Various results of number theory

April 17, 2016

Contents

1 Primes **5**
- 1.1 Primes . . . 5
  - 1.1.1 Make prime naively executable . . . 6
- 1.2 Infinitely many primes . . . 8
- 1.3 Powers of Primes . . . 9
- 1.4 Chinese Remainder Theorem Variants . . . 11

2 Congruence **13**
- 2.1 Turn off *One-nat-def* . . . 13
- 2.2 Main definitions . . . 13
- 2.3 Set up Transfer . . . 14
- 2.4 Congruence . . . 14

3 Unique factorization for the natural numbers and the integers **30**
- 3.1 Unique factorization: multiset version . . . 30
- 3.2 Prime factors and multiplicity for nat and int . . . 33
- 3.3 Set up transfer . . . 34
- 3.4 Properties of prime factors and multiplicity for nat and int . 34
- 3.5 An application . . . 44

4 Things that can be added to the Algebra library **47**
- 4.1 Finiteness stuff . . . 47
- 4.2 The rest is for the algebra libraries . . . 47
  - 4.2.1 These go in Group.thy . . . 47
  - 4.2.2 Miscellaneous . . . 50
  - 4.2.3 This goes in FiniteProduct . . . 51

5 Residue rings **53**
- 5.1 A locale for residue rings . . . 53
- 5.2 Prime residues . . . 57

6 Test cases: Euler's theorem and Wilson's theorem **58**
6.1 Euler's theorem . . . 58
6.2 Wilson's theorem . . . 60

7 Pocklington's Theorem for Primes **62**
7.1 Lemmas about previously defined terms . . . 62
7.2 Some basic theorems about solving congruences . . . 63
7.3 Lucas's theorem . . . 64
7.4 Definition of the order of a number mod n (0 in non-coprime case) . . . 68
7.5 Another trivial primality characterization . . . 71
7.6 Pocklington theorem . . . 73
7.7 Prime factorizations . . . 75

8 Gauss' Lemma **78**
8.1 Basic properties of p . . . 79
8.2 Basic Properties of the Gauss Sets . . . 79
8.3 Relationships Between Gauss Sets . . . 82
8.4 Gauss' Lemma . . . 85

9 The fibonacci function **86**
9.1 Fibonacci numbers . . . 86
9.2 Basic Properties . . . 86
9.3 A Few Elementary Results . . . 86
9.4 Law 6.111 of Concrete Mathematics . . . 87
9.5 Fibonacci and Binomial Coefficients . . . 88

10 The sieve of Eratosthenes **89**
10.1 Preliminary: strict divisibility . . . 89
10.2 Main corpus . . . 89
10.3 Application: smallest prime beyond a certain number . . . 95

11 Comprehensive number theory **96**

12 Less common functions on lists **97**

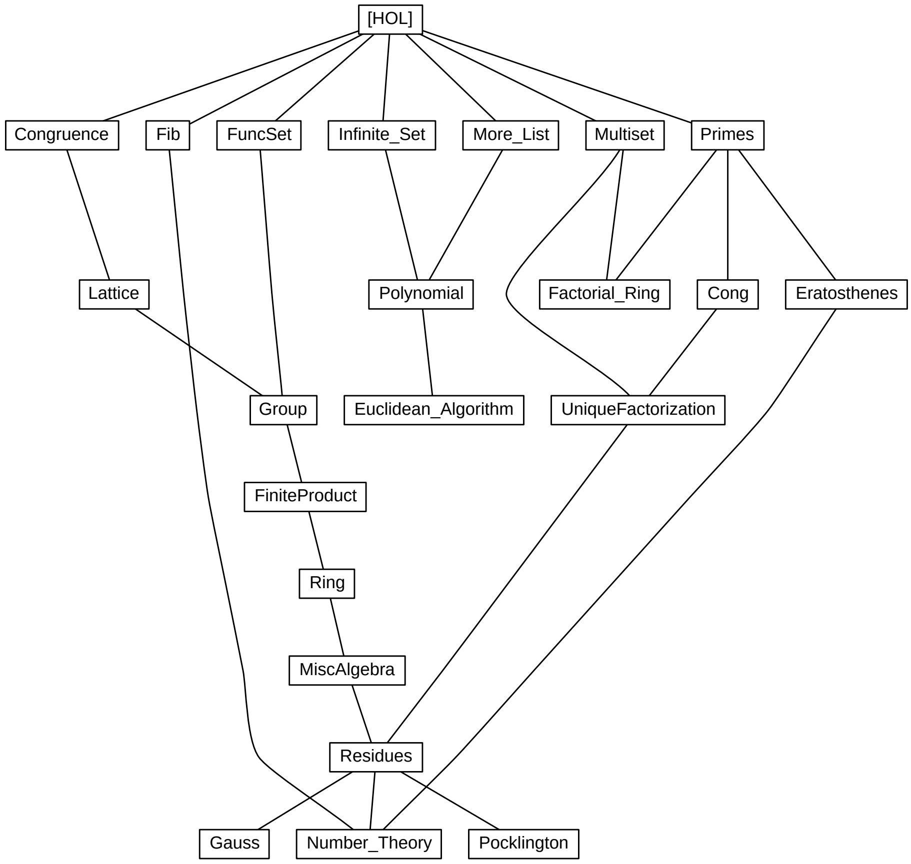
13 Infinite Sets and Related Concepts **104**
13.1 Infinitely Many and Almost All . . . 105
13.2 Enumeration of an Infinite Set . . . 107

14 Polynomials as type over a ring structure **110**
14.1 Auxiliary: operations for lists (later) representing coefficients 110
14.2 Definition of type *poly* . . . 111
14.3 Degree of a polynomial . . . 111
14.4 The zero polynomial . . . 112

14.5 List-style constructor for polynomials . . . 113
14.6 Quickcheck generator for polynomials . . . 115
14.7 List-style syntax for polynomials . . . 115
14.8 Representation of polynomials by lists of coefficients . . . 115
14.9 Fold combinator for polynomials . . . 119
14.10 Canonical morphism on polynomials – evaluation . . . 119
14.11 Monomials . . . 120
14.12 Addition and subtraction . . . 121
14.13 Multiplication by a constant, polynomial multiplication and the unit polynomial . . . 126
14.14 Conversions from natural numbers . . . 130
14.15 Lemmas about divisibility . . . 131
14.16 Polynomials form an integral domain . . . 132
14.17 Polynomials form an ordered integral domain . . . 133
14.18 Synthetic division and polynomial roots . . . 135
14.19 Long division of polynomials . . . 137
14.20 Order of polynomial roots . . . 147
14.21 Additional induction rules on polynomials . . . 149
14.22 Composition of polynomials . . . 150
14.23 Leading coefficient . . . 153
14.24 Derivatives of univariate polynomials . . . 155
14.25 Algebraic numbers . . . 161
14.26 Algebraic numbers . . . 161

15 Abstract euclidean algorithm **168**
15.1 Typical instances . . . 177

16 Factorial (semi)rings **180**


[HOL]
Congruence
Fib
FuncSet
Infinite_Set
More_List
Multiset
Primes
Lattice
Polynomial
Factorial_Ring
Cong
Eratosthenes
Group
Euclidean_Algorithm
UniqueFactorization
FiniteProduct
Ring
MiscAlgebra
Residues
Gauss
Number_Theory
Pocklington

1 Primes

theory *Primes*
imports *~~/src/HOL/GCD ~~/src/HOL/Binomial*
begin

declare [[*coercion int*]]
declare [[*coercion-enabled*]]

definition *prime* :: *nat* ⇒ *bool*
  where *prime p* = (*1* < *p* ∧ (∀ *m*. *m dvd p* ⟶ *m* = *1* ∨ *m* = *p*))

1.1 Primes

lemma *prime-odd-nat*: *prime p* ⟹ *p* > *2* ⟹ *odd p*
  using *nat-dvd-1-iff-1 odd-one prime-def* **by** *blast*

lemma *prime-gt-0-nat*: *prime p* ⟹ *p* > *0*
  unfolding *prime-def* **by** *auto*

lemma *prime-ge-1-nat*: *prime p* ⟹ *p* >= *1*
  unfolding *prime-def* **by** *auto*

lemma *prime-gt-1-nat*: *prime p* ⟹ *p* > *1*
  unfolding *prime-def* **by** *auto*

lemma *prime-ge-Suc-0-nat*: *prime p* ⟹ *p* >= *Suc 0*
  unfolding *prime-def* **by** *auto*

lemma *prime-gt-Suc-0-nat*: *prime p* ⟹ *p* > *Suc 0*
  unfolding *prime-def* **by** *auto*

lemma *prime-ge-2-nat*: *prime p* ⟹ *p* >= *2*
  unfolding *prime-def* **by** *auto*

lemma *prime-imp-coprime-nat*: *prime p* ⟹ ¬ *p dvd n* ⟹ *coprime p n*
  apply (*unfold prime-def*)
  apply (*metis gcd-dvd1 gcd-dvd2*)
  done

lemma *prime-int-altdef*:
  prime p = (*1* < *p* ∧ (∀ *m*::*int*. *m* ≥ *0* ⟶ *m dvd p* ⟶
    m = *1* ∨ *m* = *p*))
  apply (*simp add*: *prime-def*)
  apply (*auto simp add*:)
  apply (*metis One-nat-def of-nat-1 nat-0-le nat-dvd-iff*)
  apply (*metis zdvd-int One-nat-def le0 of-nat-0 of-nat-1 of-nat-eq-iff of-nat-le-iff*)
  done

lemma *prime-imp-coprime-int*:

fixes *n*::*int* **shows** *prime p* $\Longrightarrow$ $\neg$ *p dvd n* $\Longrightarrow$ *coprime p n*
apply (*unfold prime-int-altdef*)
apply (*metis gcd-dvd1 gcd-dvd2 gcd-ge-0-int*)
done

lemma *prime-dvd-mult-nat*: *prime p* $\Longrightarrow$ *p dvd m* $*$ *n* $\Longrightarrow$ *p dvd m* $\vee$ *p dvd n*
by (*blast intro*: *coprime-dvd-mult prime-imp-coprime-nat*)

lemma *prime-dvd-mult-int*:
fixes *n*::*int* **shows** *prime p* $\Longrightarrow$ *p dvd m* $*$ *n* $\Longrightarrow$ *p dvd m* $\vee$ *p dvd n*
by (*blast intro*: *coprime-dvd-mult prime-imp-coprime-int*)

lemma *prime-dvd-mult-eq-nat* [*simp*]: *prime p* $\Longrightarrow$
p dvd m $*$ *n* = (*p dvd m* $\vee$ *p dvd n*)
by (*rule iffI*, *rule prime-dvd-mult-nat*, *auto*)

lemma *prime-dvd-mult-eq-int* [*simp*]:
fixes *n*::*int*
shows *prime p* $\Longrightarrow$ *p dvd m* $*$ *n* = (*p dvd m* $\vee$ *p dvd n*)
by (*rule iffI*, *rule prime-dvd-mult-int*, *auto*)

lemma *not-prime-eq-prod-nat*:
1 < *n* $\Longrightarrow$ $\neg$ *prime n* $\Longrightarrow$
$\exists$ *m k*. *n* = *m* $*$ *k* $\wedge$ *1* < *m* $\wedge$ *m* < *n* $\wedge$ *1* < *k* $\wedge$ *k* < *n*
unfolding *prime-def dvd-def* **apply** (*auto simp add*: *ac-simps*)
by (*metis Suc-lessD Suc-lessI n-less-m-mult-n n-less-n-mult-m nat-0-less-mult-iff*)

lemma *prime-dvd-power-nat*: *prime p* $\Longrightarrow$ *p dvd x*^*n* $\Longrightarrow$ *p dvd x*
by (*induct n*) *auto*

lemma *prime-dvd-power-int*:
fixes *x*::*int* **shows** *prime p* $\Longrightarrow$ *p dvd x*^*n* $\Longrightarrow$ *p dvd x*
by (*induct n*) *auto*

lemma *prime-dvd-power-nat-iff*: *prime p* $\Longrightarrow$ *n* > *0* $\Longrightarrow$
p dvd x^*n* $\longleftrightarrow$ *p dvd x*
by (*cases n*) (*auto elim*: *prime-dvd-power-nat*)

lemma *prime-dvd-power-int-iff*:
fixes *x*::*int*
shows *prime p* $\Longrightarrow$ *n* > *0* $\Longrightarrow$ *p dvd x*^*n* $\longleftrightarrow$ *p dvd x*
by (*cases n*) (*auto elim*: *prime-dvd-power-int*)

1.1.1 Make prime naively executable

lemma *zero-not-prime-nat* [*simp*]: ~*prime* (*0*::*nat*)
by (*simp add*: *prime-def*)

lemma *one-not-prime-nat* [*simp*]: ~*prime* (*1*::*nat*)

by (*simp add*: *prime-def*)

lemma *Suc-0-not-prime-nat* [*simp*]: ~*prime* (*Suc 0*)
by (*simp add*: *prime-def*)

lemma *prime-nat-code* [*code*]:
prime p ⟷ *p* > *1* ∧ (∀ *n* ∈ {*1*<..<*p*}. ~ *n dvd p*)
apply (*simp add*: *Ball-def*)
apply (*metis One-nat-def less-not-refl prime-def dvd-triv-right not-prime-eq-prod-nat*)
done

lemma *prime-nat-simp*:
prime p ⟷ *p* > *1* ∧ (∀ *n* ∈ *set* [*2*..<*p*]. ¬ *n dvd p*)
by (*auto simp add*: *prime-nat-code*)

lemmas *prime-nat-simp-numeral* [*simp*] = *prime-nat-simp* [*of numeral m*] **for** *m*

lemma *two-is-prime-nat* [*simp*]: *prime* (*2*::*nat*)
by *simp*

A bit of regression testing:

lemma *prime*(*97*::*nat*) **by** *simp*
lemma *prime*(*997*::*nat*) **by** *eval*

lemma *prime-imp-power-coprime-nat*: *prime p* ⟹ ~ *p dvd a* ⟹ *coprime a* (*p*^*m*)
by (*metis coprime-exp gcd.commute prime-imp-coprime-nat*)

lemma *prime-imp-power-coprime-int*:
fixes *a*::*int* **shows** *prime p* ⟹ ~ *p dvd a* ⟹ *coprime a* (*p*^*m*)
by (*metis coprime-exp gcd.commute prime-imp-coprime-int*)

lemma *primes-coprime-nat*: *prime p* ⟹ *prime q* ⟹ *p* ≠ *q* ⟹ *coprime p q*
by (*metis gcd-nat.absorb1 gcd-nat.absorb2 prime-imp-coprime-nat*)

lemma *primes-imp-powers-coprime-nat*:
prime p ⟹ *prime q* ⟹ *p* ~= *q* ⟹ *coprime* (*p*^*m*) (*q*^*n*)
by (*rule coprime-exp2-nat*, *rule primes-coprime-nat*)

lemma *prime-factor-nat*:
n ≠ (*1*::*nat*) ⟹ ∃ *p*. *prime p* ∧ *p dvd n*
proof (*induct n rule*: *nat-less-induct*)
case (*1 n*) **show** *?case*
proof (*cases n* = *0*)
case *True* **then show** *?thesis*
by (*auto intro*: *two-is-prime-nat*)
next
case *False* **with** *1.prems* **have** *n* > *1* **by** *simp*
with *1.hyps* **show** *?thesis*

```
    by (metis One-nat-def dvd-mult dvd-refl not-prime-eq-prod-nat order-less-irrefl)
  qed
qed
```

One property of coprimality is easier to prove via prime factors.

```
lemma prime-divprod-pow-nat:
  assumes p: prime p and ab: coprime a b and pab: p^n dvd a * b
  shows p^n dvd a ∨ p^n dvd b
proof−
  { assume n = 0 ∨ a = 1 ∨ b = 1 with pab have ?thesis
      apply (cases n=0, simp-all)
      apply (cases a=1, simp-all)
      done }
  moreover
  { assume n: n ≠ 0 and a: a≠1 and b: b≠1
    then obtain m where m: n = Suc m by (cases n) auto
    from n have p dvd p^n apply (intro dvd-power) apply auto done
    also note pab
    finally have pab': p dvd a * b.
    from prime-dvd-mult-nat[OF p pab']
    have p dvd a ∨ p dvd b .
    moreover
    { assume pa: p dvd a
     from coprime-common-divisor-nat [OF ab, OF pa] p have ¬ p dvd b by auto
      with p have coprime b p
        by (subst gcd.commute, intro prime-imp-coprime-nat)
      then have pnb: coprime (p^n) b
        by (subst gcd.commute, rule coprime-exp)
      from coprime-dvd-mult[OF pnb pab] have ?thesis by blast }
    moreover
    { assume pb: p dvd b
      have pnba: p^n dvd b*a using pab by (simp add: mult.commute)
      from coprime-common-divisor-nat [OF ab, of p] pb p have ¬ p dvd a
        by auto
      with p have coprime a p
        by (subst gcd.commute, intro prime-imp-coprime-nat)
      then have pna: coprime (p^n) a
        by (subst gcd.commute, rule coprime-exp)
      from coprime-dvd-mult[OF pna pnba] have ?thesis by blast }
    ultimately have ?thesis by blast }
  ultimately show ?thesis by blast
qed
```

1.2 Infinitely many primes

```
lemma next-prime-bound: ∃ p. prime p ∧ n < p ∧ p <= fact n + 1
proof−
  have f1: fact n + 1 ≠ (1::nat) using fact-ge-1 [of n, where 'a=nat] by arith
  from prime-factor-nat [OF f1]
```

```
  obtain p where prime p and p dvd fact n + 1 by auto
  then have p ≤ fact n + 1 apply (intro dvd-imp-le) apply auto done
  { assume p ≤ n
    from ‹prime p› have p ≥ 1
      by (cases p, simp-all)
    with ‹p <= n› have p dvd fact n
      by (intro dvd-fact)
    with ‹p dvd fact n + 1› have p dvd fact n + 1 − fact n
      by (rule dvd-diff-nat)
    then have p dvd 1 by simp
    then have p <= 1 by auto
    moreover from ‹prime p› have p > 1
      using prime-def by blast
    ultimately have False by auto}
  then have n < p by presburger
  with ‹prime p› and ‹p <= fact n + 1› show ?thesis by auto
qed

lemma bigger-prime: ∃ p. prime p ∧ p > (n::nat)
  using next-prime-bound by auto

lemma primes-infinite: ¬ (finite {(p::nat). prime p})
proof
  assume finite {(p::nat). prime p}
  with Max-ge have (EX b. (ALL x : {(p::nat). prime p}. x <= b))
    by auto
  then obtain b where ALL (x::nat). prime x ⟶ x <= b
    by auto
  with bigger-prime [of b] show False
    by auto
qed
```

1.3 Powers of Primes

Versions for type nat only

```
lemma prime-product:
  fixes p::nat
  assumes prime (p ∗ q)
  shows p = 1 ∨ q = 1
proof −
  from assms have
    1 < p ∗ q and P: ⋀m. m dvd p ∗ q ⟹ m = 1 ∨ m = p ∗ q
    unfolding prime-def by auto
  from ‹1 < p ∗ q› have p ≠ 0 by (cases p) auto
  then have Q: p = p ∗ q ⟷ q = 1 by auto
  have p dvd p ∗ q by simp
  then have p = 1 ∨ p = p ∗ q by (rule P)
  then show ?thesis by (simp add: Q)
qed
```

```
lemma prime-exp:
  fixes p::nat
  shows prime (p^n) ⟷ prime p ∧ n = 1
proof(induct n)
  case 0 thus ?case by simp
next
  case (Suc n)
  {assume p = 0 hence ?case by simp}
  moreover
  {assume p=1 hence ?case by simp}
  moreover
  {assume p: p ≠ 0 p≠1
    {assume pp: prime (p^Suc n)
      hence p = 1 ∨ p^n = 1 using prime-product[of p p^n] by simp
      with p have n: n = 0
        by (metis One-nat-def nat-power-eq-Suc-0-iff)
      with pp have prime p ∧ Suc n = 1 by simp}
    moreover
    {assume n: prime p ∧ Suc n = 1 hence prime (p^Suc n) by simp}
    ultimately have ?case by blast}
  ultimately show ?case by blast
qed

lemma prime-power-mult:
  fixes p::nat
  assumes p: prime p and xy: x * y = p ^ k
  shows ∃ i j. x = p ^i ∧ y = p^ j
using xy
proof(induct k arbitrary: x y)
  case 0 thus ?case apply simp by (rule exI[where x=0], simp)
next
  case (Suc k x y)
  from Suc.prems have pxy: p dvd x*y by auto
  from Primes.prime-dvd-mult-nat [OF p pxy] have pxyc: p dvd x ∨ p dvd y .
  from p have p0: p ≠ 0 by − (rule ccontr, simp)
  {assume px: p dvd x
    then obtain d where d: x = p*d unfolding dvd-def by blast
    from Suc.prems d  have p*d*y = p^Suc k by simp
    hence th: d*y = p^k using p0 by simp
    from Suc.hyps[OF th] obtain i j where ij: d = p^i y = p^j by blast
    with d have x = p^Suc i by simp
    with ij(2) have ?case by blast}
  moreover
  {assume px: p dvd y
    then obtain d where d: y = p*d unfolding dvd-def by blast
    from Suc.prems d  have p*d*x = p^Suc k by (simp add: mult.commute)
    hence th: d*x = p^k using p0 by simp
    from Suc.hyps[OF th] obtain i j where ij: d = p^i x = p^j by blast
```

```
    with d have y = p ^Suc i by simp
    with ij(2) have ?case by blast}
  ultimately show ?case using pxyc by blast
qed

lemma prime-power-exp:
  fixes p::nat
  assumes p: prime p and n: n ≠ 0
    and xn: x ^n = p ^k shows ∃ i. x = p ^i
  using n xn
proof(induct n arbitrary: k)
  case 0 thus ?case by simp
next
  case (Suc n k) hence th: x*x ^n = p ^k by simp
  {assume n = 0 with Suc have ?case by simp (rule exI[where x=k], simp)}
  moreover
  {assume n: n ≠ 0
    from prime-power-mult[OF p th]
    obtain i j where ij: x = p ^i x ^n = p ^jby blast
    from Suc.hyps[OF n ij(2)] have ?case .}
  ultimately show ?case by blast
qed

lemma divides-primepow:
  fixes p::nat
  assumes p: prime p
  shows d dvd p ^k ⟷ (∃ i. i ≤ k ∧ d = p ^i)
proof
  assume H: d dvd p ^k then obtain e where e: d*e = p ^k
    unfolding dvd-def apply (auto simp add: mult.commute) by blast
  from prime-power-mult[OF p e] obtain i j where ij: d = p ^i e=p ^j by blast
  from e ij have p ^(i + j) = p ^k by (simp add: power-add)
  hence i + j = k using p prime-gt-1-nat power-inject-exp[of p i+j k] by simp
  hence i ≤ k by arith
  with ij(1) show ∃ i≤k. d = p ^ i by blast
next
  {fix i assume H: i ≤ k d = p ^i
    then obtain j where j: k = i + j
      by (metis le-add-diff-inverse)
    hence p ^k = p ^j*d using H(2) by (simp add: power-add)
    hence d dvd p ^k unfolding dvd-def by auto}
  thus ∃ i≤k. d = p ^ i ⟹ d dvd p ^ k by blast
qed
```

1.4 Chinese Remainder Theorem Variants

```
lemma bezout-gcd-nat:
  fixes a::nat shows ∃ x y. a * x − b * y = gcd a b ∨ b * x − a * y = gcd a b
  using bezout-nat[of a b]
```

```
by (metis bezout-nat diff-add-inverse gcd-add-mult gcd.commute
  gcd-nat.right-neutral mult-0)

lemma gcd-bezout-sum-nat:
  fixes a::nat
  assumes a ∗ x + b ∗ y = d
  shows gcd a b dvd d
proof−
  let ?g = gcd a b
    have dv: ?g dvd a∗x ?g dvd b ∗ y
      by simp-all
    from dvd-add[OF dv] assms
    show ?thesis by auto
qed
```

A binary form of the Chinese Remainder Theorem.

```
lemma chinese-remainder:
  fixes a::nat  assumes ab: coprime a b and a: a ≠ 0 and b: b ≠ 0
  shows ∃ x q1 q2. x = u + q1 ∗ a ∧ x = v + q2 ∗ b
proof−
  from bezout-add-strong-nat[OF a, of b] bezout-add-strong-nat[OF b, of a]
  obtain d1 x1 y1 d2 x2 y2 where dxy1: d1 dvd a d1 dvd b a ∗ x1 = b ∗ y1 + d1
    and dxy2: d2 dvd b d2 dvd a b ∗ x2 = a ∗ y2 + d2 by blast
  then have d12: d1 = 1 d2 =1
    by (metis ab coprime-nat)+
  let ?x = v ∗ a ∗ x1 + u ∗ b ∗ x2
  let ?q1 = v ∗ x1 + u ∗ y2
  let ?q2 = v ∗ y1 + u ∗ x2
  from dxy2(3)[simplified d12] dxy1(3)[simplified d12]
  have ?x = u + ?q1 ∗ a ?x = v + ?q2 ∗ b
    by algebra+
  thus ?thesis by blast
qed
```

Primality

```
lemma coprime-bezout-strong:
  fixes a::nat assumes coprime a b  b ≠ 1
  shows ∃ x y. a ∗ x = b ∗ y + 1
by (metis assms bezout-nat gcd-nat.left-neutral)

lemma bezout-prime:
  assumes p: prime p and pa: ¬ p dvd a
  shows ∃ x y. a∗x = Suc (p∗y)
proof −
  have ap: coprime a p
    by (metis gcd.commute p pa prime-imp-coprime-nat)
  moreover from p have p ≠ 1 by auto
  ultimately have ∃ x y. a ∗ x = p ∗ y + 1
    by (rule coprime-bezout-strong)
```

then show *?thesis* **by** *simp*
qed

end

2 Congruence

theory *Cong*
imports *Primes*
begin

2.1 Turn off *One-nat-def*

lemma *power-eq-one-eq-nat* [*simp*]: ((*x*::*nat*) ^*m* = *1*) = (*m* = *0* | *x* = *1*)
  by (*induct m*) *auto*

declare *mod-pos-pos-trivial* [*simp*]

2.2 Main definitions

class *cong* =
  fixes *cong* :: ′*a* ⇒ ′*a* ⇒ ′*a* ⇒ *bool* ((*1*[- = -] ′(()*mod* -′)))
begin

abbreviation *notcong* :: ′*a* ⇒ ′*a* ⇒ ′*a* ⇒ *bool* ((*1*[- ≠ -] ′(()*mod* -′)))
  where *notcong x y m* ≡ ¬ *cong x y m*

end

instantiation *nat* :: *cong*
begin

definition *cong-nat* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool*
  where *cong-nat x y m* = ((*x mod m*) = (*y mod m*))

instance ..

end

instantiation *int* :: *cong*
begin

definition *cong-int* :: *int* ⇒ *int* ⇒ *int* ⇒ *bool*
  where *cong-int x y m* = ((*x mod m*) = (*y mod m*))

instance ..

end

2.3 Set up Transfer

lemma *transfer-nat-int-cong*:
 (*x*::*int*) >= *0* ⟹ *y* >= *0* ⟹ *m* >= *0* ⟹
  ([(*nat x*) = (*nat y*)] (*mod* (*nat m*))) = ([*x* = *y*] (*mod m*))
 unfolding *cong-int-def cong-nat-def*
 by (*metis Divides.transfer-int-nat-functions*(*2*) *nat-0-le nat-mod-distrib*)

declare *transfer-morphism-nat-int*[*transfer add return*:
  transfer-nat-int-cong]

lemma *transfer-int-nat-cong*:
 [(*int x*) = (*int y*)] (*mod* (*int m*)) = [*x* = *y*] (*mod m*)
 apply (*auto simp add*: *cong-int-def cong-nat-def*)
 apply (*auto simp add*: *zmod-int* [*symmetric*])
 done

declare *transfer-morphism-int-nat*[*transfer add return*:
  transfer-int-nat-cong]

2.4 Congruence

lemma *cong-0-nat* [*simp*, *presburger*]: ([(*a*::*nat*) = *b*] (*mod 0*)) = (*a* = *b*)
 unfolding *cong-nat-def* **by** *auto*

lemma *cong-0-int* [*simp*, *presburger*]: ([(*a*::*int*) = *b*] (*mod 0*)) = (*a* = *b*)
 unfolding *cong-int-def* **by** *auto*

lemma *cong-1-nat* [*simp*, *presburger*]: [(*a*::*nat*) = *b*] (*mod 1*)
 unfolding *cong-nat-def* **by** *auto*

lemma *cong-Suc-0-nat* [*simp*, *presburger*]: [(*a*::*nat*) = *b*] (*mod Suc 0*)
 unfolding *cong-nat-def* **by** *auto*

lemma *cong-1-int* [*simp*, *presburger*]: [(*a*::*int*) = *b*] (*mod 1*)
 unfolding *cong-int-def* **by** *auto*

lemma *cong-refl-nat* [*simp*]: [(*k*::*nat*) = *k*] (*mod m*)
 unfolding *cong-nat-def* **by** *auto*

lemma *cong-refl-int* [*simp*]: [(*k*::*int*) = *k*] (*mod m*)
 unfolding *cong-int-def* **by** *auto*

lemma *cong-sym-nat*: [(*a*::*nat*) = *b*] (*mod m*) ⟹ [*b* = *a*] (*mod m*)

unfolding *cong-nat-def* **by** *auto*

lemma *cong-sym-int*: $[(a::int) = b] \ (mod \ m) \Longrightarrow [b = a] \ (mod \ m)$
unfolding *cong-int-def* **by** *auto*

lemma *cong-sym-eq-nat*: $[(a::nat) = b] \ (mod \ m) = [b = a] \ (mod \ m)$
unfolding *cong-nat-def* **by** *auto*

lemma *cong-sym-eq-int*: $[(a::int) = b] \ (mod \ m) = [b = a] \ (mod \ m)$
unfolding *cong-int-def* **by** *auto*

lemma *cong-trans-nat* [*trans*]:
$[(a::nat) = b] \ (mod \ m) \Longrightarrow [b = c] \ (mod \ m) \Longrightarrow [a = c] \ (mod \ m)$
unfolding *cong-nat-def* **by** *auto*

lemma *cong-trans-int* [*trans*]:
$[(a::int) = b] \ (mod \ m) \Longrightarrow [b = c] \ (mod \ m) \Longrightarrow [a = c] \ (mod \ m)$
unfolding *cong-int-def* **by** *auto*

lemma *cong-add-nat*:
$[(a::nat) = b] \ (mod \ m) \Longrightarrow [c = d] \ (mod \ m) \Longrightarrow [a + c = b + d] \ (mod \ m)$
unfolding *cong-nat-def* **by** (*metis mod-add-cong*)

lemma *cong-add-int*:
$[(a::int) = b] \ (mod \ m) \Longrightarrow [c = d] \ (mod \ m) \Longrightarrow [a + c = b + d] \ (mod \ m)$
unfolding *cong-int-def* **by** (*metis mod-add-cong*)

lemma *cong-diff-int*:
$[(a::int) = b] \ (mod \ m) \Longrightarrow [c = d] \ (mod \ m) \Longrightarrow [a - c = b - d] \ (mod \ m)$
unfolding *cong-int-def* **by** (*metis mod-diff-cong*)

lemma *cong-diff-aux-int*:
$[(a::int) = b] \ (mod \ m) \Longrightarrow [c = d] \ (mod \ m) \Longrightarrow$
$(a::int) >= c \Longrightarrow b >= d \Longrightarrow [tsub \ a \ c = tsub \ b \ d] \ (mod \ m)$
by (*metis cong-diff-int tsub-eq*)

lemma *cong-diff-nat*:
assumes$[a = b] \ (mod \ m) \ [c = d] \ (mod \ m) \ (a::nat) >= c \ b >= d$
shows $[a - c = b - d] \ (mod \ m)$
using *assms* **by** (*rule cong-diff-aux-int* [*transferred*])

lemma *cong-mult-nat*:
$[(a::nat) = b] \ (mod \ m) \Longrightarrow [c = d] \ (mod \ m) \Longrightarrow [a * c = b * d] \ (mod \ m)$
unfolding *cong-nat-def* **by** (*metis mod-mult-cong*)

lemma *cong-mult-int*:
$[(a::int) = b] \ (mod \ m) \Longrightarrow [c = d] \ (mod \ m) \Longrightarrow [a * c = b * d] \ (mod \ m)$
unfolding *cong-int-def* **by** (*metis mod-mult-cong*)

lemma *cong-exp-nat*: $[(x::nat) = y] \ (mod\ n) \Longrightarrow [x\hat{}k = y\hat{}k] \ (mod\ n)$
by (*induct k*) (*auto simp add*: *cong-mult-nat*)

lemma *cong-exp-int*: $[(x::int) = y] \ (mod\ n) \Longrightarrow [x\hat{}k = y\hat{}k] \ (mod\ n)$
by (*induct k*) (*auto simp add*: *cong-mult-int*)

lemma *cong-setsum-nat* [*rule-format*]:
$(\forall x{\in}A.\ [((f\ x)::nat) = g\ x] \ (mod\ m)) \longrightarrow$
$[(\sum x{\in}A.\ f\ x) = (\sum x{\in}A.\ g\ x)] \ (mod\ m)$
apply (*cases finite A*)
apply (*induct set*: *finite*)
apply (*auto intro*: *cong-add-nat*)
done

lemma *cong-setsum-int* [*rule-format*]:
$(\forall x{\in}A.\ [((f\ x)::int) = g\ x] \ (mod\ m)) \longrightarrow$
$[(\sum x{\in}A.\ f\ x) = (\sum x{\in}A.\ g\ x)] \ (mod\ m)$
apply (*cases finite A*)
apply (*induct set*: *finite*)
apply (*auto intro*: *cong-add-int*)
done

lemma *cong-setprod-nat* [*rule-format*]:
$(\forall x{\in}A.\ [((f\ x)::nat) = g\ x] \ (mod\ m)) \longrightarrow$
$[(\prod x{\in}A.\ f\ x) = (\prod x{\in}A.\ g\ x)] \ (mod\ m)$
apply (*cases finite A*)
apply (*induct set*: *finite*)
apply (*auto intro*: *cong-mult-nat*)
done

lemma *cong-setprod-int* [*rule-format*]:
$(\forall x{\in}A.\ [((f\ x)::int) = g\ x] \ (mod\ m)) \longrightarrow$
$[(\prod x{\in}A.\ f\ x) = (\prod x{\in}A.\ g\ x)] \ (mod\ m)$
apply (*cases finite A*)
apply (*induct set*: *finite*)
apply (*auto intro*: *cong-mult-int*)
done

lemma *cong-scalar-nat*: $[(a::nat){=}\ b] \ (mod\ m) \Longrightarrow [a * k = b * k] \ (mod\ m)$
by (*rule cong-mult-nat*) *simp-all*

lemma *cong-scalar-int*: $[(a::int){=}\ b] \ (mod\ m) \Longrightarrow [a * k = b * k] \ (mod\ m)$
by (*rule cong-mult-int*) *simp-all*

lemma *cong-scalar2-nat*: $[(a::nat){=}\ b] \ (mod\ m) \Longrightarrow [k * a = k * b] \ (mod\ m)$
by (*rule cong-mult-nat*) *simp-all*

lemma *cong-scalar2-int*: $[(a::int){=}\ b] \ (mod\ m) \Longrightarrow [k * a = k * b] \ (mod\ m)$
by (*rule cong-mult-int*) *simp-all*

lemma *cong-mult-self-nat*: [(*a*::*nat*) ∗ *m* = *0*] (*mod m*)
unfolding *cong-nat-def* **by** *auto*

lemma *cong-mult-self-int*: [(*a*::*int*) ∗ *m* = *0*] (*mod m*)
unfolding *cong-int-def* **by** *auto*

lemma *cong-eq-diff-cong-0-int*: [(*a*::*int*) = *b*] (*mod m*) = [*a* − *b* = *0*] (*mod m*)
by (*metis cong-add-int cong-diff-int cong-refl-int diff-add-cancel diff-self*)

lemma *cong-eq-diff-cong-0-aux-int*: *a* >= *b* ⟹
[(*a*::*int*) = *b*] (*mod m*) = [*tsub a b* = *0*] (*mod m*)
by (*subst tsub-eq*, *assumption*, *rule cong-eq-diff-cong-0-int*)

lemma *cong-eq-diff-cong-0-nat*:
assumes (*a*::*nat*) >= *b*
shows [*a* = *b*] (*mod m*) = [*a* − *b* = *0*] (*mod m*)
using *assms* **by** (*rule cong-eq-diff-cong-0-aux-int* [*transferred*])

lemma *cong-diff-cong-0'-nat*:
[(*x*::*nat*) = *y*] (*mod n*) ⟷
(*if x* <= *y then* [*y* − *x* = *0*] (*mod n*) *else* [*x* − *y* = *0*] (*mod n*))
by (*metis cong-eq-diff-cong-0-nat cong-sym-nat nat-le-linear*)

lemma *cong-altdef-nat*: (*a*::*nat*) >= *b* ⟹ [*a* = *b*] (*mod m*) = (*m dvd* (*a* − *b*))
apply (*subst cong-eq-diff-cong-0-nat*, *assumption*)
apply (*unfold cong-nat-def*)
apply (*simp add*: *dvd-eq-mod-eq-0* [*symmetric*])
done

lemma *cong-altdef-int*: [(*a*::*int*) = *b*] (*mod m*) = (*m dvd* (*a* − *b*))
by (*metis cong-int-def zmod-eq-dvd-iff*)

lemma *cong-abs-int*: [(*x*::*int*) = *y*] (*mod abs m*) = [*x* = *y*] (*mod m*)
by (*simp add*: *cong-altdef-int*)

lemma *cong-square-int*:
fixes *a*::*int*
shows ⟦ *prime p*; *0* < *a*; [*a* ∗ *a* = *1*] (*mod p*) ⟧
⟹ [*a* = *1*] (*mod p*) ∨ [*a* = − *1*] (*mod p*)
apply (*simp only*: *cong-altdef-int*)
apply (*subst prime-dvd-mult-eq-int* [*symmetric*], *assumption*)
apply (*auto simp add*: *field-simps*)
done

lemma *cong-mult-rcancel-int*:
coprime k (*m*::*int*) ⟹ [*a* ∗ *k* = *b* ∗ *k*] (*mod m*) = [*a* = *b*] (*mod m*)
by (*metis cong-altdef-int left-diff-distrib coprime-dvd-mult-iff gcd*.*commute*)

lemma *cong-mult-rcancel-nat*:
  coprime k (*m*::*nat*) $\Longrightarrow$ [*a* $*$ *k* = *b* $*$ *k*] (*mod m*) = [*a* = *b*] (*mod m*)
  by (*metis cong-mult-rcancel-int* [*transferred*])

lemma *cong-mult-lcancel-nat*:
  coprime k (*m*::*nat*) $\Longrightarrow$ [*k* $*$ *a* = *k* $*$ *b*] (*mod m*) = [*a* = *b*] (*mod m*)
  by (*simp add*: *mult.commute cong-mult-rcancel-nat*)

lemma *cong-mult-lcancel-int*:
  coprime k (*m*::*int*) $\Longrightarrow$ [*k* $*$ *a* = *k* $*$ *b*] (*mod m*) = [*a* = *b*] (*mod m*)
  by (*simp add*: *mult.commute cong-mult-rcancel-int*)

lemma *coprime-cong-mult-int*:
  [(*a*::*int*) = *b*] (*mod m*) $\Longrightarrow$ [*a* = *b*] (*mod n*) $\Longrightarrow$ *coprime m n*
  $\Longrightarrow$ [*a* = *b*] (*mod m* $*$ *n*)
by (*metis divides-mult cong-altdef-int*)

lemma *coprime-cong-mult-nat*:
  assumes [(*a*::*nat*) = *b*] (*mod m*) **and** [*a* = *b*] (*mod n*) **and** *coprime m n*
  shows [*a* = *b*] (*mod m* $*$ *n*)
  by (*metis assms coprime-cong-mult-int* [*transferred*])

lemma *cong-less-imp-eq-nat*: *0* $\leq$ (*a*::*nat*) $\Longrightarrow$
  a < *m* $\Longrightarrow$ *0* $\leq$ *b* $\Longrightarrow$ *b* < *m* $\Longrightarrow$ [*a* = *b*] (*mod m*) $\Longrightarrow$ *a* = *b*
  by (*auto simp add*: *cong-nat-def*)

lemma *cong-less-imp-eq-int*: *0* $\leq$ (*a*::*int*) $\Longrightarrow$
  a < *m* $\Longrightarrow$ *0* $\leq$ *b* $\Longrightarrow$ *b* < *m* $\Longrightarrow$ [*a* = *b*] (*mod m*) $\Longrightarrow$ *a* = *b*
  by (*auto simp add*: *cong-int-def*)

lemma *cong-less-unique-nat*:
  0 < (*m*::*nat*) $\Longrightarrow$ ($\exists$!*b*. *0* $\leq$ *b* $\wedge$ *b* < *m* $\wedge$ [*a* = *b*] (*mod m*))
  by (*auto simp*: *cong-nat-def*) (*metis mod-less-divisor mod-mod-trivial*)

lemma *cong-less-unique-int*:
  0 < (*m*::*int*) $\Longrightarrow$ ($\exists$!*b*. *0* $\leq$ *b* $\wedge$ *b* < *m* $\wedge$ [*a* = *b*] (*mod m*))
  by (*auto simp*: *cong-int-def*) (*metis mod-mod-trivial pos-mod-conj*)

lemma *cong-iff-lin-int*: ([(*a*::*int*) = *b*] (*mod m*)) = ($\exists$ *k*. *b* = *a* + *m* $*$ *k*)
  apply (*auto simp add*: *cong-altdef-int dvd-def*)
  apply (*rule-tac* [!] *x* = $-$*k* **in** *exI*, *auto*)
  done

lemma *cong-iff-lin-nat*:
  ([(*a*::*nat*) = *b*] (*mod m*)) $\longleftrightarrow$ ($\exists$ *k1 k2*. *b* + *k1* $*$ *m* = *a* + *k2* $*$ *m*) (**is** *?lhs* = *?rhs*)
proof (*rule iffI*)
  assume *eqm*: *?lhs*

show *?rhs*
proof (*cases* $b \le a$)
case *True*
then show *?rhs* **using** *eqm*
by (*metis cong-altdef-nat dvd-def le-add-diff-inverse add-0-right mult-0 mult.commute*)
next
case *False*
then show *?rhs* **using** *eqm*
apply (*subst* (*asm*) *cong-sym-eq-nat*)
apply (*auto simp*: *cong-altdef-nat*)
apply (*metis add-0-right add-diff-inverse dvd-div-mult-self less-or-eq-imp-le mult-0*)
done
qed
next
assume *?rhs*
then show *?lhs*
by (*metis cong-nat-def mod-mult-self2 mult.commute*)
qed

lemma *cong-gcd-eq-int*: $[(a::int) = b]$ $(mod\ m) \Longrightarrow gcd\ a\ m = gcd\ b\ m$
by (*metis cong-int-def gcd-red-int*)

lemma *cong-gcd-eq-nat*:
$[(a::nat) = b]$ $(mod\ m) \Longrightarrow gcd\ a\ m = gcd\ b\ m$
by (*metis assms cong-gcd-eq-int* [*transferred*])

lemma *cong-imp-coprime-nat*: $[(a::nat) = b]$ $(mod\ m) \Longrightarrow coprime\ a\ m \Longrightarrow coprime\ b\ m$
by (*auto simp add*: *cong-gcd-eq-nat*)

lemma *cong-imp-coprime-int*: $[(a::int) = b]$ $(mod\ m) \Longrightarrow coprime\ a\ m \Longrightarrow coprime\ b\ m$
by (*auto simp add*: *cong-gcd-eq-int*)

lemma *cong-cong-mod-nat*: $[(a::nat) = b]$ $(mod\ m) = [a\ mod\ m = b\ mod\ m]$ $(mod\ m)$
by (*auto simp add*: *cong-nat-def*)

lemma *cong-cong-mod-int*: $[(a::int) = b]$ $(mod\ m) = [a\ mod\ m = b\ mod\ m]$ $(mod\ m)$
by (*auto simp add*: *cong-int-def*)

lemma *cong-minus-int* [*iff*]: $[(a::int) = b]$ $(mod\ -m) = [a = b]$ $(mod\ m)$
by (*metis cong-iff-lin-int minus-equation-iff mult-minus-left mult-minus-right*)

lemma *cong-add-lcancel-nat*:

[(a::nat) + x = a + y] (mod n) ⟷ [x = y] (mod n)
by (*simp add*: *cong-iff-lin-nat*)

lemma *cong-add-lcancel-int*:
[(a::int) + x = a + y] (mod n) ⟷ [x = y] (mod n)
by (*simp add*: *cong-iff-lin-int*)

lemma *cong-add-rcancel-nat*: [(x::nat) + a = y + a] (mod n) ⟷ [x = y] (mod n)
by (*simp add*: *cong-iff-lin-nat*)

lemma *cong-add-rcancel-int*: [(x::int) + a = y + a] (mod n) ⟷ [x = y] (mod n)
by (*simp add*: *cong-iff-lin-int*)

lemma *cong-add-lcancel-0-nat*: [(a::nat) + x = a] (mod n) ⟷ [x = 0] (mod n)
by (*simp add*: *cong-iff-lin-nat*)

lemma *cong-add-lcancel-0-int*: [(a::int) + x = a] (mod n) ⟷ [x = 0] (mod n)
by (*simp add*: *cong-iff-lin-int*)

lemma *cong-add-rcancel-0-nat*: [x + (a::nat) = a] (mod n) ⟷ [x = 0] (mod n)
by (*simp add*: *cong-iff-lin-nat*)

lemma *cong-add-rcancel-0-int*: [x + (a::int) = a] (mod n) ⟷ [x = 0] (mod n)
by (*simp add*: *cong-iff-lin-int*)

lemma *cong-dvd-modulus-nat*: [(x::nat) = y] (mod m) ⟹ n dvd m ⟹
[x = y] (mod n)
apply (*auto simp add*: *cong-iff-lin-nat dvd-def*)
apply (*rule-tac x=k1 * k* **in** *exI*)
apply (*rule-tac x=k2 * k* **in** *exI*)
apply (*simp add*: *field-simps*)
done

lemma *cong-dvd-modulus-int*: [(x::int) = y] (mod m) ⟹ n dvd m ⟹ [x = y] (mod n)
by (*auto simp add*: *cong-altdef-int dvd-def*)

lemma *cong-dvd-eq-nat*: [(x::nat) = y] (mod n) ⟹ n dvd x ⟷ n dvd y
unfolding *cong-nat-def* **by** (*auto simp add*: *dvd-eq-mod-eq-0*)

lemma *cong-dvd-eq-int*: [(x::int) = y] (mod n) ⟹ n dvd x ⟷ n dvd y
unfolding *cong-int-def* **by** (*auto simp add*: *dvd-eq-mod-eq-0*)

lemma *cong-mod-nat*: (n::nat) ~= 0 ⟹ [a mod n = a] (mod n)
by (*simp add*: *cong-nat-def*)

lemma *cong-mod-int*: (n::int) ~= 0 ⟹ [a mod n = a] (mod n)

by (*simp add*: *cong-int-def*)

lemma *mod-mult-cong-nat*: (*a*::*nat*) ~= *0* ⟹ *b* ~= *0*
⟹ [*x mod* (*a* ∗ *b*) = *y*] (*mod a*) ⟷ [*x* = *y*] (*mod a*)
by (*simp add*: *cong-nat-def mod-mult2-eq mod-add-left-eq*)

lemma *neg-cong-int*: ([(*a*::*int*) = *b*] (*mod m*)) = ([−*a* = −*b*] (*mod m*))
by (*metis cong-int-def minus-minus zminus-zmod*)

lemma *cong-modulus-neg-int*: ([(*a*::*int*) = *b*] (*mod m*)) = ([*a* = *b*] (*mod* −*m*))
by (*auto simp add*: *cong-altdef-int*)

lemma *mod-mult-cong-int*: (*a*::*int*) ~= *0* ⟹ *b* ~= *0*
⟹ [*x mod* (*a* ∗ *b*) = *y*] (*mod a*) ⟷ [*x* = *y*] (*mod a*)
apply (*cases b* > *0*, *simp add*: *cong-int-def mod-mod-cancel mod-add-left-eq*)
apply (*subst* (*1 2*) *cong-modulus-neg-int*)
apply (*unfold cong-int-def*)
apply (*subgoal-tac a* ∗ *b* = (−*a* ∗ −*b*))
apply (*erule ssubst*)
apply (*subst zmod-zmult2-eq*)
apply (*auto simp add*: *mod-add-left-eq mod-minus-right div-minus-right*)
apply (*metis mod-diff-left-eq mod-diff-right-eq mod-mult-self1-is-0 diff-zero*)+
done

lemma *cong-to-1-nat*: ([(*a*::*nat*) = *1*] (*mod n*)) ⟹ (*n dvd* (*a* − *1*))
apply (*cases a* = *0*, *force*)
by (*metis cong-altdef-nat leI less-one*)

lemma *cong-0-1-nat'*: [(*0*::*nat*) = *Suc 0*] (*mod n*) = (*n* = *Suc 0*)
unfolding *cong-nat-def* **by** *auto*

lemma *cong-0-1-nat*: [(*0*::*nat*) = *1*] (*mod n*) = (*n* = *1*)
unfolding *cong-nat-def* **by** *auto*

lemma *cong-0-1-int*: [(*0*::*int*) = *1*] (*mod n*) = ((*n* = *1*) | (*n* = −*1*))
unfolding *cong-int-def* **by** (*auto simp add*: *zmult-eq-1-iff*)

lemma *cong-to-1'-nat*: [(*a*::*nat*) = *1*] (*mod n*) ⟷
a = *0* ∧ *n* = *1* ∨ (∃ *m*. *a* = *1* + *m* ∗ *n*)
by (*metis add.right-neutral cong-0-1-nat cong-iff-lin-nat cong-to-1-nat dvd-div-mult-self leI le-add-diff-inverse less-one mult-eq-if*)

lemma *cong-le-nat*: (*y*::*nat*) <= *x* ⟹ [*x* = *y*] (*mod n*) ⟷ (∃ *q*. *x* = *q* ∗ *n* + *y*)
by (*metis cong-altdef-nat Nat.le-imp-diff-is-add dvd-def mult.commute*)

lemma *cong-solve-nat*: (*a*::*nat*) ≠ *0* ⟹ *EX x*. [*a* ∗ *x* = *gcd a n*] (*mod n*)
apply (*cases n* = *0*)
apply *force*
apply (*frule bezout-nat* [*of a n*], *auto*)

by (*metis cong-add-rcancel-0-nat cong-mult-self-nat mult.commute*)

lemma *cong-solve-int*: $(a::int) \neq 0 \Longrightarrow EX\ x.\ [a * x = gcd\ a\ n]\ (mod\ n)$
apply (*cases* $n = 0$)
apply (*cases* $a \geq 0$)
apply *auto*
apply (*rule-tac* $x = -1$ **in** *exI*)
apply *auto*
apply (*insert bezout-int* [*of a n*], *auto*)
by (*metis cong-iff-lin-int mult.commute*)

lemma *cong-solve-dvd-nat*:
assumes *a*: $(a::nat) \neq 0$ **and** *b*: *gcd a n dvd d*
shows $EX\ x.\ [a * x = d]\ (mod\ n)$
proof −
from *cong-solve-nat* [*OF a*] **obtain** *x* **where** $[a * x = gcd\ a\ n](mod\ n)$
by *auto*
then have $[(d\ div\ gcd\ a\ n) * (a * x) = (d\ div\ gcd\ a\ n) * gcd\ a\ n]\ (mod\ n)$
by (*elim cong-scalar2-nat*)
also from *b* **have** $(d\ div\ gcd\ a\ n) * gcd\ a\ n = d$
by (*rule dvd-div-mult-self*)
also have $(d\ div\ gcd\ a\ n) * (a * x) = a * (d\ div\ gcd\ a\ n * x)$
by *auto*
finally show *?thesis*
by *auto*
qed

lemma *cong-solve-dvd-int*:
assumes *a*: $(a::int) \neq 0$ **and** *b*: *gcd a n dvd d*
shows $EX\ x.\ [a * x = d]\ (mod\ n)$
proof −
from *cong-solve-int* [*OF a*] **obtain** *x* **where** $[a * x = gcd\ a\ n](mod\ n)$
by *auto*
then have $[(d\ div\ gcd\ a\ n) * (a * x) = (d\ div\ gcd\ a\ n) * gcd\ a\ n]\ (mod\ n)$
by (*elim cong-scalar2-int*)
also from *b* **have** $(d\ div\ gcd\ a\ n) * gcd\ a\ n = d$
by (*rule dvd-div-mult-self*)
also have $(d\ div\ gcd\ a\ n) * (a * x) = a * (d\ div\ gcd\ a\ n * x)$
by *auto*
finally show *?thesis*
by *auto*
qed

lemma *cong-solve-coprime-nat*: *coprime* $(a::nat)\ n \Longrightarrow EX\ x.\ [a * x = 1]\ (mod\ n)$
apply (*cases* $a = 0$)
apply *force*
apply (*metis cong-solve-nat*)
done

lemma *cong-solve-coprime-int*: *coprime* (*a*::*int*) *n* ⟹ *EX x*. [*a* ∗ *x* = *1*] (*mod n*)
apply (*cases a* = *0*)
apply *auto*
apply (*cases n* ≥ *0*)
apply *auto*
apply (*metis cong-solve-int*)
done

lemma *coprime-iff-invertible-nat*:
m > *0* ⟹ *coprime a m* = (*EX x*. [*a* ∗ *x* = *Suc 0*] (*mod m*))
by (*metis One-nat-def cong-gcd-eq-nat cong-solve-coprime-nat coprime-lmult gcd.commute gcd-Suc-0*)

lemma *coprime-iff-invertible-int*: *m* > (*0*::*int*) ⟹ *coprime a m* = (*EX x*. [*a* ∗ *x* = *1*] (*mod m*))
apply (*auto intro*: *cong-solve-coprime-int*)
apply (*metis cong-int-def coprime-mul-eq gcd-1-int gcd.commute gcd-red-int*)
done

lemma *coprime-iff-invertible′-nat*: *m* > *0* ⟹ *coprime a m* =
(*EX x*. *0* ≤ *x* & *x* < *m* & [*a* ∗ *x* = *Suc 0*] (*mod m*))
apply (*subst coprime-iff-invertible-nat*)
apply *auto*
apply (*auto simp add*: *cong-nat-def*)
apply (*metis mod-less-divisor mod-mult-right-eq*)
done

lemma *coprime-iff-invertible′-int*: *m* > (*0*::*int*) ⟹ *coprime a m* =
(*EX x*. *0* <= *x* & *x* < *m* & [*a* ∗ *x* = *1*] (*mod m*))
apply (*subst coprime-iff-invertible-int*)
apply (*auto simp add*: *cong-int-def*)
apply (*metis mod-mult-right-eq pos-mod-conj*)
done

lemma *cong-cong-lcm-nat*: [(*x*::*nat*) = *y*] (*mod a*) ⟹
[*x* = *y*] (*mod b*) ⟹ [*x* = *y*] (*mod lcm a b*)
apply (*cases y* ≤ *x*)
apply (*metis cong-altdef-nat lcm-least*)
apply (*meson cong-altdef-nat cong-sym-nat lcm-least-iff nat-le-linear*)
done

lemma *cong-cong-lcm-int*: [(*x*::*int*) = *y*] (*mod a*) ⟹
[*x* = *y*] (*mod b*) ⟹ [*x* = *y*] (*mod lcm a b*)
by (*auto simp add*: *cong-altdef-int lcm-least*) [*1*]

lemma *cong-cong-setprod-coprime-nat* [*rule-format*]: *finite A* ⟹
(∀ *i*∈*A*. (∀ *j*∈*A*. *i* ≠ *j* ⟶ *coprime* (*m i*) (*m j*))) ⟶
(∀ *i*∈*A*. [(*x*::*nat*) = *y*] (*mod m i*)) ⟶

