apply (simp add: rcoset_mult_add a_rcos_sum l_distr)
  apply clarify
  apply (simp add: rcoset_mult_add a_rcos_sum r_distr)
done

```

This is a ring homomorphism

```

lemma (in ideal) rcos_ring_hom:
  "(op +> I) ∈ ring_hom R (R Quot I)"
apply (rule ring_hom_memI)
  apply (simp add: FactRing_def a_rcosetsI[OF a_subset])
  apply (simp add: FactRing_def rcoset_mult_add)
  apply (simp add: FactRing_def a_rcos_sum)
  apply (simp add: FactRing_def)
done

```

```

lemma (in ideal) rcos_ring_hom_ring:
  "ring_hom_ring R (R Quot I) (op +> I)"
apply (rule ring_hom_ringI)
  apply (rule is_ring, rule quotient_is_ring)
  apply (simp add: FactRing_def a_rcosetsI[OF a_subset])

```

```

    apply (simp add: FactRing_def rcoset_mult_add)
    apply (simp add: FactRing_def a_rcos_sum)
  apply (simp add: FactRing_def)
done

```

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)
  apply (rule quotient_is_ring)
  apply (rule ring.axioms[OF quotient_is_ring])
  apply (simp add: FactRing_def A_RCOSSETS_defs a_r_coset_def[symmetric])
  apply clarify
  apply (simp add: 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) (op +> 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

```

11.4 Factorization over Prime Ideals

The quotient ring generated by a prime ideal is a domain

```

lemma (in primeideal) quotient_is_domain:
  shows "domain (R Quot I)"
  apply (rule domain.intro)
  apply (rule quotient_is_cring, rule is_cring)
  apply (rule domain_axioms.intro)
  apply (simp add: FactRing_def) defer 1
  apply (simp add: FactRing_def A_RCOSSETS_defs a_r_coset_def[symmetric],
  clarify)
  apply (simp add: rcoset_mult_add) defer 1
proof (rule ccontr, clarsimp)
  assume "I +> 1 = I"

```

```

hence "1 ∈ I" by (simp only: a_coset_join1 one_closed a_subgroup)
hence "carrier R ⊆ I" by (subst one_imp_carrier, simp, fast)
from this and a_subset
  have "I = carrier R" by fast
from this and I_notcarr
  show "False" by fast
next
fix x y
assume carr: "x ∈ carrier R" "y ∈ carrier R"
  and a: "I +> x ⊗ y = I"
  and b: "I +> y ≠ I"

have ynI: "y ∉ I"
proof (rule ccontr, simp)
  assume "y ∈ I"
  hence "I +> y = I" by (rule a_rcos_const)
  from this and b
    show "False" by simp
qed

from carr
  have "x ⊗ y ∈ I +> x ⊗ y" by (simp add: a_rcos_self)
from this
  have xyI: "x ⊗ y ∈ I" by (simp add: a)

from xyI and carr
  have xI: "x ∈ I ∨ y ∈ I" by (simp add: I_prime)
from this and ynI
  have "x ∈ I" by fast
thus "I +> x = I" by (rule a_rcos_const)
qed

```

Generating right cosets of a prime ideal is a homomorphism on commutative rings

```

lemma (in primeideal) rcos_ring_hom_cring:
  shows "ring_hom_cring R (R Quot I) (op +> I)"
by (rule rcos_ring_hom_cring, rule is_cring)

```

11.5 Factorization over Maximal Ideals

In a commutative ring, the quotient ring over a maximal ideal is a field. The proof follows “W. Adkins, S. Weintraub: Algebra – An Approach via Module Theory”

```

lemma (in maximalideal) quotient_is_field:
  assumes "cring R"
  shows "field (R Quot I)"
proof -
  interpret cring R by fact

```

```

show ?thesis apply (intro cring.cring_fieldI2)
apply (rule quotient_is_cring, rule is_cring)
defer 1
apply (simp add: FactRing_def A_RCOSSETS_defs a_r_coset_def[symmetric],
clarsimp)
apply (simp add: rcoset_mult_add) defer 1
proof (rule ccontr, simp)
  — Quotient is not empty
  assume "0R Quot I = 1R Quot I"
  hence II1: "I = I +> 1" by (simp add: FactRing_def)
  from a_rcos_self[OF one_closed]
  have "1 ∈ I" by (simp add: II1[symmetric])
  hence "I = carrier R" by (rule one_imp_carrier)
  from this and I_notcarr
  show "False" by simp
next
  — Existence of Inverse
  fix a
  assume IanI: "I +> a ≠ I"
    and acarr: "a ∈ carrier R"

  — Helper ideal J
  def J ≡ "(carrier R #> a) <+> I :: 'a set"
  have idealJ: "ideal J R"
    apply (unfold J_def, rule add_ideals)
    apply (simp only: cgenideal_eq_rcos[symmetric], rule cgenideal_ideal,
rule acarr)
    apply (rule is_ideal)
    done

  — Showing J not smaller than I
  have IinJ: "I ⊆ J"
  proof (rule, simp add: J_def r_coset_def set_add_defs)
    fix x
    assume xI: "x ∈ I"
    have Zcarr: "0 ∈ carrier R" by fast
    from xI[THEN a_Hcarr] acarr
    have "x = 0 ⊗ a ⊕ x" by algebra

    from Zcarr and xI and this
    show "∃ xa ∈ carrier R. ∃ k ∈ I. x = xa ⊗ a ⊕ k" by fast
  qed

  — Showing J ≠ I
  have anI: "a ∉ I"
  proof (rule ccontr, simp)
    assume "a ∈ I"
    hence "I +> a = I" by (rule a_rcos_const)
    from this and IanI

```



```

    show "False" by simp
  qed

  have aJ: "a ∈ J"
  proof (simp add: J_def r_coset_def set_add_defs)
    from acarr
    have "a = 1 ⊗ a ⊕ 0" by algebra
    from one_closed and additive_subgroup.zero_closed[OF is_additive_subgroup]
  and this
    show "∃x∈carrier R. ∃k∈I. a = x ⊗ a ⊕ k" by fast
  qed

  from aJ and anI
  have JnI: "J ≠ I" by fast

  — Deducing J = carrier R because I is maximal
  from idealJ and IinJ
  have "J = I ∨ J = carrier R"
  proof (rule I_maximal, unfold J_def)
    have "carrier R #> a ⊆ carrier R"
      using subset_refl acarr
      by (rule r_coset_subset_G)
    from this and a_subset
    show "carrier R #> a <+> I ⊆ carrier R" by (rule set_add_closed)
  qed

  from this and JnI
  have Jcarr: "J = carrier R" by simp

  — Calculating an inverse for a
  from one_closed[folded Jcarr]
  have "∃r∈carrier R. ∃i∈I. 1 = r ⊗ a ⊕ i"
    by (simp add: J_def r_coset_def set_add_defs)
  from this
  obtain r i
    where rcarr: "r ∈ carrier R"
      and iI: "i ∈ I"
      and one: "1 = r ⊗ a ⊕ i"
    by fast
  from one and rcarr and acarr and iI[THEN a_Hcarr]
  have rail: "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)
  from this and rail
  have "a ⊗ r ∈ I +> 1" by simp
  from this 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
qed

end

```

```

theory IntRing
imports QuotRing Lattice Int Primes
begin

```

12 The Ring of Integers

12.1 Some properties of int

```

lemma dvd_imp_abseq:
  "[1 dvd k; k dvd 1] ⇒ abs 1 = abs (k::int)"
apply (subst abs_split, rule conjI)
apply (clarsimp, subst abs_split, rule conjI)
  apply (clarsimp)
  apply (cases "k=0", simp)
  apply (cases "1=0", simp)
  apply (simp add: zdvd_anti_sym)
apply clarsimp
apply (cases "k=0", simp)
apply (simp add: zdvd_anti_sym)
apply (clarsimp, subst abs_split, rule conjI)
  apply (clarsimp)
  apply (cases "1=0", simp)
  apply (simp add: zdvd_anti_sym)
apply (clarsimp)
apply (subgoal_tac "-1 = -k", simp)
apply (intro zdvd_anti_sym, simp+)
done

lemma abseq_imp_dvd:
  assumes a_lk: "abs 1 = abs (k::int)"
  shows "1 dvd k"
proof -
  from a_lk
  have "nat (abs 1) = nat (abs k)" by simp
  hence "nat (abs 1) dvd nat (abs k)" by simp
  hence "int (nat (abs 1)) dvd k" by (subst int_dvd_iff)
  hence "abs 1 dvd k" by simp
  thus "1 dvd k"

```

```

apply (unfold dvd_def, cases "l<0")
  defer 1 apply clarsimp
proof (clarsimp)
  fix k
  assume l0: "l < 0"
  have "- (l * k) = l * (-k)" by simp
  thus "∃ka. - (l * k) = l * ka" by fast
qed
qed

lemma dvds_eq_abseq:
  "(l dvd k ∧ k dvd l) = (abs l = abs (k::int))"
apply rule
  apply (simp add: dvds_imp_abseq)
apply (rule conjI)
  apply (simp add: abseq_imp_dvd)+
done

```

12.2 \mathbb{Z} : The Set of Integers as Algebraic Structure

```

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

lemma int_Zcarr [intro!, simp]:
  "k ∈ carrier ℤ"
  by (simp add: int_ring_def)

lemma int_is_cring:
  "cring ℤ"
unfolding int_ring_def
apply (rule cringI)
  apply (rule abelian_groupI, simp_all)
  defer 1
  apply (rule comm_monoidI, simp_all)
  apply (rule zadd_zmult_distrib)
apply (fast intro: zadd_zminus_inverse2)
done

```

12.3 Interpretations

Since definitions of derived operations are global, their interpretation needs to be done as early as possible — that is, with as few assumptions as possible.

```

interpretation int: monoid ℤ
  where "carrier ℤ = UNIV"
    and "mult ℤ x y = x * y"
    and "one ℤ = 1"
    and "pow ℤ x n = x^n"

```

```

proof -
  — Specification
  show "monoid  $\mathcal{Z}$ " proof qed (auto simp: int_ring_def)
  then interpret int: monoid  $\mathcal{Z}$  .

  — Carrier
  show "carrier  $\mathcal{Z}$  = UNIV" by (simp add: int_ring_def)

  — Operations
  { fix x y show "mult  $\mathcal{Z}$  x y = x * y" by (simp add: int_ring_def) }
  note mult = this
  show one: "one  $\mathcal{Z}$  = 1" by (simp add: int_ring_def)
  show "pow  $\mathcal{Z}$  x n = x^n" by (induct n) (simp, simp add: int_ring_def)+
qed

interpretation int: comm_monoid  $\mathcal{Z}$ 
  where "finprod  $\mathcal{Z}$  f A = (if finite A then setprod f A else undefined)"
proof -
  — Specification
  show "comm_monoid  $\mathcal{Z}$ " proof qed (auto simp: int_ring_def)
  then interpret int: comm_monoid  $\mathcal{Z}$  .

  — Operations
  { fix x y have "mult  $\mathcal{Z}$  x y = x * y" by (simp add: int_ring_def) }
  note mult = this
  have one: "one  $\mathcal{Z}$  = 1" by (simp add: int_ring_def)
  show "finprod  $\mathcal{Z}$  f A = (if finite A then setprod f A else undefined)"
  proof (cases "finite A")
    case True then show ?thesis proof induct
      case empty show ?case by (simp add: one)
    next
      case insert then show ?case by (simp add: Pi_def mult)
    qed
  next
    case False then show ?thesis by (simp add: finprod_def)
  qed
qed

interpretation int: abelian_monoid  $\mathcal{Z}$ 
  where "zero  $\mathcal{Z}$  = 0"
  and "add  $\mathcal{Z}$  x y = x + y"
  and "finsum  $\mathcal{Z}$  f A = (if finite A then setsum f A else undefined)"
proof -
  — Specification
  show "abelian_monoid  $\mathcal{Z}$ " proof qed (auto simp: int_ring_def)
  then interpret int: abelian_monoid  $\mathcal{Z}$  .

  — Operations
  { fix x y show "add  $\mathcal{Z}$  x y = x + y" by (simp add: int_ring_def) }

```

```

note add = this
show zero: "zero  $\mathcal{Z} = 0$ " by (simp add: int_ring_def)
show "finsum  $\mathcal{Z}$  f A = (if finite A then setsum f A else undefined)"
proof (cases "finite A")
  case True then show ?thesis proof induct
    case empty show ?case by (simp add: zero)
  next
    case insert then show ?case by (simp add: Pi_def add)
  qed
next
  case False then show ?thesis by (simp add: finsum_def finprod_def)
qed
qed

interpretation int: abelian_group  $\mathcal{Z}$ 
  where "a_inv  $\mathcal{Z}$  x = - x"
    and "a_minus  $\mathcal{Z}$  x y = x - y"
proof -
  — Specification
  show "abelian_group  $\mathcal{Z}$ "
  proof (rule abelian_groupI)
    show "!!x. x  $\in$  carrier  $\mathcal{Z} \implies \exists y : \text{carrier } \mathcal{Z}. y \oplus_{\mathcal{Z}} x = 0_{\mathcal{Z}}$ "
      by (simp add: int_ring_def) arith
    qed (auto simp: int_ring_def)
  then interpret int: abelian_group  $\mathcal{Z}$  .

  — Operations
  { fix x y have "add  $\mathcal{Z}$  x y = x + y" by (simp add: int_ring_def) }
  note add = this
  have zero: "zero  $\mathcal{Z} = 0$ " by (simp add: int_ring_def)
  { fix x
    have "add  $\mathcal{Z}$  (-x) x = zero  $\mathcal{Z}$ " by (simp add: add zero)
    then show "a_inv  $\mathcal{Z}$  x = - x" by (simp add: int.minus_equality) }
  note a_inv = this
  show "a_minus  $\mathcal{Z}$  x y = x - y" by (simp add: int.minus_eq add a_inv)
qed

interpretation int: "domain"  $\mathcal{Z}$ 
  proof qed (auto simp: int_ring_def left_distrib right_distrib)

Removal of occurrences of UNIV in interpretation result — experimental.

lemma UNIV:
  "x  $\in$  UNIV = True"
  "A  $\subseteq$  UNIV = True"
  "(ALL x : UNIV. P x) = (ALL x. P x)"
  "(EX x : UNIV. P x) = (EX x. P x)"
  "(True  $\dashrightarrow$  Q) = Q"
  "(True  $\implies$  PROP R) == PROP R"
  by simp_all