[*x* = *y*] (*mod* (∏ *i*∈*A*. *m i*))
apply (*induct set*: *finite*)
apply *auto*
apply (*metis One-nat-def coprime-cong-mult-nat gcd.commute setprod-coprime*)
done

lemma *cong-cong-setprod-coprime-int* [*rule-format*]: *finite A* ⟹
(∀ *i*∈*A*. (∀ *j*∈*A*. *i* ≠ *j* ⟶ *coprime* (*m i*) (*m j*))) ⟶
(∀ *i*∈*A*. [(*x*::*int*) = *y*] (*mod m i*)) ⟶
[*x* = *y*] (*mod* (∏ *i*∈*A*. *m i*))
apply (*induct set*: *finite*)
apply *auto*
apply (*metis coprime-cong-mult-int gcd.commute setprod-coprime*)
done

lemma *binary-chinese-remainder-aux-nat*:
assumes *a*: *coprime* (*m1*::*nat*) *m2*
shows *EX b1 b2*. [*b1* = *1*] (*mod m1*) ∧ [*b1* = *0*] (*mod m2*) ∧
[*b2* = *0*] (*mod m1*) ∧ [*b2* = *1*] (*mod m2*)
proof −
from *cong-solve-coprime-nat* [*OF a*] **obtain** *x1* **where** *one*: [*m1* ∗ *x1* = *1*] (*mod m2*)
by *auto*
from *a* **have** *b*: *coprime m2 m1*
by (*subst gcd.commute*)
from *cong-solve-coprime-nat* [*OF b*] **obtain** *x2* **where** *two*: [*m2* ∗ *x2* = *1*] (*mod m1*)
by *auto*
have [*m1* ∗ *x1* = *0*] (*mod m1*)
by (*subst mult.commute*, *rule cong-mult-self-nat*)
moreover have [*m2* ∗ *x2* = *0*] (*mod m2*)
by (*subst mult.commute*, *rule cong-mult-self-nat*)
moreover note *one two*
ultimately show *?thesis* **by** *blast*
qed

lemma *binary-chinese-remainder-aux-int*:
assumes *a*: *coprime* (*m1*::*int*) *m2*
shows *EX b1 b2*. [*b1* = *1*] (*mod m1*) ∧ [*b1* = *0*] (*mod m2*) ∧
[*b2* = *0*] (*mod m1*) ∧ [*b2* = *1*] (*mod m2*)
proof −
from *cong-solve-coprime-int* [*OF a*] **obtain** *x1* **where** *one*: [*m1* ∗ *x1* = *1*] (*mod m2*)
by *auto*
from *a* **have** *b*: *coprime m2 m1*
by (*subst gcd.commute*)
from *cong-solve-coprime-int* [*OF b*] **obtain** *x2* **where** *two*: [*m2* ∗ *x2* = *1*] (*mod m1*)
by *auto*

```
  have [m1 * x1 = 0] (mod m1)
    by (subst mult.commute, rule cong-mult-self-int)
  moreover have [m2 * x2 = 0] (mod m2)
    by (subst mult.commute, rule cong-mult-self-int)
  moreover note one two
  ultimately show ?thesis by blast
qed

lemma binary-chinese-remainder-nat:
  assumes a: coprime (m1::nat) m2
  shows EX x. [x = u1] (mod m1) ∧ [x = u2] (mod m2)
proof −
  from binary-chinese-remainder-aux-nat [OF a] obtain b1 b2
      where [b1 = 1] (mod m1) and [b1 = 0] (mod m2) and
            [b2 = 0] (mod m1) and [b2 = 1] (mod m2)
    by blast
  let ?x = u1 * b1 + u2 * b2
  have [?x = u1 * 1 + u2 * 0] (mod m1)
    apply (rule cong-add-nat)
    apply (rule cong-scalar2-nat)
    apply (rule ‹[b1 = 1] (mod m1)›)
    apply (rule cong-scalar2-nat)
    apply (rule ‹[b2 = 0] (mod m1)›)
    done
  then have [?x = u1] (mod m1) by simp
  have [?x = u1 * 0 + u2 * 1] (mod m2)
    apply (rule cong-add-nat)
    apply (rule cong-scalar2-nat)
    apply (rule ‹[b1 = 0] (mod m2)›)
    apply (rule cong-scalar2-nat)
    apply (rule ‹[b2 = 1] (mod m2)›)
    done
  then have [?x = u2] (mod m2) by simp
  with ‹[?x = u1] (mod m1)› show ?thesis by blast
qed

lemma binary-chinese-remainder-int:
  assumes a: coprime (m1::int) m2
  shows EX x. [x = u1] (mod m1) ∧ [x = u2] (mod m2)
proof −
  from binary-chinese-remainder-aux-int [OF a] obtain b1 b2
    where [b1 = 1] (mod m1) and [b1 = 0] (mod m2) and
          [b2 = 0] (mod m1) and [b2 = 1] (mod m2)
    by blast
  let ?x = u1 * b1 + u2 * b2
  have [?x = u1 * 1 + u2 * 0] (mod m1)
    apply (rule cong-add-int)
    apply (rule cong-scalar2-int)
    apply (rule ‹[b1 = 1] (mod m1)›)
```

apply (*rule cong-scalar2-int*)
apply (*rule* ‹[*b2* = *0*] (*mod m1*)›)
done
then have [*?x* = *u1*] (*mod m1*) **by** *simp*
have [*?x* = *u1* ∗ *0* + *u2* ∗ *1*] (*mod m2*)
apply (*rule cong-add-int*)
apply (*rule cong-scalar2-int*)
apply (*rule* ‹[*b1* = *0*] (*mod m2*)›)
apply (*rule cong-scalar2-int*)
apply (*rule* ‹[*b2* = *1*] (*mod m2*)›)
done
then have [*?x* = *u2*] (*mod m2*) **by** *simp*
with ‹[*?x* = *u1*] (*mod m1*)› **show** *?thesis* **by** *blast*
qed

lemma *cong-modulus-mult-nat*: [(*x*::*nat*) = *y*] (*mod m* ∗ *n*) ⟹
[*x* = *y*] (*mod m*)
apply (*cases y* ≤ *x*)
apply (*simp add*: *cong-altdef-nat*)
apply (*erule dvd-mult-left*)
apply (*rule cong-sym-nat*)
apply (*subst* (*asm*) *cong-sym-eq-nat*)
apply (*simp add*: *cong-altdef-nat*)
apply (*erule dvd-mult-left*)
done

lemma *cong-modulus-mult-int*: [(*x*::*int*) = *y*] (*mod m* ∗ *n*) ⟹
[*x* = *y*] (*mod m*)
apply (*simp add*: *cong-altdef-int*)
apply (*erule dvd-mult-left*)
done

lemma *cong-less-modulus-unique-nat*:
[(*x*::*nat*) = *y*] (*mod m*) ⟹ *x* < *m* ⟹ *y* < *m* ⟹ *x* = *y*
by (*simp add*: *cong-nat-def*)

lemma *binary-chinese-remainder-unique-nat*:
assumes *a*: *coprime* (*m1*::*nat*) *m2*
and *nz*: *m1* ≠ *0* *m2* ≠ *0*
shows *EX*! *x*. *x* < *m1* ∗ *m2* ∧ [*x* = *u1*] (*mod m1*) ∧ [*x* = *u2*] (*mod m2*)
proof −
from *binary-chinese-remainder-nat* [*OF a*] **obtain** *y* **where**
[*y* = *u1*] (*mod m1*) **and** [*y* = *u2*] (*mod m2*)
by *blast*
let *?x* = *y mod* (*m1* ∗ *m2*)
from *nz* **have** *less*: *?x* < *m1* ∗ *m2*
by *auto*
have *one*: [*?x* = *u1*] (*mod m1*)
apply (*rule cong-trans-nat*)

```
    prefer 2
    apply (rule ‹[y = u1] (mod m1)›)
    apply (rule cong-modulus-mult-nat)
    apply (rule cong-mod-nat)
    using nz apply auto
    done
  have two: [?x = u2] (mod m2)
    apply (rule cong-trans-nat)
    prefer 2
    apply (rule ‹[y = u2] (mod m2)›)
    apply (subst mult.commute)
    apply (rule cong-modulus-mult-nat)
    apply (rule cong-mod-nat)
    using nz apply auto
    done
  have ALL z. z < m1 * m2 ∧ [z = u1] (mod m1) ∧ [z = u2] (mod m2) ⟶ z
= ?x
  proof clarify
    fix z
    assume z < m1 * m2
    assume [z = u1] (mod m1) and  [z = u2] (mod m2)
    have [?x = z] (mod m1)
      apply (rule cong-trans-nat)
      apply (rule ‹[?x = u1] (mod m1)›)
      apply (rule cong-sym-nat)
      apply (rule ‹[z = u1] (mod m1)›)
      done
    moreover have [?x = z] (mod m2)
      apply (rule cong-trans-nat)
      apply (rule ‹[?x = u2] (mod m2)›)
      apply (rule cong-sym-nat)
      apply (rule ‹[z = u2] (mod m2)›)
      done
    ultimately have [?x = z] (mod m1 * m2)
      by (auto intro: coprime-cong-mult-nat a)
    with ‹z < m1 * m2› ‹?x < m1 * m2› show z = ?x
      apply (intro cong-less-modulus-unique-nat)
      apply (auto, erule cong-sym-nat)
      done
  qed
  with less one two show ?thesis by auto
 qed

lemma chinese-remainder-aux-nat:
  fixes A :: 'a set
    and m :: 'a ⇒ nat
  assumes fin: finite A
    and cop: ALL i : A. (ALL j : A. i ≠ j ⟶ coprime (m i) (m j))
  shows EX b. (ALL i : A. [b i = 1] (mod m i) ∧ [b i = 0] (mod (∏j ∈ A − {i}.
```

```
m j)))
proof (rule finite-set-choice, rule fin, rule ballI)
  fix i
  assume i : A
  with cop have coprime (∏ j ∈ A − {i}. m j) (m i)
    by (intro setprod-coprime, auto)
  then have EX x. [(∏ j ∈ A − {i}. m j) ∗ x = 1] (mod m i)
    by (elim cong-solve-coprime-nat)
  then obtain x where [(∏ j ∈ A − {i}. m j) ∗ x = 1] (mod m i)
    by auto
  moreover have [(∏ j ∈ A − {i}. m j) ∗ x = 0]
    (mod (∏ j ∈ A − {i}. m j))
    by (subst mult.commute, rule cong-mult-self-nat)
  ultimately show ∃ a. [a = 1] (mod m i) ∧ [a = 0]
      (mod setprod m (A − {i}))
    by blast
qed

lemma chinese-remainder-nat:
  fixes A :: 'a set
    and m :: 'a ⇒ nat
    and u :: 'a ⇒ nat
  assumes fin: finite A
    and cop: ALL i:A. (ALL j : A. i ≠ j ⟶ coprime (m i) (m j))
  shows EX x. (ALL i:A. [x = u i] (mod m i))
proof −
  from chinese-remainder-aux-nat [OF fin cop] obtain b where
    bprop: ALL i:A. [b i = 1] (mod m i) ∧
      [b i = 0] (mod (∏ j ∈ A − {i}. m j))
    by blast
  let ?x = ∑ i∈A. (u i) ∗ (b i)
  show ?thesis
  proof (rule exI, clarify)
    fix i
    assume a: i : A
    show [?x = u i] (mod m i)
    proof −
      from fin a have ?x = (∑ j ∈ {i}. u j ∗ b j) +
          (∑ j ∈ A − {i}. u j ∗ b j)
        by (subst setsum.union-disjoint [symmetric], auto intro: setsum.cong)
      then have [?x = u i ∗ b i + (∑ j ∈ A − {i}. u j ∗ b j)] (mod m i)
        by auto
      also have [u i ∗ b i + (∑ j ∈ A − {i}. u j ∗ b j) =
                  u i ∗ 1 + (∑ j ∈ A − {i}. u j ∗ 0)] (mod m i)
        apply (rule cong-add-nat)
        apply (rule cong-scalar2-nat)
        using bprop a apply blast
        apply (rule cong-setsum-nat)
        apply (rule cong-scalar2-nat)
```

using *bprop* **apply** *auto*
apply (*rule cong-dvd-modulus-nat*)
apply (*drule* (*1*) *bspec*)
apply (*erule conjE*)
apply *assumption*
apply *rule*
using *fin a* **apply** *auto*
done
finally show *?thesis*
by *simp*
qed
qed
qed

lemma *coprime-cong-prod-nat* [*rule-format*]: *finite A* $\Longrightarrow$
($\forall$ *i*$\in$*A*. ($\forall$ *j*$\in$*A*. *i* $\neq$ *j* $\longrightarrow$ *coprime* (*m i*) (*m j*))) $\longrightarrow$
($\forall$ *i*$\in$*A*. [(*x*::*nat*) = *y*] (*mod m i*)) $\longrightarrow$
[*x* = *y*] (*mod* ($\prod$ *i*$\in$*A*. *m i*))
apply (*induct set*: *finite*)
apply *auto*
apply (*metis One-nat-def coprime-cong-mult-nat gcd.commute setprod-coprime*)
done

lemma *chinese-remainder-unique-nat*:
fixes *A* :: *'a set*
and *m* :: *'a* $\Rightarrow$ *nat*
and *u* :: *'a* $\Rightarrow$ *nat*
assumes *fin*: *finite A*
and *nz*: $\forall$ *i*$\in$*A*. *m i* $\neq$ *0*
and *cop*: $\forall$ *i*$\in$*A*. ($\forall$ *j*$\in$*A*. *i* $\neq$ *j* $\longrightarrow$ *coprime* (*m i*) (*m j*))
shows *EX*! *x*. *x* < ($\prod$ *i*$\in$*A*. *m i*) $\wedge$ ($\forall$ *i*$\in$*A*. [*x* = *u i*] (*mod m i*))
proof −
from *chinese-remainder-nat* [*OF fin cop*]
obtain *y* **where** *one*: (*ALL i*:*A*. [*y* = *u i*] (*mod m i*))
by *blast*
let *?x* = *y mod* ($\prod$ *i*$\in$*A*. *m i*)
from *fin nz* **have** *prodnz*: ($\prod$ *i*$\in$*A*. *m i*) $\neq$ *0*
by *auto*
then have *less*: *?x* < ($\prod$ *i*$\in$*A*. *m i*)
by *auto*
have *cong*: *ALL i*:*A*. [*?x* = *u i*] (*mod m i*)
apply *auto*
apply (*rule cong-trans-nat*)
prefer *2*
using *one* **apply** *auto*
apply (*rule cong-dvd-modulus-nat*)
apply (*rule cong-mod-nat*)
using *prodnz* **apply** *auto*
apply *rule*

```
    apply (rule fin)
    apply assumption
    done
  have unique: ALL z. z < (∏ i∈A. m i) ∧
      (ALL i:A. [z = u i] (mod m i)) ⟶ z = ?x
  proof (clarify)
    fix z
    assume zless: z < (∏ i∈A. m i)
    assume zcong: (ALL i:A. [z = u i] (mod m i))
    have ALL i:A. [?x = z] (mod m i)
      apply clarify
      apply (rule cong-trans-nat)
      using cong apply (erule bspec)
      apply (rule cong-sym-nat)
      using zcong apply auto
      done
    with fin cop have [?x = z] (mod (∏ i∈A. m i))
      apply (intro coprime-cong-prod-nat)
      apply auto
      done
    with zless less show z = ?x
      apply (intro cong-less-modulus-unique-nat)
      apply (auto, erule cong-sym-nat)
      done
  qed
  from less cong unique show ?thesis by blast
qed

end
```

3 Unique factorization for the natural numbers and the integers

```
theory UniqueFactorization
imports Cong ~~/src/HOL/Library/Multiset
begin
```

3.1 Unique factorization: multiset version

```
lemma multiset-prime-factorization-exists:
  n > 0 ⟹ (∃ M. (∀ p::nat ∈ set-mset M. prime p) ∧ n = (∏ i ∈# M. i))
proof (induct n rule: nat-less-induct)
  fix n :: nat
  assume ih: ∀ m < n. 0 < m ⟶ (∃ M. (∀ p∈set-mset M. prime p) ∧ m = (∏ i
∈# M. i))
  assume n > 0
  then consider n = 1 | n > 1 prime n | n > 1 ¬ prime n
    by arith
```

```
  then show ∃M. (∀p ∈ set-mset M. prime p) ∧ n = (∏i∈#M. i)
  proof cases
    case 1
    then have (∀p∈set-mset {#}. prime p) ∧ n = (∏i ∈# {#}. i)
      by auto
    then show ?thesis ..
  next
    case 2
    then have (∀p∈set-mset {#n#}. prime p) ∧ n = (∏i ∈# {#n#}. i)
      by auto
    then show ?thesis ..
  next
    case 3
    with not-prime-eq-prod-nat
    obtain m k where n: n = m ∗ k 1 < m m < n 1 < k k < n
      by blast
    with ih obtain Q R where (∀p ∈ set-mset Q. prime p) ∧ m = (∏i∈#Q. i)
        and (∀p∈set-mset R. prime p) ∧ k = (∏i∈#R. i)
      by blast
    then have (∀p∈set-mset (Q + R). prime p) ∧ n = (∏i ∈# Q + R. i)
      by (auto simp add: n msetprod-Un)
    then show ?thesis ..
  qed
qed

lemma multiset-prime-factorization-unique-aux:
  fixes a :: nat
  assumes ∀p∈set-mset M. prime p
    and ∀p∈set-mset N. prime p
    and (∏i ∈# M. i) dvd (∏i ∈# N. i)
  shows count M a ≤ count N a
proof (cases a ∈ set-mset M)
  case True
  with assms have a: prime a
    by auto
  with True have a ^ count M a dvd (∏i ∈# M. i)
    by (auto simp add: msetprod-multiplicity)
  also have . . . dvd (∏i ∈# N. i)
    by (rule assms)
  also have . . . = (∏i ∈ set-mset N. i ^ count N i)
    by (simp add: msetprod-multiplicity)
  also have . . . = a ^ count N a ∗ (∏i ∈ (set-mset N − {a}). i ^ count N i)
  proof (cases a ∈ set-mset N)
    case True
    then have b: set-mset N = {a} ∪ (set-mset N − {a})
      by auto
    then show ?thesis
      by (subst (1) b, subst setprod.union-disjoint, auto)
  next
```

```
    case False
    then show ?thesis
      by (auto simp add: not-in-iff)
  qed
  finally have a ^ count M a dvd a ^ count N a * (∏ i ∈ (set-mset N − {a}). i ^
count N i) .
  moreover
  have coprime (a ^ count M a) (∏ i ∈ (set-mset N − {a}). i ^ count N i)
    apply (subst gcd.commute)
    apply (rule setprod-coprime)
    apply (rule primes-imp-powers-coprime-nat)
    using assms True
    apply auto
    done
  ultimately have a ^ count M a dvd a ^ count N a
    by (elim coprime-dvd-mult)
  with a show ?thesis
    using power-dvd-imp-le prime-def by blast
next
  case False
  then show ?thesis
    by (auto simp add: not-in-iff)
qed

lemma multiset-prime-factorization-unique:
  assumes ∀ p::nat ∈ set-mset M. prime p
    and ∀ p ∈ set-mset N. prime p
    and (∏ i ∈# M. i) = (∏ i ∈# N. i)
  shows M = N
proof −
  have count M a = count N a for a
  proof −
    from assms have count M a ≤ count N a
      by (intro multiset-prime-factorization-unique-aux, auto)
    moreover from assms have count N a ≤ count M a
      by (intro multiset-prime-factorization-unique-aux, auto)
    ultimately show ?thesis
      by auto
  qed
  then show ?thesis
    by (simp add: multiset-eq-iff)
qed

definition multiset-prime-factorization :: nat ⇒ nat multiset
where
  multiset-prime-factorization n =
    (if n > 0
     then THE M. (∀ p ∈ set-mset M. prime p) ∧ n = (∏ i ∈# M. i)
     else {#})
```

lemma *multiset-prime-factorization*: $n > 0 \Longrightarrow$
$(\forall\, p \in \textit{set-mset}\ (\textit{multiset-prime-factorization}\ n).\ \textit{prime}\ p) \wedge$
$n = (\prod i \in\#\ (\textit{multiset-prime-factorization}\ n).\ i)$
apply (*unfold multiset-prime-factorization-def*)
apply *clarsimp*
apply (*frule multiset-prime-factorization-exists*)
apply *clarify*
apply (*rule theI*)
apply (*insert multiset-prime-factorization-unique*)
apply *auto*
done

3.2 Prime factors and multiplicity for nat and int

class *unique-factorization* =
fixes *multiplicity* :: $'a \Rightarrow 'a \Rightarrow \textit{nat}$
and *prime-factors* :: $'a \Rightarrow 'a\ \textit{set}$

Definitions for the natural numbers.

instantiation *nat* :: *unique-factorization*
begin

definition *multiplicity-nat* :: $\textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{nat}$
where *multiplicity-nat* $p\ n$ = *count* (*multiset-prime-factorization* n) p

definition *prime-factors-nat* :: $\textit{nat} \Rightarrow \textit{nat set}$
where *prime-factors-nat* n = *set-mset* (*multiset-prime-factorization* n)

instance ..

end

Definitions for the integers.

instantiation *int* :: *unique-factorization*
begin

definition *multiplicity-int* :: $\textit{int} \Rightarrow \textit{int} \Rightarrow \textit{nat}$
where *multiplicity-int* $p\ n$ = *multiplicity* (*nat* p) (*nat* n)

definition *prime-factors-int* :: $\textit{int} \Rightarrow \textit{int set}$
where *prime-factors-int* n = *int* ‘ (*prime-factors* (*nat* n))

instance ..

end

3.3 Set up transfer

lemma *transfer-nat-int-prime-factors*: *prime-factors* (*nat n*) = *nat* ‘ *prime-factors* *n*
  unfolding *prime-factors-int-def*
  apply *auto*
  apply (*subst transfer-int-nat-set-return-embed*)
  apply *assumption*
  done

lemma *transfer-nat-int-prime-factors-closure*: $n \geq 0 \Longrightarrow$ *nat-set* (*prime-factors* *n*)
  by (*auto simp add*: *nat-set-def prime-factors-int-def*)

lemma *transfer-nat-int-multiplicity*:
  $p \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$ *multiplicity* (*nat p*) (*nat n*) = *multiplicity p n*
  by (*auto simp add*: *multiplicity-int-def*)

declare *transfer-morphism-nat-int*[*transfer add return*:
  transfer-nat-int-prime-factors transfer-nat-int-prime-factors-closure
  transfer-nat-int-multiplicity]

lemma *transfer-int-nat-prime-factors*: *prime-factors* (*int n*) = *int* ‘ *prime-factors* *n*
  unfolding *prime-factors-int-def* **by** *auto*

lemma *transfer-int-nat-prime-factors-closure*: *is-nat n* $\Longrightarrow$ *nat-set* (*prime-factors* *n*)
  by (*simp only*: *transfer-nat-int-prime-factors-closure is-nat-def*)

lemma *transfer-int-nat-multiplicity*: *multiplicity* (*int p*) (*int n*) = *multiplicity p n*
  by (*auto simp add*: *multiplicity-int-def*)

declare *transfer-morphism-int-nat*[*transfer add return*:
  transfer-int-nat-prime-factors transfer-int-nat-prime-factors-closure
  transfer-int-nat-multiplicity]

3.4 Properties of prime factors and multiplicity for nat and int

lemma *prime-factors-ge-0-int* [*elim*]:
  fixes *n* :: *int*
  shows $p \in$ *prime-factors n* $\Longrightarrow p \geq 0$
  unfolding *prime-factors-int-def* **by** *auto*

lemma *prime-factors-prime-nat* [*intro*]:
  fixes *n* :: *nat*
  shows $p \in$ *prime-factors n* $\Longrightarrow$ *prime p*
  apply (*cases n* = *0*)
  apply (*simp add*: *prime-factors-nat-def multiset-prime-factorization-def*)

apply (*auto simp add*: *prime-factors-nat-def multiset-prime-factorization*)
done

lemma *prime-factors-prime-int* [*intro*]:
fixes *n* :: *int*
assumes $n \geq 0$ **and** $p \in$ *prime-factors n*
shows *prime p*
apply (*rule prime-factors-prime-nat* [*transferred*, *of n p*, *simplified*])
using *assms* **apply** *auto*
done

lemma *prime-factors-gt-0-nat*:
fixes *p* :: *nat*
shows $p \in$ *prime-factors x* $\Longrightarrow p > 0$
using *prime-factors-prime-nat* **by** *force*

lemma *prime-factors-gt-0-int*:
shows $x \geq 0 \Longrightarrow p \in$ *prime-factors x* $\Longrightarrow$ *int p* $>$ (*0*::*int*)
by (*simp add*: *prime-factors-gt-0-nat*)

lemma *prime-factors-finite-nat* [*iff*]:
fixes *n* :: *nat*
shows *finite* (*prime-factors n*)
unfolding *prime-factors-nat-def* **by** *auto*

lemma *prime-factors-finite-int* [*iff*]:
fixes *n* :: *int*
shows *finite* (*prime-factors n*)
unfolding *prime-factors-int-def* **by** *auto*

lemma *prime-factors-altdef-nat*:
fixes *n* :: *nat*
shows *prime-factors n* = {*p*. *multiplicity p n* > *0*}
by (*force simp add*: *prime-factors-nat-def multiplicity-nat-def*)

lemma *prime-factors-altdef-int*:
fixes *n* :: *int*
shows *prime-factors n* = {*p*. $p \geq 0 \wedge$ *multiplicity p n* > *0*}
apply (*unfold prime-factors-int-def multiplicity-int-def*)
apply (*subst prime-factors-altdef-nat*)
apply (*auto simp add*: *image-def*)
done

lemma *prime-factorization-nat*:
fixes *n* :: *nat*
shows $n > 0 \Longrightarrow n = (\prod p \in$ *prime-factors n*. *p* ^ *multiplicity p n*)
apply (*frule multiset-prime-factorization*)
apply (*simp add*: *prime-factors-nat-def multiplicity-nat-def msetprod-multiplicity*)
done

lemma *prime-factorization-int*:
  fixes *n* :: *int*
  assumes *n* > *0*
  shows *n* = (∏ *p* ∈ *prime-factors n*. *p* ˆ *multiplicity p n*)
  apply (*rule prime-factorization-nat* [*transferred*, *of n*])
  using *assms* **apply** *auto*
  done

lemma *prime-factorization-unique-nat*:
  fixes *f* :: *nat* ⇒ *-*
  assumes *S-eq*: *S* = {*p*. *0* < *f p*}
    and *finite S*
    and *S*: ∀ *p*∈*S*. *prime p n* = (∏ *p*∈*S*. *p* ˆ *f p*)
  shows *S* = *prime-factors n* ∧ (∀ *p*. *f p* = *multiplicity p n*)
proof −
  from *assms* **have** *f* ∈ *multiset*
    by (*auto simp add*: *multiset-def*)
  moreover from *assms* **have** *n* > *0*
    by (*auto intro*: *prime-gt-0-nat*)
  ultimately have *multiset-prime-factorization n* = *Abs-multiset f*
    apply (*unfold multiset-prime-factorization-def*)
    apply (*subst if-P*, *assumption*)
    apply (*rule the1-equality*)
    apply (*rule ex-ex1I*)
    apply (*rule multiset-prime-factorization-exists*, *assumption*)
    apply (*rule multiset-prime-factorization-unique*)
    apply *force*
    apply *force*
    apply *force*
    using *S S-eq* **apply** (*simp add*: *set-mset-def msetprod-multiplicity*)
    done
  with ‹*f* ∈ *multiset*› **have** *count* (*multiset-prime-factorization n*) = *f*
    by *simp*
  with *S-eq* **show** *?thesis*
    by (*simp add*: *set-mset-def multiset-def prime-factors-nat-def multiplicity-nat-def*)
qed

lemma *prime-factors-characterization-nat*:
  S = {*p*. *0* < *f* (*p*::*nat*)} =⇒
    finite S =⇒ ∀ *p*∈*S*. *prime p* =⇒ *n* = (∏ *p*∈*S*. *p* ˆ *f p*) =⇒ *prime-factors n* =
S
  by (*rule prime-factorization-unique-nat* [*THEN conjunct1*, *symmetric*])

lemma *prime-factors-characterization′-nat*:
  finite {*p*. *0* < *f* (*p*::*nat*)} =⇒
    (∀ *p*. *0* < *f p* −→ *prime p*) =⇒
      prime-factors (∏ *p* | *0* < *f p*. *p* ˆ *f p*) = {*p*. *0* < *f p*}
  by (*rule prime-factors-characterization-nat*) *auto*

thm *prime-factors-characterization′-nat*
[**where** $f = \lambda x.\ f\ (int\ (x{::}nat))$,
transferred direction: *nat op* $\leq$ $(0{::}int)$, *rule-format*]

lemma *primes-characterization′-int* [*rule-format*]:
$finite\ \{p.\ p \geq 0 \wedge 0 < f\ (p{::}int)\} \Longrightarrow \forall p.\ 0 < f\ p \longrightarrow prime\ p \Longrightarrow$
$prime\text{-}factors\ (\prod p \mid p \geq 0 \wedge 0 < f\ p.\ p\ \hat{}\ f\ p) = \{p.\ p \geq 0 \wedge 0 < f\ p\}$
using *prime-factors-characterization′-nat*
[**where** $f = \lambda x.\ f\ (int\ (x{::}nat))$,
transferred direction: *nat op* $\leq$ $(0{::}int)$]
by *auto*

lemma *prime-factors-characterization-int*:
$S = \{p.\ 0 < f\ (p{::}int)\} \Longrightarrow finite\ S \Longrightarrow$
$\forall p{\in}S.\ prime\ (nat\ p) \Longrightarrow n = (\prod p{\in}S.\ p\ \hat{}\ f\ p) \Longrightarrow prime\text{-}factors\ n = S$
apply *simp*
apply (*subgoal-tac* $\{p.\ 0 < f\ p\} = \{p.\ 0 \leq p \wedge 0 < f\ p\}$)
apply (*simp only*:)
apply (*subst primes-characterization′-int*)
apply *simp-all*
apply (*metis nat-int*)
apply (*metis le-cases nat-le-0 zero-not-prime-nat*)
done

lemma *multiplicity-characterization-nat*:
$S = \{p.\ 0 < f\ (p{::}nat)\} \Longrightarrow finite\ S \Longrightarrow \forall p{\in}S.\ prime\ p \Longrightarrow$
$n = (\prod p{\in}S.\ p\ \hat{}\ f\ p) \Longrightarrow multiplicity\ p\ n = f\ p$
apply (*frule prime-factorization-unique-nat* [*THEN conjunct2*, *rule-format*, *symmetric*])
apply *auto*
done

lemma *multiplicity-characterization′-nat*: $finite\ \{p.\ 0 < f\ (p{::}nat)\} \longrightarrow$
$(\forall p.\ 0 < f\ p \longrightarrow prime\ p) \longrightarrow$
$multiplicity\ p\ (\prod p \mid 0 < f\ p.\ p\ \hat{}\ f\ p) = f\ p$
apply (*intro impI*)
apply (*rule multiplicity-characterization-nat*)
apply *auto*
done

lemma *multiplicity-characterization′-int* [*rule-format*]:
$finite\ \{p.\ p \geq 0 \wedge 0 < f\ (p{::}int)\} \Longrightarrow$
$(\forall p.\ 0 < f\ p \longrightarrow prime\ p) \Longrightarrow p \geq 0 \Longrightarrow$
$multiplicity\ p\ (\prod p \mid p \geq 0 \wedge 0 < f\ p.\ p\ \hat{}\ f\ p) = f\ p$
apply (*insert multiplicity-characterization′-nat*

[**where** $f = \lambda x.\ f\ (int\ (x::nat))$,
transferred direction: $nat\ op \leq (0::int)$, *rule-format*])
apply *auto*
done

lemma *multiplicity-characterization-int*: $S = \{p.\ 0 < f\ (p::int)\} \Longrightarrow$
$finite\ S \Longrightarrow \forall\, p{\in}S.\ prime\ (nat\ p) \Longrightarrow n = (\prod p{\in}S.\ p\ \hat{}\ f\ p) \Longrightarrow$
$p \geq 0 \Longrightarrow multiplicity\ p\ n = f\ p$
apply *simp*
apply (*subgoal-tac* $\{p.\ 0 < f\ p\} = \{p.\ 0 \leq p \wedge 0 < f\ p\}$)
apply (*simp only*:)
apply (*subst multiplicity-characterization'-int*)
apply *simp-all*
apply (*metis nat-int*)
apply (*metis le-cases nat-le-0 zero-not-prime-nat*)
done

lemma *multiplicity-zero-nat* [*simp*]: $multiplicity\ (p::nat)\ 0 = 0$
by (*simp add*: *multiplicity-nat-def multiset-prime-factorization-def*)

lemma *multiplicity-zero-int* [*simp*]: $multiplicity\ (p::int)\ 0 = 0$
by (*simp add*: *multiplicity-int-def*)

lemma *multiplicity-one-nat'*: $multiplicity\ p\ (1::nat) = 0$
by (*subst multiplicity-characterization-nat* [**where** $f = \lambda x.\ 0$], *auto*)

lemma *multiplicity-one-nat* [*simp*]: $multiplicity\ p\ (Suc\ 0) = 0$
by (*metis One-nat-def multiplicity-one-nat'*)

lemma *multiplicity-one-int* [*simp*]: $multiplicity\ p\ (1::int) = 0$
by (*metis multiplicity-int-def multiplicity-one-nat' transfer-nat-int-numerals*(*2*))

lemma *multiplicity-prime-nat* [*simp*]: $prime\ p \Longrightarrow multiplicity\ p\ p = 1$
apply (*subst multiplicity-characterization-nat* [**where** $f = \lambda q.\ if\ q = p\ then\ 1$ $else\ 0$])
apply *auto*
apply (*metis* (*full-types*) *less-not-refl*)
done

lemma *multiplicity-prime-power-nat* [*simp*]: $prime\ p \Longrightarrow multiplicity\ p\ (p\ \hat{}\ n) =$ n
apply (*cases* $n = 0$)
apply *auto*
apply (*subst multiplicity-characterization-nat* [**where** $f = \lambda q.\ if\ q = p\ then\ n$ $else\ 0$])
apply *auto*
apply (*metis* (*full-types*) *less-not-refl*)
done

```
lemma multiplicity-prime-power-int [simp]: prime p ⟹ multiplicity p (int p ˆ n)
= n
  by (metis multiplicity-prime-power-nat of-nat-power transfer-int-nat-multiplicity)

lemma multiplicity-nonprime-nat [simp]:
  fixes p n :: nat
  shows ¬ prime p ⟹ multiplicity p n = 0
  apply (cases n = 0)
  apply auto
  apply (frule multiset-prime-factorization)
  apply (auto simp add: multiplicity-nat-def count-eq-zero-iff)
  done

lemma multiplicity-not-factor-nat [simp]:
  fixes p n :: nat
  shows p ∉ prime-factors n ⟹ multiplicity p n = 0
  apply (subst (asm) prime-factors-altdef-nat)
  apply auto
  done

lemma multiplicity-not-factor-int [simp]:
  fixes n :: int
  shows p ≥ 0 ⟹ p ∉ prime-factors n ⟹ multiplicity p n = 0
  apply (subst (asm) prime-factors-altdef-int)
  apply auto
  done


lemma multiplicity-product-aux-nat: (k::nat) > 0 ⟹ l > 0 ⟹
    (prime-factors k) ∪ (prime-factors l) = prime-factors (k ∗ l) ∧
    (∀ p. multiplicity p k + multiplicity p l = multiplicity p (k ∗ l))
  apply (rule prime-factorization-unique-nat)
  apply (simp only: prime-factors-altdef-nat)
  apply auto
  apply (subst power-add)
  apply (subst setprod.distrib)
  apply (rule arg-cong2 [where f = λx y. x∗y])
  apply (subgoal-tac prime-factors k ∪ prime-factors l = prime-factors k ∪
      (prime-factors l − prime-factors k))
  apply (erule ssubst)
  apply (subst setprod.union-disjoint)
  apply auto
  apply (metis One-nat-def nat-mult-1-right prime-factorization-nat setprod.neutral-const)
  apply (subgoal-tac prime-factors k ∪ prime-factors l = prime-factors l ∪
      (prime-factors k − prime-factors l))
  apply (erule ssubst)
  apply (subst setprod.union-disjoint)
  apply auto
  apply (subgoal-tac (∏ p∈prime-factors k − prime-factors l. p ˆ multiplicity p l)
```

=
 ($\prod p\in$*prime-factors k* $-$ *prime-factors l*. *1*))
 apply *auto*
 apply (*metis One-nat-def nat-mult-1-right prime-factorization-nat setprod.neutral-const*)
 done

lemma *multiplicity-product-aux-int*:
 assumes (k::*int*) $>$ *0* **and** $l > 0$
 shows *prime-factors k* $\cup$ *prime-factors l* = *prime-factors* ($k * l$) $\wedge$
 ($\forall p \geq 0$. *multiplicity p k* + *multiplicity p l* = *multiplicity p* ($k * l$))
 apply (*rule multiplicity-product-aux-nat* [*transferred*, *of l k*])
 using *assms* **apply** *auto*
 done

lemma *prime-factors-product-nat*: (k::*nat*) $>$ *0* $\Longrightarrow$ $l > 0$ $\Longrightarrow$ *prime-factors* ($k *$ l) =
 prime-factors k $\cup$ *prime-factors l*
 by (*rule multiplicity-product-aux-nat* [*THEN conjunct1*, *symmetric*])

lemma *prime-factors-product-int*: (k::*int*) $>$ *0* $\Longrightarrow$ $l > 0$ $\Longrightarrow$ *prime-factors* ($k *$ l) =
 prime-factors k $\cup$ *prime-factors l*
 by (*rule multiplicity-product-aux-int* [*THEN conjunct1*, *symmetric*])

lemma *multiplicity-product-nat*: (k::*nat*) $>$ *0* $\Longrightarrow$ $l > 0$ $\Longrightarrow$ *multiplicity p* ($k * l$) =
 multiplicity p k + *multiplicity p l*
 by (*rule multiplicity-product-aux-nat* [*THEN conjunct2*, *rule-format*, *symmetric*])

lemma *multiplicity-product-int*: (k::*int*) $>$ *0* $\Longrightarrow$ $l > 0$ $\Longrightarrow$ $p \geq 0$ $\Longrightarrow$
 multiplicity p ($k * l$) = *multiplicity p k* + *multiplicity p l*
 by (*rule multiplicity-product-aux-int* [*THEN conjunct2*, *rule-format*, *symmetric*])

lemma *multiplicity-setprod-nat*: *finite S* $\Longrightarrow$ $\forall x\in S$. $f\ x > 0$ $\Longrightarrow$
 multiplicity (p::*nat*) ($\prod x \in S$. $f\ x$) = ($\sum x \in S$. *multiplicity p* ($f\ x$))
 apply (*induct set*: *finite*)
 apply *auto*
 apply (*subst multiplicity-product-nat*)
 apply *auto*
 done

lemma *transfer-nat-int-sum-prod-closure3*: ($\sum x \in A$. *int* ($f\ x$)) $\geq$ *0* ($\prod x \in A$. *int* ($f\ x$)) $\geq$ *0*
 apply (*rule setsum-nonneg*; *auto*)
 apply (*rule setprod-nonneg*; *auto*)

done

declare *transfer-morphism-nat-int*[*transfer*
add return: *transfer-nat-int-sum-prod-closure3*
del: *transfer-nat-int-sum-prod2* (*1*)]

lemma *multiplicity-setprod-int*: $p \geq 0 \Longrightarrow$ *finite* $S \Longrightarrow \forall x{\in}S.\ f\ x > 0 \Longrightarrow$
multiplicity (*p*::*int*) ($\prod x \in S.\ f\ x$) = ($\sum x \in S.$ *multiplicity* *p* (*f* *x*))
apply (*frule multiplicity-setprod-nat*
[**where** $f = \lambda x.\ nat(int(nat(f\ x)))$,
transferred direction: *nat op* $\leq$ (*0*::*int*)])
apply *auto*
apply (*subst* (*asm*) *setprod.cong*)
apply (*rule refl*)
apply (*rule if-P*)
apply *auto*
apply (*rule setsum.cong*)
apply *auto*
done

declare *transfer-morphism-nat-int*[*transfer*
add return: *transfer-nat-int-sum-prod2* (*1*)]

lemma *multiplicity-prod-prime-powers-nat*:
finite $S \Longrightarrow \forall p{\in}S.$ *prime* (*p*::*nat*) $\Longrightarrow$
multiplicity *p* ($\prod p \in S.\ p\ \hat{}\ f\ p$) = (*if* $p \in S$ *then* *f* *p* *else* *0*)
apply (*subgoal-tac* ($\prod p \in S.\ p\ \hat{}\ f\ p$) = ($\prod p \in S.\ p\ \hat{}$ ($\lambda x.$ *if* $x \in S$ *then* *f* *x* *else* *0*) *p*))
apply (*erule ssubst*)
apply (*subst multiplicity-characterization-nat*)
prefer *5* **apply** (*rule refl*)
apply (*rule refl*)
apply *auto*
apply (*subst setprod.mono-neutral-right*)
apply *assumption*
prefer *3*
apply (*rule setprod.cong*)
apply (*rule refl*)
apply *auto*
done

lemma *multiplicity-prod-prime-powers-int*:
(*p*::*int*) $\geq 0 \Longrightarrow$ *finite* $S \Longrightarrow \forall p{\in}S.$ *prime* (*nat* *p*) $\Longrightarrow$
multiplicity *p* ($\prod p \in S.\ p\ \hat{}\ f\ p$) = (*if* $p \in S$ *then* *f* *p* *else* *0*)
apply (*subgoal-tac int* ‘ *nat* ‘ *S* = *S*)
apply (*frule multiplicity-prod-prime-powers-nat*
[**where** $f = \lambda x.\ f(int\ x)$ **and** *S* = *nat* ‘ *S*, *transferred*])

apply *auto*
apply (*metis linear nat-0-iff zero-not-prime-nat*)
apply (*metis* (*full-types*) *image-iff int-nat-eq less-le less-linear nat-0-iff zero-not-prime-nat*)
done

lemma *multiplicity-distinct-prime-power-nat*:
$prime\ p \Longrightarrow prime\ q \Longrightarrow p \neq q \Longrightarrow multiplicity\ p\ (q\ \hat{}\ n) = 0$
apply (*subgoal-tac* $q\ \hat{}\ n = setprod\ (\lambda x.\ x\ \hat{}\ n)\ \{q\}$)
apply (*erule ssubst*)
apply (*subst multiplicity-prod-prime-powers-nat*)
apply *auto*
done

lemma *multiplicity-distinct-prime-power-int*:
$prime\ p \Longrightarrow prime\ q \Longrightarrow p \neq q \Longrightarrow multiplicity\ p\ (int\ q\ \hat{}\ n) = 0$
by (*metis multiplicity-distinct-prime-power-nat of-nat-power transfer-int-nat-multiplicity*)

lemma *dvd-multiplicity-nat*:
fixes *x y* :: *nat*
shows $0 < y \Longrightarrow x\ dvd\ y \Longrightarrow multiplicity\ p\ x \leq multiplicity\ p\ y$
apply (*cases* $x = 0$)
apply (*auto simp add*: *dvd-def multiplicity-product-nat*)
done

lemma *dvd-multiplicity-int*:
fixes *p x y* :: *int*
shows $0 < y \Longrightarrow 0 \leq x \Longrightarrow x\ dvd\ y \Longrightarrow p \geq 0 \Longrightarrow multiplicity\ p\ x \leq multiplicity\ p\ y$
apply (*cases* $x = 0$)
apply (*auto simp add*: *dvd-def*)
apply (*subgoal-tac* $0 < k$)
apply (*auto simp add*: *multiplicity-product-int*)
apply (*erule zero-less-mult-pos*)
apply *arith*
done

lemma *dvd-prime-factors-nat* [*intro*]:
fixes *x y* :: *nat*
shows $0 < y \Longrightarrow x\ dvd\ y \Longrightarrow prime\text{-}factors\ x \leq prime\text{-}factors\ y$
apply (*simp only*: *prime-factors-altdef-nat*)
apply *auto*
apply (*metis dvd-multiplicity-nat le-0-eq neq0-conv*)
done

lemma *dvd-prime-factors-int* [*intro*]:
fixes *x y* :: *int*
shows $0 < y \Longrightarrow 0 \leq x \Longrightarrow x\ dvd\ y \Longrightarrow prime\text{-}factors\ x \leq prime\text{-}factors\ y$
apply (*auto simp add*: *prime-factors-altdef-int*)
apply (*metis dvd-multiplicity-int le-0-eq neq0-conv*)

done

lemma *multiplicity-dvd-nat*:
fixes *x y* :: *nat*
shows $0 < x \Longrightarrow 0 < y \Longrightarrow \forall p.\ multiplicity\ p\ x \leq multiplicity\ p\ y \Longrightarrow x\ dvd\ y$
apply (*subst prime-factorization-nat* [*of x*], *assumption*)
apply (*subst prime-factorization-nat* [*of y*], *assumption*)
apply (*rule setprod-dvd-setprod-subset2*)
apply *force*
apply (*subst prime-factors-altdef-nat*)+
apply *auto*
apply (*metis gr0I le-0-eq less-not-refl*)
apply (*metis le-imp-power-dvd*)
done

lemma *multiplicity-dvd-int*:
fixes *x y* :: *int*
shows $0 < x \Longrightarrow 0 < y \Longrightarrow \forall p \geq 0.\ multiplicity\ p\ x \leq multiplicity\ p\ y \Longrightarrow x$
$dvd\ y$
apply (*subst prime-factorization-int* [*of x*], *assumption*)
apply (*subst prime-factorization-int* [*of y*], *assumption*)
apply (*rule setprod-dvd-setprod-subset2*)
apply *force*
apply (*subst prime-factors-altdef-int*)+
apply *auto*
apply (*metis le-imp-power-dvd prime-factors-ge-0-int*)
done

lemma *multiplicity-dvd′-nat*:
fixes *x y* :: *nat*
assumes $0 < x$
assumes $\forall p.\ prime\ p \longrightarrow multiplicity\ p\ x \leq multiplicity\ p\ y$
shows *x dvd y*
using *dvd-0-right assms* **by** (*metis* (*no-types*) *le0 multiplicity-dvd-nat multiplicity-nonprime-nat not-gr0*)

lemma *multiplicity-dvd′-int*:
fixes *x y* :: *int*
shows $0 < x \Longrightarrow 0 \leq y \Longrightarrow$
$\forall p.\ prime\ p \longrightarrow multiplicity\ p\ x \leq multiplicity\ p\ y \Longrightarrow x\ dvd\ y$
by (*metis dvd-int-iff abs-of-nat multiplicity-dvd′-nat multiplicity-int-def nat-int zero-le-imp-eq-int zero-less-imp-eq-int*)

lemma *dvd-multiplicity-eq-nat*:
fixes *x y* :: *nat*
shows $0 < x \Longrightarrow 0 < y \Longrightarrow x\ dvd\ y \longleftrightarrow (\forall p.\ multiplicity\ p\ x \leq multiplicity\ p$
$y)$
by (*auto intro*: *dvd-multiplicity-nat multiplicity-dvd-nat*)

lemma *dvd-multiplicity-eq-int*: $0 < (x::int) \Longrightarrow 0 < y \Longrightarrow$
  $(x\ dvd\ y) = (\forall p \geq 0.\ multiplicity\ p\ x \leq multiplicity\ p\ y)$
 by (*auto intro*: *dvd-multiplicity-int multiplicity-dvd-int*)

lemma *prime-factors-altdef2-nat*:
 fixes *n* :: *nat*
 shows $n > 0 \Longrightarrow p \in prime\text{-}factors\ n \longleftrightarrow prime\ p \wedge p\ dvd\ n$
 apply (*cases prime p*)
 apply *auto*
 apply (*subst prime-factorization-nat* [**where** *n* = *n*], *assumption*)
 apply (*rule dvd-trans*)
 apply (*rule dvd-power* [**where** *x* = *p* **and** *n* = *multiplicity p n*])
 apply (*subst* (*asm*) *prime-factors-altdef-nat*, *force*)
 apply *rule*
 apply *auto*
 apply (*metis One-nat-def Zero-not-Suc dvd-multiplicity-nat le0*
  le-antisym multiplicity-not-factor-nat multiplicity-prime-nat)
 done

lemma *prime-factors-altdef2-int*:
 fixes *n* :: *int*
 assumes $n > 0$
 shows $p \in prime\text{-}factors\ n \longleftrightarrow prime\ p \wedge p\ dvd\ n$
 using *assms* **by** (*simp add*: *prime-factors-altdef2-nat* [*transferred*])

lemma *multiplicity-eq-nat*:
 fixes *x* **and** *y*::*nat*
 assumes [*arith*]: $x > 0\ y > 0$
  and *mult-eq* [*simp*]: $\bigwedge p.\ prime\ p \Longrightarrow multiplicity\ p\ x = multiplicity\ p\ y$
 shows $x = y$
 apply (*rule dvd-antisym*)
 apply (*auto intro*: *multiplicity-dvd′-nat*)
 done

lemma *multiplicity-eq-int*:
 fixes *x y* :: *int*
 assumes [*arith*]: $x > 0\ y > 0$
  and *mult-eq* [*simp*]: $\bigwedge p.\ prime\ p \Longrightarrow multiplicity\ p\ x = multiplicity\ p\ y$
 shows $x = y$
 apply (*rule dvd-antisym* [*transferred*])
 apply (*auto intro*: *multiplicity-dvd′-int*)
 done

3.5 An application

lemma *gcd-eq-nat*:
 fixes *x y* :: *nat*
 assumes *pos* [*arith*]: $x > 0\ y > 0$
 shows *gcd x y* =

(∏ *p* ∈ *prime-factors x* ∪ *prime-factors y*. *p* ^ *min* (*multiplicity p x*) (*multiplicity p y*))
(**is** - = *?z*)
proof −
have [*arith*]: *?z* > *0*
using *prime-factors-gt-0-nat* **by** *auto*
have *aux*: ⋀*p*. *prime p* ⟹ *multiplicity p ?z* = *min* (*multiplicity p x*) (*multiplicity p y*)
apply (*subst multiplicity-prod-prime-powers-nat*)
apply *auto*
done
have *?z dvd x*
by (*intro multiplicity-dvd'-nat*) (*auto simp add*: *aux intro*: *prime-gt-0-nat*)
moreover have *?z dvd y*
by (*intro multiplicity-dvd'-nat*) (*auto simp add*: *aux intro*: *prime-gt-0-nat*)
moreover have *w dvd x* ∧ *w dvd y* ⟶ *w dvd ?z* **for** *w*
proof (*cases w* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then show *?thesis*
apply *auto*
apply (*erule multiplicity-dvd'-nat*)
apply (*auto intro*: *dvd-multiplicity-nat simp add*: *aux*)
done
qed
ultimately have *?z* = *gcd x y*
by (*subst gcd-unique-nat* [*symmetric*], *blast*)
then show *?thesis*
by *auto*
qed

lemma *lcm-eq-nat*:
assumes *pos* [*arith*]: *x* > *0 y* > *0*
shows *lcm* (*x*::*nat*) *y* =
(∏ *p* ∈ *prime-factors x* ∪ *prime-factors y*. *p* ^ *max* (*multiplicity p x*) (*multiplicity p y*))
(**is** - = *?z*)
proof −
have [*arith*]: *?z* > *0*
by (*auto intro*: *prime-gt-0-nat*)
have *aux*: ⋀*p*. *prime p* ⟹ *multiplicity p ?z* = *max* (*multiplicity p x*) (*multiplicity p y*)
apply (*subst multiplicity-prod-prime-powers-nat*)
apply *auto*
done
have *x dvd ?z*
by (*intro multiplicity-dvd'-nat*) (*auto simp add*: *aux*)

```
  moreover have y dvd ?z
    by (intro multiplicity-dvd'-nat) (auto simp add: aux)
  moreover have x dvd w ∧ y dvd w ⟶ ?z dvd w for w
  proof (cases w = 0)
    case True
    then show ?thesis by auto
  next
    case False
    then show ?thesis
      apply auto
      apply (rule multiplicity-dvd'-nat)
      apply (auto intro: prime-gt-0-nat dvd-multiplicity-nat simp add: aux)
      done
  qed
  ultimately have ?z = lcm x y
    by (subst lcm-unique-nat [symmetric], blast)
  then show ?thesis
    by auto
qed

lemma multiplicity-gcd-nat:
  fixes p x y :: nat
  assumes [arith]: x > 0 y > 0
  shows multiplicity p (gcd x y) = min (multiplicity p x) (multiplicity p y)
  apply (subst gcd-eq-nat)
  apply auto
  apply (subst multiplicity-prod-prime-powers-nat)
  apply auto
  done

lemma multiplicity-lcm-nat:
  fixes p x y :: nat
  assumes [arith]: x > 0 y > 0
  shows multiplicity p (lcm x y) = max (multiplicity p x) (multiplicity p y)
  apply (subst lcm-eq-nat)
  apply auto
  apply (subst multiplicity-prod-prime-powers-nat)
  apply auto
  done

lemma gcd-lcm-distrib-nat:
  fixes x y z :: nat
  shows gcd x (lcm y z) = lcm (gcd x y) (gcd x z)
  apply (cases x = 0 | y = 0 | z = 0)
  apply auto
  apply (rule multiplicity-eq-nat)
  apply (auto simp add: multiplicity-gcd-nat multiplicity-lcm-nat lcm-pos-nat)
  done
```

lemma *gcd-lcm-distrib-int*:
  fixes *x y z* :: *int*
  shows *gcd x* (*lcm y z*) = *lcm* (*gcd x y*) (*gcd x z*)
  apply (*subst* (*1 2 3*) *gcd-abs-int*)
  apply (*subst lcm-abs-int*)
  apply (*subst* (*2*) *abs-of-nonneg*)
  apply *force*
  apply (*rule gcd-lcm-distrib-nat* [*transferred*])
  apply *auto*
  done

end

4 Things that can be added to the Algebra library

theory *MiscAlgebra*
imports
  ~~/*src*/*HOL*/*Algebra*/*Ring*
  ~~/*src*/*HOL*/*Algebra*/*FiniteProduct*
begin

4.1 Finiteness stuff

lemma *bounded-set1-int* [*intro*]: *finite* {(*x*::*int*). *a* < *x* & *x* < *b* & *P x*}
  apply (*subgoal-tac* {*x*. *a* < *x* & *x* < *b* & *P x*} <= {*a*<..<*b*})
  apply (*erule finite-subset*)
  apply *auto*
  done

4.2 The rest is for the algebra libraries

4.2.1 These go in Group.thy

Show that the units in any monoid give rise to a group.