```

```

interpretation int :
  partial_order "(| carrier = UNIV::int set, eq = op =, le = op ≤ |)"
  where "carrier (| carrier = UNIV::int set, eq = op =, le = op ≤ |)
= UNIV"
    and "le (| carrier = UNIV::int set, eq = op =, le = op ≤ |) x y =
(x ≤ y)"
    and "lless (| carrier = UNIV::int set, eq = op =, le = op ≤ |) x
y = (x < y)"
  proof -
    show "partial_order (| carrier = UNIV::int set, eq = op =, le = op
≤ |)"
      proof qed simp_all
    show "carrier (| carrier = UNIV::int set, eq = op =, le = op ≤ |) =
UNIV"
      by simp
    show "le (| carrier = UNIV::int set, eq = op =, le = op ≤ |) x y =
(x ≤ y)"
      by simp
    show "lless (| carrier = UNIV::int set, eq = op =, le = op ≤ |) x y
= (x < y)"
      by (simp add: lless_def) auto
  qed

```

```

interpretation int :
  lattice "(| carrier = UNIV::int set, eq = op =, le = op ≤ |)"
  where "join (| carrier = UNIV::int set, eq = op =, le = op ≤ |) x y
= max x y"
    and "meet (| carrier = UNIV::int set, eq = op =, le = op ≤ |) x y
= min x y"
  proof -
    let ?Z = "(| carrier = UNIV::int set, eq = op =, le = op ≤ |)"
    show "lattice ?Z"
      apply unfold_locales
      apply (simp add: least_def Upper_def)
      apply arith
      apply (simp add: greatest_def Lower_def)
      apply arith
      done
    then interpret int: lattice "?Z" .
    show "join ?Z x y = max x y"
      apply (rule int.joinI)
      apply (simp_all add: least_def Upper_def)
      apply arith
      done
    show "meet ?Z x y = min x y"
      apply (rule int.meetI)
      apply (simp_all add: greatest_def Lower_def)
      apply arith

```

```

done
qed

```

```

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

```

12.4 Generated Ideals of \mathcal{Z}

```

lemma int_Idl:
  "Idl $\mathcal{Z}$  {a} = {x * a | x. True}"
  apply (subst int.cgenideal_eq_genideal[symmetric]) apply (simp add:
int_ring_def)
  apply (simp add: cgenideal_def int_ring_def)
  done

```

```

lemma multiples_principalideal:
  "principalideal {x * a | x. True }  $\mathcal{Z}$ "
  apply (subst int_Idl[symmetric], rule principalidealI)
  apply (rule int.genideal_ideal, simp)
  apply fast
  done

```

```

lemma prime_primeideal:
  assumes prime: "prime (nat p)"
  shows "primeideal (Idl $\mathcal{Z}$  {p})  $\mathcal{Z}$ "
  apply (rule primeidealI)
  apply (rule int.genideal_ideal, simp)
  apply (rule int_is_cring)
  apply (simp add: int.cgenideal_eq_genideal[symmetric] cgenideal_def)
  apply (simp add: int_ring_def)
  apply clarsimp defer 1
  apply (simp add: int.cgenideal_eq_genideal[symmetric] cgenideal_def)
  apply (simp add: int_ring_def)
  apply (elim exE)
proof -
  fix a b x

  from prime
  have ppos: "0 ≤ p" by (simp add: prime_def)
  have unnat: "!!x. nat p dvd nat (abs x) ==> p dvd x"
  proof -
    fix x
    assume "nat p dvd nat (abs x)"
    hence "int (nat p) dvd x" by (simp add: int_dvd_iff[symmetric])
    thus "p dvd x" by (simp add: ppos)
  qed

```

```

    assume "a * b = x * p"
    hence "p dvd a * b" by simp
    hence "nat p dvd nat (abs (a * b))" using ppos by (simp add: nat_dvd_iff)
    hence "nat p dvd (nat (abs a) * nat (abs b))" by (simp add: nat_abs_mult_distrib)
    hence "nat p dvd nat (abs a) | nat p dvd nat (abs b)" by (rule prime_dvd_mult[OF
prime])
    hence "p dvd a | p dvd b" by (fast intro: unnat)
    thus "(EX x. a = x * p) | (EX x. b = x * p)"
  proof
    assume "p dvd a"
    hence "EX x. a = p * x" by (simp add: dvd_def)
    from this obtain x
      where "a = p * x" by fast
    hence "a = x * p" by simp
    hence "EX x. a = x * p" by simp
    thus "(EX x. a = x * p) | (EX x. b = x * p)" ..
  next
    assume "p dvd b"
    hence "EX x. b = p * x" by (simp add: dvd_def)
    from this obtain x
      where "b = p * x" by fast
    hence "b = x * p" by simp
    hence "EX x. b = x * p" by simp
    thus "(EX x. a = x * p) | (EX x. b = x * p)" ..
  qed
next
  assume "UNIV = {uu. EX x. uu = x * p}"
  from this obtain x
    where "1 = x * p" by best
  from this [symmetric]
    have "p * x = 1" by (subst zmult_commute)
  hence "|p * x| = 1" by simp
  hence "|p| = 1" by (rule abs_zmult_eq_1)
  from this and prime
    show "False" by (simp add: prime_def)
qed

```

12.5 Ideals and Divisibility

lemma int_Idl_subset_ideal:

```

  "IdlZ {k} ⊆ IdlZ {l} = (k ∈ IdlZ {l})"
by (rule int.Idl_subset_ideal', simp+)

```

lemma Idl_subset_eq_dvd:

```

  "(IdlZ {k} ⊆ IdlZ {l}) = (l dvd k)"
apply (subst int_Idl_subset_ideal, subst int_Idl, simp)
apply (rule, clarify)
apply (simp add: dvd_def)