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

definition *units-of* :: ('*a*, '*b*) *monoid-scheme* => '*a monoid* **where**
  units-of G == (| *carrier* = *Units G*,
    Group.monoid.mult = *Group.monoid.mult G*,
    one = *one G* |)

lemma (**in** *monoid*) *units-group*: *group*(*units-of G*)
  apply (*unfold units-of-def*)
  apply (*rule groupI*)
  apply *auto*
  apply (*subst m-assoc*)

apply *auto*
apply (*rule-tac x* = *inv x* **in** *bexI*)
apply *auto*
done

lemma (**in** *comm-monoid*) *units-comm-group*: *comm-group*(*units-of G*)
apply (*rule group.group-comm-groupI*)
apply (*rule units-group*)
apply (*insert comm-monoid-axioms*)
apply (*unfold units-of-def Units-def comm-monoid-def comm-monoid-axioms-def*)
apply *auto*
done

lemma *units-of-carrier*: *carrier* (*units-of G*) = *Units G*
unfolding *units-of-def* **by** *auto*

lemma *units-of-mult*: *mult*(*units-of G*) = *mult G*
unfolding *units-of-def* **by** *auto*

lemma *units-of-one*: *one*(*units-of G*) = *one G*
unfolding *units-of-def* **by** *auto*

lemma (**in** *monoid*) *units-of-inv*: *x* : *Units G* ==> *m-inv* (*units-of G*) *x* = *m-inv G x*
apply (*rule sym*)
apply (*subst m-inv-def*)
apply (*rule the1-equality*)
apply (*rule ex-ex1I*)
apply (*subst* (*asm*) *Units-def*)
apply *auto*
apply (*erule inv-unique*)
apply *auto*
apply (*rule Units-closed*)
apply (*simp-all only*: *units-of-carrier* [*symmetric*])
apply (*insert units-group*)
apply *auto*
apply (*subst units-of-mult* [*symmetric*])
apply (*subst units-of-one* [*symmetric*])
apply (*erule group.r-inv*, *assumption*)
apply (*subst units-of-mult* [*symmetric*])
apply (*subst units-of-one* [*symmetric*])
apply (*erule group.l-inv*, *assumption*)
done

lemma (**in** *group*) *inj-on-const-mult*: *a*: (*carrier G*) ==> *inj-on* (%*x*. *a* ⊗ *x*) (*carrier G*)
unfolding *inj-on-def* **by** *auto*

lemma (**in** *group*) *surj-const-mult*: *a* : (*carrier G*) ==> (%*x*. *a* ⊗ *x*) ‘ (*carrier*

G) = (*carrier G*)
  apply (*auto simp add*: *image-def*)
  apply (*rule-tac x* = (*m-inv G a*) ⊗ *x* **in** *bexI*)
  apply *auto*

  apply (*subst m-assoc* [*symmetric*])
  apply *auto*
  done

lemma (**in** *group*) *l-cancel-one* [*simp*]:
    x : *carrier G* ⟹ *a* : *carrier G* ⟹ (*x* ⊗ *a* = *x*) = (*a* = *one G*)
  apply *auto*
  apply (*subst l-cancel* [*symmetric*])
  prefer *4*
  apply (*erule ssubst*)
  apply *auto*
  done

lemma (**in** *group*) *r-cancel-one* [*simp*]: *x* : *carrier G* ⟹ *a* : *carrier G* ⟹
    (*a* ⊗ *x* = *x*) = (*a* = *one G*)
  apply *auto*
  apply (*subst r-cancel* [*symmetric*])
  prefer *4*
  apply (*erule ssubst*)
  apply *auto*
  done

lemma (**in** *group*) *l-cancel-one′* [*simp*]: *x* : *carrier G* ⟹ *a* : *carrier G* ⟹
    (*x* = *x* ⊗ *a*) = (*a* = *one G*)
  apply (*subst eq-commute*)
  apply *simp*
  done

lemma (**in** *group*) *r-cancel-one′* [*simp*]: *x* : *carrier G* ⟹ *a* : *carrier G* ⟹
    (*x* = *a* ⊗ *x*) = (*a* = *one G*)
  apply (*subst eq-commute*)
  apply *simp*
  done

lemma (**in** *comm-group*) *power-order-eq-one*:
  assumes *fin* [*simp*]: *finite* (*carrier G*)
    and *a* [*simp*]: *a* : *carrier G*
  shows *a* (ˆ) *card*(*carrier G*) = *one G*
proof −
  have (⨂*x*∈*carrier G*. *x*) = (⨂*x*∈*carrier G*. *a* ⊗ *x*)
    by (*subst* (*2*) *finprod-reindex* [*symmetric*],

auto simp add: *Pi-def inj-on-const-mult surj-const-mult*)
also have ... = ($\bigotimes$ *x*∈*carrier G*. *a*) ⊗ ($\bigotimes$ *x*∈*carrier G*. *x*)
by (*auto simp add*: *finprod-multf Pi-def*)
also have ($\bigotimes$ *x*∈*carrier G*. *a*) = *a* (^) *card*(*carrier G*)
by (*auto simp add*: *finprod-const*)
finally show *?thesis*

by *auto*
qed

4.2.2 Miscellaneous

lemma (**in** *cring*) *field-intro2*: $\mathbf{0}_R$ ~= $\mathbf{1}_R$ ⟹ ∀ *x* ∈ *carrier R* − {$\mathbf{0}_R$}. *x* ∈ *Units R* ⟹ *field R*
apply (*unfold-locales*)
apply (*insert cring-axioms*, *auto*)
apply (*rule trans*)
apply (*subgoal-tac a* = (*a* ⊗ *b*) ⊗ *inv b*)
apply *assumption*
apply (*subst m-assoc*)
apply *auto*
apply (*unfold Units-def*)
apply *auto*
done

lemma (**in** *monoid*) *inv-char*: *x* : *carrier G* ⟹ *y* : *carrier G* ⟹
x ⊗ *y* = **1** ⟹ *y* ⊗ *x* = **1** ⟹ *inv x* = *y*
apply (*subgoal-tac x* : *Units G*)
apply (*subgoal-tac y* = *inv x* ⊗ **1**)
apply *simp*
apply (*erule subst*)
apply (*subst m-assoc* [*symmetric*])
apply *auto*
apply (*unfold Units-def*)
apply *auto*
done

lemma (**in** *comm-monoid*) *comm-inv-char*: *x* : *carrier G* ⟹ *y* : *carrier G* ⟹
x ⊗ *y* = **1** ⟹ *inv x* = *y*
apply (*rule inv-char*)
apply *auto*
apply (*subst m-comm*, *auto*)
done

lemma (**in** *ring*) *inv-neg-one* [*simp*]: *inv* (⊖ **1**) = ⊖ **1**
apply (*rule inv-char*)
apply (*auto simp add*: *l-minus r-minus*)
done

lemma (**in** *monoid*) *inv-eq-imp-eq*: x : *Units G* $\Longrightarrow$ y : *Units G* $\Longrightarrow$
inv x = *inv y* $\Longrightarrow$ $x = y$
apply (*subgoal-tac inv*(*inv x*) = *inv*(*inv y*))
apply (*subst* (*asm*) *Units-inv-inv*)+
apply *auto*
done

lemma (**in** *ring*) *Units-minus-one-closed* [*intro*]: $\ominus$ **1** : *Units R*
apply (*unfold Units-def*)
apply *auto*
apply (*rule-tac* $x = \ominus$ **1** **in** *bexI*)
apply *auto*
apply (*simp add*: *l-minus r-minus*)
done

lemma (**in** *monoid*) *inv-one* [*simp*]: *inv* **1** = **1**
apply (*rule inv-char*)
apply *auto*
done

lemma (**in** *ring*) *inv-eq-neg-one-eq*: x : *Units R* $\Longrightarrow$ (*inv x* = $\ominus$ **1**) = (x = $\ominus$ **1**)
apply *auto*
apply (*subst Units-inv-inv* [*symmetric*])
apply *auto*
done

lemma (**in** *monoid*) *inv-eq-one-eq*: x : *Units G* $\Longrightarrow$ (*inv x* = **1**) = (x = **1**)
by (*metis Units-inv-inv inv-one*)

4.2.3 This goes in FiniteProduct

lemma (**in** *comm-monoid*) *finprod-UN-disjoint*:
finite I $\Longrightarrow$ (*ALL i*:*I*. *finite* (*A i*)) $\longrightarrow$ (*ALL i*:*I*. *ALL j*:*I*. i ~= j $\longrightarrow$
(*A i*) *Int* (*A j*) = {}) $\longrightarrow$
(*ALL i*:*I*. *ALL x*: (*A i*). *g x* : *carrier G*) $\longrightarrow$
finprod G g (*UNION I A*) = *finprod G* (%*i*. *finprod G g* (*A i*)) *I*
apply (*induct set*: *finite*)
apply *force*
apply *clarsimp*
apply (*subst finprod-Un-disjoint*)
apply *blast*
apply (*erule finite-UN-I*)
apply *blast*
apply (*fastforce*)
apply (*auto intro*!: *funcsetI finprod-closed*)
done

lemma (**in** *comm-monoid*) *finprod-Union-disjoint*:
[| *finite C*; (*ALL A*:*C*. *finite A* & (*ALL x*:*A*. *f x* : *carrier G*));

(ALL A:C. ALL B:C. A ~= B --> A Int B = {}) |]
==> finprod G f ($\bigcup$ *C*) = *finprod G* (*finprod G f*) *C*
apply (*frule finprod-UN-disjoint* [*of C id f*])
apply *auto*
done

lemma (**in** *comm-monoid*) *finprod-one*:
finite A $\Longrightarrow$ ($\bigwedge x.\ x{:}A \Longrightarrow f\ x = \mathbf{1}$) $\Longrightarrow$ *finprod G f A* = $\mathbf{1}$
by (*induct set*: *finite*) *auto*

lemma (**in** *cring*) *sum-zero-eq-neg*: x : *carrier R* $\Longrightarrow$ y : *carrier R* $\Longrightarrow$ $x \oplus y$ = $\mathbf{0}$ $\Longrightarrow$ $x = \ominus\ y$
by (*metis minus-equality*)

lemma (**in** *domain*) *square-eq-one*:
fixes x
assumes [*simp*]: x : *carrier R*
and $x \otimes x = \mathbf{1}$
shows $x = \mathbf{1} \mid x = \ominus\mathbf{1}$
proof −
have $(x \oplus \mathbf{1}) \otimes (x \oplus \ominus\ \mathbf{1}) = x \otimes x \oplus \ominus\ \mathbf{1}$
by (*simp add*: *ring-simprules*)
also from ‹$x \otimes x = \mathbf{1}$› **have** ... = $\mathbf{0}$
by (*simp add*: *ring-simprules*)
finally have $(x \oplus \mathbf{1}) \otimes (x \oplus \ominus\ \mathbf{1}) = \mathbf{0}$ **.**
then have $(x \oplus \mathbf{1}) = \mathbf{0} \mid (x \oplus \ominus\ \mathbf{1}) = \mathbf{0}$
by (*intro integral*, *auto*)
then show *?thesis*
apply *auto*
apply (*erule notE*)
apply (*rule sum-zero-eq-neg*)
apply *auto*
apply (*subgoal-tac* $x = \ominus\ (\ominus\ \mathbf{1})$)
apply (*simp add*: *ring-simprules*)
apply (*rule sum-zero-eq-neg*)
apply *auto*
done
qed

lemma (**in** *Ring.domain*) *inv-eq-self*: x : *Units R* $\Longrightarrow$ x = *inv* x $\Longrightarrow$ $x = \mathbf{1} \vee x = \ominus\mathbf{1}$
by (*metis Units-closed Units-l-inv square-eq-one*)

The following translates theorems about groups to the facts about the units of a ring. (The list should be expanded as more things are needed.)

lemma (**in** *ring*) *finite-ring-finite-units* [*intro*]: *finite* (*carrier R*) $\Longrightarrow$ *finite* (*Units R*)
by (*rule finite-subset*) *auto*

lemma (**in** *monoid*) *units-of-pow*:
fixes *n* :: *nat*
shows $x \in Units\ G \Longrightarrow x\ (\hat{})_{units\text{-}of\ G}\ n = x\ (\hat{})_G\ n$
apply (*induct n*)
apply (*auto simp add*: *units-group group.is-monoid monoid.nat-pow-0 monoid.nat-pow-Suc units-of-one units-of-mult*)
done

lemma (**in** *cring*) *units-power-order-eq-one*: *finite* (*Units R*) $\Longrightarrow$ *a* : *Units R* $\Longrightarrow$ *a* (^) *card*(*Units R*) = **1**
apply (*subst units-of-carrier* [*symmetric*])
apply (*subst units-of-one* [*symmetric*])
apply (*subst units-of-pow* [*symmetric*])
apply *assumption*
apply (*rule comm-group.power-order-eq-one*)
apply (*rule units-comm-group*)
apply (*unfold units-of-def*, *auto*)
done

end

5 Residue rings

theory *Residues*
imports *UniqueFactorization MiscAlgebra*
begin

5.1 A locale for residue rings

definition *residue-ring* :: *int* ⇒ *int ring*
where
residue-ring m =
(|*carrier* = {*0..m* − *1*},
mult = λ*x y*. (*x* ∗ *y*) *mod m*,
one = *1*,
zero = *0*,
add = λ*x y*. (*x* + *y*) *mod m*|)

locale *residues* =
fixes *m* :: *int* **and** *R* (**structure**)
assumes *m-gt-one*: *m* > *1*
defines *R* ≡ *residue-ring m*
begin

lemma *abelian-group*: *abelian-group R*

apply (*insert m-gt-one*)
apply (*rule abelian-groupI*)
apply (*unfold R-def residue-ring-def*)
apply (*auto simp add*: *mod-add-right-eq* [*symmetric*] *ac-simps*)
apply (*case-tac* $x = 0$)
apply *force*
apply (*subgoal-tac* $(x + (m - x))$ *mod* $m = 0$)
apply (*erule bexI*)
apply *auto*
done

lemma *comm-monoid*: *comm-monoid R*
apply (*insert m-gt-one*)
apply (*unfold R-def residue-ring-def*)
apply (*rule comm-monoidI*)
apply *auto*
apply (*subgoal-tac* $x * y$ *mod* $m * z$ *mod* $m = z * (x * y$ *mod* $m)$ *mod* m)
apply (*erule ssubst*)
apply (*subst mod-mult-right-eq* [*symmetric*])+
apply (*simp-all only*: *ac-simps*)
done

lemma *cring*: *cring R*
apply (*rule cringI*)
apply (*rule abelian-group*)
apply (*rule comm-monoid*)
apply (*unfold R-def residue-ring-def*, *auto*)
apply (*subst mod-add-eq* [*symmetric*])
apply (*subst mult.commute*)
apply (*subst mod-mult-right-eq* [*symmetric*])
apply (*simp add*: *field-simps*)
done

end

sublocale *residues* < *cring*
by (*rule cring*)

context *residues*
begin

These lemmas translate back and forth between internal and external concepts.

lemma *res-carrier-eq*: *carrier R* = $\{0..m - 1\}$
unfolding *R-def residue-ring-def* **by** *auto*

lemma *res-add-eq*: $x \oplus y = (x + y)$ *mod* m
unfolding *R-def residue-ring-def* **by** *auto*

lemma *res-mult-eq*: $x \otimes y = (x * y)$ *mod* m
  unfolding *R-def residue-ring-def* **by** *auto*

lemma *res-zero-eq*: $\mathbf{0} = 0$
  unfolding *R-def residue-ring-def* **by** *auto*

lemma *res-one-eq*: $\mathbf{1} = 1$
  unfolding *R-def residue-ring-def units-of-def* **by** *auto*

lemma *res-units-eq*: *Units R* = $\{x.\ 0 < x \wedge x < m \wedge$ *coprime x m*$\}$
  apply (*insert m-gt-one*)
  apply (*unfold Units-def R-def residue-ring-def*)
  apply *auto*
  apply (*subgoal-tac* $x \neq 0$)
  apply *auto*
  apply (*metis invertible-coprime-int*)
  apply (*subst* (*asm*) *coprime-iff-invertible′-int*)
  apply (*auto simp add*: *cong-int-def mult.commute*)
  done

lemma *res-neg-eq*: $\ominus\ x = (-\ x)$ *mod* m
  apply (*insert m-gt-one*)
  apply (*unfold R-def a-inv-def m-inv-def residue-ring-def*)
  apply *auto*
  apply (*rule the-equality*)
  apply *auto*
  apply (*subst mod-add-right-eq* [*symmetric*])
  apply *auto*
  apply (*subst mod-add-left-eq* [*symmetric*])
  apply *auto*
  apply (*subgoal-tac* y *mod* $m = -\ x$ *mod* m)
  apply *simp*
  apply (*metis minus-add-cancel mod-mult-self1 mult.commute*)
  done

lemma *finite* [*iff*]: *finite* (*carrier R*)
  by (*subst res-carrier-eq*) *auto*

lemma *finite-Units* [*iff*]: *finite* (*Units R*)
  by (*subst res-units-eq*) *auto*

The function $a \mapsto a$ *mod* m maps the integers to the residue classes. The following lemmas show that this mapping respects addition and multiplication on the integers.

lemma *mod-in-carrier* [*iff*]: a *mod* $m \in$ *carrier R*
  unfolding *res-carrier-eq*
  using *insert m-gt-one* **by** *auto*

lemma *add-cong*: (*x mod m*) ⊕ (*y mod m*) = (*x* + *y*) *mod m*
unfolding *R-def residue-ring-def*
apply *auto*
apply *presburger*
done

lemma *mult-cong*: (*x mod m*) ⊗ (*y mod m*) = (*x* ∗ *y*) *mod m*
unfolding *R-def residue-ring-def*
by *auto* (*metis mod-mult-eq*)

lemma *zero-cong*: **0** = *0*
unfolding *R-def residue-ring-def* **by** *auto*

lemma *one-cong*: **1** = *1 mod m*
using *m-gt-one* **unfolding** *R-def residue-ring-def* **by** *auto*

lemma *pow-cong*: (*x mod m*) (ˆ) *n* = *x*ˆ*n mod m*
apply (*insert m-gt-one*)
apply (*induct n*)
apply (*auto simp add*: *nat-pow-def one-cong*)
apply (*metis mult.commute mult-cong*)
done

lemma *neg-cong*: ⊖ (*x mod m*) = (− *x*) *mod m*
by (*metis mod-minus-eq res-neg-eq*)

lemma (**in** *residues*) *prod-cong*: *finite A* ⟹ (⨂*i*∈*A*. (*f i*) *mod m*) = (∏*i*∈*A*. *f i*) *mod m*
by (*induct set*: *finite*) (*auto simp*: *one-cong mult-cong*)

lemma (**in** *residues*) *sum-cong*: *finite A* ⟹ (⨁*i*∈*A*. (*f i*) *mod m*) = (∑*i*∈*A*. *f i*) *mod m*
by (*induct set*: *finite*) (*auto simp*: *zero-cong add-cong*)

lemma *mod-in-res-units* [*simp*]:
assumes *1* < *m* **and** *coprime a m*
shows *a mod m* ∈ *Units R*
proof (*cases a mod m* = *0*)
case *True* **with** *assms* **show** *?thesis*
by (*auto simp add*: *res-units-eq gcd-red-int* [*symmetric*])
next
case *False*
from *assms* **have** *0* < *m* **by** *simp*
with *pos-mod-sign* [*of m a*] **have** *0* ≤ *a mod m* **.**
with *False* **have** *0* < *a mod m* **by** *simp*
with *assms* **show** *?thesis*
by (*auto simp add*: *res-units-eq gcd-red-int* [*symmetric*] *ac-simps*)
qed

lemma *res-eq-to-cong*: (*a mod m*) = (*b mod m*) ⟷ [*a* = *b*] (*mod m*)
unfolding *cong-int-def* **by** *auto*

Simplifying with these will translate a ring equation in R to a congruence.

lemmas *res-to-cong-simps* = *add-cong mult-cong pow-cong one-cong prod-cong sum-cong neg-cong res-eq-to-cong*

Other useful facts about the residue ring.

lemma *one-eq-neg-one*: **1** = ⊖ **1** ⟹ *m* = *2*
apply (*simp add*: *res-one-eq res-neg-eq*)
apply (*metis add.commute add-diff-cancel mod-mod-trivial one-add-one uminus-add-conv-diff zero-neq-one zmod-zminus1-eq-if*)
done

end

5.2 Prime residues

locale *residues-prime* =
fixes *p* **and** *R* (**structure**)
assumes *p-prime* [*intro*]: *prime p*
defines *R* ≡ *residue-ring p*

sublocale *residues-prime* < *residues p*
apply (*unfold R-def residues-def*)
using *p-prime* **apply** *auto*
apply (*metis* (*full-types*) *of-nat-1 of-nat-less-iff prime-gt-1-nat*)
done

context *residues-prime*
begin

lemma *is-field*: *field R*
apply (*rule cring.field-intro2*)
apply (*rule cring*)
apply (*auto simp add*: *res-carrier-eq res-one-eq res-zero-eq res-units-eq*)
apply (*rule classical*)
apply (*erule notE*)
apply (*subst gcd.commute*)
apply (*rule prime-imp-coprime-int*)
apply (*rule p-prime*)
apply (*rule notI*)
apply (*frule zdvd-imp-le*)
apply *auto*
done

lemma *res-prime-units-eq*: *Units R* = {*1..p* − *1*}
apply (*subst res-units-eq*)

```
  apply auto
  apply (subst gcd.commute)
  apply (auto simp add: p-prime prime-imp-coprime-int zdvd-not-zless)
  done

end

sublocale residues-prime < field
  by (rule is-field)
```

6 Test cases: Euler's theorem and Wilson's theorem

6.1 Euler's theorem

The definition of the phi function.

```
definition phi :: int ⇒ nat
  where phi m = card {x. 0 < x ∧ x < m ∧ gcd x m = 1}

lemma phi-def-nat: phi m = card {x. 0 < x ∧ x < nat m ∧ gcd x (nat m) = 1}
  apply (simp add: phi-def)
  apply (rule bij-betw-same-card [of nat])
  apply (auto simp add: inj-on-def bij-betw-def image-def)
  apply (metis dual-order.irrefl dual-order.strict-trans leI nat-1 transfer-nat-int-gcd(1))
  apply (metis One-nat-def of-nat-0 of-nat-1 of-nat-less-0-iff int-nat-eq nat-int
    transfer-int-nat-gcd(1) of-nat-less-iff)
  done

lemma prime-phi:
  assumes 2 ≤ p phi p = p − 1
  shows prime p
proof −
  have ∗: {x. 0 < x ∧ x < p ∧ coprime x p} = {1..p − 1}
    using assms unfolding phi-def-nat
    by (intro card-seteq) fastforce+
  have False if ∗∗: 1 < x x < p and x dvd p for x :: nat
  proof −
    from ∗ have cop: x ∈ {1..p − 1} =⇒ coprime x p
      by blast
    have coprime x p
      apply (rule cop)
      using ∗∗ apply auto
      done
    with ‹x dvd p› ‹1 < x› show ?thesis
      by auto
  qed
  then show ?thesis
    using ‹2 ≤ p›
```

```
    by (simp add: prime-def)
      (metis One-nat-def dvd-pos-nat nat-dvd-not-less nat-neq-iff not-gr0
          not-numeral-le-zero one-dvd)
qed

lemma phi-zero [simp]: phi 0 = 0
  unfolding phi-def

  apply (auto simp add: card-eq-0-iff)

  done

lemma phi-one [simp]: phi 1 = 0
  by (auto simp add: phi-def card-eq-0-iff)

lemma (in residues) phi-eq: phi m = card (Units R)
  by (simp add: phi-def res-units-eq)

lemma (in residues) euler-theorem1:
  assumes a: gcd a m = 1
  shows [a ^phi m = 1] (mod m)
proof -
  from a m-gt-one have [simp]: a mod m ∈ Units R
    by (intro mod-in-res-units)
  from phi-eq have (a mod m) (^) (phi m) = (a mod m) (^) (card (Units R))
    by simp
  also have ... = 1
    by (intro units-power-order-eq-one) auto
  finally show ?thesis
    by (simp add: res-to-cong-simps)
qed
```

Outside the locale, we can relax the restriction $m > 1$.

```
lemma euler-theorem:
  assumes m ≥ 0
    and gcd a m = 1
  shows [a ^phi m = 1] (mod m)
proof (cases m = 0 | m = 1)
  case True
  then show ?thesis by auto
next
  case False
  with assms show ?thesis
    by (intro residues.euler-theorem1, unfold residues-def, auto)
qed

lemma (in residues-prime) phi-prime: phi p = nat p - 1
  apply (subst phi-eq)
  apply (subst res-prime-units-eq)
```

apply *auto*
done

lemma *phi-prime*: *prime p* $\Longrightarrow$ *phi p* = *nat p* − *1*
apply (*rule residues-prime.phi-prime*)
apply (*erule residues-prime.intro*)
done

lemma *fermat-theorem*:
fixes *a* :: *int*
assumes *prime p*
and ¬ *p dvd a*
shows [*a*ˆ(*p* − *1*) = *1*] (*mod p*)
proof −
from *assms* **have** [*a* ˆ *phi p* = *1*] (*mod p*)
by (*auto intro*!: *euler-theorem dest*!: *prime-imp-coprime-int simp add*: *ac-simps*)
also have *phi p* = *nat p* − *1*
by (*rule phi-prime*) (*rule assms*)
finally show *?thesis*
by (*metis nat-int*)
qed

lemma *fermat-theorem-nat*:
assumes *prime p* **and** ¬ *p dvd a*
shows [*a* ˆ (*p* − *1*) = *1*] (*mod p*)
using *fermat-theorem* [*of p a*] *assms*
by (*metis of-nat-1 of-nat-power transfer-int-nat-cong zdvd-int*)

6.2 Wilson's theorem

lemma (**in** *field*) *inv-pair-lemma*: *x* ∈ *Units R* $\Longrightarrow$ *y* ∈ *Units R* $\Longrightarrow$
{*x*, *inv x*} ≠ {*y*, *inv y*} $\Longrightarrow$ {*x*, *inv x*} ∩ {*y*, *inv y*} = {}
apply *auto*
apply (*metis Units-inv-inv*)+
done

lemma (**in** *residues-prime*) *wilson-theorem1*:
assumes *a*: *p* > *2*
shows [*fact* (*p* − *1*) = (−*1*::*int*)] (*mod p*)
proof −
let *?Inverse-Pairs* = {{*x*, *inv x*}| *x*. *x* ∈ *Units R* − {**1**, ⊖ **1**}}
have *UR*: *Units R* = {**1**, ⊖ **1**} ∪ ⋃ *?Inverse-Pairs*
by *auto*
have (⨂ *i*∈*Units R*. *i*) = (⨂ *i*∈{**1**, ⊖ **1**}. *i*) ⊗ (⨂ *i*∈⋃ *?Inverse-Pairs*. *i*)
apply (*subst UR*)
apply (*subst finprod-Un-disjoint*)
apply (*auto intro*: *funcsetI*)
using *inv-one* **apply** *auto*[*1*]
using *inv-eq-neg-one-eq* **apply** *auto*

done
also have ($\bigotimes i\in\{\mathbf{1}, \ominus \mathbf{1}\}.\ i) = \ominus \mathbf{1}$
apply (*subst finprod-insert*)
apply *auto*
apply (*frule one-eq-neg-one*)
using *a* **apply** *force*
done
also have ($\bigotimes i\in(\bigcup$ *?Inverse-Pairs*). $i) = (\bigotimes A\in$ *?Inverse-Pairs*. $(\bigotimes y\in A.\ y))$
apply (*subst finprod-Union-disjoint*)
apply *auto*
apply (*metis Units-inv-inv*)+
done
also have ... $= \mathbf{1}$
apply (*rule finprod-one*)
apply *auto*
apply (*subst finprod-insert*)
apply *auto*
apply (*metis inv-eq-self*)
done
finally have ($\bigotimes i\in$ *Units R*. $i) = \ominus \mathbf{1}$
by *simp*
also have ($\bigotimes i\in$ *Units R*. $i) = (\bigotimes i\in$ *Units R*. *i mod p*)
apply (*rule finprod-cong'*)
apply *auto*
apply (*subst* (*asm*) *res-prime-units-eq*)
apply *auto*
done
also have ... $= (\prod i\in$ *Units R*. $i)$ *mod p*
apply (*rule prod-cong*)
apply *auto*
done
also have ... = *fact* (*p* − *1*) *mod p*
apply (*subst fact-altdef-nat*)
apply (*insert assms*)
apply (*subst res-prime-units-eq*)
apply (*simp add*: *int-setprod zmod-int setprod-int-eq*)
done
finally have *fact* (*p* − *1*) *mod p* $= \ominus \mathbf{1}$ **.**
then show *?thesis*
by (*metis of-nat-fact Divides.transfer-int-nat-functions*(*2*)
cong-int-def res-neg-eq res-one-eq)
qed

lemma *wilson-theorem*:
assumes *prime p*
shows [*fact* (*p* − *1*) = − *1*] (*mod p*)
proof (*cases p* = *2*)
case *True*
then show *?thesis*

```
    by (simp add: cong-int-def fact-altdef-nat)
next
  case False
  then show ?thesis
    using assms prime-ge-2-nat
    by (metis residues-prime.wilson-theorem1 residues-prime.intro le-eq-less-or-eq)
qed

end
```

7 Pocklington's Theorem for Primes

```
theory Pocklington
imports Residues
begin
```

7.1 Lemmas about previously defined terms

```
lemma prime:
  prime p ⟷ p ≠ 0 ∧ p ≠ 1 ∧ (∀m. 0 < m ∧ m < p ⟶ coprime p m)
  (is ?lhs ⟷ ?rhs)
proof−
  {assume p=0 ∨ p=1 hence ?thesis
    by (metis one-not-prime-nat zero-not-prime-nat)}
  moreover
  {assume p0: p≠0 p≠1
    {assume H: ?lhs
      {fix m assume m: m > 0 m < p
        {assume m=1 hence coprime p m by simp}
        moreover
        {assume p dvd m hence p ≤ m using dvd-imp-le m by blast with m(2)
          have coprime p m by simp}
        ultimately have coprime p m
          by (metis H prime-imp-coprime-nat)}
      hence ?rhs using p0 by auto}
    moreover
    { assume H: ∀m. 0 < m ∧ m < p ⟶ coprime p m
      obtain q where q: prime q q dvd p
        by (metis p0(2) prime-factor-nat)
      have q0: q > 0
        by (metis prime-gt-0-nat q(1))
      from dvd-imp-le[OF q(2)] p0 have qp: q ≤ p by arith
      {assume q = p hence ?lhs using q(1) by blast}
      moreover
      {assume q≠p with qp have qplt: q < p by arith
        from H qplt q0 have coprime p q by arith
       hence ?lhs using q
         by (auto dest: gcd-nat.absorb2)}
      ultimately have ?lhs by blast}
```

ultimately have *?thesis* **by** *blast*}
ultimately show *?thesis* **by** (*casesp=0 ∨ p=1*, *auto*)
qed

lemma *finite-number-segment*: *card* { *m*. *0* < *m* ∧ *m* < *n* } = *n* − *1*
proof−
have { *m*. *0* < *m* ∧ *m* < *n* } = {*1*..<*n*} **by** *auto*
thus *?thesis* **by** *simp*
qed

7.2 Some basic theorems about solving congruences

lemma *cong-solve*:
fixes *n*::*nat* **assumes** *an*: *coprime a n* **shows** ∃ *x*. [*a* ∗ *x* = *b*] (*mod n*)
proof−
{**assume** *a=0* **hence** *?thesis* **using** *an* **by** (*simp add*: *cong-nat-def*)}
moreover
{**assume** *az*: *a≠0*
from *bezout-add-strong-nat*[*OF az*, *of n*]
obtain *d x y* **where** *dxy*: *d dvd a d dvd n a*∗*x* = *n*∗*y* + *d* **by** *blast*
from *dxy*(*1*,*2*) **have** *d1*: *d* = *1*
by (*metis assms coprime-nat*)
hence *a*∗*x*∗*b* = (*n*∗*y* + *1*)∗*b* **using** *dxy*(*3*) **by** *simp*
hence *a*∗(*x*∗*b*) = *n*∗(*y*∗*b*) + *b*
by (*auto simp add*: *algebra-simps*)
hence *a*∗(*x*∗*b*) *mod n* = (*n*∗(*y*∗*b*) + *b*) *mod n* **by** *simp*
hence *a*∗(*x*∗*b*) *mod n* = *b mod n* **by** (*simp add*: *mod-add-left-eq*)
hence [*a*∗(*x*∗*b*) = *b*] (*mod n*) **unfolding** *cong-nat-def* **.**
hence *?thesis* **by** *blast*}
ultimately show *?thesis* **by** *blast*
qed

lemma *cong-solve-unique*:
fixes *n*::*nat* **assumes** *an*: *coprime a n* **and** *nz*: *n* ≠ *0*
shows ∃!*x*. *x* < *n* ∧ [*a* ∗ *x* = *b*] (*mod n*)
proof−
let *?P* = λ*x*. *x* < *n* ∧ [*a* ∗ *x* = *b*] (*mod n*)
from *cong-solve*[*OF an*] **obtain** *x* **where** *x*: [*a*∗*x* = *b*] (*mod n*) **by** *blast*
let *?x* = *x mod n*
from *x* **have** *th*: [*a* ∗ *?x* = *b*] (*mod n*)
by (*simp add*: *cong-nat-def mod-mult-right-eq*[*of a x n*])
from *mod-less-divisor*[*of n x*] *nz th* **have** *Px*: *?P ?x* **by** *simp*
{**fix** *y* **assume** *Py*: *y* < *n* [*a* ∗ *y* = *b*] (*mod n*)
from *Py*(*2*) *th* **have** [*a* ∗ *y* = *a*∗*?x*] (*mod n*) **by** (*simp add*: *cong-nat-def*)
hence [*y* = *?x*] (*mod n*)
by (*metis an cong-mult-lcancel-nat*)
with *mod-less*[*OF Py*(*1*)] *mod-less-divisor*[*of n x*] *nz*
have *y* = *?x* **by** (*simp add*: *cong-nat-def*)}
with *Px* **show** *?thesis* **by** *blast*

qed

lemma *cong-solve-unique-nontrivial*:
  assumes *p*: *prime p* **and** *pa*: *coprime p a* **and** *x0*: $0 < x$ **and** *xp*: $x < p$
  shows $\exists! y.\ 0 < y \wedge y < p \wedge [x * y = a]\ (mod\ p)$
proof−
  from *pa* **have** *ap*: *coprime a p*
    by (*metis gcd.commute*)
  have *px*:*coprime x p*
    by (*metis gcd.commute p prime x0 xp*)
  obtain *y* **where** *y*: $y < p$ $[x * y = a]\ (mod\ p)$ $\forall z.\ z < p \wedge [x * z = a]\ (mod\ p) \longrightarrow z = y$
    by (*metis cong-solve-unique neq0-conv p prime-gt-0-nat px*)
  {**assume** *y0*: $y = 0$
    with *y*(*2*) **have** *th*: *p dvd a*
      by (*auto dest*: *cong-dvd-eq-nat*)
    have *False*
      by (*metis gcd-nat.absorb1 one-not-prime-nat p pa th*)}
  with *y* **show** *?thesis* **unfolding** *Ex1-def* **using** *neq0-conv* **by** *blast*
qed

lemma *cong-unique-inverse-prime*:
  assumes *prime p* **and** $0 < x$ **and** $x < p$
  shows $\exists! y.\ 0 < y \wedge y < p \wedge [x * y = 1]\ (mod\ p)$
  by (*rule cong-solve-unique-nontrivial*) (*insert assms*, *simp-all*)

lemma *chinese-remainder-coprime-unique*:
  fixes *a*::*nat*
  assumes *ab*: *coprime a b* **and** *az*: $a \neq 0$ **and** *bz*: $b \neq 0$
  and *ma*: *coprime m a* **and** *nb*: *coprime n b*
  shows $\exists! x.\ coprime\ x\ (a * b) \wedge x < a * b \wedge [x = m]\ (mod\ a) \wedge [x = n]\ (mod\ b)$
proof−
  let $?P = \lambda x.\ x < a * b \wedge [x = m]\ (mod\ a) \wedge [x = n]\ (mod\ b)$
  from *binary-chinese-remainder-unique-nat*[*OF ab az bz*]
  obtain *x* **where** *x*: $x < a * b$ $[x = m]\ (mod\ a)$ $[x = n]\ (mod\ b)$
    $\forall y.\ ?P\ y \longrightarrow y = x$ **by** *blast*
  from *ma nb x*
  have *coprime x a coprime x b*
    by (*metis cong-gcd-eq-nat*)+
  then have *coprime x* (*a*∗*b*)
    by (*metis coprime-mul-eq*)
  with *x* **show** *?thesis* **by** *blast*
qed

7.3 Lucas's theorem

lemma *phi-limit-strong*: $phi(n) \leq n - 1$
proof −

have *phi n* = *card* {*x*. *0* < *x* ∧ *x* < *int n* ∧ *coprime x* (*int n*)}
  by (*simp add*: *phi-def*)
also have ... ≤ *card* {*0* <..< *int n*}
  by (*rule card-mono*) *auto*
also have ... = *card* {*0* <..< *n*}
  by (*simp add*: *transfer-nat-int-set-functions*)
also have ... ≤ *n* − *1*
  by (*metis card-greaterThanLessThan le-refl One-nat-def*)
finally show *?thesis* **.**
qed

lemma *phi-lowerbound-1*: **assumes** *n*: *n* ≥ *2*
  shows *phi n* ≥ *1*
proof −
  have *1* ≤ *card* {*0*::*int* <.. *1*}
    by *auto*
  also have ... ≤ *card* {*x*. *0* < *x* ∧ *x* < *n* ∧ *coprime x n*}
    apply (*rule card-mono*) **using** *assms*
   by *auto* (*metis dual-order.antisym gcd-1-int gcd.commute int-one-le-iff-zero-less*)
  also have ... = *phi n*
    by (*simp add*: *phi-def*)
  finally show *?thesis* **.**
qed

lemma *phi-lowerbound-1-nat*: **assumes** *n*: *n* ≥ *2*
  shows *phi*(*int n*) ≥ *1*
by (*metis n nat-le-iff nat-numeral phi-lowerbound-1*)

lemma *euler-theorem-nat*:
  fixes *m*::*nat*
  assumes *coprime a m*
  shows [*a* ˆ *phi m* = *1*] (*mod m*)
by (*metis assms le0 euler-theorem* [*transferred*])

lemma *lucas-coprime-lemma*:
  fixes *n*::*nat*
  assumes *m*: *m*≠*0* **and** *am*: [*a*ˆ*m* = *1*] (*mod n*)
  shows *coprime a n*
proof−
  {**assume** *n*=*1* **hence** *?thesis* **by** *simp*}
  moreover
  {**assume** *n* = *0* **hence** *?thesis* **using** *am m*
    by (*metis am cong-0-nat gcd-nat.right-neutral power-eq-one-eq-nat*)}
  moreover
  {**assume** *n*: *n*≠*0 n*≠*1*
    from *m* **obtain** *m′* **where** *m′*: *m* = *Suc m′* **by** (*cases m*, *blast*+)
    {**fix** *d*
      assume *d*: *d dvd a d dvd n*
      from *n* **have** *n1*: *1* < *n* **by** *arith*

```
    from am mod-less[OF n1] have am1: a^m mod n = 1 unfolding cong-nat-def
by simp
     from dvd-mult2[OF d(1), of a^m] have dam:d dvd a^m by (simp add: m')
     from dvd-mod-iff[OF d(2), of a^m] dam am1
     have d = 1 by simp }
   hence ?thesis by auto
  }
  ultimately show ?thesis by blast
qed

lemma lucas-weak:
  fixes n::nat
  assumes n: n ≥ 2 and an:[a^(n - 1) = 1] (mod n)
  and nm: ∀ m. 0 <m ∧ m < n - 1 ⟶ ¬ [a^m = 1] (mod n)
  shows prime n
proof-
  from n have n1: n ≠ 1 n≠0 n - 1 ≠ 0 n - 1 > 0 n - 1 < n by arith+
  from lucas-coprime-lemma[OF n1(3) an] have can: coprime a n .
  from euler-theorem-nat[OF can] have afn: [a ^ phi n = 1] (mod n)
    by auto
  {assume phi n ≠ n - 1
    with phi-limit-strong phi-lowerbound-1-nat [OF n]
    have c:phi n > 0 ∧ phi n < n - 1
      by (metis gr0I leD less-linear not-one-le-zero)
    from nm[rule-format, OF c] afn have False ..}
  hence phi n = n - 1 by blast
  with prime-phi phi-prime n1(1,2) show ?thesis
    by auto
qed

lemma nat-exists-least-iff: (∃ (n::nat). P n) ⟷ (∃ n. P n ∧ (∀ m < n. ¬ P m))
  by (metis ex-least-nat-le not-less0)

lemma nat-exists-least-iff': (∃ (n::nat). P n) ⟷ (P (Least P) ∧ (∀ m < (Least
P). ¬ P m))
  (is ?lhs ⟷ ?rhs)
proof-
  {assume ?rhs hence ?lhs by blast}
  moreover
  { assume H: ?lhs then obtain n where n: P n by blast
    let ?x = Least P
    {fix m assume m: m < ?x
      from not-less-Least[OF m] have ¬ P m .}
    with LeastI-ex[OF H] have ?rhs by blast}
  ultimately show ?thesis by blast
qed

theorem lucas:
  assumes n2: n ≥ 2 and an1: [a^(n - 1) = 1] (mod n)
```

and *pn*: $\forall p.$ *prime p* $\wedge$ *p dvd n* $-$ *1* $\longrightarrow$ [*a*ˆ((*n* $-$ *1*) *div p*) $\neq$ *1*] (*mod n*)
shows *prime n*
proof$-$
from *n2* **have** *n01*: *n*$\neq$*0 n*$\neq$*1 n* $-$ *1* $\neq$ *0* **by** *arith*+
from *mod-less-divisor*[*of n 1*] *n01* **have** *onen*: *1 mod n* = *1* **by** *simp*
from *lucas-coprime-lemma*[*OF n01*(*3*) *an1*] *cong-imp-coprime-nat an1*
have *an*: *coprime a n coprime* (*a*ˆ(*n* $-$ *1*)) *n*
by (*auto simp add*: *coprime-exp gcd.commute*)
{**assume** *H0*: $\exists$ *m*. *0* < *m* $\wedge$ *m* < *n* $-$ *1* $\wedge$ [*a* ˆ *m* = *1*] (*mod n*) (**is** *EX m*. *?P m*)
from *H0*[*unfolded nat-exists-least-iff*[*of ?P*]] **obtain** *m* **where**
m: *0* < *m m* < *n* $-$ *1* [*a* ˆ *m* = *1*] (*mod n*) $\forall$ *k* <*m*. $\neg$*?P k* **by** *blast*
{**assume** *nm1*: (*n* $-$ *1*) *mod m* > *0*
from *mod-less-divisor*[*OF m*(*1*)] **have** *th0*:(*n* $-$ *1*) *mod m* < *m* **by** *blast*
let *?y* = *a*ˆ ((*n* $-$ *1*) *div m* $*$ *m*)
note *mdeq* = *mod-div-equality*[*of* (*n* $-$ *1*) *m*]
have *yn*: *coprime ?y n*
by (*metis an*(*1*) *coprime-exp gcd.commute*)
have *?y mod n* = (*a*ˆ*m*) ˆ((*n* $-$ *1*) *div m*) *mod n*
by (*simp add*: *algebra-simps power-mult*)
also have ... = (*a*ˆ*m mod n*) ˆ((*n* $-$ *1*) *div m*) *mod n*
using *power-mod*[*of a*ˆ*m n* (*n* $-$ *1*) *div m*] **by** *simp*
also have ... = *1* **using** *m*(*3*)[*unfolded cong-nat-def onen*] *onen*
by (*metis power-one*)
finally have *th3*: *?y mod n* = *1* **.**
have *th2*: [*?y* $*$ *a* ˆ ((*n* $-$ *1*) *mod m*) = *?y*$*$ *1*] (*mod n*)
using *an1*[*unfolded cong-nat-def onen*] *onen*
mod-div-equality[*of* (*n* $-$ *1*) *m*, *symmetric*]
by (*simp add*:*power-add*[*symmetric*] *cong-nat-def th3 del*: *One-nat-def*)
have *th1*: [*a* ˆ ((*n* $-$ *1*) *mod m*) = *1*] (*mod n*)
by (*metis cong-mult-rcancel-nat mult.commute th2 yn*)
from *m*(*4*)[*rule-format*, *OF th0*] *nm1*
less-trans[*OF mod-less-divisor*[*OF m*(*1*), *of n* $-$ *1*] *m*(*2*)] *th1*
have *False* **by** *blast* }
hence (*n* $-$ *1*) *mod m* = *0* **by** *auto*
then have *mn*: *m dvd n* $-$ *1* **by** *presburger*
then obtain *r* **where** *r*: *n* $-$ *1* = *m*$*$*r* **unfolding** *dvd-def* **by** *blast*
from *n01 r m*(*2*) **have** *r01*: *r*$\neq$*0 r*$\neq$*1* **by** $-$ (*rule ccontr*, *simp*)+
obtain *p* **where** *p*: *prime p p dvd r*
by (*metis prime-factor-nat r01*(*2*))
hence *th*: *prime p* $\wedge$ *p dvd n* $-$ *1* **unfolding** *r* **by** (*auto intro*: *dvd-mult*)
have (*a* ˆ ((*n* $-$ *1*) *div p*)) *mod n* = (*a*ˆ(*m*$*$*r div p*)) *mod n* **using** *r*
by (*simp add*: *power-mult*)
also have ... = (*a*ˆ(*m*$*$(*r div p*))) *mod n*
using *div-mult1-eq*[*of m r p*] *p*(*2*)[*unfolded dvd-eq-mod-eq-0*]
by *simp*
also have ... = ((*a*ˆ*m*) ˆ(*r div p*)) *mod n* **by** (*simp add*: *power-mult*)
also have ... = ((*a*ˆ*m mod n*) ˆ(*r div p*)) *mod n* **using** *power-mod* **..**
also have ... = *1* **using** *m*(*3*) *onen* **by** (*simp add*: *cong-nat-def*)

finally have $[(a \,\hat{}\, ((n - 1)\ div\ p)) = 1]\ (mod\ n)$
  using *onen* **by** (*simp add*: *cong-nat-def*)
with *pn th* **have** *False* **by** *blast*}
hence *th*: $\forall m.\ 0 < m \wedge m < n - 1 \longrightarrow \neg\ [a \,\hat{}\, m = 1]\ (mod\ n)$ **by** *blast*
from *lucas-weak*[*OF n2 an1 th*] **show** *?thesis* **.**
qed

7.4 Definition of the order of a number mod n (0 in non-coprime case)

definition *ord n a* = (*if coprime n a then Least* ($\lambda d.\ d > 0 \wedge [a \,\hat{}d = 1]\ (mod\ n)$) *else 0*)

lemma *coprime-ord*:
fixes *n*::*nat*
assumes *coprime n a*
shows $ord\ n\ a > 0 \wedge [a \,\hat{}(ord\ n\ a) = 1]\ (mod\ n) \wedge (\forall m.\ 0 < m \wedge m < ord\ n\ a \longrightarrow [a\hat{}\ m \neq 1]\ (mod\ n))$
proof−
let $?P = \lambda d.\ 0 < d \wedge [a \,\hat{}\, d = 1]\ (mod\ n)$
from *bigger-prime*[*of a*] **obtain** *p* **where** *p*: *prime p a* < *p* **by** *blast*
from *assms* **have** *o*: *ord n a* = *Least ?P* **by** (*simp add*: *ord-def*)
{**assume** $n=0 \vee n=1$ **with** *assms* **have** $\exists m>0.\ ?P\ m$
  by *auto*}
moreover
{**assume** $n\neq 0 \wedge n\neq 1$ **hence** *n2*:$n \geq 2$ **by** *arith*
from *assms* **have** *na'*: *coprime a n*
  by (*metis gcd.commute*)
from *phi-lowerbound-1-nat*[*OF n2*] *euler-theorem-nat* [*OF na'*]
have *ex*: $\exists m>0.\ ?P\ m$ **by** − (*rule exI*[**where** *x=phi n*], *auto*) }
ultimately have *ex*: $\exists m>0.\ ?P\ m$ **by** *blast*
from *nat-exists-least-iff'*[*of ?P*] *ex assms* **show** *?thesis*
  unfolding *o*[*symmetric*] **by** *auto*
qed

lemma *ord-works*:
fixes *n*::*nat*
shows $[a \,\hat{}\, (ord\ n\ a) = 1]\ (mod\ n) \wedge (\forall m.\ 0 < m \wedge m < ord\ n\ a \longrightarrow \sim[a\hat{}\ m = 1]\ (mod\ n))$
apply (*cases coprime n a*)
using *coprime-ord*[*of n a*]
by (*auto simp add*: *ord-def cong-nat-def*)

lemma *ord*:
fixes *n*::*nat*
shows $[a\hat{}(ord\ n\ a) = 1]\ (mod\ n)$ **using** *ord-works* **by** *blast*

```
lemma ord-minimal:
  fixes n::nat
  shows 0 < m ⟹ m < ord n a ⟹ ~[a^m = 1] (mod n)
  using ord-works by blast

lemma ord-eq-0:
  fixes n::nat
  shows ord n a = 0 ⟷ ~coprime n a
by (cases coprime n a, simp add: coprime-ord, simp add: ord-def)

lemma divides-rexp:
  x dvd y ⟹ (x::nat) dvd (y^(Suc n))
  by (simp add: dvd-mult2[of x y])

lemma ord-divides:
  fixes n::nat
  shows [a ^ d = 1] (mod n) ⟷ ord n a dvd d (is ?lhs ⟷ ?rhs)
proof
  assume rh: ?rhs
  then obtain k where d = ord n a * k unfolding dvd-def by blast
  hence [a ^ d = (a ^ (ord n a) mod n)^k] (mod n)
    by (simp add : cong-nat-def power-mult power-mod)
  also have [(a ^ (ord n a) mod n)^k = 1] (mod n)
    using ord[of a n, unfolded cong-nat-def]
    by (simp add: cong-nat-def power-mod)
  finally show ?lhs .
next
  assume lh: ?lhs
  { assume H: ¬ coprime n a
    hence o: ord n a = 0 by (simp add: ord-def)
    {assume d: d=0 with o H have ?rhs by (simp add: cong-nat-def)}
    moreover
    {assume d0: d≠0 then obtain d' where d': d = Suc d' by (cases d, auto)
      from H
      obtain p where p: p dvd n p dvd a p ≠ 1 by auto
      from lh
      obtain q1 q2 where q12:a ^ d + n * q1 = 1 + n * q2
        by (metis H d0 gcd.commute lucas-coprime-lemma)
      hence a ^ d + n * q1 − n * q2 = 1 by simp
      with dvd-diff-nat [OF dvd-add [OF divides-rexp]]  dvd-mult2  d' p
      have p dvd 1
        by metis
      with p(3) have False by simp
      hence ?rhs ..}
    ultimately have ?rhs by blast}
  moreover
  {assume H: coprime n a
    let ?o = ord n a
```

```
    let ?q = d div ord n a
    let ?r = d mod ord n a
    have eqo: [(a^?o)^?q = 1] (mod n)
      by (metis cong-exp-nat ord power-one)
    from H have onz: ?o ≠ 0 by (simp add: ord-eq-0)
    hence op: ?o > 0 by simp
    from mod-div-equality[of d ord n a] lh
   have [a^(?o*?q + ?r) = 1] (mod n) by (simp add: cong-nat-def mult.commute)
    hence [(a^?o)^?q * (a^?r) = 1] (mod n)
      by (simp add: cong-nat-def power-mult[symmetric] power-add[symmetric])
    hence th: [a^?r = 1] (mod n)
      using eqo mod-mult-left-eq[of (a^?o)^?q a^?r n]
      apply (simp add: cong-nat-def del: One-nat-def)
      by (simp add: mod-mult-left-eq[symmetric])
    {assume r: ?r = 0 hence ?rhs by (simp add: dvd-eq-mod-eq-0)}
    moreover
    {assume r: ?r ≠ 0
      with mod-less-divisor[OF op, of d] have r0o:?r >0 ∧ ?r < ?o by simp
      from conjunct2[OF ord-works[of a n], rule-format, OF r0o] th
      have ?rhs by blast}
    ultimately have ?rhs by blast}
  ultimately  show ?rhs by blast
qed

lemma order-divides-phi:
  fixes n::nat shows coprime n a ⟹ ord n a dvd phi n
  by (metis ord-divides euler-theorem-nat gcd.commute)

lemma order-divides-expdiff:
  fixes n::nat and a::nat assumes na: coprime n a
  shows [a^d = a^e] (mod n) ⟷ [d = e] (mod (ord n a))
proof−
  {fix n::nat and a::nat and d::nat and e::nat
    assume na: coprime n a and ed: (e::nat) ≤ d
    hence ∃ c. d = e + c by presburger
    then obtain c where c: d = e + c by presburger
    from na have an: coprime a n
      by (metis gcd.commute)
    have aen: coprime (a^e) n
      by (metis coprime-exp gcd.commute na)
    have acn: coprime (a^c) n
      by (metis coprime-exp gcd.commute na)
    have [a^d = a^e] (mod n) ⟷ [a^(e + c) = a^(e + 0)] (mod n)
      using c by simp
    also have ... ⟷ [a^e* a^c = a^e *a^0] (mod n) by (simp add: power-add)
    also have  ... ⟷ [a ^ c = 1] (mod n)
      using cong-mult-lcancel-nat [OF aen, of a^c a^0] by simp
    also  have ... ⟷ ord n a dvd c by (simp only: ord-divides)
    also have ... ⟷ [e + c = e + 0] (mod ord n a)
```

using *cong-add-lcancel-nat*
by (*metis cong-dvd-eq-nat dvd-0-right cong-dvd-modulus-nat cong-mult-self-nat nat-mult-1*)
finally have $[a\hat{\ }d = a\hat{\ }e]$ (*mod n*) $\longleftrightarrow$ $[d = e]$ (*mod* (*ord n a*))
using *c* **by** *simp* **}**
note *th* = *this*
have $e \leq d \vee d \leq e$ **by** *arith*
moreover
{assume *ed*: $e \leq d$ **from** *th*[*OF na ed*] **have** *?thesis* **.}**
moreover
{assume *de*: $d \leq e$
from *th*[*OF na de*] **have** *?thesis*
by (*metis cong-sym-nat*)**}**
ultimately show *?thesis* **by** *blast*
qed

7.5 Another trivial primality characterization

lemma *prime-prime-factor*:
prime n $\longleftrightarrow$ $n \neq 1 \wedge (\forall p.\ prime\ p \wedge p\ dvd\ n \longrightarrow p = n)$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof (*cases* $n=0 \vee n=1$)
case *True*
then show *?thesis*
by (*metis bigger-prime dvd-0-right one-not-prime-nat zero-not-prime-nat*)
next
case *False*
show *?thesis*
proof
assume *prime n*
then show *?rhs*
by (*metis one-not-prime-nat prime-def*)
next
assume *?rhs*
with *False* **show** *prime n*
by (*auto simp*: *prime-def*) (*metis One-nat-def prime-factor-nat prime-def*)
qed
qed

lemma *prime-divisor-sqrt*:
prime n $\longleftrightarrow$ $n \neq 1 \wedge (\forall d.\ d\ dvd\ n \wedge d^2 \leq n \longrightarrow d = 1)$
proof −
{assume $n=0 \vee n=1$ **hence** *?thesis*
by *auto***}**
moreover
{assume *n*: $n \neq 0$ $n \neq 1$
hence *np*: $n > 1$ **by** *arith*
{fix *d* **assume** *d*: *d dvd n* $d^2 \leq n$ **and** *H*: $\forall m.\ m\ dvd\ n \longrightarrow m=1 \vee m=n$
from *H d* **have** *d1n*: $d = 1 \vee d=n$ **by** *blast*

```
      {assume dn: d=n
        have n^2 > n*1 using n by (simp add: power2-eq-square)
        with dn d(2) have d=1 by simp}
      with d1n have d = 1 by blast }
    moreover
    {fix d assume d: d dvd n and H: ∀ d'. d' dvd n ∧ d'^2 ≤ n ⟶ d' = 1
      from d n have d ≠ 0
        by (metis dvd-0-left-iff)
      hence dp: d > 0 by simp
      from d[unfolded dvd-def] obtain e where e: n= d*e by blast
      from n dp e have ep:e > 0 by simp
      have d^2 ≤ n ∨ e^2 ≤ n using dp ep
        by (auto simp add: e power2-eq-square mult-le-cancel-left)
      moreover
      {assume h: d^2 ≤ n
        from H[rule-format, of d] h d have d = 1 by blast}
      moreover
      {assume h: e^2 ≤ n
        from e have e dvd n unfolding dvd-def by (simp add: mult.commute)
        with H[rule-format, of e] h have e=1 by simp
        with e have d = n by simp}
      ultimately have d=1 ∨ d=n  by blast}
    ultimately have ?thesis unfolding prime-def using np n(2) by blast}
  ultimately show ?thesis by auto
qed

lemma prime-prime-factor-sqrt:
  prime n ⟷ n ≠ 0 ∧ n ≠ 1 ∧ ¬ (∃ p. prime p ∧ p dvd n ∧ p^2 ≤ n)
  (is ?lhs ⟷?rhs)
proof−
  {assume n=0 ∨ n=1
   hence ?thesis
     by (metis one-not-prime-nat zero-not-prime-nat)}
  moreover
  {assume n: n≠0 n≠1
    {assume H: ?lhs
      from H[unfolded prime-divisor-sqrt] n
      have ?rhs
        by (metis prime-prime-factor) }
    moreover
    {assume H: ?rhs
      {fix d assume d: d dvd n d^2 ≤ n d≠1
        then obtain p where p: prime p p dvd d
          by (metis prime-factor-nat)
        from d(1) n have dp: d > 0
          by (metis dvd-0-left neq0-conv)
       from mult-mono[OF dvd-imp-le[OF p(2) dp] dvd-imp-le[OF p(2) dp]] d(2)
        have p^2 ≤ n unfolding power2-eq-square by arith
        with H n p(1) dvd-trans[OF p(2) d(1)] have False  by blast}
```

with *n prime-divisor-sqrt* **have** *?lhs* **by** *auto*}
ultimately have *?thesis* **by** *blast* }
ultimately show *?thesis* **by** (*cases n=0* ∨ *n=1*, *auto*)
qed