```



```

apply (simp add: dvd_def mult_ac)
done

lemma dvds_eq_Idl:
  "(l dvd k ∧ k dvd l) = (IdlZ {k} = IdlZ {l})"
proof -
  have a: "l dvd k = (IdlZ {k} ⊆ IdlZ {l})" by (rule Idl_subset_eq_dvd[symmetric])
  have b: "k dvd l = (IdlZ {l} ⊆ IdlZ {k})" by (rule Idl_subset_eq_dvd[symmetric])

  have "(l dvd k ∧ k dvd l) = ((IdlZ {k} ⊆ IdlZ {l}) ∧ (IdlZ {l} ⊆
IdlZ {k}))"
  by (subst a, subst b, simp)
  also have "((IdlZ {k} ⊆ IdlZ {l}) ∧ (IdlZ {l} ⊆ IdlZ {k})) = (IdlZ
{k} = IdlZ {l})" by (rule, fast+)
  finally
    show ?thesis .
qed

lemma Idl_eq_abs:
  "(IdlZ {k} = IdlZ {l}) = (abs l = abs k)"
apply (subst dvds_eq_abseq[symmetric])
apply (rule dvds_eq_Idl[symmetric])
done

```

12.6 Ideals and the Modulus

```

constdefs
  ZMod :: "int => int => int set"
  "ZMod k r == (IdlZ {k}) +>Z r"

lemmas ZMod_defs =
  ZMod_def genideal_def

lemma rcos_zfact:
  assumes kIl: "k ∈ ZMod l r"
  shows "EX x. k = x * l + r"
proof -
  from kIl[unfolded ZMod_def]
    have "∃ x1 ∈ IdlZ {l}. k = x1 + r" by (simp add: a_r_coset_defs int_ring_def)
  from this obtain x1
    where x1: "x1 ∈ IdlZ {l}"
    and k: "k = x1 + r"
    by auto
  from x1 obtain x
    where "x1 = x * l"
    by (simp add: int_Idl, fast)
  from k and this
    have "k = x * l + r" by simp
  thus "∃ 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 "IdlZ {m}"  $\mathcal{Z}$  by (rule int.genideal_ideal, fast)
  from zmods
    have "b  $\in$  ZMod m a"
    unfolding ZMod_def
    by (simp add: a_repr_independenceD)
  from this
    have "EX x. b = x * m + a" by (rule rcos_zfact)
  from this obtain x
    where "b = x * m + a"
    by fast

  hence "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" .
  thus "a mod m = b mod m" ..
qed

```

```

lemma ZMod_mod:
  shows "ZMod m a = ZMod m (a mod m)"
proof -
  interpret ideal "IdlZ {m}"  $\mathcal{Z}$  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)
    apply (simp add: int_ring_def)
  proof -
    have "a = m * (a div m) + (a mod m)" by (simp add: zmod_zdiv_equality)
    hence "a = (a div m) * m + (a mod m)" by simp
    thus " $\exists h. (\exists x. h = x * m) \wedge a = h + a \text{ mod } m$ " 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:
  shows "(ZMod m a = ZMod m b) = (a mod m = b mod m)"
by (rule, erule ZMod_imp_zmod, erule zmod_imp_ZMod)

```

12.7 Factorization

```

constdefs
  ZFact :: "int  $\Rightarrow$  int set ring"
  "ZFact k ==  $\mathbb{Z}$  Quot (Idl_ $\mathbb{Z}$  {k})"

lemmas ZFact_defs = ZFact_def FactRing_def

lemma ZFact_is_cring:
  shows "cring (ZFact k)"
apply (unfold ZFact_def)
apply (rule ideal.quotient_is_cring)
apply (intro ring.genideal_ideal)
apply (simp add: cring.axioms[OF int_is_cring] ring.intro)
apply simp
apply (rule int_is_cring)
done

lemma ZFact_zero:
  "carrier (ZFact 0) = ( $\bigcup$  a. {a})"
apply (insert int.genideal_zero)
apply (simp add: ZFact_defs A_RCOSSETS_defs r_coset_def int_ring_def ring_record_simps)
done

lemma ZFact_one:
  "carrier (ZFact 1) = {UNIV}"
apply (simp only: ZFact_defs A_RCOSSETS_defs r_coset_def int_ring_def ring_record_simps)
apply (subst int.genideal_one[unfolded int_ring_def, simplified ring_record_simps])
apply (rule, rule, clarsimp)
apply (rule, rule, clarsimp)
apply (rule, clarsimp, arith)
apply (rule, clarsimp)
apply (rule exI[of _ "0"], clarsimp)
done

lemma ZFact_prime_is_domain:
  assumes pprime: "prime (nat p)"
  shows "domain (ZFact p)"
apply (unfold ZFact_def)
apply (rule primeideal.quotient_is_domain)
apply (rule prime_primeideal[OF pprime])

```

done

end

theory Module imports Ring begin

13 Modules over an Abelian Group

13.1 Definitions

```

record ('a, 'b) module = "'b ring" +
  smult :: "'a, 'b] => 'b" (infixl "⊙" 70)

locale module = R: cring + M: abelian_group M for M (structure) +
  assumes smult_closed [simp, intro]:
    "[| a ∈ carrier R; x ∈ carrier M |] ==> a ⊙M x ∈ carrier M"
  and smult_l_distr:
    "[| a ∈ carrier R; b ∈ carrier R; x ∈ carrier M |] ==>
    (a ⊕ b) ⊙M x = a ⊙M x ⊕M b ⊙M x"
  and smult_r_distr:
    "[| a ∈ carrier R; x ∈ carrier M; y ∈ carrier M |] ==>
    a ⊙M (x ⊕M y) = a ⊙M x ⊕M a ⊙M y"
  and smult_assoc1:
    "[| a ∈ carrier R; b ∈ carrier R; x ∈ carrier M |] ==>
    (a ⊗ b) ⊙M x = a ⊙M (b ⊙M x)"
  and smult_one [simp]:
    "x ∈ carrier M ==> 1 ⊙M x = x"

locale algebra = module + cring M +
  assumes smult_assoc2:
    "[| a ∈ carrier R; x ∈ carrier M; y ∈ carrier M |] ==>
    (a ⊙M x) ⊗M y = a ⊙M (x ⊗M y)"

lemma moduleI:
  fixes R (structure) and M (structure)
  assumes cring: "cring R"
    and abelian_group: "abelian_group M"
    and smult_closed:
      "!!a x. [| a ∈ carrier R; x ∈ carrier M |] ==> a ⊙M x ∈ carrier
M"
  and smult_l_distr:
    "!!a b x. [| a ∈ carrier R; b ∈ carrier R; x ∈ carrier M |] ==>
    (a ⊕ b) ⊙M x = (a ⊙M x) ⊕M (b ⊙M x)"
  and smult_r_distr:
    "!!a x y. [| a ∈ carrier R; x ∈ carrier M; y ∈ carrier M |] ==>
    a ⊙M (x ⊕M y) = (a ⊙M x) ⊕M (a ⊙M y)"
  and smult_assoc1:

```

```

    "!!a b x. [| a ∈ carrier R; b ∈ carrier R; x ∈ carrier M |] ==>
      (a ⊗ b) ⊙M x = a ⊙M (b ⊙M x)"
  and smult_one:
    "!!x. x ∈ carrier M ==> 1 ⊙M x = x"
  shows "module R M"
  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)
```