7.6 Pocklington theorem

lemma *pocklington-lemma*:
assumes *n*: *n* ≥ *2* **and** *nqr*: *n* − *1* = *q*∗*r* **and** *an*: [*a*ˆ (*n* − *1*) = *1*] (*mod n*)
and *aq*:∀ *p*. *prime p* ∧ *p dvd q* ⟶ *coprime* (*a*ˆ ((*n* − *1*) *div p*) − *1*) *n*
and *pp*: *prime p* **and** *pn*: *p dvd n*
shows [*p* = *1*] (*mod q*)
proof −
have *p01*: *p* ≠ *0 p* ≠ *1* **using** *pp one-not-prime-nat zero-not-prime-nat* **by** (*auto intro*: *prime-gt-0-nat*)
obtain *k* **where** *k*: *a* ˆ (*q* ∗ *r*) − *1* = *n*∗*k*
by (*metis an cong-to-1-nat dvd-def nqr*)
from *pn*[*unfolded dvd-def*] **obtain** *l* **where** *l*: *n* = *p*∗*l* **by** *blast*
{**assume** *a0*: *a* = *0*
hence *a*ˆ (*n* − *1*) = *0* **using** *n* **by** (*simp add*: *power-0-left*)
with *n an mod-less*[*of 1 n*] **have** *False* **by** (*simp add*: *power-0-left cong-nat-def*)}
hence *a0*: *a*≠*0* **..**
from *n nqr* **have** *aqr0*: *a* ˆ (*q* ∗ *r*) ≠ *0* **using** *a0* **by** *simp*
hence (*a* ˆ (*q* ∗ *r*) − *1*) + *1* = *a* ˆ (*q* ∗ *r*) **by** *simp*
with *k l* **have** *a* ˆ (*q* ∗ *r*) = *p*∗*l*∗*k* + *1* **by** *simp*
hence *a* ˆ (*r* ∗ *q*) + *p* ∗ *0* = *1* + *p* ∗ (*l*∗*k*) **by** (*simp add*: *ac-simps*)
hence *odq*: *ord p* (*a*ˆ*r*) *dvd q*
unfolding *ord-divides*[*symmetric*] *power-mult*[*symmetric*]
by (*metis an cong-dvd-modulus-nat mult.commute nqr pn*)
from *odq*[*unfolded dvd-def*] **obtain** *d* **where** *d*: *q* = *ord p* (*a*ˆ*r*) ∗ *d* **by** *blast*
{**assume** *d1*: *d* ≠ *1*
obtain *P* **where** *P*: *prime P P dvd d*
by (*metis d1 prime-factor-nat*)
from *d dvd-mult*[*OF P*(*2*), *of ord p* (*a*ˆ*r*)] **have** *Pq*: *P dvd q* **by** *simp*
from *aq P*(*1*) *Pq* **have** *caP*:*coprime* (*a*ˆ ((*n* − *1*) *div P*) − *1*) *n* **by** *blast*
from *Pq* **obtain** *s* **where** *s*: *q* = *P*∗*s* **unfolding** *dvd-def* **by** *blast*
have *P0*: *P* ≠ *0* **using** *P*(*1*)
by (*metis zero-not-prime-nat*)
from *P*(*2*) **obtain** *t* **where** *t*: *d* = *P*∗*t* **unfolding** *dvd-def* **by** *blast*
from *d s t P0* **have** *s'*: *ord p* (*a*ˆ*r*) ∗ *t* = *s*
by (*metis mult.commute mult-cancel1 mult.assoc*)
have *ord p* (*a*ˆ*r*) ∗ *t*∗*r* = *r* ∗ *ord p* (*a*ˆ*r*) ∗ *t*
by (*metis mult.assoc mult.commute*)
hence *exps*: *a*ˆ(*ord p* (*a*ˆ*r*) ∗ *t*∗*r*) = ((*a* ˆ *r*) ˆ *ord p* (*a*ˆ*r*)) ˆ *t*
by (*simp only*: *power-mult*)
then have *th*: [((*a* ˆ *r*) ˆ *ord p* (*a*ˆ*r*)) ˆ *t*= *1*] (*mod p*)
by (*metis cong-exp-nat ord power-one*)
have *pd0*: *p dvd a*ˆ(*ord p* (*a*ˆ*r*) ∗ *t*∗*r*) − *1*
by (*metis cong-to-1-nat exps th*)

from *nqr s s'* **have** *(n − 1) div P = ord p (a ^r) ∗ t∗r* **using** *P0* **by** *simp*
with *caP* **have** *coprime (a ^(ord p (a ^r) ∗ t∗r) − 1) n* **by** *simp*
with *p01 pn pd0 coprime-common-divisor-nat* **have** *False*
by *auto*}
hence *d1*: *d = 1* **by** *blast*
hence *o*: *ord p (a ^r) = q* **using** *d* **by** *simp*
from *pp phi-prime[of p]* **have** *phip*: *phi p = p − 1* **by** *simp*
{**fix** *d* **assume** *d*: *d dvd p d dvd a d ≠ 1*
from *pp[unfolded prime-def] d* **have** *dp*: *d = p* **by** *blast*
from *n* **have** *n ≠ 0* **by** *simp*
then have *False* **using** *d dp pn*
by *auto (metis One-nat-def Suc-pred an dvd-1-iff-1 gcd-greatest-iff lucas-coprime-lemma)*}

hence *cpa*: *coprime p a* **by** *auto*
have *arp*: *coprime (a ^r) p*
by *(metis coprime-exp cpa gcd.commute)*
from *euler-theorem-nat[OF arp, simplified ord-divides] o phip*
have *q dvd (p − 1)* **by** *simp*
then obtain *d* **where** *d*:*p − 1 = q ∗ d*
unfolding *dvd-def* **by** *blast*
have *p0*:*p ≠ 0*
by *(metis p01(1))*
from *p0 d* **have** *p + q ∗ 0 = 1 + q ∗ d* **by** *simp*
then show *?thesis*
by *(metis cong-iff-lin-nat mult.commute)*
qed

theorem *pocklington*:
assumes *n*: *n ≥ 2* **and** *nqr*: *n − 1 = q∗r* **and** *sqr*: *n ≤ q*^{2}
and *an*: *[a ^ (n − 1) = 1] (mod n)*
and *aq*: *∀ p. prime p ∧ p dvd q ⟶ coprime (a ^ ((n − 1) div p) − 1) n*
shows *prime n*
unfolding *prime-prime-factor-sqrt[of n]*
proof−
let *?ths = n ≠ 0 ∧ n ≠ 1 ∧ ¬ (∃ p. prime p ∧ p dvd n ∧ p*^{2} *≤ n)*
from *n* **have** *n01*: *n≠0 n≠1* **by** *arith+*
{**fix** *p* **assume** *p*: *prime p p dvd n p*^{2} *≤ n*
from *p(3) sqr* **have** *p ^(Suc 1) ≤ q ^(Suc 1)* **by** *(simp add: power2-eq-square)*
hence *pq*: *p ≤ q*
by *(metis le0 power-le-imp-le-base)*
from *pocklington-lemma[OF n nqr an aq p(1,2)]*
have *th*: *q dvd p − 1*
by *(metis cong-to-1-nat)*
have *p − 1 ≠ 0* **using** *prime-ge-2-nat [OF p(1)]* **by** *arith*
with *pq th* **have** *False*
by *(simp add: nat-dvd-not-less)*}
with *n01* **show** *?ths* **by** *blast*
qed

```
lemma pocklington-alt:
  assumes n: n ≥ 2 and nqr: n − 1 = q∗r and sqr: n ≤ q^2
  and an: [a^ (n − 1) = 1] (mod n)
  and aq:∀ p. prime p ∧ p dvd q ⟶ (∃ b. [a^((n − 1) div p) = b] (mod n) ∧
coprime (b − 1) n)
  shows prime n
proof−
  {fix p assume p: prime p p dvd q
    from aq[rule-format] p obtain b where
      b: [a^((n − 1) div p) = b] (mod n) coprime (b − 1) n by blast
    {assume a0: a=0
      from n an have [0 = 1] (mod n) unfolding a0 power-0-left by auto
     hence False using n by (simp add: cong-nat-def dvd-eq-mod-eq-0[symmetric])}
    hence a0: a≠ 0 ..
    hence a1: a ≥ 1 by arith
    from one-le-power[OF a1] have ath: 1 ≤ a ^ ((n − 1) div p) .
    {assume b0: b = 0
      from p(2) nqr have (n − 1) mod p = 0
        by (metis mod-0 mod-mod-cancel mod-mult-self1-is-0)
      with mod-div-equality[of n − 1 p]
      have (n − 1) div p ∗ p= n − 1 by auto
      hence eq: (a^((n − 1) div p))^p = a^(n − 1)
        by (simp only: power-mult[symmetric])
      have p − 1 ≠ 0 using prime-ge-2-nat [OF p(1)] by arith
      then have pS: Suc (p − 1) = p by arith
      from b have d: n dvd a^((n − 1) div p) unfolding b0
        by auto
      from divides-rexp[OF d, of p − 1] pS eq cong-dvd-eq-nat [OF an] n
      have False
        by simp}
    then have b0: b ≠ 0 ..
    hence b1: b ≥ 1 by arith
   from cong-imp-coprime-nat[OF Cong.cong-diff-nat[OF cong-sym-nat [OF b(1)]
cong-refl-nat[of 1] b1]]
        ath b1 b nqr
    have coprime (a ^ ((n − 1) div p) − 1) n
      by simp}
  hence th: ∀ p. prime p ∧ p dvd q ⟶ coprime (a ^ ((n − 1) div p) − 1) n
    by blast
  from pocklington[OF n nqr sqr an th] show ?thesis .
qed
```

7.7 Prime factorizations

```
definition primefact ps n = (foldr op ∗ ps  1 = n ∧ (∀ p∈ set ps. prime p))

lemma primefact: assumes n: n ≠ 0
  shows ∃ ps. primefact ps n
```

using *n*
proof(*induct n rule*: *nat-less-induct*)
  fix *n* **assume** *H*: $\forall m<n.\ m \neq 0 \longrightarrow (\exists ps.\ primefact\ ps\ m)$ **and** *n*: $n\neq 0$
  let *?ths* = $\exists ps.\ primefact\ ps\ n$
  {assume $n = 1$
    hence *primefact* [] *n* **by** (*simp add*: *primefact-def*)
    hence *?ths* **by** *blast* **}**
  moreover
  {assume *n1*: $n \neq 1$
    with *n* **have** *n2*: $n \geq 2$ **by** *arith*
    obtain *p* **where** *p*: *prime p p dvd n*
      by (*metis n1 prime-factor-nat*)
    from *p*(*2*) **obtain** *m* **where** *m*: $n = p*m$ **unfolding** *dvd-def* **by** *blast*
    from *n m* **have** *m0*: $m > 0$ $m\neq 0$ **by** *auto*
    have $1 < p$
      by (*metis p*(*1*) *prime-def*)
    with *m0 m* **have** *mn*: $m < n$ **by** *auto*
    from *H*[*rule-format*, *OF mn m0*(*2*)] **obtain** *ps* **where** *ps*: *primefact ps m* **..**
    from *ps m p*(*1*) **have** *primefact* (*p#ps*) *n* **by** (*simp add*: *primefact-def*)
    hence *?ths* **by** *blast***}**
  ultimately show *?ths* **by** *blast*
qed

lemma *primefact-contains*:
  assumes *pf*: *primefact ps n* **and** *p*: *prime p* **and** *pn*: *p dvd n*
  shows $p \in set\ ps$
  using *pf p pn*
proof(*induct ps arbitrary*: *p n*)
  case *Nil* **thus** *?case* **by** (*auto simp add*: *primefact-def*)
next
  case (*Cons q qs p n*)
  from *Cons.prems*[*unfolded primefact-def*]
  have *q*: $prime\ q\ q * foldr\ op * qs\ 1 = n\ \forall p \in set\ qs.\ prime\ p$ **and** *p*: $prime\ p\ p\ dvd\ q * foldr\ op * qs\ 1$ **by** *simp-all*
  {assume *p dvd q*
    with *p*(*1*) *q*(*1*) **have** $p = q$ **unfolding** *prime-def* **by** *auto*
    hence *?case* **by** *simp***}**
  moreover
  { assume *h*: $p\ dvd\ foldr\ op * qs\ 1$
    from *q*(*3*) **have** *pqs*: *primefact qs* ($foldr\ op * qs\ 1$)
      by (*simp add*: *primefact-def*)
    from *Cons.hyps*[*OF pqs p*(*1*) *h*] **have** *?case* **by** *simp***}**
  ultimately show *?case*
    by (*metis p prime-dvd-mult-eq-nat*)
qed

lemma *primefact-variant*: $primefact\ ps\ n \longleftrightarrow foldr\ op * ps\ 1 = n \wedge list\text{-}all\ prime\ ps$
  by (*auto simp add*: *primefact-def list-all-iff*)

```
lemma lucas-primefact:
  assumes n: n ≥ 2 and an: [aˆ(n − 1) = 1] (mod n)
  and psn: foldr op ∗ ps 1 = n − 1
  and psp: list-all (λp. prime p ∧ ¬ [aˆ((n − 1) div p) = 1] (mod n)) ps
  shows prime n
proof−
  {fix p assume p: prime p p dvd n − 1 [a ˆ ((n − 1) div p) = 1] (mod n)
    from psn psp have psn1: primefact ps (n − 1)
      by (auto simp add: list-all-iff primefact-variant)
    from p(3) primefact-contains[OF psn1 p(1,2)] psp
    have False by (induct ps, auto)}
  with lucas[OF n an] show ?thesis by blast
qed
```

```
lemma pocklington-primefact:
  assumes n: n ≥ 2 and qrn: q∗r = n − 1 and nq2: n ≤ q^2
  and arnb: (aˆr) mod n = b and psq: foldr op ∗ ps 1 = q
  and bqn: (bˆq) mod n = 1
  and psp: list-all (λp. prime p ∧ coprime ((bˆ(q div p)) mod n − 1) n) ps
  shows prime n
proof−
  from bqn psp qrn
  have bqn: a ˆ (n − 1) mod n = 1
   and psp: list-all (λp. prime p ∧ coprime (aˆ(r ∗(q div p)) mod n − 1) n) ps
    unfolding arnb[symmetric] power-mod
    by (simp-all add: power-mult[symmetric] algebra-simps)
  from n  have n0: n > 0 by arith
  from mod-div-equality[of aˆ(n − 1) n]
    mod-less-divisor[OF n0, of aˆ(n − 1)]
  have an1: [a ˆ (n − 1) = 1] (mod n)
    by (metis bqn cong-nat-def mod-mod-trivial)
  {fix p assume p: prime p p dvd q
    from psp psq have pfpsq: primefact ps q
      by (auto simp add: primefact-variant list-all-iff)
    from psp primefact-contains[OF pfpsq p]
    have p': coprime (a ˆ (r ∗ (q div p)) mod n − 1) n
      by (simp add: list-all-iff)
    from p prime-def have p01: p ≠ 0 p ≠ 1 p =Suc(p − 1)
      by auto
    from div-mult1-eq[of r q p] p(2)
    have eq1: r∗ (q div p) = (n − 1) div p
      unfolding qrn[symmetric] dvd-eq-mod-eq-0 by (simp add: mult.commute)
    have ath: ⋀a (b::nat). a <= b =⇒ a ≠ 0 ==> 1 <= a ∧ 1 <= b by arith
    {assume a ˆ ((n − 1) div p) mod n = 0
```

then obtain *s* **where** *s*: $a \hat{} ((n - 1) \ div\ p) = n*s$
unfolding *mod-eq-0-iff* **by** *blast*
hence *eq0*: $(a \hat{}((n - 1) \ div\ p)) \hat{} p = (n*s) \hat{} p$ **by** *simp*
from *qrn[symmetric]* **have** *qn1*: $q\ dvd\ n - 1$ **unfolding** *dvd-def* **by** *auto*
from *dvd-trans[OF p(2) qn1]*
have *npp*: $(n - 1)\ div\ p * p = n - 1$ **by** *simp*
with *eq0* **have** $a \hat{} (n - 1) = (n*s) \hat{} p$
by (*simp add*: *power-mult[symmetric]*)
hence $1 = (n*s) \hat{} (Suc\ (p - 1))\ mod\ n$ **using** *bqn p01* **by** *simp*
also have $\ldots = 0$ **by** (*simp add*: *mult.assoc*)
finally have *False* **by** *simp* **}**
then have *th11*: $a \hat{} ((n - 1)\ div\ p)\ mod\ n \neq 0$ **by** *auto*
have *th1*: $[a \hat{} ((n - 1)\ div\ p)\ mod\ n = a \hat{} ((n - 1)\ div\ p)]\ (mod\ n)$
unfolding *cong-nat-def* **by** *simp*
from *th1 ath[OF mod-less-eq-dividend th11]*
have *th*: $[a \hat{} ((n - 1)\ div\ p)\ mod\ n - 1 = a \hat{} ((n - 1)\ div\ p) - 1]\ (mod\ n)$
by (*metis cong-diff-nat cong-refl-nat*)
have *coprime* $(a \hat{} ((n - 1)\ div\ p) - 1)\ n$
by (*metis cong-imp-coprime-nat eq1 p' th*) **}**
with *pocklington[OF n qrn[symmetric] nq2 an1]*
show *?thesis* **by** *blast*
qed

end

8 Gauss' Lemma

theory *Gauss*
imports *Residues*
begin

lemma *cong-prime-prod-zero-nat*:
fixes *a::nat*
shows $[\![[a * b = 0]\ (mod\ p);\ prime\ p]\!] \Longrightarrow [a = 0]\ (mod\ p) \mid [b = 0]\ (mod\ p)$
by (*auto simp add*: *cong-altdef-nat*)

lemma *cong-prime-prod-zero-int*:
fixes *a::int*
shows $[\![[a * b = 0]\ (mod\ p);\ prime\ p]\!] \Longrightarrow [a = 0]\ (mod\ p) \mid [b = 0]\ (mod\ p)$
by (*auto simp add*: *cong-altdef-int*)

locale *GAUSS* =
fixes *p :: nat*
fixes *a :: int*

assumes *p-prime*: *prime p*
assumes *p-ge-2*: $2 < p$
assumes *p-a-relprime*: $[a \neq 0](mod\ p)$

assumes *a-nonzero*: *0* < *a*
begin

definition *A* = {*0*::*int* <.. ((*int p* − *1*) *div 2*)}
definition *B* = (λ*x*. *x* ∗ *a*) ‘ *A*
definition *C* = (λ*x*. *x mod p*) ‘ *B*
definition *D* = *C* ∩ {.. (*int p* − *1*) *div 2*}
definition *E* = *C* ∩ {(*int p* − *1*) *div 2* <..}
definition *F* = (λ*x*. (*int p* − *x*)) ‘ *E*

8.1 Basic properties of p

lemma *odd-p*: *odd p*
by (*metis p-prime p-ge-2 prime-odd-nat*)

lemma *p-minus-one-l*: (*int p* − *1*) *div 2* < *p*
proof −
  have (*p* − *1*) *div 2* ≤ (*p* − *1*) *div 1*
    by (*metis div-by-1 div-le-dividend*)
  also have ... = *p* − *1* **by** *simp*
  finally show *?thesis* **using** *p-ge-2* **by** *arith*
qed

lemma *p-eq2*: *int p* = (*2* ∗ ((*int p* − *1*) *div 2*)) + *1*
  using *odd-p p-ge-2 div-mult-self1-is-id* [*of 2 p* − *1*]
  by *simp*

lemma *p-odd-int*: **obtains** *z*::*int* **where** *int p* = *2*∗*z*+*1 0*<*z*
  using *odd-p p-ge-2*
  by (*auto simp add*: *even-iff-mod-2-eq-zero*) (*metis p-eq2*)

8.2 Basic Properties of the Gauss Sets

lemma *finite-A*: *finite* (*A*)
by (*auto simp add*: *A-def*)

lemma *finite-B*: *finite* (*B*)
by (*auto simp add*: *B-def finite-A*)

lemma *finite-C*: *finite* (*C*)
by (*auto simp add*: *C-def finite-B*)

lemma *finite-D*: *finite* (*D*)
by (*auto simp add*: *D-def finite-C*)

lemma *finite-E*: *finite* (*E*)
by (*auto simp add*: *E-def finite-C*)

lemma *finite-F*: *finite* (*F*)
by (*auto simp add*: *F-def finite-E*)

```
lemma C-eq: C = D ∪ E
by (auto simp add: C-def D-def E-def)

lemma A-card-eq: card A = nat ((int p - 1) div 2)
  by (auto simp add: A-def)

lemma inj-on-xa-A: inj-on (λx. x * a) A
  using a-nonzero by (simp add: A-def inj-on-def)

definition ResSet :: int => int set => bool
  where ResSet m X = (∀ y1 y2. (y1 ∈ X & y2 ∈ X & [y1 = y2] (mod m) -->
y1 = y2))

lemma ResSet-image:
  ⟦ 0 < m; ResSet m A; ∀ x ∈ A. ∀ y ∈ A. ([f x = f y](mod m) --> x = y) ⟧ ⟹
    ResSet m (f ` A)
  by (auto simp add: ResSet-def)

lemma A-res: ResSet p A
  using p-ge-2
  by (auto simp add: A-def ResSet-def intro!: cong-less-imp-eq-int)

lemma B-res: ResSet p B
proof -
  {fix x fix y
    assume a: [x * a = y * a] (mod p)
    assume b: 0 < x
    assume c: x ≤ (int p - 1) div 2
    assume d: 0 < y
    assume e: y ≤ (int p - 1) div 2
    from a p-a-relprime p-prime a-nonzero cong-mult-rcancel-int [of - a x y]
    have [x = y](mod p)
     by (metis monoid-mult-class.mult.left-neutral cong-dvd-modulus-int cong-mult-rcancel-int

             cong-mult-self-int gcd.commute prime-imp-coprime-int)
    with cong-less-imp-eq-int [of x y p] p-minus-one-l
        order-le-less-trans [of x (int p - 1) div 2 p]
        order-le-less-trans [of y (int p - 1) div 2 p]
    have x = y
      by (metis b c cong-less-imp-eq-int d e zero-less-imp-eq-int of-nat-0-le-iff)
    } note xy = this
  show ?thesis
    apply (insert p-ge-2 p-a-relprime p-minus-one-l)
    apply (auto simp add: B-def)
    apply (rule ResSet-image)
    apply (auto simp add: A-res)
    apply (auto simp add: A-def xy)
    done
```

qed

lemma *SR-B-inj*: *inj-on* (λ*x*. *x mod p*) *B*
proof −
{ fix *x* **fix** *y*
assume *a*: *x* ∗ *a mod p* = *y* ∗ *a mod p*
assume *b*: *0* < *x*
assume *c*: *x* ≤ (*int p* − *1*) *div 2*
assume *d*: *0* < *y*
assume *e*: *y* ≤ (*int p* − *1*) *div 2*
assume *f*: *x* ≠ *y*
from *a* **have** [*x* ∗ *a* = *y* ∗ *a*](*mod p*)
by (*metis cong-int-def*)
with *p-a-relprime p-prime cong-mult-rcancel-int* [*of a p x y*]
have [*x* = *y*](*mod p*)
by (*metis cong-mult-self-int dvd-div-mult-self gcd.commute prime-imp-coprime-int*)
with *cong-less-imp-eq-int* [*of x y p*] *p-minus-one-l*
order-le-less-trans [*of x* (*int p* − *1*) *div 2 p*]
order-le-less-trans [*of y* (*int p* − *1*) *div 2 p*]
have *x* = *y*
by (*metis b c cong-less-imp-eq-int d e zero-less-imp-eq-int of-nat-0-le-iff*)
then have *False*
by (*simp add*: *f*)**}**
then show *?thesis*
by (*auto simp add*: *B-def inj-on-def A-def*) *metis*
qed

lemma *inj-on-pminusx-E*: *inj-on* (λ*x*. *p* − *x*) *E*
apply (*auto simp add*: *E-def C-def B-def A-def*)
apply (*rule-tac g* = (*op* − (*int p*)) **in** *inj-on-inverseI*)
apply *auto*
done

lemma *nonzero-mod-p*:
fixes *x*::*int* **shows** ⟦*0* < *x*; *x* < *int p*⟧ ⟹ [*x* ≠ *0*](*mod p*)
by (*simp add*: *cong-int-def*)

lemma *A-ncong-p*: *x* ∈ *A* ⟹ [*x* ≠ *0*](*mod p*)
by (*rule nonzero-mod-p*) (*auto simp add*: *A-def*)

lemma *A-greater-zero*: *x* ∈ *A* ⟹ *0* < *x*
by (*auto simp add*: *A-def*)

lemma *B-ncong-p*: *x* ∈ *B* ⟹ [*x* ≠ *0*](*mod p*)
by (*auto simp add*: *B-def*) (*metis cong-prime-prod-zero-int A-ncong-p p-a-relprime p-prime*)

lemma *B-greater-zero*: *x* ∈ *B* ⟹ *0* < *x*
using *a-nonzero* **by** (*auto simp add*: *B-def A-greater-zero*)

lemma *C-greater-zero*: $y \in C \Longrightarrow 0 < y$
proof (*auto simp add*: *C-def*)
  fix *x* :: *int*
  assume *a1*: $x \in B$
  have *f2*: $\bigwedge x_1.\ int\ x_1 = 0 \vee 0 < int\ x_1$ **by** *linarith*
  have $x\ mod\ int\ p \neq 0$ **using** *a1 B-ncong-p cong-int-def* **by** *simp*
  thus $0 < x\ mod\ int\ p$ **using** *a1 f2*
   by (*metis* (*no-types*) *B-greater-zero Divides.transfer-int-nat-functions*(*2*) *zero-less-imp-eq-int*)
qed

lemma *F-subset*: $F \subseteq \{x.\ 0 < x\ \&\ x \leq ((int\ p - 1)\ div\ 2)\}$
  apply (*auto simp add*: *F-def E-def C-def*)
  apply (*metis p-ge-2 Divides.pos-mod-bound less-diff-eq nat-int plus-int-code*(*2*) *zless-nat-conj*)
  apply (*auto intro*: *p-odd-int*)
  done

lemma *D-subset*: $D \subseteq \{x.\ 0 < x\ \&\ x \leq ((p - 1)\ div\ 2)\}$
  by (*auto simp add*: *D-def C-greater-zero*)

lemma *F-eq*: $F = \{x.\ \exists\, y \in A.\ (\ x = p - ((y*a)\ mod\ p)\ \&\ (int\ p - 1)\ div\ 2 < (y*a)\ mod\ p)\}$
  by (*auto simp add*: *F-def E-def D-def C-def B-def A-def*)

lemma *D-eq*: $D = \{x.\ \exists\, y \in A.\ (\ x = (y*a)\ mod\ p\ \&\ (y*a)\ mod\ p \leq (int\ p - 1)\ div\ 2)\}$
  by (*auto simp add*: *D-def C-def B-def A-def*)

lemma *all-A-relprime*: **assumes** $x \in A$ **shows** $gcd\ x\ p = 1$
  using *p-prime A-ncong-p* [*OF assms*]
  by (*simp add*: *cong-altdef-int*) (*metis gcd.commute prime-imp-coprime-int*)

lemma *A-prod-relprime*: $gcd\ (setprod\ id\ A)\ p = 1$
  by (*metis id-def all-A-relprime setprod-coprime*)

8.3 Relationships Between Gauss Sets

lemma *StandardRes-inj-on-ResSet*: $ResSet\ m\ X \Longrightarrow (inj\text{-}on\ (\lambda b.\ b\ mod\ m)\ X)$
  by (*auto simp add*: *ResSet-def inj-on-def cong-int-def*)

lemma *B-card-eq-A*: $card\ B = card\ A$
  using *finite-A* **by** (*simp add*: *finite-A B-def inj-on-xa-A card-image*)

lemma *B-card-eq*: $card\ B = nat\ ((int\ p - 1)\ div\ 2)$
  by (*simp add*: *B-card-eq-A A-card-eq*)

lemma *F-card-eq-E*: $card\ F = card\ E$
  using *finite-E*

by (*simp add*: *F-def inj-on-pminusx-E card-image*)

lemma *C-card-eq-B*: *card C* = *card B*
proof −
have *inj-on* (λ*x*. *x mod p*) *B*
by (*metis SR-B-inj*)
then show *?thesis*
by (*metis C-def card-image*)
qed

lemma *D-E-disj*: *D* ∩ *E* = {}
by (*auto simp add*: *D-def E-def*)

lemma *C-card-eq-D-plus-E*: *card C* = *card D* + *card E*
by (*auto simp add*: *C-eq card-Un-disjoint D-E-disj finite-D finite-E*)

lemma *C-prod-eq-D-times-E*: *setprod id E* ∗ *setprod id D* = *setprod id C*
by (*metis C-eq D-E-disj finite-D finite-E inf-commute setprod.union-disjoint sup-commute*)

lemma *C-B-zcong-prod*: [*setprod id C* = *setprod id B*] (*mod p*)
apply (*auto simp add*: *C-def*)
apply (*insert finite-B SR-B-inj*)
apply (*drule setprod.reindex* [*of* λ*x*. *x mod int p B id*])
apply *auto*
apply (*rule cong-setprod-int*)
apply (*auto simp add*: *cong-int-def*)
done

lemma *F-Un-D-subset*: (*F* ∪ *D*) ⊆ *A*
apply (*intro Un-least subset-trans* [*OF F-subset*] *subset-trans* [*OF D-subset*])
apply (*auto simp add*: *A-def*)
done

lemma *F-D-disj*: (*F* ∩ *D*) = {}
proof (*auto simp add*: *F-eq D-eq*)
fix *y*::*int* **and** *z*::*int*
assume *p* − (*y*∗*a*) *mod p* = (*z*∗*a*) *mod p*
then have [(*y*∗*a*) *mod p* + (*z*∗*a*) *mod p* = *0*] (*mod p*)
by (*metis add.commute diff-eq-eq dvd-refl cong-int-def dvd-eq-mod-eq-0 mod-0*)
moreover have [*y* ∗ *a* = (*y*∗*a*) *mod p*] (*mod p*)
by (*metis cong-int-def mod-mod-trivial*)
ultimately have [*a* ∗ (*y* + *z*) = *0*] (*mod p*)
by (*metis cong-int-def mod-add-left-eq mod-add-right-eq mult.commute ring-class.ring-distribs*(*1*))
with *p-prime a-nonzero p-a-relprime*
have *a*: [*y* + *z* = *0*] (*mod p*)
by (*metis cong-prime-prod-zero-int*)
assume *b*: *y* ∈ *A* **and** *c*: *z* ∈ *A*
with *A-def* **have** *0* < *y* + *z*

by *auto*
moreover from *b c p-eq2 A-def* **have** *y* + *z* < *p*
by *auto*
ultimately show *False*
by (*metis a nonzero-mod-p*)
qed

lemma *F-Un-D-card*: *card* (*F* ∪ *D*) = *nat* ((*p* − *1*) *div 2*)
proof −
have *card* (*F* ∪ *D*) = *card E* + *card D*
by (*auto simp add*: *finite-F finite-D F-D-disj card-Un-disjoint F-card-eq-E*)
then have *card* (*F* ∪ *D*) = *card C*
by (*simp add*: *C-card-eq-D-plus-E*)
then show *card* (*F* ∪ *D*) = *nat* ((*p* − *1*) *div 2*)
by (*simp add*: *C-card-eq-B B-card-eq*)
qed

lemma *F-Un-D-eq-A*: *F* ∪ *D* = *A*
using *finite-A F-Un-D-subset A-card-eq F-Un-D-card*
by (*auto simp add*: *card-seteq*)

lemma *prod-D-F-eq-prod-A*: (*setprod id D*) ∗ (*setprod id F*) = *setprod id A*
by (*metis F-D-disj F-Un-D-eq-A Int-commute Un-commute finite-D finite-F setprod.union-disjoint*)

lemma *prod-F-zcong*: [*setprod id F* = ((−*1*) ˆ (*card E*)) ∗ (*setprod id E*)] (*mod p*)
proof −
have *FE*: *setprod id F* = *setprod* (*op* − *p*) *E*
apply (*auto simp add*: *F-def*)
apply (*insert finite-E inj-on-pminusx-E*)
apply (*drule setprod.reindex*, *auto*)
done
then have ∀ *x* ∈ *E*. [(*p*−*x*) *mod p* = − *x*](*mod p*)
by (*metis cong-int-def minus-mod-self1 mod-mod-trivial*)
then have [*setprod* ((λ*x*. *x mod p*) *o* (*op* − *p*)) *E* = *setprod* (*uminus*) *E*](*mod p*)
using *finite-E p-ge-2*
cong-setprod-int [*of E* (λ*x*. *x mod p*) *o* (*op* − *p*) *uminus p*]
by *auto*
then have *two*: [*setprod id F* = *setprod* (*uminus*) *E*](*mod p*)
by (*metis FE cong-cong-mod-int cong-refl-int cong-setprod-int minus-mod-self1*)
have *setprod uminus E* = (−*1*) ˆ (*card E*) ∗ (*setprod id E*)
using *finite-E* **by** (*induct set*: *finite*) *auto*
with *two* **show** *?thesis*
by *simp*
qed

8.4 Gauss' Lemma

lemma *aux*: *setprod id A* ∗ (− *1*) ˆ *card E* ∗ *a* ˆ *card A* ∗ (− *1*) ˆ *card E* = *setprod id A* ∗ *a* ˆ *card A*
by (*metis* (*no-types*) *minus-minus mult.commute mult.left-commute power-minus power-one*)

theorem *pre-gauss-lemma*:
  [*a* ˆ *nat*((*int p* − *1*) *div 2*) = (−*1*) ˆ (*card E*)] (*mod p*)
proof −
  have [*setprod id A* = *setprod id F* ∗ *setprod id D*](*mod p*)
    by (*auto simp add*: *prod-D-F-eq-prod-A mult.commute cong del*:*setprod.cong*)
  then have [*setprod id A* = ((−*1*) ˆ(*card E*) ∗ *setprod id E*) ∗ *setprod id D*] (*mod p*)
    apply (*rule cong-trans-int*)
    apply (*metis cong-scalar-int prod-F-zcong*)
    done
  then have [*setprod id A* = ((−*1*) ˆ(*card E*) ∗ *setprod id C*)] (*mod p*)
    by (*metis C-prod-eq-D-times-E mult.commute mult.left-commute*)
  then have [*setprod id A* = ((−*1*) ˆ(*card E*) ∗ *setprod id B*)] (*mod p*)
    by (*rule cong-trans-int*) (*metis C-B-zcong-prod cong-scalar2-int*)
  then have [*setprod id A* = ((−*1*) ˆ(*card E*) ∗
    (*setprod id* ((λ*x*. *x* ∗ *a*) ‘ *A*)))] (*mod p*)
    by (*simp add*: *B-def*)
  then have [*setprod id A* = ((−*1*) ˆ(*card E*) ∗ (*setprod* (λ*x*. *x* ∗ *a*) *A*))]
    (*mod p*)
    by (*simp add*: *inj-on-xa-A setprod.reindex*)
  moreover have *setprod* (λ*x*. *x* ∗ *a*) *A* =
    setprod (λ*x*. *a*) *A* ∗ *setprod id A*
    using *finite-A* **by** (*induct set*: *finite*) *auto*
  ultimately have [*setprod id A* = ((−*1*) ˆ(*card E*) ∗ (*setprod* (λ*x*. *a*) *A* ∗
    setprod id A))] (*mod p*)
    by *simp*
  then have [*setprod id A* = ((−*1*) ˆ(*card E*) ∗ *a*ˆ(*card A*) ∗
      setprod id A)](*mod p*)
    apply (*rule cong-trans-int*)
      apply (*simp add*: *cong-scalar2-int cong-scalar-int finite-A setprod-constant mult.assoc*)
    done
  then have *a*: [*setprod id A* ∗ (−*1*) ˆ(*card E*) =
      ((−*1*) ˆ(*card E*) ∗ *a*ˆ(*card A*) ∗ *setprod id A* ∗ (−*1*) ˆ(*card E*))](*mod p*)
    by (*rule cong-scalar-int*)
  then have [*setprod id A* ∗ (−*1*) ˆ(*card E*) = *setprod id A* ∗
      (−*1*) ˆ(*card E*) ∗ *a*ˆ(*card A*) ∗ (−*1*) ˆ(*card E*)](*mod p*)
    apply (*rule cong-trans-int*)
    apply (*simp add*: *a mult.commute mult.left-commute*)
    done
  then have [*setprod id A* ∗ (−*1*) ˆ(*card E*) = *setprod id A* ∗ *a*ˆ(*card A*)](*mod p*)
    apply (*rule cong-trans-int*)
    apply (*simp add*: *aux cong del*:*setprod.cong*)

```
    done
  with A-prod-relprime have [(− 1) ^ card E = a ^ card A](mod p)
    by (metis cong-mult-lcancel-int)
  then show ?thesis
    by (simp add: A-card-eq cong-sym-int)
qed
```

```
end

end
```

9 The fibonacci function

```
theory Fib
imports Main GCD Binomial
begin
```

9.1 Fibonacci numbers

```
fun fib :: nat ⇒ nat
where
  fib0: fib 0 = 0
| fib1: fib (Suc 0) = 1
| fib2: fib (Suc (Suc n)) = fib (Suc n) + fib n
```

9.2 Basic Properties

```
lemma fib-1 [simp]: fib (1::nat) = 1
  by (metis One-nat-def fib1)

lemma fib-2 [simp]: fib (2::nat) = 1
  using fib.simps(3) [of 0]
  by (simp add: numeral-2-eq-2)

lemma fib-plus-2: fib (n + 2) = fib (n + 1) + fib n
  by (metis Suc-eq-plus1 add-2-eq-Suc' fib.simps(3))

lemma fib-add: fib (Suc (n+k)) = fib (Suc k) ∗ fib (Suc n) + fib k ∗ fib n
  by (induct n rule: fib.induct) (auto simp add: field-simps)

lemma fib-neq-0-nat: n > 0 =⇒ fib n > 0
  by (induct n rule: fib.induct) (auto simp add: )
```

9.3 A Few Elementary Results

Concrete Mathematics, page 278: Cassini's identity. The proof is much easier using integers, not natural numbers!

lemma *fib-Cassini-int*: *int* (*fib* (*Suc* (*Suc* *n*)) ∗ *fib* *n*) − *int*((*fib* (*Suc* *n*))^2) = −((−*1*) ˆ*n*)
by (*induct* *n* *rule*: *fib.induct*) (*auto* *simp* *add*: *field-simps* *power2-eq-square* *power-add*)

lemma *fib-Cassini-nat*:
fib (*Suc* (*Suc* *n*)) ∗ *fib* *n* =
(*if* *even* *n* *then* (*fib* (*Suc* *n*))^2 − *1* *else* (*fib* (*Suc* *n*))^2 + *1*)
using *fib-Cassini-int* [*of* *n*] **by** (*auto* *simp* *del*: *of-nat-mult* *of-nat-power*)

9.4 Law 6.111 of Concrete Mathematics

lemma *coprime-fib-Suc-nat*: *coprime* (*fib* (*n*::*nat*)) (*fib* (*Suc* *n*))
apply (*induct* *n* *rule*: *fib.induct*)
apply *auto*
apply (*metis* *gcd-add1* *add.commute*)
done

lemma *gcd-fib-add*: *gcd* (*fib* *m*) (*fib* (*n* + *m*)) = *gcd* (*fib* *m*) (*fib* *n*)
apply (*simp* *add*: *gcd.commute* [*of* *fib* *m*])
apply (*cases* *m*)
apply (*auto* *simp* *add*: *fib-add*)
apply (*metis* *gcd.commute* *mult.commute* *coprime-fib-Suc-nat*
gcd-add-mult *gcd-mult-cancel* *gcd.commute*)
done

lemma *gcd-fib-diff*: *m* ≤ *n* =⇒ *gcd* (*fib* *m*) (*fib* (*n* − *m*)) = *gcd* (*fib* *m*) (*fib* *n*)
by (*simp* *add*: *gcd-fib-add* [*symmetric*, *of* - *n*−*m*])

lemma *gcd-fib-mod*: *0* < *m* =⇒ *gcd* (*fib* *m*) (*fib* (*n* *mod* *m*)) = *gcd* (*fib* *m*) (*fib* *n*)
proof (*induct* *n* *rule*: *less-induct*)
case (*less* *n*)
show *gcd* (*fib* *m*) (*fib* (*n* *mod* *m*)) = *gcd* (*fib* *m*) (*fib* *n*)
proof (*cases* *m* < *n*)
case *True*
then have *m* ≤ *n* **by** *auto*
with ‹*0* < *m*› **have** *pos-n*: *0* < *n* **by** *auto*
with ‹*0* < *m*› ‹*m* < *n*› **have** *diff*: *n* − *m* < *n* **by** *auto*
have *gcd* (*fib* *m*) (*fib* (*n* *mod* *m*)) = *gcd* (*fib* *m*) (*fib* ((*n* − *m*) *mod* *m*))
by (*simp* *add*: *mod-if* [*of* *n*]) (*insert* ‹*m* < *n*›, *auto*)
also have . . . = *gcd* (*fib* *m*) (*fib* (*n* − *m*))
by (*simp* *add*: *less.hyps* *diff* ‹*0* < *m*›)
also have . . . = *gcd* (*fib* *m*) (*fib* *n*)
by (*simp* *add*: *gcd-fib-diff* ‹*m* ≤ *n*›)
finally show *gcd* (*fib* *m*) (*fib* (*n* *mod* *m*)) = *gcd* (*fib* *m*) (*fib* *n*) **.**
next
case *False*
then show *gcd* (*fib* *m*) (*fib* (*n* *mod* *m*)) = *gcd* (*fib* *m*) (*fib* *n*)
by (*cases* *m* = *n*) *auto*

qed
qed

lemma *fib-gcd*: *fib* (*gcd m n*) = *gcd* (*fib m*) (*fib n*)
— Law 6.111
by (*induct m n rule*: *gcd-nat-induct*) (*simp-all add*: *gcd-non-0-nat gcd.commute gcd-fib-mod*)

theorem *fib-mult-eq-setsum-nat*: *fib* (*Suc n*) ∗ *fib n* = (∑ *k* ∈ {..*n*}. *fib k* ∗ *fib k*)
by (*induct n rule*: *nat.induct*) (*auto simp add*: *field-simps*)

9.5 Fibonacci and Binomial Coefficients

lemma *setsum-drop-zero*: (∑ *k* = *0*..*Suc n*. *if 0<k then* (*f* (*k* − *1*)) *else 0*) = (∑ *j* = *0*..*n*. *f j*)
by (*induct n*) *auto*

lemma *setsum-choose-drop-zero*:
(∑ *k* = *0*..*Suc n*. *if k=0 then 0 else* (*Suc n* − *k*) *choose* (*k* − *1*)) = (∑ *j* = *0*..*n*. (*n*−*j*) *choose j*)
by (*rule trans* [*OF setsum.cong setsum-drop-zero*]) *auto*

lemma *ne-diagonal-fib*: (∑ *k* = *0*..*n*. (*n*−*k*) *choose k*) = *fib* (*Suc n*)
proof (*induct n rule*: *fib.induct*)
case *1*
show *?case* **by** *simp*
next
case *2*
show *?case* **by** *simp*
next
case (*3 n*)
have (∑ *k* = *0*..*Suc n*. *Suc* (*Suc n*) − *k choose k*) =
(∑ *k* = *0*..*Suc n*. (*Suc n* − *k choose k*) + (*if k=0 then 0 else* (*Suc n* − *k choose* (*k* − *1*))))
by (*rule setsum.cong*) (*simp-all add*: *choose-reduce-nat*)
also have ... = (∑ *k* = *0*..*Suc n*. *Suc n* − *k choose k*) +
(∑ *k* = *0*..*Suc n*. *if k=0 then 0 else* (*Suc n* − *k choose* (*k* − *1*)))
by (*simp add*: *setsum.distrib*)
also have ... = (∑ *k* = *0*..*Suc n*. *Suc n* − *k choose k*) +
(∑ *j* = *0*..*n*. *n* − *j choose j*)
by (*metis setsum-choose-drop-zero*)
finally show *?case* **using** *3*
by *simp*
qed

end

10 The sieve of Eratosthenes

theory *Eratosthenes*
imports *Main Primes*
begin

10.1 Preliminary: strict divisibility

context *dvd*
begin

abbreviation *dvd-strict* :: *'a* ⇒ *'a* ⇒ *bool* (**infixl** *dvd'-strict 50*)
where
b dvd-strict a ≡ *b dvd a* ∧ ¬ *a dvd b*

end

10.2 Main corpus

The sieve is modelled as a list of booleans, where *False* means *marked out*.

type-synonym *marks* = *bool list*

definition *numbers-of-marks* :: *nat* ⇒ *marks* ⇒ *nat set*
where
numbers-of-marks n bs = *fst* ‘ {*x* ∈ *set* (*enumerate n bs*). *snd x*}

lemma *numbers-of-marks-simps* [*simp*, *code*]:
numbers-of-marks n [] = {}
numbers-of-marks n (*True* # *bs*) = *insert n* (*numbers-of-marks* (*Suc n*) *bs*)
numbers-of-marks n (*False* # *bs*) = *numbers-of-marks* (*Suc n*) *bs*
by (*auto simp add*: *numbers-of-marks-def intro*!: *image-eqI*)

lemma *numbers-of-marks-Suc*:
numbers-of-marks (*Suc n*) *bs* = *Suc* ‘ *numbers-of-marks n bs*
by (*auto simp add*: *numbers-of-marks-def enumerate-Suc-eq image-iff Bex-def*)

lemma *numbers-of-marks-replicate-False* [*simp*]:
numbers-of-marks n (*replicate m False*) = {}
by (*auto simp add*: *numbers-of-marks-def enumerate-replicate-eq*)

lemma *numbers-of-marks-replicate-True* [*simp*]:
numbers-of-marks n (*replicate m True*) = {*n*..<*n*+*m*}
by (*auto simp add*: *numbers-of-marks-def enumerate-replicate-eq image-def*)

lemma *in-numbers-of-marks-eq*:
m ∈ *numbers-of-marks n bs* ⟷ *m* ∈ {*n*..<*n* + *length bs*} ∧ *bs* ! (*m* − *n*)
by (*simp add*: *numbers-of-marks-def in-set-enumerate-eq image-iff add.commute*)

lemma *sorted-list-of-set-numbers-of-marks*:

sorted-list-of-set (*numbers-of-marks n bs*) = *map fst* (*filter snd* (*enumerate n bs*))
by (*auto simp add*: *numbers-of-marks-def distinct-map*
intro!: *sorted-filter distinct-filter inj-onI sorted-distinct-set-unique*)

Marking out multiples in a sieve

definition *mark-out* :: *nat* ⇒ *marks* ⇒ *marks*
where
mark-out n bs = *map* (λ(*q*, *b*). *b* ∧ ¬ *Suc n dvd Suc* (*Suc q*)) (*enumerate n bs*)

lemma *mark-out-Nil* [*simp*]: *mark-out n* [] = []
by (*simp add*: *mark-out-def*)

lemma *length-mark-out* [*simp*]: *length* (*mark-out n bs*) = *length bs*
by (*simp add*: *mark-out-def*)

lemma *numbers-of-marks-mark-out*:
numbers-of-marks n (*mark-out m bs*) = {*q* ∈ *numbers-of-marks n bs*. ¬ *Suc m dvd Suc q* − *n*}
by (*auto simp add*: *numbers-of-marks-def mark-out-def in-set-enumerate-eq image-iff*
nth-enumerate-eq less-eq-dvd-minus)

Auxiliary operation for efficient implementation

definition *mark-out-aux* :: *nat* ⇒ *nat* ⇒ *marks* ⇒ *marks*
where
mark-out-aux n m bs =
map (λ(*q*, *b*). *b* ∧ (*q* < *m* + *n* ∨ ¬ *Suc n dvd Suc* (*Suc q*) + (*n* − *m mod Suc n*))) (*enumerate n bs*)

lemma *mark-out-code* [*code*]: *mark-out n bs* = *mark-out-aux n n bs*
proof −
have *aux*: *False*
if *A*: *Suc n dvd Suc* (*Suc a*)
and *B*: *a* < *n* + *n*
and *C*: *n* ≤ *a*
for *a*
proof (*cases n* = *0*)
case *True*
with *A B C* **show** *?thesis* **by** *simp*
next
case *False*
def *m* ≡ *Suc n*
then have *m* > *0* **by** *simp*
from *False* **have** *n* > *0* **by** *simp*
from *A* **obtain** *q* **where** *q*: *Suc* (*Suc a*) = *Suc n* ∗ *q* **by** (*rule dvdE*)
have *q* > *0*
proof (*rule ccontr*)
assume ¬ *q* > *0*
with *q* **show** *False* **by** *simp*
qed

with ‹$n > 0$› **have** *Suc n* $*$ $q \geq 2$ **by** (*auto simp add*: *gr0-conv-Suc*)
with *q* **have** *a*: $a = Suc\ n * q - 2$ **by** *simp*
with *B* **have** $q + n * q < n + n + 2$ **by** *auto*
then have $m * q < m * 2$ **by** (*simp add*: *m-def*)
with ‹$m > 0$› **have** $q < 2$ **by** *simp*
with ‹$q > 0$› **have** $q = 1$ **by** *simp*
with *a* **have** $a = n - 1$ **by** *simp*
with ‹$n > 0$› *C* **show** *False* **by** *simp*
qed
show *?thesis*
by (*auto simp add*: *mark-out-def mark-out-aux-def in-set-enumerate-eq intro*: *aux*)
qed

lemma *mark-out-aux-simps* [*simp*, *code*]:
mark-out-aux n m [] = []
mark-out-aux n 0 (*b* # *bs*) = *False* # *mark-out-aux n n bs*
mark-out-aux n (*Suc m*) (*b* # *bs*) = *b* # *mark-out-aux n m bs*
proof *goal-cases*
case *1*
show *?case*
by (*simp add*: *mark-out-aux-def*)
next
case *2*
show *?case*
by (*auto simp add*: *mark-out-code* [*symmetric*] *mark-out-aux-def mark-out-def enumerate-Suc-eq in-set-enumerate-eq less-eq-dvd-minus*)
next
case *3*
{ def $v \equiv Suc\ m$ **and** $w \equiv Suc\ n$
fix *q*
assume $m + n \leq q$
then obtain *r* **where** *q*: $q = m + n + r$ **by** (*auto simp add*: *le-iff-add*)
{ fix *u*
from *w-def* **have** $u\ mod\ w < w$ **by** *simp*
then have $u + (w - u\ mod\ w) = w + (u - u\ mod\ w)$
by *simp*
then have $u + (w - u\ mod\ w) = w + u\ div\ w * w$
by (*simp add*: *div-mod-equality'* [*symmetric*])
}
then have $w\ dvd\ v + w + r + (w - v\ mod\ w) \longleftrightarrow w\ dvd\ m + w + r + (w - m\ mod\ w)$
by (*simp add*: *add.assoc add.left-commute* [*of m*] *add.left-commute* [*of v*] *dvd-add-left-iff dvd-add-right-iff*)
moreover from *q* **have** $Suc\ q = m + w + r$ **by** (*simp add*: *w-def*)
moreover from *q* **have** $Suc\ (Suc\ q) = v + w + r$ **by** (*simp add*: *v-def w-def*)
ultimately have $w\ dvd\ Suc\ (Suc\ (q + (w - v\ mod\ w))) \longleftrightarrow w\ dvd\ Suc\ (q + (w - m\ mod\ w))$
by (*simp only*: *add-Suc* [*symmetric*])

```
    then have Suc n dvd Suc (Suc (Suc (q + n) − Suc m mod Suc n)) ⟷
      Suc n dvd Suc (Suc (q + n − m mod Suc n))
      by (simp add: v-def w-def Suc-diff-le trans-le-add2)
  }
  then show ?case
    by (auto simp add: mark-out-aux-def
      enumerate-Suc-eq in-set-enumerate-eq not-less)
qed
```

Main entry point to sieve

```
fun sieve :: nat ⇒ marks ⇒ marks
where
  sieve n [] = []
| sieve n (False # bs) = False # sieve (Suc n) bs
| sieve n (True # bs) = True # sieve (Suc n) (mark-out n bs)
```

There are the following possible optimisations here:

- *sieve* can abort as soon as *n* is too big to let *mark-out* have any effect.
- Search for further primes can be given up as soon as the search position exceeds the square root of the maximum candidate.