13.2 Basic Properties of Algebras

```
lemma (in algebra) 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 ⊙M x ⊕M 0 ⊙M x) ⊕M ⊖M (0 ⊙M x)" by
  algebra
  also from M have "... = (0 ⊕ 0) ⊙M x ⊕M ⊖M (0 ⊙M x)"
  by (simp add: smult_l_distr del: R.l_zero R.r_zero)
  also from facts have "... = 0M" apply algebra apply algebra done
  finally show ?thesis .
qed
```

```
lemma (in algebra) 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 algebra
  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 algebra
  finally show ?thesis .
qed
```

```
lemma (in algebra) 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 algebra
  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)"
```

```

    apply algebra apply algebra done
  finally show ?thesis .
qed

lemma (in algebra) 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 algebra
  also from RM have "... = a ⊙M (⊖Mx ⊕M x) ⊕M ⊖M(a ⊙M x)"
    by (simp add: smult_r_distr)
  also from facts smult_r_null have "... = ⊖M(a ⊙M x)" by algebra
  finally show ?thesis .
qed

end

```

```

theory UnivPoly
imports Module RingHom
begin

```

14 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`.

14.1 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  $\neq$  z" shows "n  $\leq$  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_def: "up R == {f. f  $\in$  UNIV -> carrier R & ( $\exists$  n. bound 0R n f)}"

```

```

definition UP :: "('a, 'm) ring_scheme => ('a, nat => 'a) up_ring"
  where UP_def: "UP R == (|
    carrier = up R,
    mult = (%p:up R. %q:up R. %n.  $\bigoplus_{R} i \in \{..n\}. p\ i \otimes_R q\ (n-i)$ ),
    one = (%i. if i=0 then 1R else 0R),
    zero = (%i. 0R),
    add = (%p:up R. %q:up R. %i. p i  $\oplus_R$  q i),
    smult = (%a:carrier R. %p:up R. %i. a  $\otimes_R$  p i),
    monom = (%a:carrier R. %n i. if i=n then a else 0R),
    coeff = (%p:up R. %n. p n) |)"

```

Properties of the set of polynomials up.

```

lemma mem_upI [intro]:
  "[| !n. f n  $\in$  carrier R;  $\exists$  n. bound (zero R) n f |] ==> f  $\in$  up R"
  by (simp add: up_def Pi_def)

```

```

lemma mem_upD [dest]:
  "f  $\in$  up R ==> f n  $\in$  carrier R"
  by (simp add: up_def Pi_def)

```

```

context ring
begin

```

```

lemma bound_upD [dest]: "f  $\in$  up R ==>  $\exists$  n. bound 0 n f" by (simp add:
up_def)

```

```

lemma up_one_closed: "(%n. if n = 0 then 1 else 0)  $\in$  up R" using up_def
by force

```

```

lemma up_smult_closed: "[| a  $\in$  carrier R; p  $\in$  up R |] ==> (%i. a  $\otimes$  p
i)  $\in$  up R" by force

```



```

lemma up_add_closed:
  "[| p ∈ up R; q ∈ up R |] ==> (%i. 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 "EX n. bound 0 n (%i. 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) (%i. 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 fastsimp
    qed
    then show ?thesis ..
  qed
qed

lemma up_a_inv_closed:
  "p ∈ up R ==> (%i. ⊖ (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 (%i. ⊖ p i)" by auto
  then show "EX n. bound 0 n (%i. ⊖ p i)" by auto
qed auto

lemma up_minus_closed:
  "[| p ∈ up R; q ∈ up R |] ==> (%i. p i ⊖ q i) ∈ up R"
  using mem_upD [of p R] mem_upD [of q R] up_add_closed up_a_inv_closed
  a_minus_def [of _ R]
  by auto

lemma up_mult_closed:
  "[| p ∈ up R; q ∈ up R |] ==>
    (%n. ⊕ 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 "EX n. bound 0 n (%n. ⊕ 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) (%n.  $\bigoplus_{i \in \{..n\}} p\ i \otimes q\ (n - i)$ )"
  proof
    fix k assume bound: "n + m < k"
    {
      fix i
      have "p i  $\otimes$  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)  $\in$  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  $\in$  carrier R" by auto
            ultimately show ?thesis by simp
      qed
    }
    then show " $(\bigoplus_{i \in \{..k\}} p\ i \otimes q\ (k-i)) = 0$ "
      by (simp add: Pi_def)
  qed
  then show ?thesis by fast
qed
end

```

14.2 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  $P \equiv UP\ R$  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 "(%n. 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 0P n = 0" by (auto simp add: UP_def)

  lemma coeff_one [simp]: "coeff P 1P 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

```

14.3 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 "EX q : carrier P. q ⊕P p = 0P"
proof -
  let ?q = "%i. ⊖ (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
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

```

14.4 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
    {
      fix k and a b :: "nat=>'a"
      assume R: "a ∈ UNIV -> carrier R" "b ∈ UNIV -> carrier R"
      then have "k ≤ n ==>
        (⊕ i ∈ {...k}. a i ⊗ b (n-i)) = (⊕ i ∈ {...k}. a (k-i) ⊗ b (i+n-k))"
        (is "_ ==> ?eq k")
      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

```

```

    qed
  }
  note l = this
  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)

```

14.5 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 .

```

14.6 Polynomials Form an Algebra

```

context UP_ring
begin

```



```

lemma UP_smult_l_distr:
  "[| a ∈ carrier R; b ∈ carrier R; p ∈ carrier P |] ==>
   (a ⊕ b) ⊙P p = a ⊙P p ⊕ b ⊙P p"
  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 cring) cring:
  "cring R" ..

lemma (in UP_cring) UP_algebra:
  "algebra R P" by (auto intro!: algebraI R.cring 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 .

```

14.7 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: R.r_null 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)
      qed
    case False
  qed
qed

```

```

    also from True have "... = coeff P (monom P 1 n  $\otimes_P$  monom P 1 1)
k"
    by (simp add: ivl_disj_un_one)
    finally show ?thesis .
qed
next
case False
note neq = False
let ?s =
  "\lambda i. (if n = i then 1 else 0)  $\otimes$  (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 "... = ( $\bigoplus i \in \{..k\}. ?s i$ )"
proof -
  have f1: "( $\bigoplus i \in \{..<n\}. ?s i$ ) = 0"
  by (simp cong: R.finsum_cong add: Pi_def)
  from neq have f2: "( $\bigoplus i \in \{n\}. ?s i$ ) = 0"
  by (simp cong: R.finsum_cong add: Pi_def) arith
  have f3: "n < k ==> ( $\bigoplus i \in \{n<..k\}. ?s i$ ) = 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 = ( $\bigoplus i \in \{..<n\} \cup \{n\}. ?s i$ )"
      by (simp cong: R.finsum_cong add: ivl_disj_int_singleton Pi_def)
      also from True have "... = ( $\bigoplus i \in \{..k\}. ?s i$ )"
      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 = ( $\bigoplus i \in \{..<n\} \cup \{n\}. ?s i$ )"
      by (simp add: R.finsum_Un_disjoint f1 f2
        ivl_disj_int_singleton Pi_def del: Un_insert_right)
      also have "... = ( $\bigoplus i \in \{..n\}. ?s i$ )"
      by (simp only: ivl_disj_un_singleton)
      also from n_less_k neq have "... = ( $\bigoplus i \in \{..n\} \cup \{n<..k\}.
?s i$ )"
      by (simp add: R.finsum_Un_disjoint f3 ivl_disj_int_one Pi_def)
      also from n_less_k have "... = ( $\bigoplus i \in \{..k\}. ?s i$ )"
      by (simp only: ivl_disj_un_one)
      finally show ?thesis .
    qed
  qed
qed