This is left as an constructive exercise to the reader.

```
lemma numbers-of-marks-sieve:
  numbers-of-marks (Suc n) (sieve n bs) =
    {q ∈ numbers-of-marks (Suc n) bs. ∀ m ∈ numbers-of-marks (Suc n) bs. ¬ m
dvd-strict q}
proof (induct n bs rule: sieve.induct)
  case 1
  show ?case by simp
next
  case 2
  then show ?case by simp
next
  case (3 n bs)
  have aux: n ∈ Suc ‘ M ⟷ n > 0 ∧ n − 1 ∈ M (is ?lhs ⟷ ?rhs) for M n
  proof
    show ?rhs if ?lhs using that by auto
    show ?lhs if ?rhs
    proof −
      from that have n > 0 and n − 1 ∈ M by auto
      then have Suc (n − 1) ∈ Suc ‘ M by blast
      with ‹n > 0› show n ∈ Suc ‘ M by simp
    qed
  qed
  have aux1: False if Suc (Suc n) ≤ m and m dvd Suc n for m :: nat
  proof −
```

```
    from ‹m dvd Suc n› obtain q where Suc n = m * q ..
    with ‹Suc (Suc n) ≤ m› have Suc (m * q) ≤ m by simp
    then have m * q < m by arith
    then have q = 0 by simp
    with ‹Suc n = m * q› show ?thesis by simp
  qed
  have aux2: m dvd q
    if 1: ∀ q>0. 1 < q ⟶ Suc n < q ⟶ q ≤ Suc (n + length bs) ⟶
      bs ! (q − Suc (Suc n)) ⟶ ¬ Suc n dvd q ⟶ q dvd m ⟶ m dvd q
    and 2: ¬ Suc n dvd m q dvd m
    and 3: Suc n < q q ≤ Suc (n + length bs) bs ! (q − Suc (Suc n))
    for m q :: nat
  proof −
    from 1 have *: ⋀q. Suc n < q ⟹ q ≤ Suc (n + length bs) ⟹
      bs ! (q − Suc (Suc n)) ⟹ ¬ Suc n dvd q ⟹ q dvd m ⟹ m dvd q
      by auto
    from 2 have ¬ Suc n dvd q by (auto elim: dvdE)
    moreover note 3
    moreover note ‹q dvd m›
    ultimately show ?thesis by (auto intro: *)
  qed
  from 3 show ?case
   apply (simp-all add: numbers-of-marks-mark-out numbers-of-marks-Suc Compr-image-eq
      inj-image-eq-iff in-numbers-of-marks-eq Ball-def imp-conjL aux)
    apply safe
    apply (simp-all add: less-diff-conv2 le-diff-conv2 dvd-minus-self not-less)
    apply (clarsimp dest!: aux1)
    apply (simp add: Suc-le-eq less-Suc-eq-le)
    apply (rule aux2)
    apply (clarsimp dest!: aux1)+
    done
qed
```

Relation of the sieve algorithm to actual primes

```
definition primes-upto :: nat ⇒ nat list
where
  primes-upto n = sorted-list-of-set {m. m ≤ n ∧ prime m}

lemma set-primes-upto: set (primes-upto n) = {m. m ≤ n ∧ prime m}
  by (simp add: primes-upto-def)

lemma sorted-primes-upto [iff]: sorted (primes-upto n)
  by (simp add: primes-upto-def)

lemma distinct-primes-upto [iff]: distinct (primes-upto n)
  by (simp add: primes-upto-def)

lemma set-primes-upto-sieve:
  set (primes-upto n) = numbers-of-marks 2 (sieve 1 (replicate (n − 1) True))
```

```
proof -
  consider n = 0 ∨ n = 1 | n > 1 by arith
  then show ?thesis
  proof cases
    case 1
    then show ?thesis
      by (auto simp add: numbers-of-marks-sieve numeral-2-eq-2 set-primes-upto
        dest: prime-gt-Suc-0-nat)
  next
    case 2
    {
      fix m q
      assume Suc (Suc 0) ≤ q
        and q < Suc n
        and m dvd q
      then have m < Suc n by (auto dest: dvd-imp-le)
      assume *: ∀ m∈{Suc (Suc 0)..<Suc n}. m dvd q ⟶ q dvd m
        and m dvd q and m ≠ 1
      have m = q
      proof (cases m = 0)
        case True with ‹m dvd q› show ?thesis by simp
      next
        case False with ‹m ≠ 1› have Suc (Suc 0) ≤ m by arith
        with ‹m < Suc n› * ‹m dvd q› have q dvd m by simp
        with ‹m dvd q› show ?thesis by (simp add: dvd-antisym)
      qed
    }
    then have aux: ⋀m q. Suc (Suc 0) ≤ q ⟹
      q < Suc n ⟹
      m dvd q ⟹
      ∀ m∈{Suc (Suc 0)..<Suc n}. m dvd q ⟶ q dvd m ⟹
      m dvd q ⟹ m ≠ q ⟹ m = 1 by auto
    from 2 show ?thesis
     apply (auto simp add: numbers-of-marks-sieve numeral-2-eq-2 set-primes-upto
        dest: prime-gt-Suc-0-nat)
      apply (metis One-nat-def Suc-le-eq less-not-refl prime-def)
      apply (metis One-nat-def Suc-le-eq aux prime-def)
      done
  qed
qed

lemma primes-upto-sieve [code]:
  primes-upto n = map fst (filter snd (enumerate 2 (sieve 1 (replicate (n - 1)
True))))
proof -
  have primes-upto n = sorted-list-of-set (numbers-of-marks 2 (sieve 1 (replicate
(n - 1) True)))
    apply (rule sorted-distinct-set-unique)
    apply (simp-all only: set-primes-upto-sieve numbers-of-marks-def)
```

apply *auto*
done
then show *?thesis*
by (*simp add*: *sorted-list-of-set-numbers-of-marks*)
qed

lemma *prime-in-primes-upto*: *prime n* ⟷ *n* ∈ *set* (*primes-upto n*)
by (*simp add*: *set-primes-upto*)

10.3 Application: smallest prime beyond a certain number

definition *smallest-prime-beyond* :: *nat* ⇒ *nat*
where
smallest-prime-beyond n = (*LEAST p*. *prime p* ∧ *p* ≥ *n*)

lemma *prime-smallest-prime-beyond* [*iff*]: *prime* (*smallest-prime-beyond n*) (**is** *?P*)
and *smallest-prime-beyond-le* [*iff*]: *smallest-prime-beyond n* ≥ *n* (**is** *?Q*)
proof −
let *?least* = *LEAST p*. *prime p* ∧ *p* ≥ *n*
from *primes-infinite* **obtain** *q* **where** *prime q* ∧ *q* ≥ *n*
by (*metis finite-nat-set-iff-bounded-le mem-Collect-eq nat-le-linear*)
then have *prime ?least* ∧ *?least* ≥ *n*
by (*rule LeastI*)
then show *?P* **and** *?Q*
by (*simp-all add*: *smallest-prime-beyond-def*)
qed

lemma *smallest-prime-beyond-smallest*: *prime p* ⟹ *p* ≥ *n* ⟹ *smallest-prime-beyond n* ≤ *p*
by (*simp only*: *smallest-prime-beyond-def*) (*auto intro*: *Least-le*)

lemma *smallest-prime-beyond-eq*:
prime p ⟹ *p* ≥ *n* ⟹ (⋀*q*. *prime q* ⟹ *q* ≥ *n* ⟹ *q* ≥ *p*) ⟹ *smallest-prime-beyond n* = *p*
by (*simp only*: *smallest-prime-beyond-def*) (*auto intro*: *Least-equality*)

definition *smallest-prime-between* :: *nat* ⇒ *nat* ⇒ *nat option*
where
smallest-prime-between m n =
(*if* (∃ *p*. *prime p* ∧ *m* ≤ *p* ∧ *p* ≤ *n*) *then Some* (*smallest-prime-beyond m*) *else None*)

lemma *smallest-prime-between-None*:
smallest-prime-between m n = *None* ⟷ (∀ *q*. *m* ≤ *q* ∧ *q* ≤ *n* ⟶ ¬ *prime q*)
by (*auto simp add*: *smallest-prime-between-def*)

lemma *smallest-prime-betwen-Some*:
smallest-prime-between m n = *Some p* ⟷ *smallest-prime-beyond m* = *p* ∧ *p* ≤

n
 by (*auto simp add*: *smallest-prime-between-def dest*: *smallest-prime-beyond-smallest* [*of* - *m*])

lemma [*code*]: *smallest-prime-between m n* = *List.find* (*λp. p* ≥ *m*) (*primes-upto n*)
proof −
  have *List.find* (*λp. p* ≥ *m*) (*primes-upto n*) = *Some* (*smallest-prime-beyond m*)
    if *assms*: *m* ≤ *p prime p p* ≤ *n* **for** *p*
  proof −
    def *A* ≡ {*p. p* ≤ *n* ∧ *prime p* ∧ *m* ≤ *p*}
    from *assms* **have** *smallest-prime-beyond m* ≤ *p*
      by (*auto intro*: *smallest-prime-beyond-smallest*)
    from *this* ‹*p* ≤ *n*› **have** ∗: *smallest-prime-beyond m* ≤ *n*
      by (*rule order-trans*)
    from *assms* **have** *ex*: ∃ *p*≤*n. prime p* ∧ *m* ≤ *p*
      by *auto*
    then have *finite A*
      by (*auto simp add*: *A-def*)
    with ∗ **have** *Min A* = *smallest-prime-beyond m*
      by (*auto simp add*: *A-def intro*: *Min-eqI smallest-prime-beyond-smallest*)
    with *ex sorted-primes-upto* **show** *?thesis*
      by (*auto simp add*: *set-primes-upto sorted-find-Min A-def*)
  qed
  then show *?thesis*
    by (*auto simp add*: *smallest-prime-between-def find-None-iff set-primes-upto*
      intro!: *sym* [*of* - *None*])
qed

definition *smallest-prime-beyond-aux* :: *nat* ⇒ *nat* ⇒ *nat*
where
  smallest-prime-beyond-aux k n = *smallest-prime-beyond n*

lemma [*code*]:
  smallest-prime-beyond-aux k n =
    (*case smallest-prime-between n* (*k* ∗ *n*) *of*
      Some p ⇒ *p*
    | *None* ⇒ *smallest-prime-beyond-aux* (*Suc k*) *n*)
  by (*simp add*: *smallest-prime-beyond-aux-def smallest-prime-betwen-Some split*: *option.split*)

lemma [*code*]: *smallest-prime-beyond n* = *smallest-prime-beyond-aux 2 n*
  by (*simp add*: *smallest-prime-beyond-aux-def*)

end

11 Comprehensive number theory

theory *Number-Theory*

imports *Fib Residues Eratosthenes*
begin

end

12 Less common functions on lists

theory *More-List*
imports *Main*
begin

definition *strip-while* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$
where
  strip-while P = rev ∘ dropWhile P ∘ rev

lemma *strip-while-rev* [*simp*]:
  strip-while P (*rev xs*) = *rev* (*dropWhile P xs*)
  by (*simp add*: *strip-while-def*)

lemma *strip-while-Nil* [*simp*]:
  strip-while P [] = []
  by (*simp add*: *strip-while-def*)

lemma *strip-while-append* [*simp*]:
  ¬ *P x* ⟹ *strip-while P* (*xs* @ [*x*]) = *xs* @ [*x*]
  by (*simp add*: *strip-while-def*)

lemma *strip-while-append-rec* [*simp*]:
  P x ⟹ *strip-while P* (*xs* @ [*x*]) = *strip-while P xs*
  by (*simp add*: *strip-while-def*)

lemma *strip-while-Cons* [*simp*]:
  ¬ *P x* ⟹ *strip-while P* (*x* # *xs*) = *x* # *strip-while P xs*
  by (*induct xs rule*: *rev-induct*) (*simp-all add*: *strip-while-def*)

lemma *strip-while-eq-Nil* [*simp*]:
  strip-while P xs = [] ⟷ (∀ *x*∈*set xs*. *P x*)
  by (*simp add*: *strip-while-def*)

lemma *strip-while-eq-Cons-rec*:
  strip-while P (*x* # *xs*) = *x* # *strip-while P xs* ⟷ ¬ (*P x* ∧ (∀ *x*∈*set xs*. *P x*))
  by (*induct xs rule*: *rev-induct*) (*simp-all add*: *strip-while-def*)

lemma *strip-while-not-last* [*simp*]:
  ¬ *P* (*last xs*) ⟹ *strip-while P xs* = *xs*
  by (*cases xs rule*: *rev-cases*) *simp-all*

lemma *split-strip-while-append*:
  fixes *xs* :: *'a list*

obtains *ys zs* :: *'a list*
where *strip-while P xs* = *ys* **and** ∀ *x*∈*set zs*. *P x* **and** *xs* = *ys* @ *zs*
proof (*rule that*)
show *strip-while P xs* = *strip-while P xs* **..**
show ∀ *x*∈*set* (*rev* (*takeWhile P* (*rev xs*))). *P x* **by** (*simp add*: *takeWhile-eq-all-conv* [*symmetric*])
have *rev xs* = *rev* (*strip-while P xs* @ *rev* (*takeWhile P* (*rev xs*)))
by (*simp add*: *strip-while-def*)
then show *xs* = *strip-while P xs* @ *rev* (*takeWhile P* (*rev xs*))
by (*simp only*: *rev-is-rev-conv*)
qed

lemma *strip-while-snoc* [*simp*]:
strip-while P (*xs* @ [*x*]) = (*if P x then strip-while P xs else xs* @ [*x*])
by (*simp add*: *strip-while-def*)

lemma *strip-while-map*:
strip-while P (*map f xs*) = *map f* (*strip-while* (*P* ∘ *f*) *xs*)
by (*simp add*: *strip-while-def rev-map dropWhile-map*)

definition *no-leading* :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool*
where
no-leading P xs ⟷ (*xs* ≠ [] ⟶ ¬ *P* (*hd xs*))

lemma *no-leading-Nil* [*simp*, *intro*!]:
no-leading P []
by (*simp add*: *no-leading-def*)

lemma *no-leading-Cons* [*simp*, *intro*!]:
no-leading P (*x* # *xs*) ⟷ ¬ *P x*
by (*simp add*: *no-leading-def*)

lemma *no-leading-append* [*simp*]:
no-leading P (*xs* @ *ys*) ⟷ *no-leading P xs* ∧ (*xs* = [] ⟶ *no-leading P ys*)
by (*induct xs*) *simp-all*

lemma *no-leading-dropWhile* [*simp*]:
no-leading P (*dropWhile P xs*)
by (*induct xs*) *simp-all*

lemma *dropWhile-eq-obtain-leading*:
assumes *dropWhile P xs* = *ys*
obtains *zs* **where** *xs* = *zs* @ *ys* **and** ⋀*z*. *z* ∈ *set zs* ⟹ *P z* **and** *no-leading P ys*
proof −
from *assms* **have** ∃ *zs*. *xs* = *zs* @ *ys* ∧ (∀ *z* ∈ *set zs*. *P z*) ∧ *no-leading P ys*
proof (*induct xs arbitrary*: *ys*)
case *Nil* **then show** *?case* **by** *simp*

next
  case (*Cons x xs ys*)
  show *?case* **proof** (*cases P x*)
    case *True* **with** *Cons.hyps* [*of ys*] *Cons.prems*
    have $\exists$ *zs. xs = zs @ ys* $\wedge$ ($\forall$ *a*$\in$*set zs. P a*) $\wedge$ *no-leading P ys*
      by *simp*
    then obtain *zs* **where** *xs = zs @ ys* **and** $\bigwedge$*z. z* $\in$ *set zs* $\Longrightarrow$ *P z*
      and $*$: *no-leading P ys*
      by *blast*
    with *True* **have** *x # xs = (x # zs) @ ys* **and** $\bigwedge$*z. z* $\in$ *set (x # zs)* $\Longrightarrow$ *P z*
      by *auto*
    with $*$ **show** *?thesis*
      by *blast* **next**
    case *False*
    with *Cons* **show** *?thesis* **by** (*cases ys*) *simp-all*
  qed
qed
with *that* **show** *thesis*
  by *blast*
qed

lemma *dropWhile-idem-iff*:
  dropWhile P xs = xs $\longleftrightarrow$ *no-leading P xs*
  by (*cases xs*) (*auto elim*: *dropWhile-eq-obtain-leading*)

abbreviation *no-trailing* :: (*'a* $\Rightarrow$ *bool*) $\Rightarrow$ *'a list* $\Rightarrow$ *bool*
where
  no-trailing P xs $\equiv$ *no-leading P* (*rev xs*)

lemma *no-trailing-unfold*:
  no-trailing P xs $\longleftrightarrow$ (*xs* $\neq$ [] $\longrightarrow$ $\neg$ *P* (*last xs*))
  by (*induct xs*) *simp-all*

lemma *no-trailing-Nil* [*simp*, *intro*!]:
  no-trailing P []
  by *simp*

lemma *no-trailing-Cons* [*simp*]:
  no-trailing P (*x # xs*) $\longleftrightarrow$ *no-trailing P xs* $\wedge$ (*xs* = [] $\longrightarrow$ $\neg$ *P x*)
  by *simp*

lemma *no-trailing-append-Cons* [*simp*]:
  no-trailing P (*xs @ y # ys*) $\longleftrightarrow$ *no-trailing P* (*y # ys*)
  by *simp*

lemma *no-trailing-strip-while* [*simp*]:
  no-trailing P (*strip-while P xs*)
  by (*induct xs rule*: *rev-induct*) *simp-all*

lemma *strip-while-eq-obtain-trailing*:
  assumes *strip-while P xs = ys*
  obtains *zs* **where** *xs = ys @ zs* **and** $\bigwedge z.\ z \in set\ zs \Longrightarrow P\ z$ **and** *no-trailing P ys*
proof –
  from *assms* **have** *rev (rev (dropWhile P (rev xs))) = rev ys*
    by (*simp add*: *strip-while-def*)
  then have *dropWhile P (rev xs) = rev ys*
    by *simp*
  then obtain *zs* **where** *A*: *rev xs = zs @ rev ys* **and** *B*: $\bigwedge z.\ z \in set\ zs \Longrightarrow P\ z$
    and *C*: *no-trailing P ys*
    using *dropWhile-eq-obtain-leading* **by** *blast*
  from *A* **have** *rev (rev xs) = rev (zs @ rev ys)*
    by *simp*
  then have *xs = ys @ rev zs*
    by *simp*
  moreover from *B* **have** $\bigwedge z.\ z \in set\ (rev\ zs) \Longrightarrow P\ z$
    by *simp*
  ultimately show *thesis* **using** *that C* **by** *blast*
qed

lemma *strip-while-idem-iff*:
  strip-while P xs = xs ⟷ *no-trailing P xs*
proof –
  def *ys* ≡ *rev xs*
  moreover have *strip-while P (rev ys) = rev ys* ⟷ *no-trailing P (rev ys)*
    by (*simp add*: *dropWhile-idem-iff*)
  ultimately show *?thesis* **by** *simp*
qed

lemma *no-trailing-map*:
  no-trailing P (map f xs) = no-trailing (P ∘ f) xs
  by (*simp add*: *last-map no-trailing-unfold*)

lemma *no-trailing-upt* [*simp*]:
  no-trailing P [n..<m] ⟷ $(n < m \longrightarrow \neg\ P\ (m - 1))$
  by (*auto simp add*: *no-trailing-unfold*)

definition *nth-default* :: $'a \Rightarrow 'a\ list \Rightarrow nat \Rightarrow 'a$
where
  nth-default dflt xs n = (if n < length xs then xs ! n else dflt)

lemma *nth-default-nth*:
  $n < length\ xs \Longrightarrow nth\text{-}default\ dflt\ xs\ n = xs\ !\ n$
  by (*simp add*: *nth-default-def*)

lemma *nth-default-beyond*:

length xs ≤ *n* ⟹ *nth-default dflt xs n* = *dflt*
by (*simp add*: *nth-default-def*)

lemma *nth-default-Nil* [*simp*]:
nth-default dflt [] *n* = *dflt*
by (*simp add*: *nth-default-def*)

lemma *nth-default-Cons*:
nth-default dflt (*x* # *xs*) *n* = (*case n of 0* ⇒ *x* | *Suc n'* ⇒ *nth-default dflt xs n'*)
by (*simp add*: *nth-default-def split*: *nat.split*)

lemma *nth-default-Cons-0* [*simp*]:
nth-default dflt (*x* # *xs*) *0* = *x*
by (*simp add*: *nth-default-Cons*)

lemma *nth-default-Cons-Suc* [*simp*]:
nth-default dflt (*x* # *xs*) (*Suc n*) = *nth-default dflt xs n*
by (*simp add*: *nth-default-Cons*)

lemma *nth-default-replicate-dflt* [*simp*]:
nth-default dflt (*replicate n dflt*) *m* = *dflt*
by (*simp add*: *nth-default-def*)

lemma *nth-default-append*:
nth-default dflt (*xs* @ *ys*) *n* =
(*if n* < *length xs then nth xs n else nth-default dflt ys* (*n* − *length xs*))
by (*auto simp add*: *nth-default-def nth-append*)

lemma *nth-default-append-trailing* [*simp*]:
nth-default dflt (*xs* @ *replicate n dflt*) = *nth-default dflt xs*
by (*simp add*: *fun-eq-iff nth-default-append*) (*simp add*: *nth-default-def*)

lemma *nth-default-snoc-default* [*simp*]:
nth-default dflt (*xs* @ [*dflt*]) = *nth-default dflt xs*
by (*auto simp add*: *nth-default-def fun-eq-iff nth-append*)

lemma *nth-default-eq-dflt-iff*:
nth-default dflt xs k = *dflt* ⟷ (*k* < *length xs* ⟶ *xs* ! *k* = *dflt*)
by (*simp add*: *nth-default-def*)

lemma *in-enumerate-iff-nth-default-eq*:
x ≠ *dflt* ⟹ (*n*, *x*) ∈ *set* (*enumerate 0 xs*) ⟷ *nth-default dflt xs n* = *x*
by (*auto simp add*: *nth-default-def in-set-conv-nth enumerate-eq-zip*)

lemma *last-conv-nth-default*:
assumes *xs* ≠ []
shows *last xs* = *nth-default dflt xs* (*length xs* − *1*)
using *assms* **by** (*simp add*: *nth-default-def last-conv-nth*)

lemma *nth-default-map-eq*:
f dflt′ = *dflt* ⟹ *nth-default dflt* (*map f xs*) *n* = *f* (*nth-default dflt′ xs n*)
by (*simp add*: *nth-default-def*)

lemma *finite-nth-default-neq-default* [*simp*]:
finite {*k*. *nth-default dflt xs k* ≠ *dflt*}
by (*simp add*: *nth-default-def*)

lemma *sorted-list-of-set-nth-default*:
sorted-list-of-set {*k*. *nth-default dflt xs k* ≠ *dflt*} = *map fst* (*filter* (λ(-, *x*). *x* ≠ *dflt*) (*enumerate 0 xs*))
by (*rule sorted-distinct-set-unique*) (*auto simp add*: *nth-default-def in-set-conv-nth sorted-filter distinct-map-filter enumerate-eq-zip intro*: *rev-image-eqI*)

lemma *map-nth-default*:
map (*nth-default x xs*) [*0*..<*length xs*] = *xs*
proof −
have ∗: *map* (*nth-default x xs*) [*0*..<*length xs*] = *map* (*List.nth xs*) [*0*..<*length xs*]
by (*rule map-cong*) (*simp-all add*: *nth-default-nth*)
show *?thesis* **by** (*simp add*: ∗ *map-nth*)
qed

lemma *range-nth-default* [*simp*]:
range (*nth-default dflt xs*) = *insert dflt* (*set xs*)
by (*auto simp add*: *nth-default-def* [*abs-def*] *in-set-conv-nth*)

lemma *nth-strip-while*:
assumes *n* < *length* (*strip-while P xs*)
shows *strip-while P xs* ! *n* = *xs* ! *n*
proof −
have *length* (*dropWhile P* (*rev xs*)) + *length* (*takeWhile P* (*rev xs*)) = *length xs*
by (*subst add.commute*)
(*simp add*: *arg-cong* [**where** *f*=*length*, *OF takeWhile-dropWhile-id*, *unfolded length-append*])
then show *?thesis* **using** *assms*
by (*simp add*: *strip-while-def rev-nth dropWhile-nth*)
qed

lemma *length-strip-while-le*:
length (*strip-while P xs*) ≤ *length xs*
unfolding *strip-while-def o-def length-rev*
by (*subst* (*2*) *length-rev*[*symmetric*])
(*simp add*: *strip-while-def length-dropWhile-le del*: *length-rev*)

lemma *nth-default-strip-while-dflt* [*simp*]:
nth-default dflt (*strip-while* (*op* = *dflt*) *xs*) = *nth-default dflt xs*
by (*induct xs rule*: *rev-induct*) *auto*

lemma *nth-default-eq-iff*:
nth-default dflt xs = *nth-default dflt ys*
⟷ *strip-while* (*HOL.eq dflt*) *xs* = *strip-while* (*HOL.eq dflt*) *ys* (**is** *?P* ⟷ *?Q*)
proof
let *?xs* = *strip-while* (*HOL.eq dflt*) *xs* **and** *?ys* = *strip-while* (*HOL.eq dflt*) *ys*
assume *?P*
then have *eq*: *nth-default dflt ?xs* = *nth-default dflt ?ys*
by *simp*
have *len*: *length ?xs* = *length ?ys*
proof (*rule ccontr*)
assume *len*: *length ?xs* ≠ *length ?ys*
{ fix *xs ys* :: ′*a list*
let *?xs* = *strip-while* (*HOL.eq dflt*) *xs* **and** *?ys* = *strip-while* (*HOL.eq dflt*) *ys*
assume *eq*: *nth-default dflt ?xs* = *nth-default dflt ?ys*
assume *len*: *length ?xs* < *length ?ys*
then have *length ?ys* > *0* **by** *arith*
then have *?ys* ≠ [] **by** *simp*
with *last-conv-nth-default* [*of ?ys dflt*]
have *last ?ys* = *nth-default dflt ?ys* (*length ?ys* − *1*)
by *auto*
moreover from ‹*?ys* ≠ []› *no-trailing-strip-while* [*of HOL.eq dflt ys*]
have *last ?ys* ≠ *dflt* **by** (*simp add*: *no-trailing-unfold*)
ultimately have *nth-default dflt ?xs* (*length ?ys* − *1*) ≠ *dflt*
using *eq* **by** *simp*
moreover from *len* **have** *length ?ys* − *1* ≥ *length ?xs* **by** *simp*
ultimately have *False* **by** (*simp only*: *nth-default-beyond*) *simp*
}
from *this* [*of xs ys*] *this* [*of ys xs*] *len eq* **show** *False*
by (*auto simp only*: *linorder-class.neq-iff*)
qed
then show *?Q*
proof (*rule nth-equalityI* [*rule-format*])
fix *n*
assume *n* < *length ?xs*
moreover with *len* **have** *n* < *length ?ys*
by *simp*
ultimately have *xs*: *nth-default dflt ?xs n* = *?xs* ! *n*
and *ys*: *nth-default dflt ?ys n* = *?ys* ! *n*
by (*simp-all only*: *nth-default-nth*)
with *eq* **show** *?xs* ! *n* = *?ys* ! *n*
by *simp*
qed
next
assume *?Q*
then have *nth-default dflt* (*strip-while* (*HOL.eq dflt*) *xs*) = *nth-default dflt* (*strip-while* (*HOL.eq dflt*) *ys*)
by *simp*
then show *?P*

by *simp*
qed

end

13 Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *Main*
begin

The set of natural numbers is infinite.

lemma *infinite-nat-iff-unbounded-le*: *infinite* (*S*::*nat set*) ⟷ (∀ *m*. ∃ *n*≥*m*. *n* ∈ *S*)
  using *frequently-cofinite*[*of* λ*x*. *x* ∈ *S*]
  by (*simp add*: *cofinite-eq-sequentially frequently-def eventually-sequentially*)

lemma *infinite-nat-iff-unbounded*: *infinite* (*S*::*nat set*) ⟷ (∀ *m*. ∃ *n*>*m*. *n* ∈ *S*)
  using *frequently-cofinite*[*of* λ*x*. *x* ∈ *S*]
  by (*simp add*: *cofinite-eq-sequentially frequently-def eventually-at-top-dense*)

lemma *finite-nat-iff-bounded*: *finite* (*S*::*nat set*) ⟷ (∃ *k*. *S* ⊆ {..<*k*})
  using *infinite-nat-iff-unbounded-le*[*of S*] **by** (*simp add*: *subset-eq*) (*metis not-le*)

lemma *finite-nat-iff-bounded-le*: *finite* (*S*::*nat set*) ⟷ (∃ *k*. *S* ⊆ {.. *k*})
  using *infinite-nat-iff-unbounded*[*of S*] **by** (*simp add*: *subset-eq*) (*metis not-le*)

lemma *finite-nat-bounded*: *finite* (*S*::*nat set*) ⟹ ∃ *k*. *S* ⊆ {..<*k*}
  by (*simp add*: *finite-nat-iff-bounded*)

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some k, there is some larger number that is an element of the set.

lemma *unbounded-k-infinite*: ∀ *m*>*k*. ∃ *n*>*m*. *n* ∈ *S* ⟹ *infinite* (*S*::*nat set*)
apply (*clarsimp simp add*: *finite-nat-set-iff-bounded*)
apply (*drule-tac x*=*Suc* (*max m k*) **in** *spec*)
using *less-Suc-eq* **by** *fastforce*

lemma *nat-not-finite*: *finite* (*UNIV*::*nat set*) ⟹ *R*
  by *simp*

lemma *range-inj-infinite*:
  inj (*f*::*nat* ⇒ ′*a*) ⟹ *infinite* (*range f*)
proof
  assume *finite* (*range f*) **and** *inj f*
  then have *finite* (*UNIV*::*nat set*)
    by (*rule finite-imageD*)
  then show *False* **by** *simp*

qed

The set of integers is also infinite.

lemma *infinite-int-iff-infinite-nat-abs*: *infinite* (*S*::*int set*) ⟷ *infinite* ((*nat o abs*) ‘ *S*)
by (*auto simp*: *transfer-nat-int-set-relations o-def image-comp dest*: *finite-image-absD*)

proposition *infinite-int-iff-unbounded-le*: *infinite* (*S*::*int set*) ⟷ (∀ *m*. ∃ *n*. |*n*| ≥ *m* ∧ *n* ∈ *S*)
apply (*simp add*: *infinite-int-iff-infinite-nat-abs infinite-nat-iff-unbounded-le o-def image-def*)
apply (*metis abs-ge-zero nat-le-eq-zle le-nat-iff*)
done

proposition *infinite-int-iff-unbounded*: *infinite* (*S*::*int set*) ⟷ (∀ *m*. ∃ *n*. |*n*| > *m* ∧ *n* ∈ *S*)
apply (*simp add*: *infinite-int-iff-infinite-nat-abs infinite-nat-iff-unbounded o-def image-def*)
apply (*metis* (*full-types*) *nat-le-iff nat-mono not-le*)
done

proposition *finite-int-iff-bounded*: *finite* (*S*::*int set*) ⟷ (∃ *k*. *abs* ‘ *S* ⊆ {..<*k*})
using *infinite-int-iff-unbounded-le*[*of S*] **by** (*simp add*: *subset-eq*) (*metis not-le*)

proposition *finite-int-iff-bounded-le*: *finite* (*S*::*int set*) ⟷ (∃ *k*. *abs* ‘ *S* ⊆ {.. *k*})
using *infinite-int-iff-unbounded*[*of S*] **by** (*simp add*: *subset-eq*) (*metis not-le*)

13.1 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

lemma *not-INFM* [*simp*]: ¬ (*INFM x*. *P x*) ⟷ (*MOST x*. ¬ *P x*) **by** (*fact not-frequently*)
lemma *not-MOST* [*simp*]: ¬ (*MOST x*. *P x*) ⟷ (*INFM x*. ¬ *P x*) **by** (*fact not-eventually*)

lemma *INFM-const* [*simp*]: (*INFM x*::′*a*. *P*) ⟷ *P* ∧ *infinite* (*UNIV*::′*a set*)
by (*simp add*: *frequently-const-iff*)

lemma *MOST-const* [*simp*]: (*MOST x*::′*a*. *P*) ⟷ *P* ∨ *finite* (*UNIV*::′*a set*)
by (*simp add*: *eventually-const-iff*)

lemma *INFM-imp-distrib*: (*INFM x*. *P x* ⟶ *Q x*) ⟷ ((*MOST x*. *P x*) ⟶ (*INFM x*. *Q x*))
by (*simp only*: *imp-conv-disj frequently-disj-iff not-eventually*)

lemma *MOST-imp-iff*: *MOST x*. *P x* ⟹ (*MOST x*. *P x* ⟶ *Q x*) ⟷ (*MOST x*. *Q x*)

by (*auto intro*: *eventually-rev-mp eventually-mono*)

lemma *INFM-conjI*: *INFM x*. *P x* $\Longrightarrow$ *MOST x*. *Q x* $\Longrightarrow$ *INFM x*. *P x* $\wedge$ *Q x*
by (*rule frequently-rev-mp*[*of P*]) (*auto elim*: *eventually-mono*)

Properties of quantifiers with injective functions.

lemma *INFM-inj*: *INFM x*. *P* (*f x*) $\Longrightarrow$ *inj f* $\Longrightarrow$ *INFM x*. *P x*
using *finite-vimageI*[*of* {*x*. *P x*} *f*] **by** (*auto simp*: *frequently-cofinite*)

lemma *MOST-inj*: *MOST x*. *P x* $\Longrightarrow$ *inj f* $\Longrightarrow$ *MOST x*. *P* (*f x*)
using *finite-vimageI*[*of* {*x*. $\neg$ *P x*} *f*] **by** (*auto simp*: *eventually-cofinite*)

Properties of quantifiers with singletons.

lemma *not-INFM-eq* [*simp*]:
$\neg$ (*INFM x*. *x* = *a*)
$\neg$ (*INFM x*. *a* = *x*)
unfolding *frequently-cofinite* **by** *simp-all*

lemma *MOST-neq* [*simp*]:
MOST x. *x* $\neq$ *a*
MOST x. *a* $\neq$ *x*
unfolding *eventually-cofinite* **by** *simp-all*

lemma *INFM-neq* [*simp*]:
(*INFM x*::'*a*. *x* $\neq$ *a*) $\longleftrightarrow$ *infinite* (*UNIV*::'*a set*)
(*INFM x*::'*a*. *a* $\neq$ *x*) $\longleftrightarrow$ *infinite* (*UNIV*::'*a set*)
unfolding *frequently-cofinite* **by** *simp-all*

lemma *MOST-eq* [*simp*]:
(*MOST x*::'*a*. *x* = *a*) $\longleftrightarrow$ *finite* (*UNIV*::'*a set*)
(*MOST x*::'*a*. *a* = *x*) $\longleftrightarrow$ *finite* (*UNIV*::'*a set*)
unfolding *eventually-cofinite* **by** *simp-all*

lemma *MOST-eq-imp*:
MOST x. *x* = *a* $\longrightarrow$ *P x*
MOST x. *a* = *x* $\longrightarrow$ *P x*
unfolding *eventually-cofinite* **by** *simp-all*

Properties of quantifiers over the naturals.

lemma *MOST-nat*: ($\forall_\infty$*n*. *P* (*n*::*nat*)) $\longleftrightarrow$ ($\exists$ *m*. $\forall$ *n*>*m*. *P n*)
by (*auto simp add*: *eventually-cofinite finite-nat-iff-bounded-le subset-eq not-le*[*symmetric*])

lemma *MOST-nat-le*: ($\forall_\infty$*n*. *P* (*n*::*nat*)) $\longleftrightarrow$ ($\exists$ *m*. $\forall$ *n*$\geq$*m*. *P n*)
by (*auto simp add*: *eventually-cofinite finite-nat-iff-bounded subset-eq not-le*[*symmetric*])

lemma *INFM-nat*: ($\exists_\infty$*n*. *P* (*n*::*nat*)) $\longleftrightarrow$ ($\forall$ *m*. $\exists$ *n*>*m*. *P n*)
by (*simp add*: *frequently-cofinite infinite-nat-iff-unbounded*)

lemma *INFM-nat-le*: ($\exists_\infty$*n*. *P* (*n*::*nat*)) $\longleftrightarrow$ ($\forall$ *m*. $\exists$ *n*$\geq$*m*. *P n*)

by (*simp add*: *frequently-cofinite infinite-nat-iff-unbounded-le*)

lemma *MOST-INFM*: *infinite* (*UNIV*::*'a set*) $\Longrightarrow$ *MOST x*::*'a*. *P x* $\Longrightarrow$ *INFM x*::*'a*. *P x*
by (*simp add*: *eventually-frequently*)

lemma *MOST-Suc-iff*: (*MOST n*. *P* (*Suc n*)) $\longleftrightarrow$ (*MOST n*. *P n*)
by (*simp add*: *cofinite-eq-sequentially eventually-sequentially-Suc*)

lemma
shows *MOST-SucI*: *MOST n*. *P n* $\Longrightarrow$ *MOST n*. *P* (*Suc n*)
and *MOST-SucD*: *MOST n*. *P* (*Suc n*) $\Longrightarrow$ *MOST n*. *P n*
by (*simp-all add*: *MOST-Suc-iff*)

lemma *MOST-ge-nat*: *MOST n*::*nat*. $m \le n$
by (*simp add*: *cofinite-eq-sequentially eventually-ge-at-top*)

lemma *Inf-many-def*: *Inf-many P* $\longleftrightarrow$ *infinite* $\{x.\ P\ x\}$ **by** (*fact frequently-cofinite*)
lemma *Alm-all-def*: *Alm-all P* $\longleftrightarrow$ $\neg$ (*INFM x*. $\neg$ *P x*) **by** *simp*
lemma *INFM-iff-infinite*: (*INFM x*. *P x*) $\longleftrightarrow$ *infinite* $\{x.\ P\ x\}$ **by** (*fact frequently-cofinite*)
lemma *MOST-iff-cofinite*: (*MOST x*. *P x*) $\longleftrightarrow$ *finite* $\{x.\ \neg\ P\ x\}$ **by** (*fact eventually-cofinite*)
lemma *INFM-EX*: ($\exists_\infty x.\ P\ x$) $\Longrightarrow$ ($\exists x.\ P\ x$) **by** (*fact frequently-ex*)
lemma *ALL-MOST*: $\forall x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$ **by** (*fact always-eventually*)
lemma *INFM-mono*: $\exists_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \exists_\infty x.\ Q\ x$ **by** (*fact frequently-elim1*)
lemma *MOST-mono*: $\forall_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \forall_\infty x.\ Q\ x$ **by** (*fact eventually-mono*)
lemma *INFM-disj-distrib*: $(\exists_\infty x.\ P\ x \lor Q\ x) \longleftrightarrow (\exists_\infty x.\ P\ x) \lor (\exists_\infty x.\ Q\ x)$ **by** (*fact frequently-disj-iff*)
lemma *MOST-rev-mp*: $\forall_\infty x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x \longrightarrow Q\ x \Longrightarrow \forall_\infty x.\ Q\ x$ **by** (*fact eventually-rev-mp*)
lemma *MOST-conj-distrib*: $(\forall_\infty x.\ P\ x \land Q\ x) \longleftrightarrow (\forall_\infty x.\ P\ x) \land (\forall_\infty x.\ Q\ x)$ **by** (*fact eventually-conj-iff*)
lemma *MOST-conjI*: *MOST x*. *P x* $\Longrightarrow$ *MOST x*. *Q x* $\Longrightarrow$ *MOST x*. *P x* $\land$ *Q x* **by** (*fact eventually-conj*)
lemma *INFM-finite-Bex-distrib*: *finite A* $\Longrightarrow$ (*INFM y*. $\exists x{\in}A$. *P x y*) $\longleftrightarrow$ ($\exists x{\in}A$. *INFM y*. *P x y*) **by** (*fact frequently-bex-finite-distrib*)
lemma *MOST-finite-Ball-distrib*: *finite A* $\Longrightarrow$ (*MOST y*. $\forall x{\in}A$. *P x y*) $\longleftrightarrow$ ($\forall x{\in}A$. *MOST y*. *P x y*) **by** (*fact eventually-ball-finite-distrib*)
lemma *INFM-E*: *INFM x*. *P x* $\Longrightarrow$ ($\bigwedge x$. *P x* $\Longrightarrow$ *thesis*) $\Longrightarrow$ *thesis* **by** (*fact frequentlyE*)
lemma *MOST-I*: ($\bigwedge x$. *P x*) $\Longrightarrow$ *MOST x*. *P x* **by** (*rule eventuallyI*)
lemmas *MOST-iff-finiteNeg* = *MOST-iff-cofinite*

13.2 Enumeration of an Infinite Set

The set's element type must be wellordered (e.g. the natural numbers).

Could be generalized to *enumerate′ S n* = (*SOME t. t* ∈ *s* ∧ *finite* {*s* ∈ *S*. *s* < *t*} ∧ *card* {*s* ∈ *S*. *s* < *t*} = *n*).

primrec (**in** *wellorder*) *enumerate* :: *′a set* ⇒ *nat* ⇒ *′a*
where
  enumerate-0: *enumerate S 0* = (*LEAST n*. *n* ∈ *S*)
| *enumerate-Suc*: *enumerate S* (*Suc n*) = *enumerate* (*S* − {*LEAST n*. *n* ∈ *S*}) *n*

lemma *enumerate-Suc′*: *enumerate S* (*Suc n*) = *enumerate* (*S* − {*enumerate S 0*}) *n*
  by *simp*

lemma *enumerate-in-set*: *infinite S* ⟹ *enumerate S n* ∈ *S*
  apply (*induct n arbitrary*: *S*)
   apply (*fastforce intro*: *LeastI dest*!: *infinite-imp-nonempty*)
  apply *simp*
  apply (*metis DiffE infinite-remove*)
  done

declare *enumerate-0* [*simp del*] *enumerate-Suc* [*simp del*]

lemma *enumerate-step*: *infinite S* ⟹ *enumerate S n* < *enumerate S* (*Suc n*)
  apply (*induct n arbitrary*: *S*)
   apply (*rule order-le-neq-trans*)
    apply (*simp add*: *enumerate-0 Least-le enumerate-in-set*)
   apply (*simp only*: *enumerate-Suc′*)
   apply (*subgoal-tac enumerate* (*S* − {*enumerate S 0*}) *0* ∈ *S* − {*enumerate S 0*})
    apply (*blast intro*: *sym*)
   apply (*simp add*: *enumerate-in-set del*: *Diff-iff*)
  apply (*simp add*: *enumerate-Suc′*)
  done

lemma *enumerate-mono*: *m* < *n* ⟹ *infinite S* ⟹ *enumerate S m* < *enumerate S n*
  apply (*erule less-Suc-induct*)
  apply (*auto intro*: *enumerate-step*)
  done

lemma *le-enumerate*:
  assumes *S*: *infinite S*
  shows *n* ≤ *enumerate S n*
  using *S*
proof (*induct n*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc n*)
  then have *n* ≤ *enumerate S n* **by** *simp*

```
  also note enumerate-mono[of n Suc n, OF - ‹infinite S›]
  finally show ?case by simp
qed

lemma enumerate-Suc'':
  fixes S :: 'a::wellorder set
  assumes infinite S
  shows enumerate S (Suc n) = (LEAST s. s ∈ S ∧ enumerate S n < s)
  using assms
proof (induct n arbitrary: S)
  case 0
  then have ∀ s ∈ S. enumerate S 0 ≤ s
    by (auto simp: enumerate.simps intro: Least-le)
  then show ?case
    unfolding enumerate-Suc' enumerate-0[of S − {enumerate S 0}]
    by (intro arg-cong[where f = Least] ext) auto
next
  case (Suc n S)
  show ?case
    using enumerate-mono[OF zero-less-Suc ‹infinite S›, of n] ‹infinite S›
    apply (subst (1 2) enumerate-Suc')
    apply (subst Suc)
    using ‹infinite S›
    apply simp
    apply (intro arg-cong[where f = Least] ext)
    apply (auto simp: enumerate-Suc'[symmetric])
    done
qed

lemma enumerate-Ex:
  assumes S: infinite (S::nat set)
  shows s ∈ S ⟹ ∃ n. enumerate S n = s
proof (induct s rule: less-induct)
  case (less s)
  show ?case
  proof cases
    let ?y = Max {s'∈S. s' < s}
    assume ∃ y∈S. y < s
    then have y: ⋀x. ?y < x ⟷ (∀ s'∈S. s' < s ⟶ s' < x)
      by (subst Max-less-iff) auto
    then have y-in: ?y ∈ {s'∈S. s' < s}
      by (intro Max-in) auto
    with less.hyps[of ?y] obtain n where enumerate S n = ?y
      by auto
    with S have enumerate S (Suc n) = s
      by (auto simp: y less enumerate-Suc'' intro!: Least-equality)
    then show ?case by auto
  next
    assume *: ¬ (∃ y∈S. y < s)
```

then have ∀ *t*∈*S*. *s* ≤ *t* **by** *auto*
with ‹*s* ∈ *S*› **show** *?thesis*
by (*auto intro*!: *exI*[*of* - *0*] *Least-equality simp*: *enumerate-0*)
qed
qed

lemma *bij-enumerate*:
fixes *S* :: *nat set*
assumes *S*: *infinite S*
shows *bij-betw* (*enumerate S*) *UNIV S*
proof −
have ⋀*n m*. *n* ≠ *m* ⟹ *enumerate S n* ≠ *enumerate S m*
using *enumerate-mono*[*OF* - ‹*infinite S*›] **by** (*auto simp*: *neq-iff*)
then have *inj* (*enumerate S*)
by (*auto simp*: *inj-on-def*)
moreover have ∀ *s* ∈ *S*. ∃ *i*. *enumerate S i* = *s*
using *enumerate-Ex*[*OF S*] **by** *auto*
moreover note ‹*infinite S*›
ultimately show *?thesis*
unfolding *bij-betw-def* **by** (*auto intro*: *enumerate-in-set*)
qed

end

14 Polynomials as type over a ring structure

theory *Polynomial*
imports *Main* ~~/*src*/*HOL*/*Deriv* ~~/*src*/*HOL*/*Library*/*More-List*
~~/*src*/*HOL*/*Library*/*Infinite-Set*
begin

14.1 Auxiliary: operations for lists (later) representing coefficients

definition *cCons* :: *'a*::*zero* ⇒ *'a list* ⇒ *'a list* (**infixr** ## *65*)
where
x ## *xs* = (*if xs* = [] ∧ *x* = *0 then* [] *else x* # *xs*)

lemma *cCons-0-Nil-eq* [*simp*]:
0 ## [] = []
by (*simp add*: *cCons-def*)

lemma *cCons-Cons-eq* [*simp*]:
x ## *y* # *ys* = *x* # *y* # *ys*
by (*simp add*: *cCons-def*)

lemma *cCons-append-Cons-eq* [*simp*]:
x ## *xs* @ *y* # *ys* = *x* # *xs* @ *y* # *ys*
by (*simp add*: *cCons-def*)

lemma *cCons-not-0-eq* [*simp*]:
  $x \neq 0 \Longrightarrow x \mathbin{\#\#} xs = x \mathbin{\#} xs$
  by (*simp add*: *cCons-def*)

lemma *strip-while-not-0-Cons-eq* [*simp*]:
  strip-while ($\lambda x.\ x = 0$) ($x \mathbin{\#} xs$) = $x \mathbin{\#\#}$ *strip-while* ($\lambda x.\ x = 0$) *xs*
proof (*cases* $x = 0$)
  case *False* **then show** *?thesis* **by** *simp*
next
  case *True* **show** *?thesis*
  proof (*induct xs rule*: *rev-induct*)
    case *Nil* **with** *True* **show** *?case* **by** *simp*
  next
    case (*snoc y ys*) **then show** *?case*
      by (*cases* $y = 0$) (*simp-all add*: *append-Cons* [*symmetric*] *del*: *append-Cons*)
  qed
qed

lemma *tl-cCons* [*simp*]:
  tl ($x \mathbin{\#\#} xs$) = *xs*
  by (*simp add*: *cCons-def*)

14.2 Definition of type *poly*

typedef (**overloaded**) *'a poly* = $\{f :: nat \Rightarrow {'a}::zero.\ \forall_\infty\ n.\ f\ n = 0\}$
  morphisms *coeff Abs-poly* **by** (*auto intro*!: *ALL-MOST*)

setup-lifting *type-definition-poly*

lemma *poly-eq-iff*: $p = q \longleftrightarrow (\forall n.\ coeff\ p\ n = coeff\ q\ n)$
  by (*simp add*: *coeff-inject* [*symmetric*] *fun-eq-iff*)

lemma *poly-eqI*: $(\bigwedge n.\ coeff\ p\ n = coeff\ q\ n) \Longrightarrow p = q$
  by (*simp add*: *poly-eq-iff*)

lemma *MOST-coeff-eq-0*: $\forall_\infty\ n.\ coeff\ p\ n = 0$
  using *coeff* [*of p*] **by** *simp*

14.3 Degree of a polynomial

definition *degree* :: *'a::zero poly* ⇒ *nat*
where
  $degree\ p = (LEAST\ n.\ \forall i{>}n.\ coeff\ p\ i = 0)$

lemma *coeff-eq-0*:
  assumes $degree\ p < n$
  shows $coeff\ p\ n = 0$
proof −
  have $\exists n.\ \forall i{>}n.\ coeff\ p\ i = 0$

using *MOST-coeff-eq-0* **by** (*simp add*: *MOST-nat*)
then have ∀ *i*>*degree p*. *coeff p i* = *0*
unfolding *degree-def* **by** (*rule LeastI-ex*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *le-degree*: *coeff p n* ≠ *0* ⟹ *n* ≤ *degree p*
by (*erule contrapos-np*, *rule coeff-eq-0*, *simp*)

lemma *degree-le*: ∀ *i*>*n*. *coeff p i* = *0* ⟹ *degree p* ≤ *n*
unfolding *degree-def* **by** (*erule Least-le*)

lemma *less-degree-imp*: *n* < *degree p* ⟹ ∃ *i*>*n*. *coeff p i* ≠ *0*
unfolding *degree-def* **by** (*drule not-less-Least*, *simp*)

14.4 The zero polynomial

instantiation *poly* :: (*zero*) *zero*
begin

lift-definition *zero-poly* :: *'a poly*
is λ-. *0* **by** (*rule MOST-I*) *simp*

instance ..

end

lemma *coeff-0* [*simp*]:
coeff 0 n = *0*
by *transfer rule*

lemma *degree-0* [*simp*]:
degree 0 = *0*
by (*rule order-antisym* [*OF degree-le le0*]) *simp*

lemma *leading-coeff-neq-0*:
assumes *p* ≠ *0*
shows *coeff p* (*degree p*) ≠ *0*
proof (*cases degree p*)
case *0*
from ‹*p* ≠ *0*› **have** ∃ *n*. *coeff p n* ≠ *0*
by (*simp add*: *poly-eq-iff*)
then obtain *n* **where** *coeff p n* ≠ *0* ..
hence *n* ≤ *degree p* **by** (*rule le-degree*)
with ‹*coeff p n* ≠ *0*› **and** ‹*degree p* = *0*›
show *coeff p* (*degree p*) ≠ *0* **by** *simp*
next
case (*Suc n*)
from ‹*degree p* = *Suc n*› **have** *n* < *degree p* **by** *simp*

hence $\exists\, i>n.$ *coeff p i* $\neq$ *0* **by** (*rule less-degree-imp*)
then obtain *i* **where** $n < i$ **and** *coeff p i* $\neq$ *0* **by** *fast*
from ‹*degree p* = *Suc n*› **and** ‹$n < i$› **have** *degree p* $\leq$ *i* **by** *simp*
also from ‹*coeff p i* $\neq$ *0*› **have** *i* $\leq$ *degree p* **by** (*rule le-degree*)
finally have *degree p* = *i* **.**
with ‹*coeff p i* $\neq$ *0*› **show** *coeff p* (*degree p*) $\neq$ *0* **by** *simp*
qed

lemma *leading-coeff-0-iff* [*simp*]:
coeff p (*degree p*) = *0* $\longleftrightarrow$ *p* = *0*
by (*cases p* = *0*, *simp*, *simp add*: *leading-coeff-neq-0*)

14.5 List-style constructor for polynomials

lift-definition *pCons* :: *'a::zero* $\Rightarrow$ *'a poly* $\Rightarrow$ *'a poly*
is $\lambda a\ p.$ *case-nat a* (*coeff p*)
by (*rule MOST-SucD*) (*simp add*: *MOST-coeff-eq-0*)

lemmas *coeff-pCons* = *pCons.rep-eq*

lemma *coeff-pCons-0* [*simp*]:
coeff (*pCons a p*) *0* = *a*
by *transfer simp*

lemma *coeff-pCons-Suc* [*simp*]:
coeff (*pCons a p*) (*Suc n*) = *coeff p n*
by (*simp add*: *coeff-pCons*)

lemma *degree-pCons-le*:
degree (*pCons a p*) $\leq$ *Suc* (*degree p*)
by (*rule degree-le*) (*simp add*: *coeff-eq-0 coeff-pCons split*: *nat.split*)

lemma *degree-pCons-eq*:
p $\neq$ *0* $\Longrightarrow$ *degree* (*pCons a p*) = *Suc* (*degree p*)
apply (*rule order-antisym* [*OF degree-pCons-le*])
apply (*rule le-degree*, *simp*)
done

lemma *degree-pCons-0*:
degree (*pCons a 0*) = *0*
apply (*rule order-antisym* [*OF - le0*])
apply (*rule degree-le*, *simp add*: *coeff-pCons split*: *nat.split*)
done

lemma *degree-pCons-eq-if* [*simp*]:
degree (*pCons a p*) = (*if p* = *0 then 0 else Suc* (*degree p*))
apply (*cases p* = *0*, *simp-all*)
apply (*rule order-antisym* [*OF - le0*])
apply (*rule degree-le*, *simp add*: *coeff-pCons split*: *nat.split*)

apply (*rule order-antisym* [*OF degree-pCons-le*])
apply (*rule le-degree*, *simp*)
done

lemma *pCons-0-0* [*simp*]:
pCons 0 0 = *0*
by (*rule poly-eqI*) (*simp add*: *coeff-pCons split*: *nat.split*)

lemma *pCons-eq-iff* [*simp*]:
pCons a p = *pCons b q* ⟷ *a* = *b* ∧ *p* = *q*
proof *safe*
assume *pCons a p* = *pCons b q*
then have *coeff* (*pCons a p*) *0* = *coeff* (*pCons b q*) *0* **by** *simp*
then show *a* = *b* **by** *simp*
next
assume *pCons a p* = *pCons b q*
then have ∀ *n*. *coeff* (*pCons a p*) (*Suc n*) =
coeff (*pCons b q*) (*Suc n*) **by** *simp*
then show *p* = *q* **by** (*simp add*: *poly-eq-iff*)
qed

lemma *pCons-eq-0-iff* [*simp*]:
pCons a p = *0* ⟷ *a* = *0* ∧ *p* = *0*
using *pCons-eq-iff* [*of a p 0 0*] **by** *simp*

lemma *pCons-cases* [*cases type*: *poly*]:
obtains (*pCons*) *a q* **where** *p* = *pCons a q*
proof
show *p* = *pCons* (*coeff p 0*) (*Abs-poly* (λ*n*. *coeff p* (*Suc n*)))
by *transfer*
(*simp-all add*: *MOST-inj*[**where** *f*=*Suc* **and** *P*=λ*n*. *p n* = *0* **for** *p*] *fun-eq-iff*
Abs-poly-inverse
split: *nat.split*)
qed

lemma *pCons-induct* [*case-names 0 pCons*, *induct type*: *poly*]:
assumes *zero*: *P 0*
assumes *pCons*: ⋀*a p*. *a* ≠ *0* ∨ *p* ≠ *0* ⟹ *P p* ⟹ *P* (*pCons a p*)
shows *P p*
proof (*induct p rule*: *measure-induct-rule* [**where** *f*=*degree*])
case (*less p*)
obtain *a q* **where** *p* = *pCons a q* **by** (*rule pCons-cases*)
have *P q*
proof (*cases q* = *0*)
case *True*
then show *P q* **by** (*simp add*: *zero*)
next
case *False*
then have *degree* (*pCons a q*) = *Suc* (*degree q*)

```
      by (rule degree-pCons-eq)
    then have degree q < degree p
      using ‹p = pCons a q› by simp
    then show P q
      by (rule less.hyps)
  qed
  have P (pCons a q)
  proof (cases a ≠ 0 ∨ q ≠ 0)
    case True
    with ‹P q› show ?thesis by (auto intro: pCons)
  next
    case False
    with zero show ?thesis by simp
  qed
  then show ?case
    using ‹p = pCons a q› by simp
qed

lemma degree-eq-zeroE:
  fixes p :: 'a::zero poly
  assumes degree p = 0
  obtains a where p = pCons a 0
proof -
  obtain a q where p: p = pCons a q by (cases p)
  with assms have q = 0 by (cases q = 0) simp-all
  with p have p = pCons a 0 by simp
  with that show thesis .
qed
```

14.6 Quickcheck generator for polynomials

```
quickcheck-generator poly constructors: 0 :: - poly, pCons
```

14.7 List-style syntax for polynomials

```
syntax
  -poly :: args ⇒ 'a poly  ([:(-):])

translations
  [:x, xs:] == CONST pCons x [:xs:]
  [:x:] == CONST pCons x 0
  [:x:] <= CONST pCons x (-constrain 0 t)
```

14.8 Representation of polynomials by lists of coefficients

```
primrec Poly :: 'a::zero list ⇒ 'a poly
where
  [code-post]: Poly [] = 0
| [code-post]: Poly (a # as) = pCons a (Poly as)
```

lemma *Poly-replicate-0* [*simp*]:
Poly (*replicate n 0*) = *0*
by (*induct n*) *simp-all*

lemma *Poly-eq-0*:
Poly as = *0* ⟷ (∃ *n*. *as* = *replicate n 0*)
by (*induct as*) (*auto simp add*: *Cons-replicate-eq*)

lemma *degree-Poly*: *degree* (*Poly xs*) ≤ *length xs*
by (*induction xs*) *simp-all*

definition *coeffs* :: ′*a poly* ⇒ ′*a*::*zero list*
where
coeffs p = (*if p* = *0 then* [] *else map* (λ*i*. *coeff p i*) [*0* ..< *Suc* (*degree p*)])

lemma *coeffs-eq-Nil* [*simp*]:
coeffs p = [] ⟷ *p* = *0*
by (*simp add*: *coeffs-def*)

lemma *not-0-coeffs-not-Nil*:
p ≠ *0* ⟹ *coeffs p* ≠ []
by *simp*

lemma *coeffs-0-eq-Nil* [*simp*]:
coeffs 0 = []
by *simp*

lemma *coeffs-pCons-eq-cCons* [*simp*]:
coeffs (*pCons a p*) = *a* ## *coeffs p*
proof −
{ **fix** *ms* :: *nat list* **and** *f* :: *nat* ⇒ ′*a* **and** *x* :: ′*a*
assume ∀ *m*∈*set ms*. *m* > *0*
then have *map* (*case-nat x f*) *ms* = *map f* (*map* (λ*n*. *n* − *1*) *ms*)
by (*induct ms*) (*auto split*: *nat.split*)
}
note ∗ = *this*
show *?thesis*
by (*simp add*: *coeffs-def* ∗ *upt-conv-Cons coeff-pCons map-decr-upt del*: *upt-Suc*)
qed

lemma *length-coeffs*: *p* ≠ *0* ⟹ *length* (*coeffs p*) = *degree p* + *1*
by (*simp add*: *coeffs-def*)

lemma *coeffs-nth*:
assumes *p* ≠ *0 n* ≤ *degree p*
shows *coeffs p* ! *n* = *coeff p n*
using *assms* **unfolding** *coeffs-def* **by** (*auto simp del*: *upt-Suc*)

lemma *not-0-cCons-eq* [*simp*]:

p ≠ *0* ⟹ *a* ## *coeffs p* = *a* # *coeffs p*
by (*simp add*: *cCons-def*)

lemma *Poly-coeffs* [*simp*, *code abstype*]:
Poly (*coeffs p*) = *p*
by (*induct p*) *auto*

lemma *coeffs-Poly* [*simp*]:
coeffs (*Poly as*) = *strip-while* (*HOL.eq 0*) *as*
proof (*induct as*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons a as*)
have (∀ *n*. *as* ≠ *replicate n 0*) ⟷ (∃ *a*∈*set as*. *a* ≠ *0*)
using *replicate-length-same* [*of as 0*] **by** (*auto dest*: *sym* [*of - as*])
with *Cons* **show** *?case* **by** *auto*
qed

lemma *last-coeffs-not-0*:
p ≠ *0* ⟹ *last* (*coeffs p*) ≠ *0*
by (*induct p*) (*auto simp add*: *cCons-def*)

lemma *strip-while-coeffs* [*simp*]:
strip-while (*HOL.eq 0*) (*coeffs p*) = *coeffs p*
by (*cases p* = *0*) (*auto dest*: *last-coeffs-not-0 intro*: *strip-while-not-last*)

lemma *coeffs-eq-iff*:
p = *q* ⟷ *coeffs p* = *coeffs q* (**is** *?P* ⟷ *?Q*)
proof
assume *?P* **then show** *?Q* **by** *simp*
next
assume *?Q*
then have *Poly* (*coeffs p*) = *Poly* (*coeffs q*) **by** *simp*
then show *?P* **by** *simp*
qed

lemma *coeff-Poly-eq*:
coeff (*Poly xs*) *n* = *nth-default 0 xs n*
apply (*induct xs arbitrary*: *n*) **apply** *simp-all*
by (*metis nat.case not0-implies-Suc nth-default-Cons-0 nth-default-Cons-Suc pCons.rep-eq*)

lemma *nth-default-coeffs-eq*:
nth-default 0 (*coeffs p*) = *coeff p*
by (*simp add*: *fun-eq-iff coeff-Poly-eq* [*symmetric*])

lemma [*code*]:
coeff p = *nth-default 0* (*coeffs p*)
by (*simp add*: *nth-default-coeffs-eq*)

lemma *coeffs-eqI*:
  assumes *coeff*: $\bigwedge n$. *coeff p n* = *nth-default 0 xs n*
  assumes *zero*: *xs* ≠ [] ⟹ *last xs* ≠ *0*
  shows *coeffs p* = *xs*
proof −
  from *coeff* **have** *p* = *Poly xs* **by** (*simp add*: *poly-eq-iff coeff-Poly-eq*)
  with *zero* **show** *?thesis* **by** *simp* (*cases xs*, *simp-all*)
qed

lemma *degree-eq-length-coeffs* [*code*]:
  degree p = *length* (*coeffs p*) − *1*
  by (*simp add*: *coeffs-def*)

lemma *length-coeffs-degree*:
  p ≠ *0* ⟹ *length* (*coeffs p*) = *Suc* (*degree p*)
  by (*induct p*) (*auto simp add*: *cCons-def*)

lemma [*code abstract*]:
  coeffs 0 = []
  by (*fact coeffs-0-eq-Nil*)

lemma [*code abstract*]:
  coeffs (*pCons a p*) = *a* ## *coeffs p*
  by (*fact coeffs-pCons-eq-cCons*)

instantiation *poly* :: ({*zero*, *equal*}) *equal*
begin

definition
  [*code*]: *HOL.equal* (*p*::′*a poly*) *q* ⟷ *HOL.equal* (*coeffs p*) (*coeffs q*)

instance
  by *standard* (*simp add*: *equal equal-poly-def coeffs-eq-iff*)

end

lemma [*code nbe*]: *HOL.equal* (*p* :: *- poly*) *p* ⟷ *True*
  by (*fact equal-refl*)

definition *is-zero* :: ′*a*::*zero poly* ⇒ *bool*
where
  [*code*]: *is-zero p* ⟷ *List.null* (*coeffs p*)

lemma *is-zero-null* [*code-abbrev*]:
  is-zero p ⟷ *p* = *0*
  by (*simp add*: *is-zero-def null-def*)

14.9 Fold combinator for polynomials

definition *fold-coeffs* :: ('*a*::*zero* ⇒ '*b* ⇒ '*b*) ⇒ '*a poly* ⇒ '*b* ⇒ '*b*
where
fold-coeffs f p = *foldr f* (*coeffs p*)

lemma *fold-coeffs-0-eq* [*simp*]:
fold-coeffs f 0 = *id*
by (*simp add*: *fold-coeffs-def*)

lemma *fold-coeffs-pCons-eq* [*simp*]:
f 0 = *id* ⟹ *fold-coeffs f* (*pCons a p*) = *f a* ∘ *fold-coeffs f p*
by (*simp add*: *fold-coeffs-def cCons-def fun-eq-iff*)

lemma *fold-coeffs-pCons-0-0-eq* [*simp*]:
fold-coeffs f (*pCons 0 0*) = *id*
by (*simp add*: *fold-coeffs-def*)

lemma *fold-coeffs-pCons-coeff-not-0-eq* [*simp*]:
a ≠ *0* ⟹ *fold-coeffs f* (*pCons a p*) = *f a* ∘ *fold-coeffs f p*
by (*simp add*: *fold-coeffs-def*)

lemma *fold-coeffs-pCons-not-0-0-eq* [*simp*]:
p ≠ *0* ⟹ *fold-coeffs f* (*pCons a p*) = *f a* ∘ *fold-coeffs f p*
by (*simp add*: *fold-coeffs-def*)