```

```

    also have "... = coeff P (monom P 1 n  $\otimes_P$  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  $\otimes_P$  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  $\otimes_P$  monom P 1 k"
    then show "monom P 1 (Suc (Suc k)) = monom P 1 1  $\otimes_P$  monom P 1 (Suc
k)"
    proof -
      have lhs: "monom P 1 (Suc (Suc k)) = monom P 1 1  $\otimes_P$  monom P 1 k
 $\otimes_P$  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  $\otimes_P$  monom P 1 (Suc k) = monom P 1 1  $\otimes_P$  monom
P 1 k  $\otimes_P$  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 $\text{monom P 1 (Suc ?n) = monom P 1 ?n} \otimes_P \text{monom P 1 1}$ and $\text{monom P 1 (Suc ?n) = monom P 1 1} \otimes_P \text{monom P 1 ?n}$, and is trivial in UP_cring

```

corollary monom_one_comm: shows "monom P 1 k  $\otimes_P$  monom P 1 1 = monom P
1 1  $\otimes_P$  monom P 1 k"
  unfolding monom_one_Suc [symmetric] monom_one_Suc2 [symmetric] ..

```

```

lemma monom_mult_smult:
  "[| a  $\in$  carrier R; b  $\in$  carrier R |] ==> monom P (a  $\otimes$  b) n = a  $\odot_P$  monom
P b n"
  by (rule up_eqI) simp_all

```

```

lemma monom_one_mult:
  "monom P 1 (n + m) = monom P 1 n  $\otimes_P$  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  $\otimes$  monom P 1 m = monom P 1 m
 $\otimes$  monom P 1 n"
  unfolding monom_one_mult [symmetric] by (rule up_eqI) simp_all

lemma monom_mult [simp]:
  assumes a_in_R: "a  $\in$  carrier R" and b_in_R: "b  $\in$  carrier R"
  shows "monom P (a  $\otimes$  b) (n + m) = monom P a n  $\otimes$  monom P b m"
proof (rule up_eqI)
  fix k
  show "coeff P (monom P (a  $\otimes$  b) (n + m)) k = coeff P (monom P a n  $\otimes$ 
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}" "( $\lambda$ i. (if n
= i then a else 0)  $\otimes$  (if m = n + m - i then b else 0))"
        "( $\lambda$ i. if n = i then a  $\otimes$  b else 0)"]
      a_in_R b_in_R
      unfolding simp_implies_def
      using R.finsum_singleton [of n "{.. n + m}" "( $\lambda$ i. a  $\otimes$  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}" "( $\lambda$ i. (if n = i then a
else 0)  $\otimes$  (if m = k - i then b else 0))" "( $\lambda$ i. 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) simp_all

lemma monom_inj:
  "inj_on (%a. 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

```

14.8 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"
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 "EX m. deg R p - 1 < m & coeff P p m ~= 0"
    by (unfold bound_def) fast
    then have "EX 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 fastsimp
  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 "EX 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_anti_sym deg_aboveI deg_belowI)

```

Degree and polynomial operations

```

lemma deg_add [simp]:
  assumes R: "p ∈ carrier P" "q ∈ carrier P"
  shows "deg R (p ⊕P q) ≤ max (deg R p) (deg R q)"
proof (cases "deg R p ≤ deg R q")
  case True show ?thesis
    by (rule deg_aboveI) (simp_all add: True R deg_aboveD)
next
  case False show ?thesis
    by (rule deg_aboveI) (simp_all add: False R deg_aboveD)
qed

```

```

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 (fastsimp intro: le_anti_sym deg_aboveI deg_belowI)

```

```

lemma deg_const [simp]:
  assumes R: "a ∈ carrier R" shows "deg R (monom P a 0) = 0"
proof (rule le_anti_sym)
  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_anti_sym)
  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_anti_sym)
  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_anti_sym)
  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_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_anti_sym)
  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_anti_sym)
  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 = "(%i. 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 ∈ {..

```

```

    = 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 = "(%i. 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: ivl_disj_int_singleton 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: ivl_disj_int_singleton 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 = "(%i. 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

```

14.9 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  $\otimes_P$  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  $\otimes_P$  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 = ( $\bigoplus_{i \in \{..0\}}$  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 = ( $\bigoplus_{i \in \{..0\}}$  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  $\otimes$  coeff P q 0 = coeff P (p  $\otimes_P$  q) 0" by
  simp
  also from pq have "... = 0" by simp
  finally have "coeff P p 0  $\otimes$  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 fastsimp
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)+

```

14.10 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:
  "(%a. 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).
    ⊕ s i ∈ {..deg R p}. phi (coeff (UP R) p i) ⊗ s s (^)s i"

context UP
begin

lemma eval_on_carrier:
  fixes S (structure)
  shows "p ∈ carrier P ==>
    eval R S phi s p = (⊕ s i ∈ {..deg R p}. phi (coeff P p i) ⊗ s s (^)s
i)"
  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]:
  "[| finite A; f ∈ A -> carrier R |] ==>
  h (finsum R f A) = finsum S (h o f) A"
proof (induct set: finite)
  case empty then show ?case by simp
next
  case insert then show ?case by (simp add: Pi_def)
qed

```

```

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 "... =
          (⊕S i ∈ {..max (deg R p) (deg R q)}). h (coeff P p i) ⊗S s (^)S i)
⊕S
          (⊕S i ∈ {..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 le_maxI1 le_maxI2)
      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
          "(⊕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
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_rdist)
  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 "... =
    ( $\bigoplus_S i \in \{..1\}. h \text{ (coeff P (monom P 1 1) i) } \otimes_S s \text{ (}\wedge\text{)}_S i$ )"
    by (simp only: ivl_disj_un_one deg_monom_le R.one_closed)
  also have "... = s"
  proof (cases "s = 0_S")
    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 " $(\bigoplus_S i \in \{..deg R \text{ (monom P 1 1)}\}. h \text{ (coeff P (monom P 1 1) i) } \otimes_S s \text{ (}\wedge\text{)}_S i) = s$ " .
qed

end

```

Interpretation of ring homomorphism lemmas.

```

sublocale UP_univ_prop < ring_hom_cring P S Eval
  apply (unfold Eval_def)
  apply intro_locales
  apply (rule ring_hom_cring.axioms)
  apply (rule ring_hom_cring.intro)
  apply unfold_locales
  apply (rule eval_ring_hom)
  apply rule
done