14.10 Canonical morphism on polynomials – evaluation

definition *poly* :: '*a*::*comm-semiring-0 poly* ⇒ '*a* ⇒ '*a*
where
poly p = *fold-coeffs* (λ*a f x*. *a* + *x* ∗ *f x*) *p* (λ*x*. *0*) — The Horner Schema

lemma *poly-0* [*simp*]:
poly 0 x = *0*
by (*simp add*: *poly-def*)

lemma *poly-pCons* [*simp*]:
poly (*pCons a p*) *x* = *a* + *x* ∗ *poly p x*
by (*cases p* = *0* ∧ *a* = *0*) (*auto simp add*: *poly-def*)

lemma *poly-altdef*:
poly p (*x* :: '*a* :: {*comm-semiring-0*, *semiring-1*}) = (∑ *i*≤*degree p*. *coeff p i* ∗ *x*
^ *i*)
proof (*induction p rule*: *pCons-induct*)
case (*pCons a p*)
show *?case*
proof (*cases p* = *0*)
case *False*
let *?p'* = *pCons a p*
note *poly-pCons*[*of a p x*]

also note *pCons.IH*
also have $a + x * (\sum i \leq degree\ p.\ coeff\ p\ i * x\ \hat{}\ i) =$
$coeff\ ?p'\ 0 * x\hat{}0 + (\sum i \leq degree\ p.\ coeff\ ?p'\ (Suc\ i) * x\hat{}Suc\ i)$
by (*simp add*: *field-simps setsum-right-distrib coeff-pCons*)
also note *setsum-atMost-Suc-shift*[*symmetric*]
also note *degree-pCons-eq*[*OF* ‹$p \neq 0$›, *of a*, *symmetric*]
finally show *?thesis* **.**
qed *simp*
qed *simp*

lemma *poly-0-coeff-0*: *poly p 0 = coeff p 0*
by (*cases p*) (*auto simp*: *poly-altdef*)

14.11 Monomials

lift-definition *monom* :: $'a \Rightarrow nat \Rightarrow 'a::zero\ poly$
is $\lambda a\ m\ n.\ if\ m = n\ then\ a\ else\ 0$
by (*simp add*: *MOST-iff-cofinite*)

lemma *coeff-monom* [*simp*]:
coeff (*monom a m*) *n* = (*if m = n then a else 0*)
by *transfer rule*

lemma *monom-0*:
monom a 0 = pCons a 0
by (*rule poly-eqI*) (*simp add*: *coeff-pCons split*: *nat.split*)

lemma *monom-Suc*:
monom a (*Suc n*) = *pCons 0* (*monom a n*)
by (*rule poly-eqI*) (*simp add*: *coeff-pCons split*: *nat.split*)

lemma *monom-eq-0* [*simp*]: *monom 0 n = 0*
by (*rule poly-eqI*) *simp*

lemma *monom-eq-0-iff* [*simp*]: $monom\ a\ n = 0 \longleftrightarrow a = 0$
by (*simp add*: *poly-eq-iff*)

lemma *monom-eq-iff* [*simp*]: $monom\ a\ n = monom\ b\ n \longleftrightarrow a = b$
by (*simp add*: *poly-eq-iff*)

lemma *degree-monom-le*: $degree\ (monom\ a\ n) \leq n$
by (*rule degree-le*, *simp*)

lemma *degree-monom-eq*: $a \neq 0 \Longrightarrow degree\ (monom\ a\ n) = n$
apply (*rule order-antisym* [*OF degree-monom-le*])
apply (*rule le-degree*, *simp*)
done

lemma *coeffs-monom* [*code abstract*]:

coeffs (*monom a n*) = (*if a* = *0 then* [] *else replicate n 0* @ [*a*])
by (*induct n*) (*simp-all add*: *monom-0 monom-Suc*)

lemma *fold-coeffs-monom* [*simp*]:
$a \neq 0 \Longrightarrow$ *fold-coeffs f* (*monom a n*) = *f 0* ^^ *n* ∘ *f a*
by (*simp add*: *fold-coeffs-def coeffs-monom fun-eq-iff*)

lemma *poly-monom*:
fixes *a x* :: *'a*::{*comm-semiring-1*}
shows *poly* (*monom a n*) *x* = *a* ∗ *x* ^ *n*
by (*cases a* = *0*, *simp-all*)
(*induct n*, *simp-all add*: *mult.left-commute poly-def*)

14.12 Addition and subtraction

instantiation *poly* :: (*comm-monoid-add*) *comm-monoid-add*
begin

lift-definition *plus-poly* :: *'a poly* ⇒ *'a poly* ⇒ *'a poly*
is *λp q n. coeff p n* + *coeff q n*
proof −
fix *q p* :: *'a poly*
show $\forall_\infty n$. *coeff p n* + *coeff q n* = *0*
using *MOST-coeff-eq-0*[*of p*] *MOST-coeff-eq-0*[*of q*] **by** *eventually-elim simp*
qed

lemma *coeff-add* [*simp*]: *coeff* (*p* + *q*) *n* = *coeff p n* + *coeff q n*
by (*simp add*: *plus-poly.rep-eq*)

instance
proof
fix *p q r* :: *'a poly*
show (*p* + *q*) + *r* = *p* + (*q* + *r*)
by (*simp add*: *poly-eq-iff add.assoc*)
show *p* + *q* = *q* + *p*
by (*simp add*: *poly-eq-iff add.commute*)
show *0* + *p* = *p*
by (*simp add*: *poly-eq-iff*)
qed

end

instantiation *poly* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add*
begin

lift-definition *minus-poly* :: *'a poly* ⇒ *'a poly* ⇒ *'a poly*
is *λp q n. coeff p n* − *coeff q n*
proof −
fix *q p* :: *'a poly*

show $\forall_\infty n.$ *coeff p n* − *coeff q n* = *0*
using *MOST-coeff-eq-0*[*of p*] *MOST-coeff-eq-0*[*of q*] **by** *eventually-elim simp*
qed

lemma *coeff-diff* [*simp*]: *coeff* (*p* − *q*) *n* = *coeff p n* − *coeff q n*
by (*simp add*: *minus-poly.rep-eq*)

instance
proof
fix *p q r* :: *'a poly*
show *p* + *q* − *p* = *q*
by (*simp add*: *poly-eq-iff*)
show *p* − *q* − *r* = *p* − (*q* + *r*)
by (*simp add*: *poly-eq-iff diff-diff-eq*)
qed

end

instantiation *poly* :: (*ab-group-add*) *ab-group-add*
begin

lift-definition *uminus-poly* :: *'a poly* ⇒ *'a poly*
is λ*p n*. − *coeff p n*
proof −
fix *p* :: *'a poly*
show $\forall_\infty n.$ − *coeff p n* = *0*
using *MOST-coeff-eq-0* **by** *simp*
qed

lemma *coeff-minus* [*simp*]: *coeff* (− *p*) *n* = − *coeff p n*
by (*simp add*: *uminus-poly.rep-eq*)

instance
proof
fix *p q* :: *'a poly*
show − *p* + *p* = *0*
by (*simp add*: *poly-eq-iff*)
show *p* − *q* = *p* + − *q*
by (*simp add*: *poly-eq-iff*)
qed

end

lemma *add-pCons* [*simp*]:
pCons a p + *pCons b q* = *pCons* (*a* + *b*) (*p* + *q*)
by (*rule poly-eqI*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *minus-pCons* [*simp*]:
− *pCons a p* = *pCons* (− *a*) (− *p*)

by (*rule poly-eqI*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *diff-pCons* [*simp*]:
pCons a p − pCons b q = pCons (*a − b*) (*p − q*)
by (*rule poly-eqI*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *degree-add-le-max*: *degree* (*p + q*) ≤ *max* (*degree p*) (*degree q*)
by (*rule degree-le*, *auto simp add*: *coeff-eq-0*)

lemma *degree-add-le*:
⟦*degree p ≤ n*; *degree q ≤ n*⟧ ⟹ *degree* (*p + q*) ≤ *n*
by (*auto intro*: *order-trans degree-add-le-max*)

lemma *degree-add-less*:
⟦*degree p < n*; *degree q < n*⟧ ⟹ *degree* (*p + q*) < *n*
by (*auto intro*: *le-less-trans degree-add-le-max*)

lemma *degree-add-eq-right*:
degree p < degree q ⟹ *degree* (*p + q*) = *degree q*
apply (*cases q = 0*, *simp*)
apply (*rule order-antisym*)
apply (*simp add*: *degree-add-le*)
apply (*rule le-degree*)
apply (*simp add*: *coeff-eq-0*)
done

lemma *degree-add-eq-left*:
degree q < degree p ⟹ *degree* (*p + q*) = *degree p*
using *degree-add-eq-right* [*of q p*]
by (*simp add*: *add.commute*)

lemma *degree-minus* [*simp*]:
degree (− *p*) = *degree p*
unfolding *degree-def* **by** *simp*

lemma *degree-diff-le-max*:
fixes *p q* :: *'a* :: *ab-group-add poly*
shows *degree* (*p − q*) ≤ *max* (*degree p*) (*degree q*)
using *degree-add-le* [**where** *p=p* **and** *q=−q*]
by *simp*

lemma *degree-diff-le*:
fixes *p q* :: *'a* :: *ab-group-add poly*
assumes *degree p ≤ n* **and** *degree q ≤ n*
shows *degree* (*p − q*) ≤ *n*
using *assms degree-add-le* [*of p n − q*] **by** *simp*

lemma *degree-diff-less*:
fixes *p q* :: *'a* :: *ab-group-add poly*

```
  assumes degree p < n and degree q < n
  shows degree (p − q) < n
  using assms degree-add-less [of p n − q] by simp

lemma add-monom: monom a n + monom b n = monom (a + b) n
  by (rule poly-eqI) simp

lemma diff-monom: monom a n − monom b n = monom (a − b) n
  by (rule poly-eqI) simp

lemma minus-monom: − monom a n = monom (−a) n
  by (rule poly-eqI) simp

lemma coeff-setsum: coeff (∑x∈A. p x) i = (∑x∈A. coeff (p x) i)
  by (cases finite A, induct set: finite, simp-all)

lemma monom-setsum: monom (∑x∈A. a x) n = (∑x∈A. monom (a x) n)
  by (rule poly-eqI) (simp add: coeff-setsum)

fun plus-coeffs :: 'a::comm-monoid-add list ⇒ 'a list ⇒ 'a list
where
  plus-coeffs xs [] = xs
| plus-coeffs [] ys = ys
| plus-coeffs (x # xs) (y # ys) = (x + y) ## plus-coeffs xs ys

lemma coeffs-plus-eq-plus-coeffs [code abstract]:
  coeffs (p + q) = plus-coeffs (coeffs p) (coeffs q)
proof −
  { fix xs ys :: 'a list and n
    have nth-default 0 (plus-coeffs xs ys) n = nth-default 0 xs n + nth-default 0 ys
n
    proof (induct xs ys arbitrary: n rule: plus-coeffs.induct)
      case (3 x xs y ys n)
      then show ?case by (cases n) (auto simp add: cCons-def)
    qed simp-all }
  note ∗ = this
  { fix xs ys :: 'a list
    assume xs ≠ [] ⟹ last xs ≠ 0 and ys ≠ [] ⟹ last ys ≠ 0
    moreover assume plus-coeffs xs ys ≠ []
    ultimately have last (plus-coeffs xs ys) ≠ 0
    proof (induct xs ys rule: plus-coeffs.induct)
      case (3 x xs y ys) then show ?case by (auto simp add: cCons-def) metis
    qed simp-all }
  note ∗∗ = this
  show ?thesis
    apply (rule coeffs-eqI)
    apply (simp add: ∗ nth-default-coeffs-eq)
    apply (rule ∗∗)
    apply (auto dest: last-coeffs-not-0)
```

done
qed

lemma *coeffs-uminus* [*code abstract*]:
coeffs (− *p*) = *map* (λ*a*. − *a*) (*coeffs p*)
by (*rule coeffs-eqI*)
(*simp-all add*: *not-0-coeffs-not-Nil last-map last-coeffs-not-0 nth-default-map-eq nth-default-coeffs-eq*)

lemma [*code*]:
fixes *p q* :: ′*a*::*ab-group-add poly*
shows *p* − *q* = *p* + − *q*
by (*fact diff-conv-add-uminus*)

lemma *poly-add* [*simp*]: *poly* (*p* + *q*) *x* = *poly p x* + *poly q x*
apply (*induct p arbitrary*: *q*, *simp*)
apply (*case-tac q*, *simp*, *simp add*: *algebra-simps*)
done

lemma *poly-minus* [*simp*]:
fixes *x* :: ′*a*::*comm-ring*
shows *poly* (− *p*) *x* = − *poly p x*
by (*induct p*) *simp-all*

lemma *poly-diff* [*simp*]:
fixes *x* :: ′*a*::*comm-ring*
shows *poly* (*p* − *q*) *x* = *poly p x* − *poly q x*
using *poly-add* [*of p* − *q x*] **by** *simp*

lemma *poly-setsum*: *poly* ($\sum$ *k*∈*A*. *p k*) *x* = ($\sum$ *k*∈*A*. *poly* (*p k*) *x*)
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *degree-setsum-le*: *finite S* ⟹ ($\bigwedge$ *p* . *p* ∈ *S* ⟹ *degree* (*f p*) ≤ *n*)
⟹ *degree* (*setsum f S*) ≤ *n*
proof (*induct S rule*: *finite-induct*)
case (*insert p S*)
hence *degree* (*setsum f S*) ≤ *n degree* (*f p*) ≤ *n* **by** *auto*
thus *?case* **unfolding** *setsum.insert*[*OF insert*(*1*−*2*)] **by** (*metis degree-add-le*)
qed *simp*

lemma *poly-as-sum-of-monoms*′:
assumes *n*: *degree p* ≤ *n*
shows ($\sum$ *i*≤*n*. *monom* (*coeff p i*) *i*) = *p*
proof −
have *eq*: $\bigwedge$*i*. {..*n*} ∩ {*i*} = (*if i* ≤ *n then* {*i*} *else* {})
by *auto*
show *?thesis*
using *n* **by** (*simp add*: *poly-eq-iff coeff-setsum coeff-eq-0 setsum.If-cases eq if-distrib*[**where** *f*=λ*x*. *x* ∗ *a* **for** *a*])

qed

lemma *poly-as-sum-of-monoms*: $(\sum i \le degree\ p.\ monom\ (coeff\ p\ i)\ i) = p$
by (*intro poly-as-sum-of-monoms' order-refl*)

lemma *Poly-snoc*: *Poly* (*xs* @ [*x*]) = *Poly xs* + *monom x* (*length xs*)
by (*induction xs*) (*simp-all add*: *monom-0 monom-Suc*)

14.13 Multiplication by a constant, polynomial multiplication and the unit polynomial

lift-definition *smult* :: *'a::comm-semiring-0* ⇒ *'a poly* ⇒ *'a poly*
is $\lambda a\ p\ n.\ a * coeff\ p\ n$
proof −
fix *a* :: *'a* **and** *p* :: *'a poly* **show** $\forall_\infty\ i.\ a * coeff\ p\ i = 0$
using *MOST-coeff-eq-0*[*of p*] **by** *eventually-elim simp*
qed

lemma *coeff-smult* [*simp*]:
coeff (*smult a p*) *n* = *a* ∗ *coeff p n*
by (*simp add*: *smult.rep-eq*)

lemma *degree-smult-le*: *degree* (*smult a p*) ≤ *degree p*
by (*rule degree-le*, *simp add*: *coeff-eq-0*)

lemma *smult-smult* [*simp*]: *smult a* (*smult b p*) = *smult* (*a* ∗ *b*) *p*
by (*rule poly-eqI*, *simp add*: *mult.assoc*)

lemma *smult-0-right* [*simp*]: *smult a 0* = *0*
by (*rule poly-eqI*, *simp*)

lemma *smult-0-left* [*simp*]: *smult 0 p* = *0*
by (*rule poly-eqI*, *simp*)

lemma *smult-1-left* [*simp*]: *smult* (*1::'a::comm-semiring-1*) *p* = *p*
by (*rule poly-eqI*, *simp*)

lemma *smult-add-right*:
smult a (*p* + *q*) = *smult a p* + *smult a q*
by (*rule poly-eqI*, *simp add*: *algebra-simps*)

lemma *smult-add-left*:
smult (*a* + *b*) *p* = *smult a p* + *smult b p*
by (*rule poly-eqI*, *simp add*: *algebra-simps*)

lemma *smult-minus-right* [*simp*]:
smult (*a::'a::comm-ring*) (− *p*) = − *smult a p*
by (*rule poly-eqI*, *simp*)

lemma *smult-minus-left* [*simp*]:
smult (− *a*::′*a*::*comm-ring*) *p* = − *smult a p*
by (*rule poly-eqI*, *simp*)

lemma *smult-diff-right*:
smult (*a*::′*a*::*comm-ring*) (*p* − *q*) = *smult a p* − *smult a q*
by (*rule poly-eqI*, *simp add*: *algebra-simps*)

lemma *smult-diff-left*:
smult (*a* − *b*::′*a*::*comm-ring*) *p* = *smult a p* − *smult b p*
by (*rule poly-eqI*, *simp add*: *algebra-simps*)

lemmas *smult-distribs* =
smult-add-left smult-add-right
smult-diff-left smult-diff-right

lemma *smult-pCons* [*simp*]:
smult a (*pCons b p*) = *pCons* (*a* ∗ *b*) (*smult a p*)
by (*rule poly-eqI*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *smult-monom*: *smult a* (*monom b n*) = *monom* (*a* ∗ *b*) *n*
by (*induct n*, *simp add*: *monom-0*, *simp add*: *monom-Suc*)

lemma *degree-smult-eq* [*simp*]:
fixes *a* :: ′*a*::*idom*
shows *degree* (*smult a p*) = (*if a* = *0 then 0 else degree p*)
by (*cases a* = *0*, *simp*, *simp add*: *degree-def*)

lemma *smult-eq-0-iff* [*simp*]:
fixes *a* :: ′*a*::*idom*
shows *smult a p* = *0* ⟷ *a* = *0* ∨ *p* = *0*
by (*simp add*: *poly-eq-iff*)

lemma *coeffs-smult* [*code abstract*]:
fixes *p* :: ′*a*::*idom poly*
shows *coeffs* (*smult a p*) = (*if a* = *0 then* [] *else map* (*Groups.times a*) (*coeffs p*))
by (*rule coeffs-eqI*)
(*auto simp add*: *not-0-coeffs-not-Nil last-map last-coeffs-not-0 nth-default-map-eq nth-default-coeffs-eq*)

instantiation *poly* :: (*comm-semiring-0*) *comm-semiring-0*
begin

definition
p ∗ *q* = *fold-coeffs* (λ*a p*. *smult a q* + *pCons 0 p*) *p 0*

lemma *mult-poly-0-left*: (*0*::′*a poly*) ∗ *q* = *0*
by (*simp add*: *times-poly-def*)

lemma *mult-pCons-left* [*simp*]:
pCons a p ∗ *q* = *smult a q* + *pCons 0* (*p* ∗ *q*)
by (*cases p* = *0* ∧ *a* = *0*) (*auto simp add*: *times-poly-def*)

lemma *mult-poly-0-right*: *p* ∗ (*0*::′*a poly*) = *0*
by (*induct p*) (*simp add*: *mult-poly-0-left*, *simp*)

lemma *mult-pCons-right* [*simp*]:
p ∗ *pCons a q* = *smult a p* + *pCons 0* (*p* ∗ *q*)
by (*induct p*) (*simp add*: *mult-poly-0-left*, *simp add*: *algebra-simps*)

lemmas *mult-poly-0* = *mult-poly-0-left mult-poly-0-right*

lemma *mult-smult-left* [*simp*]:
smult a p ∗ *q* = *smult a* (*p* ∗ *q*)
by (*induct p*) (*simp add*: *mult-poly-0*, *simp add*: *smult-add-right*)

lemma *mult-smult-right* [*simp*]:
p ∗ *smult a q* = *smult a* (*p* ∗ *q*)
by (*induct q*) (*simp add*: *mult-poly-0*, *simp add*: *smult-add-right*)

lemma *mult-poly-add-left*:
fixes *p q r* :: ′*a poly*
shows (*p* + *q*) ∗ *r* = *p* ∗ *r* + *q* ∗ *r*
by (*induct r*) (*simp add*: *mult-poly-0*, *simp add*: *smult-distribs algebra-simps*)

instance
proof
fix *p q r* :: ′*a poly*
show *0*: *0* ∗ *p* = *0*
by (*rule mult-poly-0-left*)
show *p* ∗ *0* = *0*
by (*rule mult-poly-0-right*)
show (*p* + *q*) ∗ *r* = *p* ∗ *r* + *q* ∗ *r*
by (*rule mult-poly-add-left*)
show (*p* ∗ *q*) ∗ *r* = *p* ∗ (*q* ∗ *r*)
by (*induct p*, *simp add*: *mult-poly-0*, *simp add*: *mult-poly-add-left*)
show *p* ∗ *q* = *q* ∗ *p*
by (*induct p*, *simp add*: *mult-poly-0*, *simp*)
qed

end

instance *poly* :: (*comm-semiring-0-cancel*) *comm-semiring-0-cancel* **..**

lemma *coeff-mult*:
coeff (*p* ∗ *q*) *n* = (∑ *i*≤*n*. *coeff p i* ∗ *coeff q* (*n*−*i*))
proof (*induct p arbitrary*: *n*)

case *0* **show** *?case* **by** *simp*
next
case (*pCons a p n*) **thus** *?case*
by (*cases n*, *simp*, *simp add*: *setsum-atMost-Suc-shift*
del: *setsum-atMost-Suc*)
qed

lemma *degree-mult-le*: *degree* (*p* ∗ *q*) ≤ *degree p* + *degree q*
apply (*rule degree-le*)
apply (*induct p*)
apply *simp*
apply (*simp add*: *coeff-eq-0 coeff-pCons split*: *nat.split*)
done

lemma *mult-monom*: *monom a m* ∗ *monom b n* = *monom* (*a* ∗ *b*) (*m* + *n*)
by (*induct m*) (*simp add*: *monom-0 smult-monom*, *simp add*: *monom-Suc*)

instantiation *poly* :: (*comm-semiring-1*) *comm-semiring-1*
begin

definition *one-poly-def*: *1* = *pCons 1 0*

instance
proof
show *1* ∗ *p* = *p* **for** *p* :: *'a poly*
unfolding *one-poly-def* **by** *simp*
show *0* ≠ (*1*::*'a poly*)
unfolding *one-poly-def* **by** *simp*
qed

end

instance *poly* :: (*comm-ring*) *comm-ring* **..**

instance *poly* :: (*comm-ring-1*) *comm-ring-1* **..**

lemma *coeff-1* [*simp*]: *coeff 1 n* = (*if n* = *0 then 1 else 0*)
unfolding *one-poly-def*
by (*simp add*: *coeff-pCons split*: *nat.split*)

lemma *monom-eq-1* [*simp*]:
monom 1 0 = *1*
by (*simp add*: *monom-0 one-poly-def*)

lemma *degree-1* [*simp*]: *degree 1* = *0*
unfolding *one-poly-def*
by (*rule degree-pCons-0*)

lemma *coeffs-1-eq* [*simp*, *code abstract*]:

coeffs 1 = [*1*]
by (*simp add*: *one-poly-def*)

lemma *degree-power-le*:
degree (*p* ^ *n*) ≤ *degree p* ∗ *n*
by (*induct n*) (*auto intro*: *order-trans degree-mult-le*)

lemma *poly-smult* [*simp*]:
poly (*smult a p*) *x* = *a* ∗ *poly p x*
by (*induct p*, *simp*, *simp add*: *algebra-simps*)

lemma *poly-mult* [*simp*]:
poly (*p* ∗ *q*) *x* = *poly p x* ∗ *poly q x*
by (*induct p*, *simp-all*, *simp add*: *algebra-simps*)

lemma *poly-1* [*simp*]:
poly 1 x = *1*
by (*simp add*: *one-poly-def*)

lemma *poly-power* [*simp*]:
fixes *p* :: *'a*::{*comm-semiring-1*} *poly*
shows *poly* (*p* ^ *n*) *x* = *poly p x* ^ *n*
by (*induct n*) *simp-all*

lemma *poly-setprod*: *poly* (∏ *k*∈*A*. *p k*) *x* = (∏ *k*∈*A*. *poly* (*p k*) *x*)
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *degree-setprod-setsum-le*: *finite S* ⟹ *degree* (*setprod f S*) ≤ *setsum* (*degree o f*) *S*
proof (*induct S rule*: *finite-induct*)
case (*insert a S*)
show *?case* **unfolding** *setprod.insert*[*OF insert*(*1*−*2*)] *setsum.insert*[*OF insert*(*1*−*2*)]
by (*rule le-trans*[*OF degree-mult-le*], *insert insert*, *auto*)
qed *simp*

14.14 Conversions from natural numbers

lemma *of-nat-poly*: *of-nat n* = [:*of-nat n* :: *'a* :: *comm-semiring-1*:]
proof (*induction n*)
case (*Suc n*)
hence *of-nat* (*Suc n*) = *1* + (*of-nat n* :: *'a poly*)
by *simp*
also have (*of-nat n* :: *'a poly*) = [: *of-nat n* :]
by (*subst Suc*) (*rule refl*)
also have *1* = [:*1*:] **by** (*simp add*: *one-poly-def*)
finally show *?case* **by** (*subst* (*asm*) *add-pCons*) *simp*
qed *simp*

lemma *degree-of-nat* [*simp*]: *degree* (*of-nat n*) = *0*
by (*simp add*: *of-nat-poly*)

lemma *degree-numeral* [*simp*]: *degree* (*numeral n*) = *0*
by (*subst of-nat-numeral* [*symmetric*], *subst of-nat-poly*) *simp*

lemma *numeral-poly*: *numeral n* = [:*numeral n*:]
by (*subst of-nat-numeral* [*symmetric*], *subst of-nat-poly*) *simp*

14.15 Lemmas about divisibility

lemma *dvd-smult*: *p dvd q* ⟹ *p dvd smult a q*
proof −
assume *p dvd q*
then obtain *k* **where** *q* = *p* ∗ *k* **..**
then have *smult a q* = *p* ∗ *smult a k* **by** *simp*
then show *p dvd smult a q* **..**
qed

lemma *dvd-smult-cancel*:
fixes *a* :: ′*a* :: *field*
shows *p dvd smult a q* ⟹ *a* ≠ *0* ⟹ *p dvd q*
by (*drule dvd-smult* [**where** *a*=*inverse a*]) *simp*

lemma *dvd-smult-iff*:
fixes *a* :: ′*a*::*field*
shows *a* ≠ *0* ⟹ *p dvd smult a q* ⟷ *p dvd q*
by (*safe elim*!: *dvd-smult dvd-smult-cancel*)

lemma *smult-dvd-cancel*:
smult a p dvd q ⟹ *p dvd q*
proof −
assume *smult a p dvd q*
then obtain *k* **where** *q* = *smult a p* ∗ *k* **..**
then have *q* = *p* ∗ *smult a k* **by** *simp*
then show *p dvd q* **..**
qed

lemma *smult-dvd*:
fixes *a* :: ′*a*::*field*
shows *p dvd q* ⟹ *a* ≠ *0* ⟹ *smult a p dvd q*
by (*rule smult-dvd-cancel* [**where** *a*=*inverse a*]) *simp*

lemma *smult-dvd-iff*:
fixes *a* :: ′*a*::*field*
shows *smult a p dvd q* ⟷ (*if a* = *0 then q* = *0 else p dvd q*)
by (*auto elim*: *smult-dvd smult-dvd-cancel*)

14.16 Polynomials form an integral domain

lemma *coeff-mult-degree-sum*:
  coeff (*p* ∗ *q*) (*degree p* + *degree q*) =
   coeff p (*degree p*) ∗ *coeff q* (*degree q*)
  by (*induct p*, *simp*, *simp add*: *coeff-eq-0*)

instance *poly* :: (*idom*) *idom*
proof
  fix *p q* :: ′*a poly*
  assume *p* ≠ *0* **and** *q* ≠ *0*
  have *coeff* (*p* ∗ *q*) (*degree p* + *degree q*) =
      coeff p (*degree p*) ∗ *coeff q* (*degree q*)
   by (*rule coeff-mult-degree-sum*)
  also have *coeff p* (*degree p*) ∗ *coeff q* (*degree q*) ≠ *0*
   using ‹*p* ≠ *0*› **and** ‹*q* ≠ *0*› **by** *simp*
  finally have ∃ *n*. *coeff* (*p* ∗ *q*) *n* ≠ *0* **..**
  thus *p* ∗ *q* ≠ *0* **by** (*simp add*: *poly-eq-iff*)
qed

lemma *degree-mult-eq*:
  fixes *p q* :: ′*a*::*semidom poly*
  shows ⟦*p* ≠ *0*; *q* ≠ *0*⟧ ⟹ *degree* (*p* ∗ *q*) = *degree p* + *degree q*
apply (*rule order-antisym* [*OF degree-mult-le le-degree*])
apply (*simp add*: *coeff-mult-degree-sum*)
done

lemma *degree-mult-right-le*:
  fixes *p q* :: ′*a*::*semidom poly*
  assumes *q* ≠ *0*
  shows *degree p* ≤ *degree* (*p* ∗ *q*)
  using *assms* **by** (*cases p* = *0*) (*simp-all add*: *degree-mult-eq*)

lemma *coeff-degree-mult*:
  fixes *p q* :: ′*a*::*semidom poly*
  shows *coeff* (*p* ∗ *q*) (*degree* (*p* ∗ *q*)) =
   coeff q (*degree q*) ∗ *coeff p* (*degree p*)
  by (*cases p* = *0* ∨ *q* = *0*) (*auto simp add*: *degree-mult-eq coeff-mult-degree-sum mult-ac*)

lemma *dvd-imp-degree-le*:
  fixes *p q* :: ′*a*::*semidom poly*
  shows ⟦*p dvd q*; *q* ≠ *0*⟧ ⟹ *degree p* ≤ *degree q*
  by (*erule dvdE*, *hypsubst*, *subst degree-mult-eq*) *auto*

lemma *divides-degree*:
  assumes *pq*: *p dvd* (*q* :: ′*a* :: *semidom poly*)
  shows *degree p* ≤ *degree q* ∨ *q* = *0*
  by (*metis dvd-imp-degree-le pq*)

14.17 Polynomials form an ordered integral domain

definition *pos-poly* :: *'a*::*linordered-idom poly* ⇒ *bool*
where
  pos-poly p ⟷ *0* < *coeff p* (*degree p*)

lemma *pos-poly-pCons*:
  pos-poly (*pCons a p*) ⟷ *pos-poly p* ∨ (*p* = *0* ∧ *0* < *a*)
  unfolding *pos-poly-def* **by** *simp*

lemma *not-pos-poly-0* [*simp*]: ¬ *pos-poly 0*
  unfolding *pos-poly-def* **by** *simp*

lemma *pos-poly-add*: ⟦*pos-poly p*; *pos-poly q*⟧ ⟹ *pos-poly* (*p* + *q*)
  apply (*induct p arbitrary*: *q*, *simp*)
  apply (*case-tac q*, *force simp add*: *pos-poly-pCons add-pos-pos*)
  done

lemma *pos-poly-mult*: ⟦*pos-poly p*; *pos-poly q*⟧ ⟹ *pos-poly* (*p* ∗ *q*)
  unfolding *pos-poly-def*
  apply (*subgoal-tac p* ≠ *0* ∧ *q* ≠ *0*)
  apply (*simp add*: *degree-mult-eq coeff-mult-degree-sum*)
  apply *auto*
  done

lemma *pos-poly-total*: *p* = *0* ∨ *pos-poly p* ∨ *pos-poly* (− *p*)
by (*induct p*) (*auto simp add*: *pos-poly-pCons*)

lemma *last-coeffs-eq-coeff-degree*:
  p ≠ *0* ⟹ *last* (*coeffs p*) = *coeff p* (*degree p*)
  by (*simp add*: *coeffs-def*)

lemma *pos-poly-coeffs* [*code*]:
  pos-poly p ⟷ (*let as* = *coeffs p in as* ≠ [] ∧ *last as* > *0*) (**is** *?P* ⟷ *?Q*)
proof
 assume *?Q* **then show** *?P* **by** (*auto simp add*: *pos-poly-def last-coeffs-eq-coeff-degree*)
next
  assume *?P* **then have** ∗: *0* < *coeff p* (*degree p*) **by** (*simp add*: *pos-poly-def*)
  then have *p* ≠ *0* **by** *auto*
  with ∗ **show** *?Q* **by** (*simp add*: *last-coeffs-eq-coeff-degree*)
qed

instantiation *poly* :: (*linordered-idom*) *linordered-idom*
begin

definition
  x < *y* ⟷ *pos-poly* (*y* − *x*)

definition
  x ≤ *y* ⟷ *x* = *y* ∨ *pos-poly* (*y* − *x*)

definition
  $|x{::}'a\ poly| = (if\ x < 0\ then\ -\ x\ else\ x)$

definition
  $sgn\ (x{::}'a\ poly) = (if\ x = 0\ then\ 0\ else\ if\ 0 < x\ then\ 1\ else\ -\ 1)$

instance
proof
  fix *x y z* :: *'a poly*
  show $x < y \longleftrightarrow x \leq y \land \neg\ y \leq x$
    unfolding *less-eq-poly-def less-poly-def*
    apply *safe*
    apply *simp*
    apply (*drule* (*1*) *pos-poly-add*)
    apply *simp*
    done
  show $x \leq x$
    unfolding *less-eq-poly-def* **by** *simp*
  show $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
    unfolding *less-eq-poly-def*
    apply *safe*
    apply (*drule* (*1*) *pos-poly-add*)
    apply (*simp add*: *algebra-simps*)
    done
  show $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
    unfolding *less-eq-poly-def*
    apply *safe*
    apply (*drule* (*1*) *pos-poly-add*)
    apply *simp*
    done
  show $x \leq y \Longrightarrow z + x \leq z + y$
    unfolding *less-eq-poly-def*
    apply *safe*
    apply (*simp add*: *algebra-simps*)
    done
  show $x \leq y \lor y \leq x$
    unfolding *less-eq-poly-def*
    using *pos-poly-total* [*of* $x - y$]
    by *auto*
  show $x < y \Longrightarrow 0 < z \Longrightarrow z * x < z * y$
    unfolding *less-poly-def*
    by (*simp add*: *right-diff-distrib* [*symmetric*] *pos-poly-mult*)
  show $|x| = (if\ x < 0\ then\ -\ x\ else\ x)$
    by (*rule abs-poly-def*)
  show $sgn\ x = (if\ x = 0\ then\ 0\ else\ if\ 0 < x\ then\ 1\ else\ -\ 1)$
    by (*rule sgn-poly-def*)
qed

end

TODO: Simplification rules for comparisons

14.18 Synthetic division and polynomial roots

Synthetic division is simply division by the linear polynomial $x - c$.

definition *synthetic-divmod* :: *'a::comm-semiring-0 poly ⇒ 'a ⇒ 'a poly × 'a*
where
synthetic-divmod p c = fold-coeffs (λa (q, r). (pCons r q, a + c ∗ r)) p (0, 0)

definition *synthetic-div* :: *'a::comm-semiring-0 poly ⇒ 'a ⇒ 'a poly*
where
synthetic-div p c = fst (synthetic-divmod p c)

lemma *synthetic-divmod-0* [*simp*]:
synthetic-divmod 0 c = (0, 0)
by (*simp add*: *synthetic-divmod-def*)

lemma *synthetic-divmod-pCons* [*simp*]:
synthetic-divmod (pCons a p) c = (λ(q, r). (pCons r q, a + c ∗ r)) (synthetic-divmod p c)
by (*cases p = 0 ∧ a = 0*) (*auto simp add*: *synthetic-divmod-def*)

lemma *synthetic-div-0* [*simp*]:
synthetic-div 0 c = 0
unfolding *synthetic-div-def* **by** *simp*

lemma *synthetic-div-unique-lemma*: *smult c p = pCons a p ⟹ p = 0*
by (*induct p arbitrary*: *a*) *simp-all*

lemma *snd-synthetic-divmod*:
snd (synthetic-divmod p c) = poly p c
by (*induct p*, *simp*, *simp add*: *split-def*)

lemma *synthetic-div-pCons* [*simp*]:
synthetic-div (pCons a p) c = pCons (poly p c) (synthetic-div p c)
unfolding *synthetic-div-def*
by (*simp add*: *split-def snd-synthetic-divmod*)

lemma *synthetic-div-eq-0-iff*:
synthetic-div p c = 0 ⟷ degree p = 0
by (*induct p*, *simp*, *case-tac p*, *simp*)

lemma *degree-synthetic-div*:
degree (synthetic-div p c) = degree p − 1
by (*induct p*, *simp*, *simp add*: *synthetic-div-eq-0-iff*)

lemma *synthetic-div-correct*:

$p + smult\ c\ (synthetic\text{-}div\ p\ c) = pCons\ (poly\ p\ c)\ (synthetic\text{-}div\ p\ c)$
by (*induct p*) *simp-all*

lemma *synthetic-div-unique*:
$p + smult\ c\ q = pCons\ r\ q \Longrightarrow r = poly\ p\ c \wedge q = synthetic\text{-}div\ p\ c$
apply (*induct p arbitrary*: *q r*)
apply (*simp*, *frule synthetic-div-unique-lemma*, *simp*)
apply (*case-tac q*, *force*)
done

lemma *synthetic-div-correct'*:
fixes *c* :: *'a::comm-ring-1*
shows $[:-c,\ 1:] * synthetic\text{-}div\ p\ c + [:poly\ p\ c:] = p$
using *synthetic-div-correct* [*of p c*]
by (*simp add*: *algebra-simps*)

lemma *poly-eq-0-iff-dvd*:
fixes *c* :: *'a::idom*
shows $poly\ p\ c = 0 \longleftrightarrow [:-c,\ 1:]\ dvd\ p$
proof
assume $poly\ p\ c = 0$
with *synthetic-div-correct'* [*of c p*]
have $p = [:-c,\ 1:] * synthetic\text{-}div\ p\ c$ **by** *simp*
then show $[:-c,\ 1:]\ dvd\ p$ **..**
next
assume $[:-c,\ 1:]\ dvd\ p$
then obtain *k* **where** $p = [:-c,\ 1:] * k$ **by** (*rule dvdE*)
then show $poly\ p\ c = 0$ **by** *simp*
qed

lemma *dvd-iff-poly-eq-0*:
fixes *c* :: *'a::idom*
shows $[:c,\ 1:]\ dvd\ p \longleftrightarrow poly\ p\ (-c) = 0$
by (*simp add*: *poly-eq-0-iff-dvd*)

lemma *poly-roots-finite*:
fixes *p* :: *'a::idom poly*
shows $p \neq 0 \Longrightarrow finite\ \{x.\ poly\ p\ x = 0\}$
proof (*induct* $n \equiv degree\ p$ *arbitrary*: *p*)
case (*0 p*)
then obtain *a* **where** $a \neq 0$ **and** $p = [:a:]$
by (*cases p*, *simp split*: *if-splits*)
then show $finite\ \{x.\ poly\ p\ x = 0\}$ **by** *simp*
next
case (*Suc n p*)
show $finite\ \{x.\ poly\ p\ x = 0\}$
proof (*cases* $\exists x.\ poly\ p\ x = 0$)
case *False*
then show $finite\ \{x.\ poly\ p\ x = 0\}$ **by** *simp*

next
case *True*
then obtain *a* **where** *poly p a = 0* **..**
then have [:−*a*, *1*:] *dvd p* **by** (*simp only*: *poly-eq-0-iff-dvd*)
then obtain *k* **where** *k*: *p* = [:−*a*, *1*:] ∗ *k* **..**
with ‹*p* ≠ *0*› **have** *k* ≠ *0* **by** *auto*
with *k* **have** *degree p* = *Suc* (*degree k*)
by (*simp add*: *degree-mult-eq del*: *mult-pCons-left*)
with ‹*Suc n* = *degree p*› **have** *n* = *degree k* **by** *simp*
then have *finite* {*x*. *poly k x* = *0*} **using** ‹*k* ≠ *0*› **by** (*rule Suc.hyps*)
then have *finite* (*insert a* {*x*. *poly k x* = *0*}) **by** *simp*
then show *finite* {*x*. *poly p x* = *0*}
by (*simp add*: *k Collect-disj-eq del*: *mult-pCons-left*)
qed
qed

lemma *poly-eq-poly-eq-iff*:
fixes *p q* :: ′*a*::{*idom*,*ring-char-0*} *poly*
shows *poly p* = *poly q* ⟷ *p* = *q* (**is** *?P* ⟷ *?Q*)
proof
assume *?Q* **then show** *?P* **by** *simp*
next
{ **fix** *p* :: ′*a*::{*idom*,*ring-char-0*} *poly*
have *poly p* = *poly 0* ⟷ *p* = *0*
apply (*cases p* = *0*, *simp-all*)
apply (*drule poly-roots-finite*)
apply (*auto simp add*: *infinite-UNIV-char-0*)
done
} **note** *this* [*of p* − *q*]
moreover assume *?P*
ultimately show *?Q* **by** *auto*
qed

lemma *poly-all-0-iff-0*:
fixes *p* :: ′*a*::{*ring-char-0*, *idom*} *poly*
shows (∀ *x*. *poly p x* = *0*) ⟷ *p* = *0*
by (*auto simp add*: *poly-eq-poly-eq-iff* [*symmetric*])

14.19 Long division of polynomials

definition *pdivmod-rel* :: ′*a*::*field poly* ⇒ ′*a poly* ⇒ ′*a poly* ⇒ ′*a poly* ⇒ *bool*
where
pdivmod-rel x y q r ⟷
x = *q* ∗ *y* + *r* ∧ (*if y* = *0 then q* = *0 else r* = *0* ∨ *degree r* < *degree y*)

lemma *pdivmod-rel-0*:
pdivmod-rel 0 y 0 0
unfolding *pdivmod-rel-def* **by** *simp*

```
lemma pdivmod-rel-by-0:
  pdivmod-rel x 0 0 x
  unfolding pdivmod-rel-def by simp

lemma eq-zero-or-degree-less:
  assumes degree p ≤ n and coeff p n = 0
  shows p = 0 ∨ degree p < n
proof (cases n)
  case 0
  with ‹degree p ≤ n› and ‹coeff p n = 0›
  have coeff p (degree p) = 0 by simp
  then have p = 0 by simp
  then show ?thesis ..
next
  case (Suc m)
  have ∀ i>n. coeff p i = 0
    using ‹degree p ≤ n› by (simp add: coeff-eq-0)
  then have ∀ i≥n. coeff p i = 0
    using ‹coeff p n = 0› by (simp add: le-less)
  then have ∀ i>m. coeff p i = 0
    using ‹n = Suc m› by (simp add: less-eq-Suc-le)
  then have degree p ≤ m
    by (rule degree-le)
  then have degree p < n
    using ‹n = Suc m› by (simp add: less-Suc-eq-le)
  then show ?thesis ..
qed

lemma pdivmod-rel-pCons:
  assumes rel: pdivmod-rel x y q r
  assumes y: y ≠ 0
  assumes b: b = coeff (pCons a r) (degree y) / coeff y (degree y)
  shows pdivmod-rel (pCons a x) y (pCons b q) (pCons a r − smult b y)
    (is pdivmod-rel ?x y ?q ?r)
proof −
  have x: x = q ∗ y + r and r: r = 0 ∨ degree r < degree y
    using assms unfolding pdivmod-rel-def by simp-all

  have 1: ?x = ?q ∗ y + ?r
    using b x by simp

  have 2: ?r = 0 ∨ degree ?r < degree y
  proof (rule eq-zero-or-degree-less)
    show degree ?r ≤ degree y
    proof (rule degree-diff-le)
      show degree (pCons a r) ≤ degree y
        using r by auto
      show degree (smult b y) ≤ degree y
        by (rule degree-smult-le)
```

```
    qed
  next
    show coeff ?r (degree y) = 0
      using ‹y ≠ 0› unfolding b by simp
  qed

  from 1 2 show ?thesis
    unfolding pdivmod-rel-def
    using ‹y ≠ 0› by simp
qed

lemma pdivmod-rel-exists: ∃ q r. pdivmod-rel x y q r
apply (cases y = 0)
apply (fast intro!: pdivmod-rel-by-0)
apply (induct x)
apply (fast intro!: pdivmod-rel-0)
apply (fast intro!: pdivmod-rel-pCons)
done

lemma pdivmod-rel-unique:
  assumes 1: pdivmod-rel x y q1 r1
  assumes 2: pdivmod-rel x y q2 r2
  shows q1 = q2 ∧ r1 = r2
proof (cases y = 0)
  assume y = 0 with assms show ?thesis
    by (simp add: pdivmod-rel-def)
next
  assume [simp]: y ≠ 0
  from 1 have q1: x = q1 * y + r1 and r1: r1 = 0 ∨ degree r1 < degree y
    unfolding pdivmod-rel-def by simp-all
  from 2 have q2: x = q2 * y + r2 and r2: r2 = 0 ∨ degree r2 < degree y
    unfolding pdivmod-rel-def by simp-all
  from q1 q2 have q3: (q1 − q2) * y = r2 − r1
    by (simp add: algebra-simps)
  from r1 r2 have r3: (r2 − r1) = 0 ∨ degree (r2 − r1) < degree y
    by (auto intro: degree-diff-less)

  show q1 = q2 ∧ r1 = r2
  proof (rule ccontr)
    assume ¬ (q1 = q2 ∧ r1 = r2)
    with q3 have q1 ≠ q2 and r1 ≠ r2 by auto
    with r3 have degree (r2 − r1) < degree y by simp
    also have degree y ≤ degree (q1 − q2) + degree y by simp
    also have ... = degree ((q1 − q2) * y)
      using ‹q1 ≠ q2› by (simp add: degree-mult-eq)
    also have ... = degree (r2 − r1)
      using q3 by simp
    finally have degree (r2 − r1) < degree (r2 − r1) .
    then show False by simp
```

qed
qed

lemma *pdivmod-rel-0-iff*: *pdivmod-rel 0 y q r* ⟷ *q* = *0* ∧ *r* = *0*
by (*auto dest*: *pdivmod-rel-unique intro*: *pdivmod-rel-0*)

lemma *pdivmod-rel-by-0-iff*: *pdivmod-rel x 0 q r* ⟷ *q* = *0* ∧ *r* = *x*
by (*auto dest*: *pdivmod-rel-unique intro*: *pdivmod-rel-by-0*)

lemmas *pdivmod-rel-unique-div* = *pdivmod-rel-unique* [*THEN conjunct1*]

lemmas *pdivmod-rel-unique-mod* = *pdivmod-rel-unique* [*THEN conjunct2*]

instantiation *poly* :: (*field*) *ring-div*
begin

definition *divide-poly* **where**
div-poly-def: *x div y* = (*THE q*. ∃ *r*. *pdivmod-rel x y q r*)

definition *mod-poly* **where**
x mod y = (*THE r*. ∃ *q*. *pdivmod-rel x y q r*)

lemma *div-poly-eq*:
pdivmod-rel x y q r ⟹ *x div y* = *q*
unfolding *div-poly-def*
by (*fast elim*: *pdivmod-rel-unique-div*)

lemma *mod-poly-eq*:
pdivmod-rel x y q r ⟹ *x mod y* = *r*
unfolding *mod-poly-def*
by (*fast elim*: *pdivmod-rel-unique-mod*)

lemma *pdivmod-rel*:
pdivmod-rel x y (*x div y*) (*x mod y*)
proof −
from *pdivmod-rel-exists*
obtain *q r* **where** *pdivmod-rel x y q r* **by** *fast*
thus *?thesis*
by (*simp add*: *div-poly-eq mod-poly-eq*)
qed

instance
proof
fix *x y* :: *'a poly*
show *x div y* ∗ *y* + *x mod y* = *x*
using *pdivmod-rel* [*of x y*]
by (*simp add*: *pdivmod-rel-def*)
next
fix *x* :: *'a poly*

```
  have pdivmod-rel x 0 0 x
    by (rule pdivmod-rel-by-0)
  thus x div 0 = 0
    by (rule div-poly-eq)
next
  fix y :: 'a poly
  have pdivmod-rel 0 y 0 0
    by (rule pdivmod-rel-0)
  thus 0 div y = 0
    by (rule div-poly-eq)
next
  fix x y z :: 'a poly
  assume y ≠ 0
  hence pdivmod-rel (x + z * y) y (z + x div y) (x mod y)
    using pdivmod-rel [of x y]
    by (simp add: pdivmod-rel-def distrib-right)
  thus (x + z * y) div y = z + x div y
    by (rule div-poly-eq)
next
  fix x y z :: 'a poly
  assume x ≠ 0
  show (x * y) div (x * z) = y div z
  proof (cases y ≠ 0 ∧ z ≠ 0)
    have ⋀x::'a poly. pdivmod-rel x 0 0 x
      by (rule pdivmod-rel-by-0)
    then have [simp]: ⋀x::'a poly. x div 0 = 0
      by (rule div-poly-eq)
    have ⋀x::'a poly. pdivmod-rel 0 x 0 0
      by (rule pdivmod-rel-0)
    then have [simp]: ⋀x::'a poly. 0 div x = 0
      by (rule div-poly-eq)
    case False then show ?thesis by auto
  next
    case True then have y ≠ 0 and z ≠ 0 by auto
    with ‹x ≠ 0›
    have ⋀q r. pdivmod-rel y z q r ⟹ pdivmod-rel (x * y) (x * z) q (x * r)
      by (auto simp add: pdivmod-rel-def algebra-simps)
        (rule classical, simp add: degree-mult-eq)
    moreover from pdivmod-rel have pdivmod-rel y z (y div z) (y mod z) .
    ultimately have pdivmod-rel (x * y) (x * z) (y div z) (x * (y mod z)) .
    then show ?thesis by (simp add: div-poly-eq)
  qed
qed

end

lemma is-unit-monom-0:
  fixes a :: 'a::field
  assumes a ≠ 0
```

shows *is-unit* (*monom a 0*)
proof
from *assms* **show** *1* = *monom a 0* ∗ *monom* (*inverse a*) *0*
by (*simp add*: *mult-monom*)
qed

lemma *is-unit-triv*:
fixes *a* :: *'a*::*field*
assumes *a* ≠ *0*
shows *is-unit* [:*a*:]
using *assms* **by** (*simp add*: *is-unit-monom-0 monom-0* [*symmetric*])

lemma *is-unit-iff-degree*:
assumes *p* ≠ *0*
shows *is-unit p* ⟷ *degree p* = *0* (**is** *?P* ⟷ *?Q*)
proof
assume *?Q*
then obtain *a* **where** *p* = [:*a*:] **by** (*rule degree-eq-zeroE*)
with *assms* **show** *?P* **by** (*simp add*: *is-unit-triv*)
next
assume *?P*
then obtain *q* **where** *q* ≠ *0 p* ∗ *q* = *1* **..**
then have *degree* (*p* ∗ *q*) = *degree 1*
by *simp*
with ‹*p* ≠ *0*› ‹*q* ≠ *0*› **have** *degree p* + *degree q* = *0*
by (*simp add*: *degree-mult-eq*)
then show *?Q* **by** *simp*
qed

lemma *is-unit-pCons-iff*:
is-unit (*pCons a p*) ⟷ *p* = *0* ∧ *a* ≠ *0* (**is** *?P* ⟷ *?Q*)
by (*cases p* = *0*) (*auto simp add*: *is-unit-triv is-unit-iff-degree*)

lemma *is-unit-monom-trival*:
fixes *p* :: *'a*::*field poly*
assumes *is-unit p*
shows *monom* (*coeff p* (*degree p*)) *0* = *p*
using *assms* **by** (*cases p*) (*simp-all add*: *monom-0 is-unit-pCons-iff*)

lemma *is-unit-polyE*:
assumes *is-unit p*
obtains *a* **where** *p* = *monom a 0* **and** *a* ≠ *0*
proof −
obtain *a q* **where** *p* = *pCons a q* **by** (*cases p*)
with *assms* **have** *p* = [:*a*:] **and** *a* ≠ *0*
by (*simp-all add*: *is-unit-pCons-iff*)
with *that* **show** *thesis* **by** (*simp add*: *monom-0*)
qed

instantiation *poly* :: (*field*) *normalization-semidom*
begin

definition *normalize-poly* :: *'a poly* ⇒ *'a poly*
  where *normalize-poly p* = *smult* (*inverse* (*coeff p* (*degree p*))) *p*

definition *unit-factor-poly* :: *'a poly* ⇒ *'a poly*
  where *unit-factor-poly p* = *monom* (*coeff p* (*degree p*)) *0*

instance
proof
  fix *p* :: *'a poly*
  show *unit-factor p* ∗ *normalize p* = *p*
    by (*cases p* = *0*)
      (*simp-all add*: *normalize-poly-def unit-factor-poly-def*,
      simp only: *mult-smult-left* [*symmetric*] *smult-monom*, *simp*)
next
  show *normalize 0* = (*0*::*'a poly*)
    by (*simp add*: *normalize-poly-def*)
next
  show *unit-factor 0* = (*0*::*'a poly*)
    by (*simp add*: *unit-factor-poly-def*)
next
  fix *p* :: *'a poly*
  assume *is-unit p*
  then obtain *a* **where** *p* = *monom a 0* **and** *a* ≠ *0*
    by (*rule is-unit-polyE*)
  then show *normalize p* = *1*
    by (*auto simp add*: *normalize-poly-def smult-monom degree-monom-eq*)
next
  fix *p q* :: *'a poly*
  assume *q* ≠ *0*
  from ‹*q* ≠ *0*› **have** *is-unit* (*monom* (*coeff q* (*degree q*)) *0*)
    by (*auto intro*: *is-unit-monom-0*)
  then show *is-unit* (*unit-factor q*)
    by (*simp add*: *unit-factor-poly-def*)
next
  fix *p q* :: *'a poly*
  have *monom* (*coeff* (*p* ∗ *q*) (*degree* (*p* ∗ *q*))) *0* =
    monom (*coeff p* (*degree p*)) *0* ∗ *monom* (*coeff q* (*degree q*)) *0*
    by (*simp add*: *monom-0 coeff-degree-mult*)
  then show *unit-factor* (*p* ∗ *q*) =
    unit-factor p ∗ *unit-factor q*
    by (*simp add*: *unit-factor-poly-def*)
qed

end

lemma *unit-factor-monom* [*simp*]:

unit-factor (*monom a n*) =
(*if a* = *0 then 0 else monom a 0*)
by (*simp add*: *unit-factor-poly-def degree-monom-eq*)

lemma *unit-factor-pCons* [*simp*]:
unit-factor (*pCons a p*) =
(*if p* = *0 then monom a 0 else unit-factor p*)
by (*simp add*: *unit-factor-poly-def*)

lemma *normalize-monom* [*simp*]:
normalize (*monom a n*) =
(*if a* = *0 then 0 else monom 1 n*)
by (*simp add*: *normalize-poly-def degree-monom-eq smult-monom*)

lemma *degree-mod-less*:
$y \neq 0 \Longrightarrow x \mathit{\ mod\ } y = 0 \vee \mathit{degree}\ (x \mathit{\ mod\ } y) < \mathit{degree}\ y$
using *pdivmod-rel* [*of x y*]
unfolding *pdivmod-rel-def* **by** *simp*

lemma *div-poly-less*: *degree x* < *degree y* ⟹ *x div y* = *0*
proof −
assume *degree x* < *degree y*
hence *pdivmod-rel x y 0 x*
by (*simp add*: *pdivmod-rel-def*)
thus *x div y* = *0* **by** (*rule div-poly-eq*)
qed

lemma *mod-poly-less*: *degree x* < *degree y* ⟹ *x mod y* = *x*
proof −
assume *degree x* < *degree y*
hence *pdivmod-rel x y 0 x*
by (*simp add*: *pdivmod-rel-def*)
thus *x mod y* = *x* **by** (*rule mod-poly-eq*)
qed

lemma *pdivmod-rel-smult-left*:
pdivmod-rel x y q r
⟹ *pdivmod-rel* (*smult a x*) *y* (*smult a q*) (*smult a r*)
unfolding *pdivmod-rel-def* **by** (*simp add*: *smult-add-right*)

lemma *div-smult-left*: (*smult a x*) *div y* = *smult a* (*x div y*)
by (*rule div-poly-eq*, *rule pdivmod-rel-smult-left*, *rule pdivmod-rel*)

lemma *mod-smult-left*: (*smult a x*) *mod y* = *smult a* (*x mod y*)
by (*rule mod-poly-eq*, *rule pdivmod-rel-smult-left*, *rule pdivmod-rel*)

lemma *poly-div-minus-left* [*simp*]:
fixes *x y* :: *'a::field poly*
shows (− *x*) *div y* = − (*x div y*)

using *div-smult-left* [*of* − *1*::′*a*] **by** *simp*

lemma *poly-mod-minus-left* [*simp*]:
fixes *x y* :: ′*a*::*field poly*
shows (− *x*) *mod y* = − (*x mod y*)
using *mod-smult-left* [*of* − *1*::′*a*] **by** *simp*

lemma *pdivmod-rel-add-left*:
assumes *pdivmod-rel x y q r*
assumes *pdivmod-rel x*′ *y q*′ *r*′
shows *pdivmod-rel* (*x* + *x*′) *y* (*q* + *q*′) (*r* + *r*′)
using *assms* **unfolding** *pdivmod-rel-def*
by (*auto simp add*: *algebra-simps degree-add-less*)

lemma *poly-div-add-left*:
fixes *x y z* :: ′*a*::*field poly*
shows (*x* + *y*) *div z* = *x div z* + *y div z*
using *pdivmod-rel-add-left* [*OF pdivmod-rel pdivmod-rel*]
by (*rule div-poly-eq*)

lemma *poly-mod-add-left*:
fixes *x y z* :: ′*a*::*field poly*
shows (*x* + *y*) *mod z* = *x mod z* + *y mod z*
using *pdivmod-rel-add-left* [*OF pdivmod-rel pdivmod-rel*]
by (*rule mod-poly-eq*)

lemma *poly-div-diff-left*:
fixes *x y z* :: ′*a*::*field poly*
shows (*x* − *y*) *div z* = *x div z* − *y div z*
by (*simp only*: *diff-conv-add-uminus poly-div-add-left poly-div-minus-left*)

lemma *poly-mod-diff-left*:
fixes *x y z* :: ′*a*::*field poly*
shows (*x* − *y*) *mod z* = *x mod z* − *y mod z*
by (*simp only*: *diff-conv-add-uminus poly-mod-add-left poly-mod-minus-left*)

lemma *pdivmod-rel-smult-right*:
⟦*a* ≠ *0*; *pdivmod-rel x y q r*⟧
⟹ *pdivmod-rel x* (*smult a y*) (*smult* (*inverse a*) *q*) *r*
unfolding *pdivmod-rel-def* **by** *simp*

lemma *div-smult-right*:
a ≠ *0* ⟹ *x div* (*smult a y*) = *smult* (*inverse a*) (*x div y*)
by (*rule div-poly-eq*, *erule pdivmod-rel-smult-right*, *rule pdivmod-rel*)

lemma *mod-smult-right*: *a* ≠ *0* ⟹ *x mod* (*smult a y*) = *x mod y*
by (*rule mod-poly-eq*, *erule pdivmod-rel-smult-right*, *rule pdivmod-rel*)

lemma *poly-div-minus-right* [*simp*]:

fixes *x y* :: *'a*::*field poly*
shows *x div* (− *y*) = − (*x div y*)
using *div-smult-right* [*of* − *1*::*'a*] **by** (*simp add*: *nonzero-inverse-minus-eq*)

lemma *poly-mod-minus-right* [*simp*]:
fixes *x y* :: *'a*::*field poly*
shows *x mod* (− *y*) = *x mod y*
using *mod-smult-right* [*of* − *1*::*'a*] **by** *simp*

lemma *pdivmod-rel-mult*:
⟦*pdivmod-rel x y q r*; *pdivmod-rel q z q' r'*⟧
⟹ *pdivmod-rel x* (*y* ∗ *z*) *q'* (*y* ∗ *r'* + *r*)
apply (*cases z* = *0*, *simp add*: *pdivmod-rel-def*)
apply (*cases y* = *0*, *simp add*: *pdivmod-rel-by-0-iff pdivmod-rel-0-iff*)
apply (*cases r* = *0*)
apply (*cases r'* = *0*)
apply (*simp add*: *pdivmod-rel-def*)
apply (*simp add*: *pdivmod-rel-def field-simps degree-mult-eq*)
apply (*cases r'* = *0*)
apply (*simp add*: *pdivmod-rel-def degree-mult-eq*)
apply (*simp add*: *pdivmod-rel-def field-simps*)
apply (*simp add*: *degree-mult-eq degree-add-less*)
done

lemma *poly-div-mult-right*:
fixes *x y z* :: *'a*::*field poly*
shows *x div* (*y* ∗ *z*) = (*x div y*) *div z*
by (*rule div-poly-eq*, *rule pdivmod-rel-mult*, (*rule pdivmod-rel*)+)

lemma *poly-mod-mult-right*:
fixes *x y z* :: *'a*::*field poly*
shows *x mod* (*y* ∗ *z*) = *y* ∗ (*x div y mod z*) + *x mod y*
by (*rule mod-poly-eq*, *rule pdivmod-rel-mult*, (*rule pdivmod-rel*)+)

lemma *mod-pCons*:
fixes *a* **and** *x*
assumes *y*: *y* ≠ *0*
defines *b*: *b* ≡ *coeff* (*pCons a* (*x mod y*)) (*degree y*) / *coeff y* (*degree y*)
shows (*pCons a x*) *mod y* = (*pCons a* (*x mod y*) − *smult b y*)
unfolding *b*
apply (*rule mod-poly-eq*)
apply (*rule pdivmod-rel-pCons* [*OF pdivmod-rel y refl*])
done

definition *pdivmod* :: *'a*::*field poly* ⇒ *'a poly* ⇒ *'a poly* × *'a poly*
where
pdivmod p q = (*p div q*, *p mod q*)

lemma *div-poly-code* [*code*]:

p div q = *fst* (*pdivmod p q*)
by (*simp add*: *pdivmod-def*)

lemma *mod-poly-code* [*code*]:
p mod q = *snd* (*pdivmod p q*)
by (*simp add*: *pdivmod-def*)

lemma *pdivmod-0*:
pdivmod 0 q = (*0*, *0*)
by (*simp add*: *pdivmod-def*)

lemma *pdivmod-pCons*:
pdivmod (*pCons a p*) *q* =
(*if q* = *0 then* (*0*, *pCons a p*) *else*
(*let* (*s*, *r*) = *pdivmod p q*;
b = *coeff* (*pCons a r*) (*degree q*) / *coeff q* (*degree q*)
in (*pCons b s*, *pCons a r* − *smult b q*)))
apply (*simp add*: *pdivmod-def Let-def*, *safe*)
apply (*rule div-poly-eq*)
apply (*erule pdivmod-rel-pCons* [*OF pdivmod-rel - refl*])
apply (*rule mod-poly-eq*)
apply (*erule pdivmod-rel-pCons* [*OF pdivmod-rel - refl*])
done

lemma *pdivmod-fold-coeffs* [*code*]:
pdivmod p q = (*if q* = *0 then* (*0*, *p*)
else fold-coeffs (λ*a* (*s*, *r*).
let b = *coeff* (*pCons a r*) (*degree q*) / *coeff q* (*degree q*)
in (*pCons b s*, *pCons a r* − *smult b q*)
) *p* (*0*, *0*))
apply (*cases q* = *0*)
apply (*simp add*: *pdivmod-def*)
apply (*rule sym*)
apply (*induct p*)
apply (*simp-all add*: *pdivmod-0 pdivmod-pCons*)
apply (*case-tac a* = *0* ∧ *p* = *0*)
apply (*auto simp add*: *pdivmod-def*)
done