```

```

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  $\otimes_S s \text{ (}\wedge\text{)}_S n$ "
proof -
  assume R: "r ∈ carrier R"
  from R have "Eval (monom P r n) = Eval (monom P r 0  $\otimes_P$  (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  $\otimes_S$  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  $\in$  carrier R" and S: "s  $\in$  carrier S"
  shows "eval R S h s (monom P r n) = h r  $\otimes_S$  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  $\in$  carrier R; p  $\in$  carrier P |] ==> Eval (r  $\odot_P$  p) = h r  $\otimes_S$  Eval p"
proof -
  assume R: "r  $\in$  carrier R" and P: "p  $\in$  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  $\in$  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  $\in$  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  $\in$  carrier R ==> Psi (monom P r 0) = h r"
    and P: "p  $\in$  carrier P" and S: "s  $\in$  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 ( $\bigoplus_{p \in \{..deg R p\}} \text{monom } P \text{ (coeff } P \text{ p } i) \text{ } 0 \otimes_P \text{monom } P \text{ } 1$ 
  1 ( $\wedge$ )p i)"
  by (simp add: up_repr P monom_mult [THEN sym] monom_pow del: monom_mult)
also
have "... =
  Psi ( $\bigoplus_{p \in \{..deg R p\}} \text{monom } P \text{ (coeff } P \text{ p } i) \text{ } 0 \otimes_P \text{monom } P \text{ } 1$ 
  1 ( $\wedge$ )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"
proof (rule ring_hom_cring.intro)
  show "ring_hom_cring_axioms P S Phi"
  by (rule ring_hom_cring_axioms.intro) (rule Phi)
qed unfold_locales

```

```

theorem UP_universal_property:
  assumes S: "s ∈ carrier S"
  shows "EX! Phi. Phi ∈ ring_hom P S ∩ extensional (carrier P) &
    Phi (monom P 1 1) = s &
    (ALL r : carrier R. Phi (monom P r 0) = h r)"
  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
  ( $\wedge$ ) (1::nat) = x"
  using nat_pow_Suc [of x 0] unfolding nat_pow_0 [of x] unfolding 1_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 ≠ 0p" shows "lcoeff p ≠ 0"
  using lcoeff_nonzero [OF p_not_zero p_in_R] .
```

14.11 The long division algorithm: some previous facts.

```
lemma coeff_minus [simp]:
  assumes p: "p ∈ carrier P" and q: "q ∈ carrier P" shows "coeff P (p
⊖ p q) n = coeff P p n ⊖ coeff P q n"
  unfolding a_minus_def [OF p q] unfolding coeff_add [OF p a_inv_closed
[OF q]] unfolding coeff_a_inv [OF q]
  using coeff_closed [OF p, of n] using coeff_closed [OF q, of n] by algebra
```

```
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 ⊙p f) ≤ deg R f"
  using deg_smult_ring [OF a_in_R f_in_P] by (cases "a = 0", auto)
```

```
lemma coeff_monom_mult: assumes R: "c ∈ carrier R" and P: "p ∈ carrier
P"
```

```
  shows "coeff P (monom P c n ⊗p p) (m + n) = c ⊗ (coeff P p m)"
proof -
  have "coeff P (monom P c n ⊗p p) (m + n) = (⊕ i ∈ {..m + n}. (if n =
i then c else 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 0) ⊗ coeff P p (m
+ n - i)) =
    (⊕ i ∈ {..m + n}. (if n = i then c ⊗ coeff P p (m + n - i) else 0))"
    using R.finsum_cong [of "{..m + n}" "{..m + n}" "(λi::nat. (if n
= i then c else 0) ⊗ coeff P p (m + n - i))"
    "(λi::nat. (if n = i then c ⊗ coeff P p (m + n - i) else 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

```

14.12 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)"
proof -
  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))"
  and ?lg = "lcoeff g"
  show ?thesis

proof (cases "deg R f < deg R g")
  case True

    have "?pred 0P f 0" using True by force
    then show ?thesis by fast

next
  case False then have deg_g_le_deg_f: "deg R g ≤ deg R f" by simp
  {

    from f_in_P deg_g_le_deg_f show ?thesis
    proof (induct n ≡ "deg R f" arbitrary: "f" rule: nat_less_induct)

```



```

fix n f
assume hypo: "∀m<n. ∀x. x ∈ carrier P →
  deg R g ≤ deg R x →
  m = deg R x →
  (∃q r (k::nat). q ∈ carrier P ∧ r ∈ carrier P ∧ lcoeff g (^)
k ⊙_P x = g ⊗_P q ⊕_P r & (r = 0_P | deg R r < deg R g))"
  and prem: "n = deg R f" and f_in_P [simp]: "f ∈ carrier P"
  and deg_g_le_deg_f: "deg R g ≤ deg R f"
  let ?k = "1::nat" and ?r = "(g ⊗_P (monom P (lcoeff f) (deg R f
- deg R g))) ⊕_P ⊖_P (lcoeff g ⊙_P f)"
  and ?q = "monom P (lcoeff f) (deg R f - deg R g)"
  show "∃ q r (k::nat). q ∈ carrier P ∧ r ∈ carrier P ∧ lcoeff
g (^) k ⊙_P f = g ⊗_P q ⊕_P r & (r = 0_P | deg R r < deg R g)"
  proof -

    have exist: "lcoeff g (^) ?k ⊙_P f = g ⊗_P ?q ⊕_P ⊖_P ?r"
      using minus_add
      using sym [OF a_assoc [of "g ⊗_P ?q" "⊖_P (g ⊗_P ?q)" "lcoeff
g ⊙_P f"]]
      using r_neg by auto
    show ?thesis
    proof (cases "deg R (⊖_P ?r) < deg R g")

      case True
      {
        show ?thesis
        proof (rule exI3 [of _ ?q "⊖_P ?r" ?k], intro conjI)
          show "lcoeff g (^) ?k ⊙_P f = g ⊗_P ?q ⊕_P ⊖_P ?r" using
exist by simp
          show "⊖_P ?r = 0_P ∨ deg R (⊖_P ?r) < deg R g" using True
by simp
          qed (simp_all)
        }
      next
      case False note n_deg_r_l_deg_g = False
      {
        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 "lcoeff g (^) (1::nat) ⊙_P f = g ⊗_P f ⊕_P 0_P"
              unfolding deg_g apply simp
              unfolding sym [OF monom_mult_is_smult [OF coeff_closed
[OF g_in_P, of 0] f_in_P]]
              using deg_zero_impl_monom [OF g_in_P deg_g] by simp
          }
        }
      }
    end
  end

```