14.20 Order of polynomial roots

definition *order* :: *'a*::*idom* ⇒ *'a poly* ⇒ *nat*
where
order a p = (*LEAST n*. ¬ [:−*a*, *1*:] ^ *Suc n dvd p*)

lemma *coeff-linear-power*:
fixes *a* :: *'a*::*comm-semiring-1*
shows *coeff* ([:*a*, *1*:] ^ *n*) *n* = *1*
apply (*induct n*, *simp-all*)

apply (*subst coeff-eq-0*)
apply (*auto intro*: *le-less-trans degree-power-le*)
done

lemma *degree-linear-power*:
  fixes *a* :: *'a::comm-semiring-1*
  shows *degree* ([:*a*, *1*:] ^ *n*) = *n*
apply (*rule order-antisym*)
apply (*rule ord-le-eq-trans* [*OF degree-power-le*], *simp*)
apply (*rule le-degree*, *simp add*: *coeff-linear-power*)
done

lemma *order-1*: [:−*a*, *1*:] ^ *order a p dvd p*
apply (*cases p* = *0*, *simp*)
apply (*cases order a p*, *simp*)
apply (*subgoal-tac nat* < (*LEAST n*. ¬ [:−*a*, *1*:] ^ *Suc n dvd p*))
apply (*drule not-less-Least*, *simp*)
apply (*fold order-def*, *simp*)
done

lemma *order-2*: *p* ≠ *0* ⟹ ¬ [:−*a*, *1*:] ^ *Suc* (*order a p*) *dvd p*
unfolding *order-def*
apply (*rule LeastI-ex*)
apply (*rule-tac x*=*degree p* **in** *exI*)
apply (*rule notI*)
apply (*drule* (*1*) *dvd-imp-degree-le*)
apply (*simp only*: *degree-linear-power*)
done

lemma *order*:
  p ≠ *0* ⟹ [:−*a*, *1*:] ^ *order a p dvd p* ∧ ¬ [:−*a*, *1*:] ^ *Suc* (*order a p*) *dvd p*
by (*rule conjI* [*OF order-1 order-2*])

lemma *order-degree*:
  assumes *p*: *p* ≠ *0*
  shows *order a p* ≤ *degree p*
proof −
  have *order a p* = *degree* ([:−*a*, *1*:] ^ *order a p*)
    by (*simp only*: *degree-linear-power*)
  also have ... ≤ *degree p*
    using *order-1 p* **by** (*rule dvd-imp-degree-le*)
  finally show *?thesis* **.**
qed

lemma *order-root*: *poly p a* = *0* ⟷ *p* = *0* ∨ *order a p* ≠ *0*
apply (*cases p* = *0*, *simp-all*)
apply (*rule iffI*)
apply (*metis order-2 not-gr0 poly-eq-0-iff-dvd power-0 power-Suc-0 power-one-right*)
unfolding *poly-eq-0-iff-dvd*

apply (*metis dvd-power dvd-trans order-1*)
done

lemma *order-0I*: *poly p a* $\neq$ *0* $\Longrightarrow$ *order a p* = *0*
  by (*subst* (*asm*) *order-root*) *auto*

14.21 Additional induction rules on polynomials

An induction rule for induction over the roots of a polynomial with a certain property. (e.g. all positive roots)

lemma *poly-root-induct* [*case-names 0 no-roots root*]:
  fixes *p* :: *'a* :: *idom poly*
  assumes *Q 0*
  assumes $\bigwedge$*p*. ($\bigwedge$*a*. *P a* $\Longrightarrow$ *poly p a* $\neq$ *0*) $\Longrightarrow$ *Q p*
  assumes $\bigwedge$*a p*. *P a* $\Longrightarrow$ *Q p* $\Longrightarrow$ *Q* ([:*a*, −*1*:] ∗ *p*)
  shows *Q p*
proof (*induction degree p arbitrary*: *p rule*: *less-induct*)
  case (*less p*)
  show *?case*
  proof (*cases p* = *0*)
    assume *nz*: *p* $\neq$ *0*
    show *?case*
    proof (*cases* $\exists$ *a*. *P a* $\wedge$ *poly p a* = *0*)
      case *False*
      thus *?thesis* **by** (*intro assms*(*2*)) *blast*
    next
      case *True*
      then obtain *a* **where** *a*: *P a poly p a* = *0*
        by *blast*
      hence −[:−*a*, *1*:] *dvd p*
        by (*subst minus-dvd-iff*) (*simp add*: *poly-eq-0-iff-dvd*)
      then obtain *q* **where** *q*: *p* = [:*a*, −*1*:] ∗ *q* **by** (*elim dvdE*) *simp*
      with *nz* **have** *q-nz*: *q* $\neq$ *0* **by** *auto*
      have *degree p* = *Suc* (*degree q*)
        by (*subst q*, *subst degree-mult-eq*) (*simp-all add*: *q-nz*)
      hence *Q q* **by** (*intro less*) *simp*
      from *a*(*1*) **and** *this* **have** *Q* ([:*a*, −*1*:] ∗ *q*)
        by (*rule assms*(*3*))
      with *q* **show** *?thesis* **by** *simp*
    qed
  qed (*simp add*: *assms*(*1*))
qed

lemma *dropWhile-replicate-append*:
  dropWhile (*op*= *a*) (*replicate n a* @ *ys*) = *dropWhile* (*op*= *a*) *ys*
  by (*induction n*) *simp-all*

lemma *Poly-append-replicate-0*: *Poly* (*xs* @ *replicate n 0*) = *Poly xs*
  by (*subst coeffs-eq-iff*) (*simp-all add*: *strip-while-def dropWhile-replicate-append*)

An induction rule for simultaneous induction over two polynomials, prepending one coefficient in each step.

lemma *poly-induct2* [*case-names 0 pCons*]:
  assumes *P 0 0* $\bigwedge$*a p b q. P p q* $\Longrightarrow$ *P* (*pCons a p*) (*pCons b q*)
  shows *P p q*
proof −
  def *n* ≡ *max* (*length* (*coeffs p*)) (*length* (*coeffs q*))
  def *xs* ≡ *coeffs p* @ (*replicate* (*n* − *length* (*coeffs p*)) *0*)
  def *ys* ≡ *coeffs q* @ (*replicate* (*n* − *length* (*coeffs q*)) *0*)
  have *length xs* = *length ys*
    by (*simp add*: *xs-def ys-def n-def*)
  hence *P* (*Poly xs*) (*Poly ys*)
    by (*induction rule*: *list-induct2*) (*simp-all add*: *assms*)
  also have *Poly xs* = *p*
    by (*simp add*: *xs-def Poly-append-replicate-0*)
  also have *Poly ys* = *q*
    by (*simp add*: *ys-def Poly-append-replicate-0*)
  finally show *?thesis* **.**
qed

14.22 Composition of polynomials

definition *pcompose* :: *'a*::*comm-semiring-0 poly* ⇒ *'a poly* ⇒ *'a poly*
where
  pcompose p q = *fold-coeffs* (λ*a c.* [:*a*:] + *q* ∗ *c*) *p 0*

notation *pcompose* (**infixl** $\circ_p$ *71*)

lemma *pcompose-0* [*simp*]:
  pcompose 0 q = *0*
  by (*simp add*: *pcompose-def*)

lemma *pcompose-pCons*:
  pcompose (*pCons a p*) *q* = [:*a*:] + *q* ∗ *pcompose p q*
  by (*cases p* = *0* ∧ *a* = *0*) (*auto simp add*: *pcompose-def*)

lemma *pcompose-1*:
  fixes *p* :: *'a* :: *comm-semiring-1 poly*
  shows *pcompose 1 p* = *1*
  unfolding *one-poly-def* **by** (*auto simp*: *pcompose-pCons*)

lemma *poly-pcompose*:
  poly (*pcompose p q*) *x* = *poly p* (*poly q x*)
  by (*induct p*) (*simp-all add*: *pcompose-pCons*)

lemma *degree-pcompose-le*:
  degree (*pcompose p q*) ≤ *degree p* ∗ *degree q*
apply (*induct p*, *simp*)
apply (*simp add*: *pcompose-pCons*, *clarify*)

apply (*rule degree-add-le*, *simp*)
apply (*rule order-trans* [*OF degree-mult-le*], *simp*)
done

lemma *pcompose-add*:
  fixes *p q r* :: *'a* :: {*comm-semiring-0*, *ab-semigroup-add*} *poly*
  shows *pcompose* (*p* + *q*) *r* = *pcompose p r* + *pcompose q r*
proof (*induction p q rule*: *poly-induct2*)
  case (*pCons a p b q*)
  have *pcompose* (*pCons a p* + *pCons b q*) *r* =
        [:*a* + *b*:] + *r* ∗ *pcompose p r* + *r* ∗ *pcompose q r*
    by (*simp-all add*: *pcompose-pCons pCons.IH algebra-simps*)
  also have [:*a* + *b*:] = [:*a*:] + [:*b*:] **by** *simp*
  also have ... + *r* ∗ *pcompose p r* + *r* ∗ *pcompose q r* =
              pcompose (*pCons a p*) *r* + *pcompose* (*pCons b q*) *r*
    by (*simp only*: *pcompose-pCons add-ac*)
  finally show *?case* **.**
qed *simp*

lemma *pcompose-uminus*:
  fixes *p r* :: *'a* :: *comm-ring poly*
  shows *pcompose* (−*p*) *r* = −*pcompose p r*
  by (*induction p*) (*simp-all add*: *pcompose-pCons*)

lemma *pcompose-diff*:
  fixes *p q r* :: *'a* :: *comm-ring poly*
  shows *pcompose* (*p* − *q*) *r* = *pcompose p r* − *pcompose q r*
  using *pcompose-add*[*of p* −*q*] **by** (*simp add*: *pcompose-uminus*)

lemma *pcompose-smult*:
  fixes *p r* :: *'a* :: *comm-semiring-0 poly*
  shows *pcompose* (*smult a p*) *r* = *smult a* (*pcompose p r*)
  by (*induction p*)
    (*simp-all add*: *pcompose-pCons pcompose-add smult-add-right*)

lemma *pcompose-mult*:
  fixes *p q r* :: *'a* :: *comm-semiring-0 poly*
  shows *pcompose* (*p* ∗ *q*) *r* = *pcompose p r* ∗ *pcompose q r*
  by (*induction p arbitrary*: *q*)
    (*simp-all add*: *pcompose-add pcompose-smult pcompose-pCons algebra-simps*)

lemma *pcompose-assoc*:
  pcompose p (*pcompose q r* :: *'a* :: *comm-semiring-0 poly*) =
    pcompose (*pcompose p q*) *r*
  by (*induction p arbitrary*: *q*)
    (*simp-all add*: *pcompose-pCons pcompose-add pcompose-mult*)

lemma *pcompose-idR*[*simp*]:
  fixes *p* :: *'a* :: *comm-semiring-1 poly*

```
  shows pcompose p [: 0, 1 :] = p
  by (induct p; simp add: pcompose-pCons)




lemma degree-mult-eq-0:
  fixes p q:: 'a :: semidom poly
  shows degree (p*q) = 0 ⟷ p=0 ∨ q=0 ∨ (p≠0 ∧ q≠0 ∧ degree p =0 ∧
degree q =0)
by (auto simp add:degree-mult-eq)

lemma pcompose-const[simp]:pcompose [:a:] q = [:a:] by (subst pcompose-pCons,simp)


lemma pcompose-0': pcompose p 0 = [:coeff p 0:]
  by (induct p) (auto simp add:pcompose-pCons)

lemma degree-pcompose:
  fixes p q:: 'a::semidom poly
  shows degree (pcompose p q) = degree p * degree q
proof (induct p)
  case 0
  thus ?case by auto
next
  case (pCons a p)
  have degree (q * pcompose p q) = 0 ⟹ ?case
    proof (cases p=0)
      case True
      thus ?thesis by auto
    next
      case False assume degree (q * pcompose p q) = 0
      hence degree q=0 ∨ pcompose p q=0 by (auto simp add: degree-mult-eq-0)
     moreover have ⟦pcompose p q=0;degree q≠0⟧ ⟹ False using pCons.hyps(2)
‹p≠0›
        proof -
          assume pcompose p q=0 degree q≠0
          hence degree p=0 using pCons.hyps(2) by auto
          then obtain a1 where p=[:a1:]
            by (metis degree-pCons-eq-if old.nat.distinct(2) pCons-cases)
          thus False using ‹pcompose p q=0› ‹p≠0› by auto
        qed
      ultimately have degree (pCons a p) * degree q=0 by auto
      moreover have degree (pcompose (pCons a p) q) = 0
        proof -
          have 0 = max (degree [:a:]) (degree (q*pcompose p q))
            using ‹degree (q * pcompose p q) = 0› by simp
          also have ... ≥ degree ([:a:] + q * pcompose p q)
            by (rule degree-add-le-max)
```

finally show *?thesis* **by** (*auto simp add*:*pcompose-pCons*)
qed
ultimately show *?thesis* **by** *simp*
qed
moreover have *degree* (*q* ∗ *pcompose p q*)>*0* ⟹ *?case*
proof −
assume *asm*:*0* < *degree* (*q* ∗ *pcompose p q*)
hence *p*≠*0 q*≠*0 pcompose p q*≠*0* **by** *auto*
have *degree* (*pcompose* (*pCons a p*) *q*) = *degree* (*q* ∗ *pcompose p q*)
unfolding *pcompose-pCons*
using *degree-add-eq-right*[*of* [:*a*:]] *asm* **by** *auto*
thus *?thesis*
using *pCons.hyps*(*2*) *degree-mult-eq*[*OF* ‹*q*≠*0*› ‹*pcompose p q*≠*0*›] **by** *auto*
qed
ultimately show *?case* **by** *blast*
qed

lemma *pcompose-eq-0*:
fixes *p q*:: ′*a* :: *semidom poly*
assumes *pcompose p q* = *0 degree q* > *0*
shows *p* = *0*
proof −
have *degree p*=*0* **using** *assms degree-pcompose*[*of p q*] **by** *auto*
then obtain *a* **where** *p*=[:*a*:]
by (*metis degree-pCons-eq-if gr0-conv-Suc neq0-conv pCons-cases*)
hence *a*=*0* **using** *assms*(*1*) **by** *auto*
thus *?thesis* **using** ‹*p*=[:*a*:]› **by** *simp*
qed

14.23 Leading coefficient

definition *lead-coeff* :: ′*a*::*zero poly* ⇒ ′*a* **where**
lead-coeff p= *coeff p* (*degree p*)

lemma *lead-coeff-pCons*[*simp*]:
p≠*0* ⟹*lead-coeff* (*pCons a p*) = *lead-coeff p*
p=*0* ⟹ *lead-coeff* (*pCons a p*) = *a*
unfolding *lead-coeff-def* **by** *auto*

lemma *lead-coeff-0*[*simp*]:*lead-coeff 0* =*0*
unfolding *lead-coeff-def* **by** *auto*

lemma *lead-coeff-mult*:
fixes *p q*::′*a* ::*idom poly*
shows *lead-coeff* (*p* ∗ *q*) = *lead-coeff p* ∗ *lead-coeff q*
by (*unfold lead-coeff-def*,*cases p*=*0* ∨ *q*=*0*,*auto simp add*:*coeff-mult-degree-sum degree-mult-eq*)

lemma *lead-coeff-add-le*:

assumes *degree p* < *degree q*
shows *lead-coeff* (*p*+*q*) = *lead-coeff q*
using *assms* **unfolding** *lead-coeff-def*
by (*metis coeff-add coeff-eq-0 monoid-add-class.add.left-neutral degree-add-eq-right*)

lemma *lead-coeff-minus*:
lead-coeff (−*p*) = − *lead-coeff p*
by (*metis coeff-minus degree-minus lead-coeff-def*)

lemma *lead-coeff-comp*:
fixes *p q*:: *'a*::*idom poly*
assumes *degree q* > *0*
shows *lead-coeff* (*pcompose p q*) = *lead-coeff p* ∗ *lead-coeff q* ˆ (*degree p*)
proof (*induct p*)
case *0*
thus *?case* **unfolding** *lead-coeff-def* **by** *auto*
next
case (*pCons a p*)
have *degree* (*q* ∗ *pcompose p q*) = *0* =⇒ *?case*
proof −
assume *degree* (*q* ∗ *pcompose p q*) = *0*
hence *pcompose p q* = *0* **by** (*metis assms degree-0 degree-mult-eq-0 neq0-conv*)
hence *p*=*0* **using** *pcompose-eq-0*[*OF* - ‹*degree q* > *0*›] **by** *simp*
thus *?thesis* **by** *auto*
qed
moreover have *degree* (*q* ∗ *pcompose p q*) > *0* =⇒ *?case*
proof −
assume *degree* (*q* ∗ *pcompose p q*) > *0*
hence *lead-coeff* (*pcompose* (*pCons a p*) *q*) =*lead-coeff* (*q* ∗ *pcompose p q*)
by (*auto simp add*:*pcompose-pCons lead-coeff-add-le*)
also have ... = *lead-coeff q* ∗ (*lead-coeff p* ∗ *lead-coeff q* ˆ *degree p*)
using *pCons.hyps*(*2*) *lead-coeff-mult*[*of q pcompose p q*] **by** *simp*
also have ... = *lead-coeff p* ∗ *lead-coeff q* ˆ (*degree p* + *1*)
by *auto*
finally show *?thesis* **by** *auto*
qed
ultimately show *?case* **by** *blast*
qed

lemma *lead-coeff-smult*:
lead-coeff (*smult c p* :: *'a* :: *idom poly*) = *c* ∗ *lead-coeff p*
proof −
have *smult c p* = [:*c*:] ∗ *p* **by** *simp*
also have *lead-coeff* . . . = *c* ∗ *lead-coeff p*
by (*subst lead-coeff-mult*) *simp-all*
finally show *?thesis* **.**
qed

lemma *lead-coeff-1* [*simp*]: *lead-coeff 1* = *1*
  by (*simp add*: *lead-coeff-def*)

lemma *lead-coeff-of-nat* [*simp*]:
  lead-coeff (*of-nat n*) = (*of-nat n* :: *'a* :: {*comm-semiring-1*,*semiring-char-0*})
  by (*induction n*) (*simp-all add*: *lead-coeff-def of-nat-poly*)

lemma *lead-coeff-numeral* [*simp*]:
  lead-coeff (*numeral n*) = *numeral n*
  unfolding *lead-coeff-def*
  by (*subst of-nat-numeral* [*symmetric*], *subst of-nat-poly*) *simp*

lemma *lead-coeff-power*:
  lead-coeff (*p* ^ *n* :: *'a* :: *idom poly*) = *lead-coeff p* ^ *n*
  by (*induction n*) (*simp-all add*: *lead-coeff-mult*)

lemma *lead-coeff-nonzero*: *p* ≠ *0* ⟹ *lead-coeff p* ≠ *0*
  by (*simp add*: *lead-coeff-def*)

14.24 Derivatives of univariate polynomials

function *pderiv* :: (*'a* :: *semidom*) *poly* ⇒ *'a poly*
where
  [*simp del*]: *pderiv* (*pCons a p*) = (*if p* = *0 then 0 else p* + *pCons 0* (*pderiv p*))
  by (*auto intro*: *pCons-cases*)

termination *pderiv*
  by (*relation measure degree*) *simp-all*

lemma *pderiv-0* [*simp*]:
  pderiv 0 = *0*
  using *pderiv.simps* [*of 0 0*] **by** *simp*

lemma *pderiv-pCons*:
  pderiv (*pCons a p*) = *p* + *pCons 0* (*pderiv p*)
  by (*simp add*: *pderiv.simps*)

lemma *pderiv-1* [*simp*]: *pderiv 1* = *0*
  unfolding *one-poly-def* **by** (*simp add*: *pderiv-pCons*)

lemma *pderiv-of-nat* [*simp*]: *pderiv* (*of-nat n*) = *0*
  and *pderiv-numeral* [*simp*]: *pderiv* (*numeral m*) = *0*
  by (*simp-all add*: *of-nat-poly numeral-poly pderiv-pCons*)

lemma *coeff-pderiv*: *coeff* (*pderiv p*) *n* = *of-nat* (*Suc n*) ∗ *coeff p* (*Suc n*)
  by (*induct p arbitrary*: *n*)
    (*auto simp add*: *pderiv-pCons coeff-pCons algebra-simps split*: *nat.split*)

fun *pderiv-coeffs-code* :: (*'a* :: *semidom*) ⇒ *'a list* ⇒ *'a list* **where**

pderiv-coeffs-code f (*x* # *xs*) = *cCons* (*f* ∗ *x*) (*pderiv-coeffs-code* (*f*+*1*) *xs*)
| *pderiv-coeffs-code f* [] = []

definition *pderiv-coeffs* :: (′*a* :: *semidom*) *list* ⇒ ′*a list* **where**
pderiv-coeffs xs = *pderiv-coeffs-code 1* (*tl xs*)

lemma *pderiv-coeffs-code*:
nth-default 0 (*pderiv-coeffs-code f xs*) *n* = (*f* + *of-nat n*) ∗ (*nth-default 0 xs n*)
proof (*induct xs arbitrary*: *f n*)
case (*Cons x xs f n*)
show *?case*
proof (*cases n*)
case *0*
thus *?thesis* **by** (*cases pderiv-coeffs-code* (*f* + *1*) *xs* = [] ∧ *f* ∗ *x* = *0*, *auto simp*: *cCons-def*)
next
case (*Suc m*) **note** *n* = *this*
show *?thesis*
proof (*cases pderiv-coeffs-code* (*f* + *1*) *xs* = [] ∧ *f* ∗ *x* = *0*)
case *False*
hence *nth-default 0* (*pderiv-coeffs-code f* (*x* # *xs*)) *n* =
nth-default 0 (*pderiv-coeffs-code* (*f* + *1*) *xs*) *m*
by (*auto simp*: *cCons-def n*)
also have . . . = (*f* + *of-nat n*) ∗ (*nth-default 0 xs m*)
unfolding *Cons* **by** (*simp add*: *n add-ac*)
finally show *?thesis* **by** (*simp add*: *n*)
next
case *True*
{
fix *g*
have *pderiv-coeffs-code g xs* = [] ⟹ *g* + *of-nat m* = *0* ∨ *nth-default 0 xs m* = *0*
proof (*induct xs arbitrary*: *g m*)
case (*Cons x xs g*)
from *Cons*(*2*) **have** *empty*: *pderiv-coeffs-code* (*g* + *1*) *xs* = []
and *g*: (*g* = *0* ∨ *x* = *0*)
by (*auto simp*: *cCons-def split*: *if-splits*)
note *IH* = *Cons*(*1*)[*OF empty*]
from *IH*[*of m*] *IH*[*of m* − *1*] *g*
show *?case* **by** (*cases m*, *auto simp*: *field-simps*)
qed *simp*
} **note** *empty* = *this*
from *True* **have** *nth-default 0* (*pderiv-coeffs-code f* (*x* # *xs*)) *n* = *0*
by (*auto simp*: *cCons-def n*)
moreover have (*f* + *of-nat n*) ∗ *nth-default 0* (*x* # *xs*) *n* = *0* **using** *True*
by (*simp add*: *n*, *insert empty*[*of f*+*1*], *auto simp*: *field-simps*)
ultimately show *?thesis* **by** *simp*
qed

qed
qed *simp*

lemma *map-upt-Suc*: *map f* [*0* ..< *Suc n*] = *f 0* # *map* (λ *i*. *f* (*Suc i*)) [*0* ..< *n*]
by (*induct n arbitrary*: *f*, *auto*)

lemma *coeffs-pderiv-code* [*code abstract*]:
coeffs (*pderiv p*) = *pderiv-coeffs* (*coeffs p*) **unfolding** *pderiv-coeffs-def*
proof (*rule coeffs-eqI*, *unfold pderiv-coeffs-code coeff-pderiv*, *goal-cases*)
case (*1 n*)
have *id*: *coeff p* (*Suc n*) = *nth-default 0* (*map* (λ*i*. *coeff p* (*Suc i*)) [*0*..<*degree p*]) *n*
by (*cases n* < *degree p*, *auto simp*: *nth-default-def coeff-eq-0*)
show *?case* **unfolding** *coeffs-def map-upt-Suc* **by** (*auto simp*: *id*)
next
case *2*
obtain *n xs* **where** *id*: *tl* (*coeffs p*) = *xs* (*1* :: *'a*) = *n* **by** *auto*
from *2* **show** *?case*
unfolding *id* **by** (*induct xs arbitrary*: *n*, *auto simp*: *cCons-def*)
qed

context
assumes *SORT-CONSTRAINT*(*'a*::{*semidom*, *semiring-char-0*})
begin

lemma *pderiv-eq-0-iff*:
pderiv (*p* :: *'a poly*) = *0* ⟷ *degree p* = *0*
apply (*rule iffI*)
apply (*cases p*, *simp*)
apply (*simp add*: *poly-eq-iff coeff-pderiv del*: *of-nat-Suc*)
apply (*simp add*: *poly-eq-iff coeff-pderiv coeff-eq-0*)
done

lemma *degree-pderiv*: *degree* (*pderiv* (*p* :: *'a poly*)) = *degree p* − *1*
apply (*rule order-antisym* [*OF degree-le*])
apply (*simp add*: *coeff-pderiv coeff-eq-0*)
apply (*cases degree p*, *simp*)
apply (*rule le-degree*)
apply (*simp add*: *coeff-pderiv del*: *of-nat-Suc*)
apply (*metis degree-0 leading-coeff-0-iff nat.distinct*(*1*))
done

lemma *not-dvd-pderiv*:
assumes *degree* (*p* :: *'a poly*) ≠ *0*
shows ¬ *p dvd pderiv p*
proof
assume *dvd*: *p dvd pderiv p*
then obtain *q* **where** *p*: *pderiv p* = *p* ∗ *q* **unfolding** *dvd-def* **by** *auto*
from *dvd* **have** *le*: *degree p* ≤ *degree* (*pderiv p*)

```
    by (simp add: assms dvd-imp-degree-le pderiv-eq-0-iff)
  from this[unfolded degree-pderiv] assms show False by auto
qed

lemma dvd-pderiv-iff [simp]: (p :: 'a poly) dvd pderiv p ⟷ degree p = 0
  using not-dvd-pderiv[of p] by (auto simp: pderiv-eq-0-iff [symmetric])

end

lemma pderiv-singleton [simp]: pderiv [:a:] = 0
by (simp add: pderiv-pCons)

lemma pderiv-add: pderiv (p + q) = pderiv p + pderiv q
by (rule poly-eqI, simp add: coeff-pderiv algebra-simps)

lemma pderiv-minus: pderiv (− p :: 'a :: idom poly) = − pderiv p
by (rule poly-eqI, simp add: coeff-pderiv algebra-simps)

lemma pderiv-diff: pderiv (p − q) = pderiv p − pderiv q
by (rule poly-eqI, simp add: coeff-pderiv algebra-simps)

lemma pderiv-smult: pderiv (smult a p) = smult a (pderiv p)
by (rule poly-eqI, simp add: coeff-pderiv algebra-simps)

lemma pderiv-mult: pderiv (p ∗ q) = p ∗ pderiv q + q ∗ pderiv p
by (induct p) (auto simp: pderiv-add pderiv-smult pderiv-pCons algebra-simps)

lemma pderiv-power-Suc:
  pderiv (p ^ Suc n) = smult (of-nat (Suc n)) (p ^ n) ∗ pderiv p
apply (induct n)
apply simp
apply (subst power-Suc)
apply (subst pderiv-mult)
apply (erule ssubst)
apply (simp only: of-nat-Suc smult-add-left smult-1-left)
apply (simp add: algebra-simps)
done

lemma pderiv-setprod: pderiv (setprod f (as)) =
  (∑ a ∈ as. setprod f (as − {a}) ∗ pderiv (f a))
proof (induct as rule: infinite-finite-induct)
  case (insert a as)
  hence id: setprod f (insert a as) = f a ∗ setprod f as
    ⋀ g. setsum g (insert a as) = g a + setsum g as
    insert a as − {a} = as
    by auto
  {
    fix b
    assume b ∈ as
```

hence *id2*: *insert a as* − {*b*} = *insert a* (*as* − {*b*}) **using** ‹*a* ∉ *as*› **by** *auto*
have *setprod f* (*insert a as* − {*b*}) = *f a* ∗ *setprod f* (*as* − {*b*})
unfolding *id2*
by (*subst setprod.insert*, *insert insert*, *auto*)
} **note** *id2* = *this*
show *?case*
unfolding *id pderiv-mult insert*(*3*) *setsum-right-distrib*
by (*auto simp add*: *ac-simps id2 intro*!: *setsum.cong*)
qed *auto*

lemma *DERIV-pow2*: *DERIV* (%*x*. *x* ˆ *Suc n*) *x* :> *real* (*Suc n*) ∗ (*x* ˆ *n*)
by (*rule DERIV-cong*, *rule DERIV-pow*, *simp*)
declare *DERIV-pow2* [*simp*] *DERIV-pow* [*simp*]

lemma *DERIV-add-const*: *DERIV f x* :> *D* ==> *DERIV* (%*x*. *a* + *f x* :: ′*a*::*real-normed-field*) *x* :> *D*
by (*rule DERIV-cong*, *rule DERIV-add*, *auto*)

lemma *poly-DERIV* [*simp*]: *DERIV* (%*x*. *poly p x*) *x* :> *poly* (*pderiv p*) *x*
by (*induct p*, *auto intro*!: *derivative-eq-intros simp add*: *pderiv-pCons*)

lemma *continuous-on-poly* [*continuous-intros*]:
fixes *p* :: ′*a* :: {*real-normed-field*} *poly*
assumes *continuous-on A f*
shows *continuous-on A* (λ*x*. *poly p* (*f x*))
proof −
have *continuous-on A* (λ*x*. (∑ *i*≤*degree p*. (*f x*) ˆ *i* ∗ *coeff p i*))
by (*intro continuous-intros assms*)
also have . . . = (λ*x*. *poly p* (*f x*)) **by** (*intro ext*) (*simp add*: *poly-altdef mult-ac*)
finally show *?thesis* **.**
qed

Consequences of the derivative theorem above

lemma *poly-differentiable*[*simp*]: (%*x*. *poly p x*) *differentiable* (*at x*::*real filter*)
apply (*simp add*: *real-differentiable-def*)
apply (*blast intro*: *poly-DERIV*)
done

lemma *poly-isCont*[*simp*]: *isCont* (%*x*. *poly p x*) (*x*::*real*)
by (*rule poly-DERIV* [*THEN DERIV-isCont*])

lemma *poly-IVT-pos*: [| *a* < *b*; *poly p* (*a*::*real*) < *0*; *0* < *poly p b* |]
==> ∃ *x*. *a* < *x* & *x* < *b* & (*poly p x* = *0*)
using *IVT-objl* [*of poly p a 0 b*]
by (*auto simp add*: *order-le-less*)

lemma *poly-IVT-neg*: [| (*a*::*real*) < *b*; *0* < *poly p a*; *poly p b* < *0* |]
==> ∃ *x*. *a* < *x* & *x* < *b* & (*poly p x* = *0*)
by (*insert poly-IVT-pos* [**where** *p* = − *p*]) *simp*

lemma *poly-IVT*:
  fixes *p*::*real poly*
  assumes $a<b$ **and** *poly p a* ∗ *poly p b* < *0*
  shows ∃ *x*>*a*. *x* < *b* ∧ *poly p x* = *0*
by (*metis assms*(*1*) *assms*(*2*) *less-not-sym mult-less-0-iff poly-IVT-neg poly-IVT-pos*)

lemma *poly-MVT*: (*a*::*real*) < *b* ==>
    ∃ *x*. *a* < *x* & *x* < *b* & (*poly p b* − *poly p a* = (*b* − *a*) ∗ *poly* (*pderiv p*) *x*)
using *MVT* [*of a b poly p*]
apply *auto*
apply (*rule-tac x* = *z* **in** *exI*)
apply (*auto simp add*: *mult-left-cancel poly-DERIV* [*THEN DERIV-unique*])
done

lemma *poly-MVT'*:
  assumes {*min a b*..*max a b*} ⊆ *A*
  shows ∃ *x*∈*A*. *poly p b* − *poly p a* = (*b* − *a*) ∗ *poly* (*pderiv p*) (*x*::*real*)
proof (*cases a b rule*: *linorder-cases*)
  case *less*
  from *poly-MVT*[*OF less*, *of p*] **guess** *x* **by** (*elim exE conjE*)
  thus *?thesis* **by** (*intro bexI*[*of - x*]) (*auto intro*!: *subsetD*[*OF assms*])

next
  case *greater*
  from *poly-MVT*[*OF greater*, *of p*] **guess** *x* **by** (*elim exE conjE*)
  thus *?thesis* **by** (*intro bexI*[*of - x*]) (*auto simp*: *algebra-simps intro*!: *subsetD*[*OF assms*])
qed (*insert assms*, *auto*)

lemma *poly-pinfty-gt-lc*:
  fixes *p*:: *real poly*
  assumes *lead-coeff p* > *0*
  shows ∃ *n*. ∀ *x* ≥ *n*. *poly p x* ≥ *lead-coeff p* **using** *assms*
proof (*induct p*)
  case *0*
  thus *?case* **by** *auto*
next
  case (*pCons a p*)
  have ⟦*a*≠*0*;*p*=*0*⟧ ⟹ *?case* **by** *auto*
  moreover have *p*≠*0* ⟹ *?case*
    proof −
      assume *p*≠*0*
      then obtain *n1* **where** *gte-lcoeff*:∀ *x*≥*n1*. *lead-coeff p* ≤ *poly p x* **using** *that pCons* **by** *auto*
      have *gt-0*:*lead-coeff p* >*0* **using** *pCons*(*3*) ‹*p*≠*0*› **by** *auto*
      def *n*≡*max n1* (*1*+ |*a*|/(*lead-coeff p*))
      show *?thesis*
        proof (*rule-tac x*=*n* **in** *exI*,*rule*,*rule*)

fix *x* **assume** $n \leq x$
hence *lead-coeff p* ≤ *poly p x*
using *gte-lcoeff* **unfolding** *n-def* **by** *auto*
hence $|a|/(lead\text{-}coeff\ p) \geq |a|/(poly\ p\ x)$ **and** *poly p x>0* **using** *gt-0*
by (*intro frac-le,auto*)
hence $x \geq 1 + |a|/(poly\ p\ x)$ **using** ‹*n≤x*›[*unfolded n-def*] **by** *auto*
thus *lead-coeff* (*pCons a p*) ≤ *poly* (*pCons a p*) *x*
using ‹*lead-coeff p* ≤ *poly p x*› ‹*poly p x>0*› ‹*p≠0*›
by (*auto simp add:field-simps*)
qed
qed
ultimately show *?case* **by** *fastforce*
qed

14.25 Algebraic numbers

Algebraic numbers can be defined in two equivalent ways: all real numbers that are roots of rational polynomials or of integer polynomials. The Algebraic-Numbers AFP entry uses the rational definition, but we need the integer definition.

The equivalence is obvious since any rational polynomial can be multiplied with the LCM of its coefficients, yielding an integer polynomial with the same roots.

14.26 Algebraic numbers

definition *algebraic* :: *'a* :: *field-char-0* ⇒ *bool* **where**
algebraic x ⟷ (∃ *p*. (∀ *i*. *coeff p i* ∈ ℤ) ∧ *p* ≠ *0* ∧ *poly p x* = *0*)

lemma *algebraicI*:
assumes ⋀*i*. *coeff p i* ∈ ℤ *p* ≠ *0 poly p x* = *0*
shows *algebraic x*
using *assms* **unfolding** *algebraic-def* **by** *blast*

lemma *algebraicE*:
assumes *algebraic x*
obtains *p* **where** ⋀*i*. *coeff p i* ∈ ℤ *p* ≠ *0 poly p x* = *0*
using *assms* **unfolding** *algebraic-def* **by** *blast*

lemma *quotient-of-denom-pos'*: *snd* (*quotient-of x*) > *0*
using *quotient-of-denom-pos*[*OF surjective-pairing*] **.**

lemma *of-int-div-in-Ints*:
b dvd a ⟹ *of-int a div of-int b* ∈ (ℤ :: *'a* :: *ring-div set*)
proof (*cases of-int b* = (*0* :: *'a*))
assume *b dvd a of-int b* ≠ (*0*::*'a*)
then obtain *c* **where** *a* = *b* ∗ *c* **by** (*elim dvdE*)
with ‹*of-int b* ≠ (*0*::*'a*)› **show** *?thesis* **by** *simp*

```
qed auto

lemma of-int-divide-in-Ints:
  b dvd a ⟹ of-int a / of-int b ∈ (ℤ :: 'a :: field set)
proof (cases of-int b = (0 :: 'a))
  assume b dvd a of-int b ≠ (0::'a)
  then obtain c where a = b * c by (elim dvdE)
  with ‹of-int b ≠ (0::'a)› show ?thesis by simp
qed auto

lemma algebraic-altdef:
  fixes p :: 'a :: field-char-0 poly
  shows algebraic x ⟷ (∃ p. (∀ i. coeff p i ∈ ℚ) ∧ p ≠ 0 ∧ poly p x = 0)
proof safe
  fix p assume rat: ∀ i. coeff p i ∈ ℚ and root: poly p x = 0 and nz: p ≠ 0
  def cs ≡ coeffs p
  from rat have ∀ c∈range (coeff p). ∃ c'. c = of-rat c' unfolding Rats-def by
blast
  then obtain f where f: ⋀i. coeff p i = of-rat (f (coeff p i))
    by (subst (asm) bchoice-iff) blast
  def cs' ≡ map (quotient-of ∘ f) (coeffs p)
  def d ≡ Lcm (set (map snd cs'))
  def p' ≡ smult (of-int d) p

  have ∀ n. coeff p' n ∈ ℤ
  proof
    fix n :: nat
    show coeff p' n ∈ ℤ
    proof (cases n ≤ degree p)
      case True
      def c ≡ coeff p n
      def a ≡ fst (quotient-of (f (coeff p n))) and b ≡ snd (quotient-of (f (coeff
p n)))
      have b-pos: b > 0 unfolding b-def using quotient-of-denom-pos' by simp
      have coeff p' n = of-int d * coeff p n by (simp add: p'-def)
      also have coeff p n = of-rat (of-int a / of-int b) unfolding a-def b-def
        by (subst quotient-of-div [of f (coeff p n), symmetric])
           (simp-all add: f [symmetric])
      also have of-int d * … = of-rat (of-int (a*d) / of-int b)
        by (simp add: of-rat-mult of-rat-divide)
      also from nz True have b ∈ snd ‘ set cs' unfolding cs'-def
        by (force simp: o-def b-def coeffs-def simp del: upt-Suc)
      hence b dvd (a * d) unfolding d-def by simp
      hence of-int (a * d) / of-int b ∈ (ℤ :: rat set)
        by (rule of-int-divide-in-Ints)
      hence of-rat (of-int (a * d) / of-int b) ∈ ℤ by (elim Ints-cases) auto
      finally show ?thesis .
    qed (auto simp: p'-def not-le coeff-eq-0)
  qed
```

moreover have *set* (*map snd cs′*) ⊆ {*0*<..}
unfolding *cs′-def* **using** *quotient-of-denom-pos′* **by** (*auto simp*: *coeffs-def simp del*: *upt-Suc*)
hence $d \neq 0$ **unfolding** *d-def* **by** (*induction cs′*) *simp-all*
with *nz* **have** $p' \neq 0$ **by** (*simp add*: *p′-def*)
moreover from *root* **have** *poly p′ x* = *0* **by** (*simp add*: *p′-def*)
ultimately show *algebraic x* **unfolding** *algebraic-def* **by** *blast*
next

assume *algebraic x*
then obtain *p* **where** *p*: ⋀*i*. *coeff p i* ∈ ℤ *poly p x* = *0 p* ≠ *0*
by (*force simp*: *algebraic-def*)
moreover have *coeff p i* ∈ ℤ ⟹ *coeff p i* ∈ ℚ **for** *i* **by** (*elim Ints-cases*) *simp*
ultimately show (∃ *p*. (∀ *i*. *coeff p i* ∈ ℚ) ∧ *p* ≠ *0* ∧ *poly p x* = *0*) **by** *auto*
qed

Lemmas for Derivatives

lemma *order-unique-lemma*:
fixes *p* :: ′*a*::*idom poly*
assumes [:−*a*, *1*:] ˆ *n dvd p* ¬ [:−*a*, *1*:] ˆ *Suc n dvd p*
shows *n* = *order a p*
unfolding *Polynomial*.*order-def*
apply (*rule Least-equality* [*symmetric*])
apply (*fact assms*)
apply (*rule classical*)
apply (*erule notE*)
unfolding *not-less-eq-eq*
using *assms*(*1*) **apply** (*rule power-le-dvd*)
apply *assumption*
done

lemma *lemma-order-pderiv1*:
pderiv ([:− *a*, *1*:] ˆ *Suc n* ∗ *q*) = [:− *a*, *1*:] ˆ *Suc n* ∗ *pderiv q* +
smult (*of-nat* (*Suc n*)) (*q* ∗ [:− *a*, *1*:] ˆ *n*)
apply (*simp only*: *pderiv-mult pderiv-power-Suc*)
apply (*simp del*: *power-Suc of-nat-Suc add*: *pderiv-pCons*)
done

lemma *lemma-order-pderiv*:
fixes *p* :: ′*a* :: *field-char-0 poly*
assumes *n*: *0* < *n*
and *pd*: *pderiv p* ≠ *0*
and *pe*: *p* = [:− *a*, *1*:] ˆ *n* ∗ *q*
and *nd*: ~ [:− *a*, *1*:] *dvd q*
shows *n* = *Suc* (*order a* (*pderiv p*))
using *n*
proof −
have *pderiv* ([:− *a*, *1*:] ˆ *n* ∗ *q*) ≠ *0*

using *assms* **by** *auto*
obtain *n′* **where** *n = Suc n′ 0 < Suc n′ pderiv* ([:− *a*, *1*:] ˆ *Suc n′* ∗ *q*) ≠ *0*
using *assms* **by** (*cases n*) *auto*
have ∗: !!*k l*. *k dvd k* ∗ *pderiv q* + *smult* (*of-nat* (*Suc n′*)) *l* ⟹ *k dvd l*
by (*auto simp del*: *of-nat-Suc simp*: *dvd-add-right-iff dvd-smult-iff*)
have *n′ = order a* (*pderiv* ([:− *a*, *1*:] ˆ *Suc n′* ∗ *q*))
proof (*rule order-unique-lemma*)
show [:− *a*, *1*:] ˆ *n′ dvd pderiv* ([:− *a*, *1*:] ˆ *Suc n′* ∗ *q*)
apply (*subst lemma-order-pderiv1*)
apply (*rule dvd-add*)
apply (*metis dvdI dvd-mult2 power-Suc2*)
apply (*metis dvd-smult dvd-triv-right*)
done
next
show ¬ [:− *a*, *1*:] ˆ *Suc n′ dvd pderiv* ([:− *a*, *1*:] ˆ *Suc n′* ∗ *q*)
apply (*subst lemma-order-pderiv1*)
by (*metis* ∗ *nd dvd-mult-cancel-right power-not-zero pCons-eq-0-iff power-Suc zero-neq-one*)
qed
then show *?thesis*
by (*metis* ‹*n = Suc n′*› *pe*)
qed

lemma *order-decomp*:
assumes *p* ≠ *0*
shows ∃ *q*. *p* = [:− *a*, *1*:] ˆ *order a p* ∗ *q* ∧ ¬ [:− *a*, *1*:] *dvd q*
proof −
from *assms* **have** *A*: [:− *a*, *1*:] ˆ *order a p dvd p*
and *B*: ¬ [:− *a*, *1*:] ˆ *Suc* (*order a p*) *dvd p* **by** (*auto dest*: *order*)
from *A* **obtain** *q* **where** *C*: *p* = [:− *a*, *1*:] ˆ *order a p* ∗ *q* **..**
with *B* **have** ¬ [:− *a*, *1*:] ˆ *Suc* (*order a p*) *dvd* [:− *a*, *1*:] ˆ *order a p* ∗ *q*
by *simp*
then have ¬ [:− *a*, *1*:] ˆ *order a p* ∗ [:− *a*, *1*:] *dvd* [:− *a*, *1*:] ˆ *order a p* ∗ *q*
by *simp*
then have *D*: ¬ [:− *a*, *1*:] *dvd q*
using *idom-class.dvd-mult-cancel-left* [*of* [:− *a*, *1*:] ˆ *order a p* [:− *a*, *1*:] *q*]
by *auto*
from *C D* **show** *?thesis* **by** *blast*
qed

lemma *order-pderiv*:
⟦*pderiv p* ≠ *0*; *order a* (*p* :: ′*a* :: *field-char-0 poly*) ≠ *0*⟧ ⟹
(*order a p* = *Suc* (*order a* (*pderiv p*)))
apply (*case-tac p* = *0*, *simp*)
apply (*drule-tac a* = *a* **and** *p* = *p* **in** *order-decomp*)
using *neq0-conv*
apply (*blast intro*: *lemma-order-pderiv*)
done

lemma *order-mult*: *p* ∗ *q* ≠ *0* ⟹ *order a* (*p* ∗ *q*) = *order a p* + *order a q*
proof −
  def *i* ≡ *order a p*
  def *j* ≡ *order a q*
  def *t* ≡ [:−*a*, *1*:]
  have *t-dvd-iff*: ⋀*u*. *t dvd u* ⟷ *poly u a* = *0*
    unfolding *t-def* **by** (*simp add*: *dvd-iff-poly-eq-0*)
  assume *p* ∗ *q* ≠ *0*
  then show *order a* (*p* ∗ *q*) = *i* + *j*
    apply *clarsimp*
    apply (*drule order* [**where** *a*=*a* **and** *p*=*p*, *folded i-def t-def*])
    apply (*drule order* [**where** *a*=*a* **and** *p*=*q*, *folded j-def t-def*])
    apply *clarify*
    apply (*erule dvdE*)+
    apply (*rule order-unique-lemma* [*symmetric*], *fold t-def*)
    apply (*simp-all add*: *power-add t-dvd-iff*)
    done
qed

lemma *order-smult*:
  assumes *c* ≠ *0*
  shows *order x* (*smult c p*) = *order x p*
proof (*cases p* = *0*)
  case *False*
  have *smult c p* = [:*c*:] ∗ *p* **by** *simp*
  also from *assms False* **have** *order x* ... = *order x* [:*c*:] + *order x p*
    by (*subst order-mult*) *simp-all*
  also from *assms* **have** *order x* [:*c*:] = *0* **by** (*intro order-0I*) *auto*
  finally show *?thesis* **by** *simp*
qed *simp*

lemma *order-1-eq-0* [*simp*]:*order x 1* = *0*
  by (*metis order-root poly-1 zero-neq-one*)

lemma *order-power-n-n*: *order a* ([:−*a*,*1*:] ^*n*)=*n*
proof (*induct n*)
  case *0*
  thus *?case* **by** (*metis order-root poly-1 power-0 zero-neq-one*)
next
  case (*Suc n*)
  have *order a* ([:− *a*, *1*:] ^ *Suc n*)=*order a* ([:− *a*, *1*:] ^ *n*) + *order a* [:−*a*,*1*:]
   by (*metis* (*no-types*, *hide-lams*) *One-nat-def add-Suc-right monoid-add-class.add.right-neutral*
     one-neq-zero order-mult pCons-eq-0-iff power-add power-eq-0-iff power-one-right)
  moreover have *order a* [:−*a*,*1*:]=*1* **unfolding** *order-def*
    proof (*rule Least-equality*,*rule ccontr*)
      assume ¬ ¬ [:− *a*, *1*:] ^ *Suc 1 dvd* [:− *a*, *1*:]
      hence [:− *a*, *1*:] ^ *Suc 1 dvd* [:− *a*, *1*:] **by** *simp*

```
    hence degree ([:- a, 1:] ^ Suc 1) ≤ degree ([:- a, 1:] )
      by (rule dvd-imp-degree-le,auto)
    thus False by auto
  next
    fix y assume asm:¬ [:- a, 1:] ^ Suc y dvd [:- a, 1:]
    show 1 ≤ y
      proof (rule ccontr)
        assume ¬ 1 ≤ y
        hence y=0 by auto
        hence [:- a, 1:] ^ Suc y dvd [:- a, 1:] by auto
        thus False using asm by auto
      qed
  qed
 ultimately show ?case using Suc by auto
qed
```

Now justify the standard squarefree decomposition, i.e. f / gcd(f,f').

```
lemma order-divides: [:-a, 1:] ^ n dvd p ⟷ p = 0 ∨ n ≤ order a p
apply (cases p = 0, auto)
apply (drule order-2 [where a=a and p=p])
apply (metis not-less-eq-eq power-le-dvd)
apply (erule power-le-dvd [OF order-1])
done

lemma poly-squarefree-decomp-order:
  assumes pderiv (p :: 'a :: field-char-0 poly) ≠ 0
  and p: p = q * d
  and p': pderiv p = e * d
  and d: d = r * p + s * pderiv p
  shows order a q = (if order a p = 0 then 0 else 1)
proof (rule classical)
  assume 1: order a q ≠ (if order a p = 0 then 0 else 1)
  from ‹pderiv p ≠ 0› have p ≠ 0 by auto
  with p have order a p = order a q + order a d
    by (simp add: order-mult)
  with 1 have order a p ≠ 0 by (auto split: if-splits)
  have order a (pderiv p) = order a e + order a d
    using ‹pderiv p ≠ 0› ‹pderiv p = e * d› by (simp add: order-mult)
  have order a p = Suc (order a (pderiv p))
    using ‹pderiv p ≠ 0› ‹order a p ≠ 0› by (rule order-pderiv)
  have d ≠ 0 using ‹p ≠ 0› ‹p = q * d› by simp
  have ([:-a, 1:] ^ (order a (pderiv p))) dvd d
    apply (simp add: d)
    apply (rule dvd-add)
    apply (rule dvd-mult)
    apply (simp add: order-divides ‹p ≠ 0›
          ‹order a p = Suc (order a (pderiv p))›)
    apply (rule dvd-mult)
    apply (simp add: order-divides)
```

done
then have *order a* (*pderiv p*) ≤ *order a d*
using ‹*d* ≠ *0*› **by** (*simp add*: *order-divides*)
show *?thesis*
using ‹*order a p* = *order a q* + *order a d*›
using ‹*order a* (*pderiv p*) = *order a e* + *order a d*›
using ‹*order a p* = *Suc* (*order a* (*pderiv p*))›
using ‹*order a* (*pderiv p*) ≤ *order a d*›
by *auto*
qed

lemma *poly-squarefree-decomp-order2*:
⟦*pderiv p* ≠ (*0* :: *'a* :: *field-char-0 poly*);
p = *q* ∗ *d*;
pderiv p = *e* ∗ *d*;
d = *r* ∗ *p* + *s* ∗ *pderiv p*
⟧ ⟹ ∀ *a*. *order a q* = (*if order a p* = *0 then 0 else 1*)
by (*blast intro*: *poly-squarefree-decomp-order*)

lemma *order-pderiv2*:
⟦*pderiv p* ≠ *0*; *order a* (*p* :: *'a* :: *field-char-0 poly*) ≠ *0*⟧
⟹ (*order a* (*pderiv p*) = *n*) = (*order a p* = *Suc n*)
by (*auto dest*: *order-pderiv*)

definition
rsquarefree :: *'a*::*idom poly* => *bool* **where**
rsquarefree p = (*p* ≠ *0* & (∀ *a*. (*order a p* = *0*) | (*order a p* = *1*)))

lemma *pderiv-iszero*: *pderiv p* = *0* ⟹ ∃ *h*. *p* = [:*h* :: *'a* :: {*semidom*,*semiring-char-0*}:]
apply (*simp add*: *pderiv-eq-0-iff*)
apply (*case-tac p*, *auto split*: *if-splits*)
done

lemma *rsquarefree-roots*:
fixes *p* :: *'a* :: *field-char-0 poly*
shows *rsquarefree p* = (∀ *a*. ¬(*poly p a* = *0* ∧ *poly* (*pderiv p*) *a* = *0*))
apply (*simp add*: *rsquarefree-def*)
apply (*case-tac p* = *0*, *simp*, *simp*)
apply (*case-tac pderiv p* = *0*)
apply *simp*
apply (*drule pderiv-iszero*, *clarsimp*)
apply (*metis coeff-0 coeff-pCons-0 degree-pCons-0 le0 le-antisym order-degree*)
apply (*force simp add*: *order-root order-pderiv2*)
done

lemma *poly-squarefree-decomp*:
assumes *pderiv* (*p* :: *'a* :: *field-char-0 poly*) ≠ *0*
and *p* = *q* ∗ *d*
and *pderiv p* = *e* ∗ *d*

and *d* = *r* ∗ *p* + *s* ∗ *pderiv p*
shows *rsquarefree q* & (∀ *a*. (*poly q a* = *0*) = (*poly p a* = *0*))
proof −
from ‹*pderiv p* ≠ *0*› **have** *p* ≠ *0* **by** *auto*
with ‹*p* = *q* ∗ *d*› **have** *q* ≠ *0* **by** *simp*
have ∀ *a*. *order a q* = (*if order a p* = *0 then 0 else 1*)
using *assms* **by** (*rule poly-squarefree-decomp-order2*)
with ‹*p* ≠ *0*› ‹*q* ≠ *0*› **show** *?thesis*
by (*simp add*: *rsquarefree-def order-root*)
qed

no-notation *cCons* (**infixr** ## *65*)

end

15 Abstract euclidean algorithm

theory *Euclidean-Algorithm*
imports ~~/*src*/*HOL*/*GCD* ~~/*src*/*HOL*/*Library*/*Polynomial*
begin