```

    then show ?thesis using f_in_P by blast
  }
next
case False note deg_f_nzero = False
{

  have deg_remainder_l_f: "deg R ( $\ominus_P$  ?r) < n"
  proof -
    have "deg R ( $\ominus_P$  ?r) = deg R ?r" using deg_uminus [of
?r] by simp
    also have "... < deg R f"
    proof (rule deg_lcoeff_cancel)
      show "deg R ( $\ominus_P$  (lcoeff g  $\odot_P$  f))  $\leq$  deg R f"
      using deg_smult_ring [of "lcoeff g" f] using prem
      using lcoeff_nonzero2 [OF g_in_P g_not_zero] by simp
      show "deg R (g  $\otimes_P$  ?q)  $\leq$  deg R f"
      using monom_deg_mult [OF _ g_in_P, of f "lcoeff f"]
and deg_g_le_deg_f
      by simp
      show "coeff P (g  $\otimes_P$  ?q) (deg R f) =  $\ominus$  coeff P ( $\ominus_P$ 
(lcoeff g  $\odot_P$  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}"]
      "( $\lambda i$ . coeff P g i  $\otimes$  (if deg R f - deg R g = deg
R f - i then lcoeff f else 0))"
      "( $\lambda i$ . if deg R g = i then coeff P g i  $\otimes$  lcoeff
f else 0)"]
      using R.finsum_singleton [of "deg R g" "{.. deg R
f}" "( $\lambda i$ . coeff P g i  $\otimes$  lcoeff f)"]
      unfolding Pi_def using deg_g_le_deg_f by force
      qed (simp_all add: deg_f_nzero)
      finally show "deg R ( $\ominus_P$  ?r) < n" unfolding prem .
    qed
    moreover have " $\ominus_P$  ?r  $\in$  carrier P" by simp
    moreover obtain m where deg_rem_eq_m: "deg R ( $\ominus_P$  ?r)
= m" by auto
    moreover have "deg R g  $\leq$  deg R ( $\ominus_P$  ?r)" using n_deg_r_l_deg_g
by simp

    ultimately obtain q' r' k'
    where rem_desc: "lcoeff g (^) (k'::nat)  $\odot_P$  ( $\ominus_P$  ?r)
= g  $\otimes_P$  q'  $\oplus_P$  r'" and rem_deg: "(r' = 0p  $\vee$  deg R r' < deg R g)"
    and q'_in_carrier: "q'  $\in$  carrier P" and r'_in_carrier:
"r'  $\in$  carrier P"

```

```

using hypo by blast

show ?thesis
proof (rule exI3 [of _ "((lcoeff g (^) k')) ⊙_p ?q ⊕_p
q')" r' "Suc k'"], intro conjI)
  show "(lcoeff g (^) (Suc k')) ⊙_p f = g ⊗_p ((lcoeff
g (^) k') ⊙_p ?q ⊕_p q') ⊕_p r'"
  proof -
    have "(lcoeff g (^) (Suc k')) ⊙_p f = (lcoeff g (^)
k') ⊙_p (g ⊗_p ?q ⊕_p ⊖_p ?r)"
    using smult_assoc1 exist by simp
    also have "... = (lcoeff g (^) k') ⊙_p (g ⊗_p ?q) ⊕_p
((lcoeff g (^) k') ⊙_p (⊖_p ?r))"
    using UP_smult_r_distr by simp
    also have "... = (lcoeff g (^) k') ⊙_p (g ⊗_p ?q) ⊕_p
(g ⊗_p q' ⊕_p r')"
    using rem_desc by simp
    also have "... = (lcoeff g (^) k') ⊙_p (g ⊗_p ?q) ⊕_p
g ⊗_p q' ⊕_p r'"
    using sym [OF a_assoc [of "lcoeff g (^) k' ⊙_p (g
⊗_p ?q)" "g ⊗_p q'" "r'"]]
    using q'_in_carrier r'_in_carrier by simp
    also have "... = (lcoeff g (^) k') ⊙_p (?q ⊗_p g) ⊕_p
q' ⊗_p g ⊕_p r'"
    using q'_in_carrier by (auto simp add: m_comm)
    also have "... = (((lcoeff g (^) k') ⊙_p ?q) ⊗_p g)
⊕_p q' ⊗_p g ⊕_p r'"
    using smult_assoc2 q'_in_carrier by auto
    also have "... = ((lcoeff g (^) 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
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 1R 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 1R 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 ≠ 0P"
  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 1R 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  $\neq$  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  $\in$  carrier R"
  shows "eval R R id a (monom P 1R 1  $\ominus_P$  monom P a 0) = 0"
  (is "eval R R id a ?g = _")
proof -
  interpret UP_pre_univ_prop R R id proof qed simp
  have eval_ring_hom: "eval R R id a  $\in$  ring_hom P R" using eval_ring_hom
[OF a] by simp
  interpret ring_hom_cring P R "eval R R id a" proof qed (simp add: eval_ring_hom)
  have mon1_closed: "monom P 1R 1  $\in$  carrier P"
    and mon0_closed: "monom P a 0  $\in$  carrier P"
    and min_mon0_closed: " $\ominus_P$  monom P a 0  $\in$  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)  $\ominus$  eval R R id
a (monom P a 0)"
    unfolding P.minus_eq [OF mon1_closed mon0_closed]
    unfolding hom_add [OF mon1_closed min_mon0_closed]
    unfolding hom_a_inv [OF mon0_closed]
    using R.minus_eq [symmetric] mon1_closed mon0_closed by auto
  also have "... = a  $\ominus$  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  $\in$  carrier P" and a: "a  $\in$  carrier R"
  and R_not_trivial: "carrier R  $\neq$  {0}"
  shows " $\exists$  q r. (q  $\in$  carrier P)  $\wedge$  (r  $\in$  carrier P)  $\wedge$  f = (monom P 1R
1  $\ominus_P$  monom P a 0)  $\otimes_P$  q  $\oplus_P$  r  $\wedge$  (deg R r = 0)"
  (is " $\exists$  q r. (q  $\in$  carrier P)  $\wedge$  (r  $\in$  carrier P)  $\wedge$  f = ?g  $\otimes_P$  q  $\oplus_P$  r  $\wedge$ 
(deg R r = 0)")
proof -
  let ?g = "monom P 1R 1  $\ominus_P$  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 (ˆ) 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 proof qed 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

```

14.13 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_def: "INTEG == (| carrier = UNIV, mult = op *, one = 1,
zero = 0, add = op + |)"

lemma INTEG_cring:
  "cring INTEG"
  by (unfold INTEG_def) (auto intro!: cringI abelian_groupI comm_monoidI
    zadd_zminus_inverse2 zadd_zmult_distrib)

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
```

References

- [1] C. Ballarin. *Computer Algebra and Theorem Proving*. PhD thesis, University of Cambridge, 1999. <http://www4.in.tum.de/~ballarin/publications.html>.
- [2] N. Jacobson. *Basic Algebra I*. Freeman, 1985.
- [3] F. Kammüller and L. C. Paulson. A formal proof of sylow's theorem: An experiment in abstract algebra with Isabelle HOL. *J. Automated Reasoning*, (23):235–264, 1999.