A Euclidean semiring is a semiring upon which the Euclidean algorithm can be implemented. It must provide:

- division with remainder
- a size function such that *size* (*a mod b*) < *size b* for any *b* ≠ (*0*::′*a*)

The existence of these functions makes it possible to derive gcd and lcm functions for any Euclidean semiring.

class *euclidean-semiring* = *semiring-div* + *normalization-semidom* +
fixes *euclidean-size* :: ′*a* ⇒ *nat*
assumes *size-0* [*simp*]: *euclidean-size 0* = *0*
assumes *mod-size-less*:
b ≠ *0* ⟹ *euclidean-size* (*a mod b*) < *euclidean-size b*
assumes *size-mult-mono*:
b ≠ *0* ⟹ *euclidean-size a* ≤ *euclidean-size* (*a* ∗ *b*)
begin

lemma *euclidean-division*:
fixes *a* :: ′*a* **and** *b* :: ′*a*
assumes *b* ≠ *0*
obtains *s* **and** *t* **where** *a* = *s* ∗ *b* + *t*
and *euclidean-size t* < *euclidean-size b*
proof −
from *div-mod-equality* [*of a b 0*]
have *a* = *a div b* ∗ *b* + *a mod b* **by** *simp*

```
  with that and assms show ?thesis by (auto simp add: mod-size-less)
qed

lemma dvd-euclidean-size-eq-imp-dvd:
 assumes a ≠ 0 and b-dvd-a: b dvd a and size-eq: euclidean-size a = euclidean-size
b
  shows a dvd b
proof (rule ccontr)
  assume ¬ a dvd b
  then have b mod a ≠ 0 by (simp add: mod-eq-0-iff-dvd)
  from b-dvd-a have b-dvd-mod: b dvd b mod a by (simp add: dvd-mod-iff)
  from b-dvd-mod obtain c where b mod a = b ∗ c unfolding dvd-def by blast
    with ‹b mod a ≠ 0› have c ≠ 0 by auto
  with ‹b mod a = b ∗ c› have euclidean-size (b mod a) ≥ euclidean-size b
      using size-mult-mono by force
  moreover from ‹¬ a dvd b› and ‹a ≠ 0›
  have euclidean-size (b mod a) < euclidean-size a
      using mod-size-less by blast
  ultimately show False using size-eq by simp
qed

function gcd-eucl :: 'a ⇒ 'a ⇒ 'a
where
  gcd-eucl a b = (if b = 0 then normalize a else gcd-eucl b (a mod b))
  by pat-completeness simp
termination
  by (relation measure (euclidean-size ∘ snd)) (simp-all add: mod-size-less)

declare gcd-eucl.simps [simp del]

lemma gcd-eucl-induct [case-names zero mod]:
  assumes H1: ⋀b. P b 0
  and H2: ⋀a b. b ≠ 0 ⟹ P b (a mod b) ⟹ P a b
  shows P a b
proof (induct a b rule: gcd-eucl.induct)
  case (1 a b)
  show ?case
  proof (cases b = 0)
    case True then show P a b by simp (rule H1)
  next
    case False
    then have P b (a mod b)
      by (rule 1.hyps)
    with ‹b ≠ 0› show P a b
      by (blast intro: H2)
  qed
qed

definition lcm-eucl :: 'a ⇒ 'a ⇒ 'a
```

where
lcm-eucl a b = *normalize* (*a* ∗ *b*) *div gcd-eucl a b*

definition *Lcm-eucl* :: *'a set* ⇒ *'a* — Somewhat complicated definition of Lcm that has the advantage of working for infinite sets as well
where
Lcm-eucl A = (*if* ∃ *l*. *l* ≠ *0* ∧ (∀ *a*∈*A*. *a dvd l*) *then*
let l = *SOME l*. *l* ≠ *0* ∧ (∀ *a*∈*A*. *a dvd l*) ∧ *euclidean-size l* =
(*LEAST n*. ∃ *l*. *l* ≠ *0* ∧ (∀ *a*∈*A*. *a dvd l*) ∧ *euclidean-size l* = *n*)
in normalize l
else 0)

definition *Gcd-eucl* :: *'a set* ⇒ *'a*
where
Gcd-eucl A = *Lcm-eucl* {*d*. ∀ *a*∈*A*. *d dvd a*}

declare *Lcm-eucl-def Gcd-eucl-def* [*code del*]

lemma *gcd-eucl-0*:
gcd-eucl a 0 = *normalize a*
by (*simp add*: *gcd-eucl.simps* [*of a 0*])

lemma *gcd-eucl-0-left*:
gcd-eucl 0 a = *normalize a*
by (*simp-all add*: *gcd-eucl-0 gcd-eucl.simps* [*of 0 a*])

lemma *gcd-eucl-non-0*:
b ≠ *0* ⟹ *gcd-eucl a b* = *gcd-eucl b* (*a mod b*)
by (*simp add*: *gcd-eucl.simps* [*of a b*] *gcd-eucl.simps* [*of b 0*])

lemma *gcd-eucl-dvd1* [*iff*]: *gcd-eucl a b dvd a*
and *gcd-eucl-dvd2* [*iff*]: *gcd-eucl a b dvd b*
by (*induct a b rule*: *gcd-eucl-induct*)
(*simp-all add*: *gcd-eucl-0 gcd-eucl-non-0 dvd-mod-iff*)

lemma *normalize-gcd-eucl* [*simp*]:
normalize (*gcd-eucl a b*) = *gcd-eucl a b*
by (*induct a b rule*: *gcd-eucl-induct*) (*simp-all add*: *gcd-eucl-0 gcd-eucl-non-0*)

lemma *gcd-eucl-greatest*:
fixes *k a b* :: *'a*
shows *k dvd a* ⟹ *k dvd b* ⟹ *k dvd gcd-eucl a b*
proof (*induct a b rule*: *gcd-eucl-induct*)
case (*zero a*) **from** *zero*(*1*) **show** *?case* **by** (*rule dvd-trans*) (*simp add*: *gcd-eucl-0*)
next
case (*mod a b*)
then show *?case*
by (*simp add*: *gcd-eucl-non-0 dvd-mod-iff*)
qed

lemma *eq-gcd-euclI*:
  fixes *gcd* :: *'a ⇒ 'a ⇒ 'a*
  assumes ⋀*a b. gcd a b dvd a* ⋀*a b. gcd a b dvd b* ⋀*a b. normalize* (*gcd a b*) = *gcd a b*
        ⋀*a b k. k dvd a* ⟹ *k dvd b* ⟹ *k dvd gcd a b*
  shows *gcd* = *gcd-eucl*
  by (*intro ext*, *rule associated-eqI*) (*simp-all add*: *gcd-eucl-greatest assms*)

lemma *gcd-eucl-zero* [*simp*]:
  gcd-eucl a b = *0* ⟷ *a* = *0* ∧ *b* = *0*
  by (*metis dvd-0-left dvd-refl gcd-eucl-dvd1 gcd-eucl-dvd2 gcd-eucl-greatest*)+

lemma *dvd-Lcm-eucl* [*simp*]: *a* ∈ *A* ⟹ *a dvd Lcm-eucl A*
  and *Lcm-eucl-least*: (⋀*a. a* ∈ *A* ⟹ *a dvd b*) ⟹ *Lcm-eucl A dvd b*
  and *unit-factor-Lcm-eucl* [*simp*]:
      unit-factor (*Lcm-eucl A*) = (*if Lcm-eucl A* = *0 then 0 else 1*)
proof −
  have (∀ *a*∈*A. a dvd Lcm-eucl A*) ∧ (∀ *l'*. (∀ *a*∈*A. a dvd l'*) ⟶ *Lcm-eucl A dvd l'*) ∧
    unit-factor (*Lcm-eucl A*) = (*if Lcm-eucl A* = *0 then 0 else 1*) (**is** *?thesis*)
  proof (*cases* ∃ *l. l* ≠ *0* ∧ (∀ *a*∈*A. a dvd l*))
    case *False*
    hence *Lcm-eucl A* = *0* **by** (*auto simp*: *Lcm-eucl-def*)
    with *False* **show** *?thesis* **by** *auto*
  next
    case *True*
    then obtain l_0 **where** l_0*-props*: l_0 ≠ *0* ∧ (∀ *a*∈*A. a dvd* l_0) **by** *blast*
    def *n* ≡ *LEAST n.* ∃ *l. l* ≠ *0* ∧ (∀ *a*∈*A. a dvd l*) ∧ *euclidean-size l* = *n*
    def *l* ≡ *SOME l. l* ≠ *0* ∧ (∀ *a*∈*A. a dvd l*) ∧ *euclidean-size l* = *n*
    have ∃ *l. l* ≠ *0* ∧ (∀ *a*∈*A. a dvd l*) ∧ *euclidean-size l* = *n*
      apply (*subst n-def*)
      apply (*rule LeastI*[*of - euclidean-size* l_0])
      apply (*rule exI*[*of -* l_0])
      apply (*simp add*: l_0*-props*)
      done
    from *someI-ex*[*OF this*] **have** *l* ≠ *0* **and** ∀ *a*∈*A. a dvd l* **and** *euclidean-size l* = *n*
      unfolding *l-def* **by** *simp-all*
    {
      fix *l'* **assume** ∀ *a*∈*A. a dvd l'*
        with ‹∀ *a*∈*A. a dvd l*› **have** ∀ *a*∈*A. a dvd gcd-eucl l l'* **by** (*auto intro*: *gcd-eucl-greatest*)
      moreover from ‹*l* ≠ *0*› **have** *gcd-eucl l l'* ≠ *0* **by** *simp*
      ultimately have ∃ *b. b* ≠ *0* ∧ (∀ *a*∈*A. a dvd b*) ∧
              euclidean-size b = *euclidean-size* (*gcd-eucl l l'*)
        by (*intro exI*[*of - gcd-eucl l l'*], *auto*)
      hence *euclidean-size* (*gcd-eucl l l'*) ≥ *n* **by** (*subst n-def*) (*rule Least-le*)

moreover have *euclidean-size* (*gcd-eucl l l'*) ≤ *n*
proof −
have *gcd-eucl l l' dvd l* **by** *simp*
then obtain *a* **where** *l* = *gcd-eucl l l'* ∗ *a* **unfolding** *dvd-def* **by** *blast*
with ‹*l* ≠ *0*› **have** *a* ≠ *0* **by** *auto*
hence *euclidean-size* (*gcd-eucl l l'*) ≤ *euclidean-size* (*gcd-eucl l l'* ∗ *a*)
by (*rule size-mult-mono*)
also have *gcd-eucl l l'* ∗ *a* = *l* **using** ‹*l* = *gcd-eucl l l'* ∗ *a*› **..**
also note ‹*euclidean-size l* = *n*›
finally show *euclidean-size* (*gcd-eucl l l'*) ≤ *n* **.**
qed
ultimately have ∗: *euclidean-size l* = *euclidean-size* (*gcd-eucl l l'*)
by (*intro le-antisym*, *simp-all add*: ‹*euclidean-size l* = *n*›)
from ‹*l* ≠ *0*› **have** *l dvd gcd-eucl l l'*
by (*rule dvd-euclidean-size-eq-imp-dvd*) (*auto simp add*: ∗)
hence *l dvd l'* **by** (*rule dvd-trans*[*OF* - *gcd-eucl-dvd2*])
}

with ‹(∀ *a*∈*A*. *a dvd l*)› **and** *unit-factor-is-unit*[*OF* ‹*l* ≠ *0*›] **and** ‹*l* ≠ *0*›
have (∀ *a*∈*A*. *a dvd normalize l*) ∧
(∀ *l'*. (∀ *a*∈*A*. *a dvd l'*) ⟶ *normalize l dvd l'*) ∧
unit-factor (*normalize l*) =
(*if normalize l* = *0 then 0 else 1*)
by (*auto simp*: *unit-simps*)
also from *True* **have** *normalize l* = *Lcm-eucl A*
by (*simp add*: *Lcm-eucl-def Let-def n-def l-def*)
finally show *?thesis* **.**
qed
note *A* = *this*

{fix *a* **assume** *a* ∈ *A* **then show** *a dvd Lcm-eucl A* **using** *A* **by** *blast***}**
{fix *b* **assume** ⋀*a*. *a* ∈ *A* ⟹ *a dvd b* **then show** *Lcm-eucl A dvd b* **using** *A* **by** *blast***}**
from *A* **show** *unit-factor* (*Lcm-eucl A*) = (*if Lcm-eucl A* = *0 then 0 else 1*) **by** *blast*
qed

lemma *normalize-Lcm-eucl* [*simp*]:
normalize (*Lcm-eucl A*) = *Lcm-eucl A*
proof (*cases Lcm-eucl A* = *0*)
case *True* **then show** *?thesis* **by** *simp*
next
case *False*
have *unit-factor* (*Lcm-eucl A*) ∗ *normalize* (*Lcm-eucl A*) = *Lcm-eucl A*
by (*fact unit-factor-mult-normalize*)
with *False* **show** *?thesis* **by** *simp*
qed

lemma *eq-Lcm-euclI*:

fixes *lcm* :: *'a set* ⇒ *'a*
assumes $\bigwedge A\ a.\ a \in A \Longrightarrow a\ dvd\ lcm\ A$ **and** $\bigwedge A\ c.\ (\bigwedge a.\ a \in A \Longrightarrow a\ dvd\ c) \Longrightarrow lcm\ A\ dvd\ c$
$\bigwedge A.\ normalize\ (lcm\ A) = lcm\ A$ **shows** *lcm* = *Lcm-eucl*
by (*intro ext*, *rule associated-eqI*) (*auto simp*: *assms intro*: *Lcm-eucl-least*)

end

class *euclidean-ring* = *euclidean-semiring* + *idom*
begin

subclass *ring-div* **..**

function *euclid-ext-aux* :: *'a* ⇒ - **where**
euclid-ext-aux r' r s' s t' t = (
if r = 0 then let c = 1 div unit-factor r' in (*s' ∗ c*, *t' ∗ c*, *normalize r'*)
else let q = r' div r
in euclid-ext-aux r (*r' mod r*) *s* (*s' − q ∗ s*) *t* (*t' − q ∗ t*))
by *auto*
termination by (*relation measure* (λ(-,*b*,-,-,-,-). *euclidean-size b*)) (*simp-all add*: *mod-size-less*)

declare *euclid-ext-aux.simps* [*simp del*]

lemma *euclid-ext-aux-correct*:
assumes *gcd-eucl r' r* = *gcd-eucl x y*
assumes *s' ∗ x* + *t' ∗ y* = *r'*
assumes *s ∗ x* + *t ∗ y* = *r*
shows *case euclid-ext-aux r' r s' s t' t of* (*a*,*b*,*c*) ⇒
a ∗ x + *b ∗ y* = *c* ∧ *c* = *gcd-eucl x y* (**is** *?P* (*euclid-ext-aux r' r s' s t' t*))
using *assms*
proof (*induction r' r s' s t' t rule*: *euclid-ext-aux.induct*)
case (*1 r' r s' s t' t*)
show *?case*
proof (*cases r = 0*)
case *True*
hence *euclid-ext-aux r' r s' s t' t* =
(*s' div unit-factor r'*, *t' div unit-factor r'*, *normalize r'*)
by (*subst euclid-ext-aux.simps*) (*simp add*: *Let-def*)
also have *?P* . . .
proof *safe*
have *s' div unit-factor r' ∗ x* + *t' div unit-factor r' ∗ y* =
(*s' ∗ x* + *t' ∗ y*) *div unit-factor r'*
by (*cases r' = 0*) (*simp-all add*: *unit-div-commute*)
also have *s' ∗ x* + *t' ∗ y* = *r'* **by** *fact*
also have . . . *div unit-factor r'* = *normalize r'* **by** *simp*
finally show *s' div unit-factor r' ∗ x* + *t' div unit-factor r' ∗ y* = *normalize r'* **.**

next
from *1.prems True* **show** *normalize r′ = gcd-eucl x y* **by** (*simp add*: *gcd-eucl-0*)
qed
finally show *?thesis* **.**
next
case *False*
hence *euclid-ext-aux r′ r s′ s t′ t =*
euclid-ext-aux r (r′ mod r) s (s′ − r′ div r ∗ s) t (t′ − r′ div r ∗ t)
by (*subst euclid-ext-aux.simps*) (*simp add*: *Let-def*)
also from *1.prems False* **have** *?P …*
proof (*intro 1.IH*)
have (*s′ − r′ div r ∗ s*) *∗ x +* (*t′ − r′ div r ∗ t*) *∗ y =*
(*s′ ∗ x + t′ ∗ y*) *− r′ div r ∗* (*s ∗ x + t ∗ y*) **by** (*simp add*: *algebra-simps*)
also have *s′ ∗ x + t′ ∗ y = r′* **by** *fact*
also have *s ∗ x + t ∗ y = r* **by** *fact*
also have *r′ − r′ div r ∗ r = r′ mod r* **using** *mod-div-equality*[*of r′ r*]
by (*simp add*: *algebra-simps*)
finally show (*s′ − r′ div r ∗ s*) *∗ x +* (*t′ − r′ div r ∗ t*) *∗ y = r′ mod r* **.**
qed (*auto simp*: *gcd-eucl-non-0 algebra-simps div-mod-equality′*)
finally show *?thesis* **.**
qed
qed

definition *euclid-ext* **where**
euclid-ext a b = euclid-ext-aux a b 1 0 0 1

lemma *euclid-ext-0*:
euclid-ext a 0 = (*1 div unit-factor a*, *0*, *normalize a*)
by (*simp add*: *euclid-ext-def euclid-ext-aux.simps*)

lemma *euclid-ext-left-0*:
euclid-ext 0 a = (*0*, *1 div unit-factor a*, *normalize a*)
by (*simp add*: *euclid-ext-def euclid-ext-aux.simps*)

lemma *euclid-ext-correct′*:
case euclid-ext x y of (*a,b,c*) *⇒ a ∗ x + b ∗ y = c ∧ c = gcd-eucl x y*
unfolding *euclid-ext-def* **by** (*rule euclid-ext-aux-correct*) *simp-all*

lemma *euclid-ext-gcd-eucl*:
(*case euclid-ext x y of* (*a,b,c*) *⇒ c*) *= gcd-eucl x y*
using *euclid-ext-correct′*[*of x y*] **by** (*simp add*: *case-prod-unfold*)

definition *euclid-ext′* **where**
euclid-ext′ x y = (*case euclid-ext x y of* (*a*, *b*, *-*) *⇒* (*a*, *b*))

lemma *euclid-ext′-correct′*:
case euclid-ext′ x y of (*a,b*) *⇒ a ∗ x + b ∗ y = gcd-eucl x y*
using *euclid-ext-correct′*[*of x y*] **by** (*simp add*: *case-prod-unfold euclid-ext′-def*)

lemma *euclid-ext′-0*: *euclid-ext′ a 0* = (*1 div unit-factor a*, *0*)
by (*simp add*: *euclid-ext′-def euclid-ext-0*)

lemma *euclid-ext′-left-0*: *euclid-ext′ 0 a* = (*0*, *1 div unit-factor a*)
by (*simp add*: *euclid-ext′-def euclid-ext-left-0*)

end

class *euclidean-semiring-gcd* = *euclidean-semiring* + *gcd* + *Gcd* +
assumes *gcd-gcd-eucl*: *gcd* = *gcd-eucl* **and** *lcm-lcm-eucl*: *lcm* = *lcm-eucl*
assumes *Gcd-Gcd-eucl*: *Gcd* = *Gcd-eucl* **and** *Lcm-Lcm-eucl*: *Lcm* = *Lcm-eucl*
begin

subclass *semiring-gcd*
by *standard* (*simp-all add*: *gcd-gcd-eucl gcd-eucl-greatest lcm-lcm-eucl lcm-eucl-def*)

subclass *semiring-Gcd*
by *standard* (*auto simp*: *Gcd-Gcd-eucl Lcm-Lcm-eucl Gcd-eucl-def intro*: *Lcm-eucl-least*)

lemma *gcd-non-0*:
$b \neq 0 \Longrightarrow$ *gcd a b* = *gcd b* (*a mod b*)
unfolding *gcd-gcd-eucl* **by** (*fact gcd-eucl-non-0*)

lemmas *gcd-0* = *gcd-0-right*
lemmas *dvd-gcd-iff* = *gcd-greatest-iff*
lemmas *gcd-greatest-iff* = *dvd-gcd-iff*

lemma *gcd-mod1* [*simp*]:
gcd (*a mod b*) *b* = *gcd a b*
by (*rule gcdI*, *metis dvd-mod-iff gcd-dvd1 gcd-dvd2*, *simp-all add*: *gcd-greatest dvd-mod-iff*)

lemma *gcd-mod2* [*simp*]:
gcd a (*b mod a*) = *gcd a b*
by (*rule gcdI*, *simp*, *metis dvd-mod-iff gcd-dvd1 gcd-dvd2*, *simp-all add*: *gcd-greatest dvd-mod-iff*)

lemma *euclidean-size-gcd-le1* [*simp*]:
assumes $a \neq 0$
shows *euclidean-size* (*gcd a b*) ≤ *euclidean-size a*
proof −
have *gcd a b dvd a* **by** (*rule gcd-dvd1*)
then obtain *c* **where** *A*: *a* = *gcd a b* ∗ *c* **unfolding** *dvd-def* **by** *blast*
with ‹$a \neq 0$› **show** *?thesis* **by** (*subst* (*2*) *A*, *intro size-mult-mono*) *auto*
qed

lemma *euclidean-size-gcd-le2* [*simp*]:
$b \neq 0 \Longrightarrow$ *euclidean-size* (*gcd a b*) ≤ *euclidean-size b*

by (*subst gcd.commute*, *rule euclidean-size-gcd-le1*)

lemma *euclidean-size-gcd-less1*:
assumes $a \neq 0$ **and** $\neg a$ *dvd* b
shows *euclidean-size* (*gcd a b*) < *euclidean-size a*
proof (*rule ccontr*)
assume ¬*euclidean-size* (*gcd a b*) < *euclidean-size a*
with ‹$a \neq 0$› **have** *A*: *euclidean-size* (*gcd a b*) = *euclidean-size a*
by (*intro le-antisym*, *simp-all*)
have *a dvd gcd a b*
by (*rule dvd-euclidean-size-eq-imp-dvd*) (*simp-all add*: *assms A*)
hence *a dvd b* **using** *dvd-gcdD2* **by** *blast*
with ‹¬*a dvd b*› **show** *False* **by** *contradiction*
qed

lemma *euclidean-size-gcd-less2*:
assumes $b \neq 0$ **and** $\neg b$ *dvd* a
shows *euclidean-size* (*gcd a b*) < *euclidean-size b*
using *assms* **by** (*subst gcd.commute*, *rule euclidean-size-gcd-less1*)

lemma *euclidean-size-lcm-le1*:
assumes $a \neq 0$ **and** $b \neq 0$
shows *euclidean-size a* ≤ *euclidean-size* (*lcm a b*)
proof −
have *a dvd lcm a b* **by** (*rule dvd-lcm1*)
then obtain *c* **where** *A*: *lcm a b* = *a* ∗ *c* **..**
with ‹$a \neq 0$› **and** ‹$b \neq 0$› **have** $c \neq 0$ **by** (*auto simp*: *lcm-eq-0-iff*)
then show *?thesis* **by** (*subst A*, *intro size-mult-mono*)
qed

lemma *euclidean-size-lcm-le2*:
$a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow$ *euclidean-size b* ≤ *euclidean-size* (*lcm a b*)
using *euclidean-size-lcm-le1* [*of b a*] **by** (*simp add*: *ac-simps*)

lemma *euclidean-size-lcm-less1*:
assumes $b \neq 0$ **and** $\neg b$ *dvd* a
shows *euclidean-size a* < *euclidean-size* (*lcm a b*)
proof (*rule ccontr*)
from *assms* **have** $a \neq 0$ **by** *auto*
assume ¬*euclidean-size a* < *euclidean-size* (*lcm a b*)
with ‹$a \neq 0$› **and** ‹$b \neq 0$› **have** *euclidean-size* (*lcm a b*) = *euclidean-size a*
by (*intro le-antisym*, *simp*, *intro euclidean-size-lcm-le1*)
with *assms* **have** *lcm a b dvd a*
by (*rule-tac dvd-euclidean-size-eq-imp-dvd*) (*auto simp*: *lcm-eq-0-iff*)
hence *b dvd a* **by** (*rule lcm-dvdD2*)
with ‹¬*b dvd a*› **show** *False* **by** *contradiction*
qed

lemma *euclidean-size-lcm-less2*:

assumes *a* ≠ *0* **and** ¬*a dvd b*
shows *euclidean-size b* < *euclidean-size* (*lcm a b*)
using *assms euclidean-size-lcm-less1* [*of a b*] **by** (*simp add*: *ac-simps*)

lemma *Lcm-eucl-set* [*code*]:
Lcm-eucl (*set xs*) = *foldl lcm-eucl 1 xs*
by (*simp add*: *Lcm-Lcm-eucl* [*symmetric*] *lcm-lcm-eucl Lcm-set*)

lemma *Gcd-eucl-set* [*code*]:
Gcd-eucl (*set xs*) = *foldl gcd-eucl 0 xs*
by (*simp add*: *Gcd-Gcd-eucl* [*symmetric*] *gcd-gcd-eucl Gcd-set*)

end

A Euclidean ring is a Euclidean semiring with additive inverses. It provides a few more lemmas; in particular, Bezout's lemma holds for any Euclidean ring.

class *euclidean-ring-gcd* = *euclidean-semiring-gcd* + *idom*
begin

subclass *euclidean-ring* **..**
subclass *ring-gcd* **..**

lemma *euclid-ext-gcd* [*simp*]:
(*case euclid-ext a b of* (-, - , *t*) ⇒ *t*) = *gcd a b*
using *euclid-ext-correct'*[*of a b*] **by** (*simp add*: *case-prod-unfold Let-def gcd-gcd-eucl*)

lemma *euclid-ext-gcd'* [*simp*]:
euclid-ext a b = (*r*, *s*, *t*) ⟹ *t* = *gcd a b*
by (*insert euclid-ext-gcd*[*of a b*], *drule* (*1*) *subst*, *simp*)

lemma *euclid-ext-correct*:
case euclid-ext x y of (*a*,*b*,*c*) ⇒ *a* ∗ *x* + *b* ∗ *y* = *c* ∧ *c* = *gcd x y*
using *euclid-ext-correct'*[*of x y*]
by (*simp add*: *gcd-gcd-eucl case-prod-unfold*)

lemma *euclid-ext'-correct*:
fst (*euclid-ext' a b*) ∗ *a* + *snd* (*euclid-ext' a b*) ∗ *b* = *gcd a b*
using *euclid-ext-correct'*[*of a b*]
by (*simp add*: *gcd-gcd-eucl case-prod-unfold euclid-ext'-def*)

lemma *bezout*: ∃ *s t*. *s* ∗ *a* + *t* ∗ *b* = *gcd a b*
using *euclid-ext'-correct* **by** *blast*

end

15.1 Typical instances

instantiation *nat* :: *euclidean-semiring*

begin

definition [*simp*]:
  euclidean-size-nat = (*id* :: *nat* ⇒ *nat*)

instance proof
qed *simp-all*

end

instantiation *int* :: *euclidean-ring*
begin

definition [*simp*]:
  euclidean-size-int = (*nat* ∘ *abs* :: *int* ⇒ *nat*)

instance
by *standard* (*auto simp add*: *abs-mult nat-mult-distrib split*: *abs-split*)

end

instantiation *poly* :: (*field*) *euclidean-ring*
begin

definition *euclidean-size-poly* :: ′*a poly* ⇒ *nat*
  where *euclidean-size p* = (*if p* = *0 then 0 else 2* ˆ *degree p*)

lemma *euclidean-size-poly-0* [*simp*]:
  euclidean-size (*0*::′*a poly*) = *0*
  by (*simp add*: *euclidean-size-poly-def*)

lemma *euclidean-size-poly-not-0* [*simp*]:
  p ≠ *0* ⟹ *euclidean-size p* = *2* ˆ *degree p*
  by (*simp add*: *euclidean-size-poly-def*)

instance
proof
  fix *p q* :: ′*a poly*
  assume *q* ≠ *0*
  then have *p mod q* = *0* ∨ *degree* (*p mod q*) < *degree q*
    by (*rule degree-mod-less* [*of q p*])
  with ‹*q* ≠ *0*› **show** *euclidean-size* (*p mod q*) < *euclidean-size q*
    by (*cases p mod q* = *0*) *simp-all*
next
  fix *p q* :: ′*a poly*
  assume *q* ≠ *0*
  from ‹*q* ≠ *0*› **have** *degree p* ≤ *degree* (*p* ∗ *q*)

by (*rule degree-mult-right-le*)
with ‹$q \neq 0$› **show** *euclidean-size p* ≤ *euclidean-size* ($p * q$)
by (*cases p* = *0*) *simp-all*
qed *simp*

end

instance *nat* :: *euclidean-semiring-gcd*
proof
show [*simp*]: *gcd* = (*gcd-eucl* :: *nat* ⇒ -) *Lcm* = (*Lcm-eucl* :: *nat set* ⇒ -)
by (*simp-all add*: *eq-gcd-euclI eq-Lcm-euclI*)
show *lcm* = (*lcm-eucl* :: *nat* ⇒ -) *Gcd* = (*Gcd-eucl* :: *nat set* ⇒ -)
by (*intro ext*, *simp add*: *lcm-eucl-def lcm-nat-def Gcd-nat-def Gcd-eucl-def*)+
qed

instance *int* :: *euclidean-ring-gcd*
proof
show [*simp*]: *gcd* = (*gcd-eucl* :: *int* ⇒ -) *Lcm* = (*Lcm-eucl* :: *int set* ⇒ -)
by (*simp-all add*: *eq-gcd-euclI eq-Lcm-euclI*)
show *lcm* = (*lcm-eucl* :: *int* ⇒ -) *Gcd* = (*Gcd-eucl* :: *int set* ⇒ -)
by (*intro ext*, *simp add*: *lcm-eucl-def lcm-altdef-int*
semiring-Gcd-class.Gcd-Lcm Gcd-eucl-def abs-mult)+
qed

instantiation *poly* :: (*field*) *euclidean-ring-gcd*
begin

definition *gcd-poly* :: *'a poly* ⇒ *'a poly* ⇒ *'a poly* **where**
gcd-poly = *gcd-eucl*

definition *lcm-poly* :: *'a poly* ⇒ *'a poly* ⇒ *'a poly* **where**
lcm-poly = *lcm-eucl*

definition *Gcd-poly* :: *'a poly set* ⇒ *'a poly* **where**
Gcd-poly = *Gcd-eucl*

definition *Lcm-poly* :: *'a poly set* ⇒ *'a poly* **where**
Lcm-poly = *Lcm-eucl*

instance by *standard* (*simp-all only*: *gcd-poly-def lcm-poly-def Gcd-poly-def Lcm-poly-def*)
end

lemma *poly-gcd-monic*:
lead-coeff (*gcd x y*) = (*if x* = *0* ∧ *y* = *0 then 0 else 1*)
using *unit-factor-gcd*[*of x y*]
by (*simp add*: *unit-factor-poly-def monom-0 one-poly-def lead-coeff-def split*: *if-split-asm*)

lemma *poly-dvd-antisym*:
  fixes *p q* :: *'a::idom poly*
  assumes *coeff*: *coeff p* (*degree p*) = *coeff q* (*degree q*)
  assumes *dvd1*: *p dvd q* **and** *dvd2*: *q dvd p* **shows** *p* = *q*
proof (*cases p* = *0*)
  case *True* **with** *coeff* **show** *p* = *q* **by** *simp*
next
  case *False* **with** *coeff* **have** $q \neq 0$ **by** *auto*
  have *degree*: *degree p* = *degree q*
    using ‹*p dvd q*› ‹*q dvd p*› ‹$p \neq 0$› ‹$q \neq 0$›
    by (*intro order-antisym dvd-imp-degree-le*)

  from ‹*p dvd q*› **obtain** *a* **where** *a*: *q* = *p* ∗ *a* **..**
  with ‹$q \neq 0$› **have** $a \neq 0$ **by** *auto*
  with *degree a* ‹$p \neq 0$› **have** *degree a* = *0*
    by (*simp add*: *degree-mult-eq*)
  with *coeff a* **show** *p* = *q*
    by (*cases a*, *auto split*: *if-splits*)
qed

lemma *poly-gcd-unique*:
  fixes *d x y* :: *- poly*
  assumes *dvd1*: *d dvd x* **and** *dvd2*: *d dvd y*
    and *greatest*: ⋀*k*. *k dvd x* ⟹ *k dvd y* ⟹ *k dvd d*
    and *monic*: *coeff d* (*degree d*) = (*if x* = *0* ∧ *y* = *0 then 0 else 1*)
  shows *d* = *gcd x y*
  using *assms* **by** (*intro gcdI*) (*auto simp*: *normalize-poly-def split*: *if-split-asm*)

lemma *poly-gcd-code* [*code*]:
  gcd x y = (*if y* = *0 then normalize x else gcd y* (*x mod* (*y* :: *- poly*)))
  by (*simp add*: *gcd-0 gcd-non-0*)

end

16 Factorial (semi)rings

theory *Factorial-Ring*
imports *Main Primes* ~~/*src*/*HOL*/*Library*/*Multiset*
begin

context *algebraic-semidom*
begin

lemma *dvd-mult-imp-div*:
  assumes *a* ∗ *c dvd b*
  shows *a dvd b div c*
proof (*cases c* = *0*)
  case *True* **then show** *?thesis* **by** *simp*

next
  case *False*
  from ‹*a* ∗ *c dvd b*› **obtain** *d* **where** $b = a * c * d$ **..**
  with *False* **show** *?thesis* **by** (*simp add*: *mult.commute* [*of a*] *mult.assoc*)
qed

end

class *factorial-semiring* = *normalization-semidom* +
  assumes *finite-divisors*: $a \neq 0 \Longrightarrow$ *finite* {*b*. *b dvd a* ∧ *normalize b* = *b*}
  fixes *is-prime* :: *'a* ⇒ *bool*
  assumes *not-is-prime-zero* [*simp*]: ¬ *is-prime 0*
    and *is-prime-not-unit*: *is-prime p* ⟹ ¬ *is-unit p*
    and *no-prime-divisorsI2*: (⋀*b*. *b dvd a* ⟹ ¬ *is-prime b*) ⟹ *is-unit a*
  assumes *is-primeI*: $p \neq 0$ ⟹ ¬ *is-unit p* ⟹ (⋀*a*. *a dvd p* ⟹ ¬ *is-unit a* ⟹ *p dvd a*) ⟹ *is-prime p*
    and *is-primeD*: *is-prime p* ⟹ *p dvd a* ∗ *b* ⟹ *p dvd a* ∨ *p dvd b*
begin

lemma *not-is-prime-one* [*simp*]:
  ¬ *is-prime 1*
  by (*auto dest*: *is-prime-not-unit*)

lemma *is-prime-not-zeroI*:
  assumes *is-prime p*
  shows $p \neq 0$
  using *assms* **by** (*auto intro*: *ccontr*)

lemma *is-prime-multD*:
  assumes *is-prime* (*a* ∗ *b*)
  shows *is-unit a* ∨ *is-unit b*
proof −
  from *assms* **have** $a \neq 0$ $b \neq 0$ **by** *auto*
  moreover from *assms is-primeD* [*of a* ∗ *b*] **have** *a* ∗ *b dvd a* ∨ *a* ∗ *b dvd b*
    by *auto*
  ultimately show *?thesis*
    using *dvd-times-left-cancel-iff* [*of a b 1*]
      dvd-times-right-cancel-iff [*of b a 1*]
    by *auto*
qed

lemma *is-primeD2*:
  assumes *is-prime p* **and** *a dvd p* **and** ¬ *is-unit a*
  shows *p dvd a*
proof −
  from ‹*a dvd p*› **obtain** *b* **where** $p = a * b$ **..**
  with ‹*is-prime p*› *is-prime-multD* ‹¬ *is-unit a*› **have** *is-unit b* **by** *auto*
  with ‹$p = a * b$› **show** *?thesis*
    by (*auto simp add*: *mult-unit-dvd-iff*)

qed

lemma *is-prime-mult-unit-left*:
  assumes *is-prime p*
    and *is-unit a*
  shows *is-prime* (*a* ∗ *p*)
proof (*rule is-primeI*)
  from *assms* **show** *a* ∗ *p* ≠ *0* ¬ *is-unit* (*a* ∗ *p*)
    by (*auto simp add*: *is-unit-mult-iff is-prime-not-unit*)
  show *a* ∗ *p dvd b* **if** *b dvd a* ∗ *p* ¬ *is-unit b* **for** *b*
  proof −
    from *that* ‹*is-unit a*› **have** *b dvd p*
      using *dvd-mult-unit-iff* [*of a b p*] **by** (*simp add*: *ac-simps*)
    with ‹*is-prime p*› ‹¬ *is-unit b*› **have** *p dvd b*
      using *is-primeD2* [*of p b*] **by** *auto*
    with ‹*is-unit a*› **show** *?thesis*
      using *mult-unit-dvd-iff* [*of a p b*] **by** (*simp add*: *ac-simps*)
  qed
qed

lemma *is-primeI2*:
  assumes *p* ≠ *0*
  assumes ¬ *is-unit p*
  assumes *P*: ⋀*a b*. *p dvd a* ∗ *b* ⟹ *p dvd a* ∨ *p dvd b*
  shows *is-prime p*
using ‹*p* ≠ *0*› ‹¬ *is-unit p*› **proof** (*rule is-primeI*)
  fix *a*
  assume *a dvd p*
  then obtain *b* **where** *p* = *a* ∗ *b* **..**
  with ‹*p* ≠ *0*› **have** *b* ≠ *0* **by** *simp*
  moreover from ‹*p* = *a* ∗ *b*› *P* **have** *p dvd a* ∨ *p dvd b* **by** *auto*
  moreover assume ¬ *is-unit a*
  ultimately show *p dvd a*
    using *dvd-times-right-cancel-iff* [*of b a 1*] ‹*p* = *a* ∗ *b*› **by** *auto*
qed

lemma *not-is-prime-divisorE*:
  assumes *a* ≠ *0* **and** ¬ *is-unit a* **and** ¬ *is-prime a*
  obtains *b* **where** *b dvd a* **and** ¬ *is-unit b* **and** ¬ *a dvd b*
proof −
  from *assms* **have** ∃ *b*. *b dvd a* ∧ ¬ *is-unit b* ∧ ¬ *a dvd b*
    by (*auto intro*: *is-primeI*)
  with *that* **show** *thesis* **by** *blast*
qed

lemma *is-prime-normalize-iff* [*simp*]:
  is-prime (*normalize p*) ⟷ *is-prime p* (**is** *?P* ⟷ *?Q*)
proof
  assume *?Q* **show** *?P*

by (*rule is-primeI*) (*insert* ‹*?Q*›, *simp-all add*: *is-prime-not-zeroI is-prime-not-unit is-primeD2*)
next
assume *?P* **show** *?Q*
by (*rule is-primeI*)
(*insert is-prime-not-zeroI* [*of normalize p*] *is-prime-not-unit* [*of normalize p*] *is-primeD2* [*of normalize p*] ‹*?P*›, *simp-all*)
qed

lemma *no-prime-divisorsI*:
assumes $\bigwedge b.\ b\ dvd\ a \Longrightarrow normalize\ b = b \Longrightarrow \neg\ is\text{-}prime\ b$
shows *is-unit a*
proof (*rule no-prime-divisorsI2*)
fix *b*
assume *b dvd a*
then have *normalize b dvd a*
by *simp*
moreover have *normalize* (*normalize b*) = *normalize b*
by *simp*
ultimately have ¬ *is-prime* (*normalize b*)
by (*rule assms*)
then show ¬ *is-prime b*
by *simp*
qed

lemma *prime-divisorE*:
assumes $a \neq 0$ **and** ¬ *is-unit a*
obtains *p* **where** *is-prime p* **and** *p dvd a*
using *assms no-prime-divisorsI* [*of a*] **by** *blast*

lemma *is-prime-associated*:
assumes *is-prime p* **and** *is-prime q* **and** *q dvd p*
shows *normalize q* = *normalize p*
using ‹*q dvd p*› **proof** (*rule associatedI*)
from ‹*is-prime q*› **have** ¬ *is-unit q*
by (*simp add*: *is-prime-not-unit*)
with ‹*is-prime p*› ‹*q dvd p*› **show** *p dvd q*
by (*blast intro*: *is-primeD2*)
qed

lemma *prime-dvd-mult-iff*:
assumes *is-prime p*
shows $p\ dvd\ a * b \longleftrightarrow p\ dvd\ a \lor p\ dvd\ b$
using *assms* **by** (*auto dest*: *is-primeD*)

lemma *prime-dvd-msetprod*:
assumes *is-prime p*
assumes *dvd*: *p dvd msetprod A*
obtains *a* **where** $a \in\# A$ **and** *p dvd a*

proof −
  from *dvd* **have** ∃ *a*. *a* ∈# *A* ∧ *p dvd a*
  proof (*induct A*)
    case *empty* **then show** *?case*
    using ‹*is-prime p*› **by** (*simp add*: *is-prime-not-unit*)
  next
    case (*add A a*)
    then have *p dvd msetprod A* ∗ *a* **by** *simp*
    with ‹*is-prime p*› **consider** (*A*) *p dvd msetprod A* | (*B*) *p dvd a*
      by (*blast dest*: *is-primeD*)
    then show *?case* **proof** *cases*
      case *B* **then show** *?thesis* **by** *auto*
    next
      case *A*
      with *add.hyps* **obtain** *b* **where** *b* ∈# *A p dvd b*
        by *auto*
      then show *?thesis* **by** *auto*
    qed
  qed
  with *that* **show** *thesis* **by** *blast*
qed

lemma *msetprod-eq-iff*:
  assumes ∀ *p*∈*set-mset P*. *is-prime p* ∧ *normalize p* = *p* **and** ∀ *p*∈*set-mset Q*. *is-prime p* ∧ *normalize p* = *p*
  shows *msetprod P* = *msetprod Q* ⟷ *P* = *Q* (**is** *?R* ⟷ *?S*)
proof
  assume *?S* **then show** *?R* **by** *simp*
next
  assume *?R* **then show** *?S* **using** *assms* **proof** (*induct P arbitrary*: *Q*)
    case *empty* **then have** *Q*: *msetprod Q* = *1* **by** *simp*
    have *Q* = {#}
    proof (*rule ccontr*)
      assume *Q* ≠ {#}
      then obtain *r R* **where** *Q* = *R* + {#*r*#}
        using *multi-nonempty-split* **by** *blast*
      moreover with *empty* **have** *is-prime r* **by** *simp*
      ultimately have *msetprod Q* = *msetprod R* ∗ *r*
        by *simp*
      with *Q* **have** *msetprod R* ∗ *r* = *1*
        by *simp*
      then have *is-unit r*
        by (*metis local.dvd-triv-right*)
      with ‹*is-prime r*› **show** *False* **by** (*simp add*: *is-prime-not-unit*)
    qed
    then show *?case* **by** *simp*
  next
    case (*add P p*)
    then have *is-prime p* **and** *normalize p* = *p*

and *msetprod Q* = *msetprod P* ∗ *p* **and** *p dvd msetprod Q*
by *auto* (*metis local.dvd-triv-right*)
with *prime-dvd-msetprod*
obtain *q* **where** *q* ∈# *Q* **and** *p dvd q*
by *blast*
with *add.prems* **have** *is-prime q* **and** *normalize q* = *q*
by *simp-all*
from ⟨*is-prime p*⟩ **have** *p* ≠ *0*
by *auto*
from ⟨*is-prime q*⟩ ⟨*is-prime p*⟩ ⟨*p dvd q*⟩
have *normalize p* = *normalize q*
by (*rule is-prime-associated*)
from ⟨*normalize p* = *p*⟩ ⟨*normalize q* = *q*⟩ **have** *p* = *q*
unfolding ⟨*normalize p* = *normalize q*⟩ **by** *simp*
with ⟨*q* ∈# *Q*⟩ **have** *p* ∈# *Q* **by** *simp*
have *msetprod P* = *msetprod* (*Q* − {#*p*#})
using ⟨*p* ∈# *Q*⟩ ⟨*p* ≠ *0*⟩ ⟨*msetprod Q* = *msetprod P* ∗ *p*⟩
by (*simp add*: *msetprod-minus*)
then have *P* = *Q* − {#*p*#}
using *add.prems*(*2*−*3*)
by (*auto intro*: *add.hyps dest*: *in-diffD*)
with ⟨*p* ∈# *Q*⟩ **show** *?case* **by** *simp*
qed
qed

lemma *prime-dvd-power-iff*:
assumes *is-prime p*
shows *p dvd a* ˆ *n* ⟷ *p dvd a* ∧ *n* > *0*
using *assms* **by** (*induct n*) (*auto dest*: *is-prime-not-unit is-primeD*)

lemma *prime-power-dvd-multD*:
assumes *is-prime p*
assumes *p* ˆ *n dvd a* ∗ *b* **and** *n* > *0* **and** ¬ *p dvd a*
shows *p* ˆ *n dvd b*
using ⟨*p* ˆ *n dvd a* ∗ *b*⟩ **and** ⟨*n* > *0*⟩ **proof** (*induct n arbitrary*: *b*)
case *0* **then show** *?case* **by** *simp*
next
case (*Suc n*) **show** *?case*
proof (*cases n* = *0*)
case *True* **with** *Suc* ⟨*is-prime p*⟩ ⟨¬ *p dvd a*⟩ **show** *?thesis*
by (*simp add*: *prime-dvd-mult-iff*)
next
case *False* **then have** *n* > *0* **by** *simp*
from ⟨*is-prime p*⟩ **have** *p* ≠ *0* **by** *auto*
from *Suc.prems* **have** ∗: *p* ∗ *p* ˆ *n dvd a* ∗ *b*
by *simp*
then have *p dvd a* ∗ *b*
by (*rule dvd-mult-left*)
with *Suc* ⟨*is-prime p*⟩ ⟨¬ *p dvd a*⟩ **have** *p dvd b*

```
      by (simp add: prime-dvd-mult-iff)
    moreover def c ≡ b div p
    ultimately have b: b = p * c by simp
    with * have p * p ^ n dvd p * (a * c)
      by (simp add: ac-simps)
    with ‹p ≠ 0› have p ^ n dvd a * c
      by simp
    with Suc.hyps ‹n > 0› have p ^ n dvd c
      by blast
    with ‹p ≠ 0› show ?thesis
      by (simp add: b)
  qed
qed

lemma is-prime-inj-power:
  assumes is-prime p
  shows inj (op ^ p)
proof (rule injI, rule ccontr)
  fix m n :: nat
  have [simp]: p ^ q = 1 ⟷ q = 0 (is ?P ⟷ ?Q) for q
  proof
    assume ?Q then show ?P by simp
  next
    assume ?P then have is-unit (p ^ q) by simp
    with assms show ?Q by (auto simp add: is-unit-power-iff is-prime-not-unit)
  qed
  have *: False if p ^ m = p ^ n and m > n for m n
  proof -
    from assms have p ≠ 0
      by (rule is-prime-not-zeroI)
    then have p ^ n ≠ 0
      by (induct n) simp-all
    from that have m = n + (m - n) and m - n > 0 by arith+
    then obtain q where m = n + q and q > 0 ..
    with that have p ^ n * p ^ q = p ^ n * 1 by (simp add: power-add)
    with ‹p ^ n ≠ 0› have p ^ q = 1
      using mult-left-cancel [of p ^ n p ^ q 1] by simp
    with ‹q > 0› show ?thesis by simp
  qed
  assume m ≠ n
  then have m > n ∨ m < n by arith
  moreover assume p ^ m = p ^ n
  ultimately show False using * [of m n] * [of n m] by auto
qed

definition factorization :: 'a ⇒ 'a multiset option
  where factorization a = (if a = 0 then None
    else Some (setsum (λp. replicate-mset (Max {n. p ^ n dvd a}) p)
      {p. p dvd a ∧ is-prime p ∧ normalize p = p}))
```

lemma *factorization-normalize* [*simp*]:
factorization (*normalize a*) = *factorization a*
by (*simp add*: *factorization-def*)

lemma *factorization-0* [*simp*]:
factorization 0 = *None*
by (*simp add*: *factorization-def*)

lemma *factorization-eq-None-iff* [*simp*]:
factorization a = *None* ⟷ *a* = *0*
by (*simp add*: *factorization-def*)

lemma *factorization-eq-Some-iff*:
factorization a = *Some P* ⟷
normalize a = *msetprod P* ∧ *0* ∉# *P* ∧ (∀ *p* ∈ *set-mset P*. *is-prime p* ∧ *normalize p* = *p*)
proof (*cases a* = *0*)
have [*simp*]: *0* = *msetprod P* ⟷ *0* ∈# *P*
using *msetprod-zero-iff* [*of P*] **by** *blast*
case *True*
then show *?thesis* **by** *auto*
next
case *False*
let *?prime-factors* = λ*a*. {*p*. *p dvd a* ∧ *is-prime p* ∧ *normalize p* = *p*}
have *?prime-factors a* ⊆ {*b*. *b dvd a* ∧ *normalize b* = *b*}
by *auto*
moreover from ‹*a* ≠ *0*› **have** *finite* {*b*. *b dvd a* ∧ *normalize b* = *b*}
by (*rule finite-divisors*)
ultimately have *finite* (*?prime-factors a*)
by (*rule finite-subset*)
then show *?thesis* **using** ‹*a* ≠ *0*›
proof (*induct ?prime-factors a arbitrary*: *a P*)
case *empty* **then have**
∗: {*p*. *p dvd a* ∧ *is-prime p* ∧ *normalize p* = *p*} = {}
and *a* ≠ *0*
by *auto*
from ‹*a* ≠ *0*› **have** *factorization a* = *Some* {#}
by (*simp only*: *factorization-def* ∗) *simp*
from ∗ **have** *normalize a* = *1*
by (*auto intro*: *is-unit-normalize no-prime-divisorsI*)
show *?case* (**is** *?lhs* ⟷ *?rhs*) **proof**
assume *?lhs* **with** ‹*factorization a* = *Some* {#}› ‹*normalize a* = *1*›
show *?rhs* **by** *simp*
next
assume *?rhs* **have** *P* = {#}
proof (*rule ccontr*)
assume *P* ≠ {#}
then obtain *q Q* **where** *P* = *Q* + {#*q*#}

using *multi-nonempty-split* **by** *blast*
with ‹*?rhs*› ‹*normalize a = 1*›
have *1 = q ∗ msetprod Q* **and** *is-prime q*
by (*simp-all add*: *ac-simps*)
then have *is-unit q* **by** (*auto intro*: *dvdI*)
with ‹*is-prime q*› **show** *False*
using *is-prime-not-unit* **by** *blast*
qed
with ‹*factorization a = Some* {#}› **show** *?lhs* **by** *simp*
qed
next
case (*insert p F*)
from ‹*insert p F = ?prime-factors a*›
have *?prime-factors a = insert p F*
by *simp*
then have *p dvd a* **and** *is-prime p* **and** *normalize p = p* **and** *p ≠ 0*
by (*auto intro*!: *is-prime-not-zeroI*)
def *n ≡ Max* {*n. p ˆ n dvd a*}
then have *n > 0* **and** *p ˆ n dvd a* **and** ¬ *p ˆ Suc n dvd a*
proof −
def *N ≡* {*n. p ˆ n dvd a*}
then have *n-M*: *n = Max N* **by** (*simp add*: *n-def*)
from *is-prime-inj-power* ‹*is-prime p*› **have** *inj* (*op ˆ p*) **.**
then have *inj-on* (*op ˆ p*) *U* **for** *U*
by (*rule subset-inj-on*) *simp*
moreover have *op ˆ p ‘ N ⊆* {*b. b dvd a ∧ normalize b = b*}
by (*auto simp add*: *normalize-power* ‹*normalize p = p*› *N-def*)
ultimately have *finite N*
by (*rule inj-on-finite*) (*simp add*: *finite-divisors* ‹*a ≠ 0*›)
from *N-def* ‹*a ≠ 0*› **have** *0 ∈ N* **by** (*simp add*: *N-def*)
then have *N ≠* {} **by** *blast*
note ∗ = ‹*finite N*› ‹*N ≠* {}›
from *N-def* ‹*p dvd a*› **have** *1 ∈ N* **by** *simp*
with ∗ **have** *Max N > 0*
by (*auto simp add*: *Max-gr-iff*)
then show *n > 0* **by** (*simp add*: *n-M*)
from ∗ **have** *Max N ∈ N* **by** (*rule Max-in*)
then have *p ˆ Max N dvd a* **by** (*simp add*: *N-def*)
then show *p ˆ n dvd a* **by** (*simp add*: *n-M*)
from ∗ **have** ∀ *n∈N. n ≤ Max N*
by (*simp add*: *Max-le-iff* [*symmetric*])
then have *p ˆ Suc* (*Max N*) *dvd a =⇒ Suc* (*Max N*) *≤ Max N*
by (*rule bspec*) (*simp add*: *N-def*)
then have ¬ *p ˆ Suc* (*Max N*) *dvd a*
by *auto*
then show ¬ *p ˆ Suc n dvd a*
by (*simp add*: *n-M*)
qed
def *b ≡ a div p ˆ n*

```
with ‹p ^ n dvd a› have a: a = p ^ n * b
  by simp
with ‹¬ p ^ Suc n dvd a› have ¬ p dvd b and b ≠ 0
  by (auto elim: dvdE simp add: ac-simps)
have ?prime-factors a = insert p (?prime-factors b)
proof (rule set-eqI)
  fix q
  show q ∈ ?prime-factors a ⟷ q ∈ insert p (?prime-factors b)
  using ‹is-prime p› ‹normalize p = p› ‹n > 0›
    by (auto simp add: a prime-dvd-mult-iff prime-dvd-power-iff)
      (auto dest: is-prime-associated)
qed
with ‹¬ p dvd b› have ?prime-factors a − {p} = ?prime-factors b
  by auto
with insert.hyps have F = ?prime-factors b
  by auto
then have ?prime-factors b = F
  by simp
 with ‹?prime-factors a = insert p (?prime-factors b)› have ?prime-factors a
= insert p F
  by simp
have equiv: (∑p∈F. replicate-mset (Max {n. p ^ n dvd a}) p) =
  (∑p∈F. replicate-mset (Max {n. p ^ n dvd b}) p)
using refl proof (rule Groups-Big.setsum.cong)
  fix q
  assume q ∈ F
  have {n. q ^ n dvd a} = {n. q ^ n dvd b}
  proof −
    have q ^ m dvd a ⟷ q ^ m dvd b (is ?R ⟷ ?S)
      for m
    proof (cases m = 0)
      case True then show ?thesis by simp
    next
      case False then have m > 0 by simp
      show ?thesis
      proof
        assume ?S then show ?R by (simp add: a)
      next
        assume ?R
        then have *: q ^ m dvd p ^ n * b by (simp add: a)
        from insert.hyps ‹q ∈ F›
        have is-prime q normalize q = q p ≠ q q dvd p ^ n * b
          by (auto simp add: a)
        from ‹is-prime q› * ‹m > 0› show ?S
        proof (rule prime-power-dvd-multD)
          have ¬ q dvd p
          proof
            assume q dvd p
            with ‹is-prime q› ‹is-prime p› have normalize q = normalize p
```

by (*blast intro*: *is-prime-associated*)
with ‹*normalize p = p*› ‹*normalize q = q*› ‹*p ≠ q*› **show** *False*
by *simp*
qed
with ‹*is-prime q*› **show** ¬ *q dvd p ^ n*
by (*simp add*: *prime-dvd-power-iff*)
qed
qed
qed
then show *?thesis* **by** *auto*
qed
then show
replicate-mset (*Max* {*n*. *q ^ n dvd a*}) *q* = *replicate-mset* (*Max* {*n*. *q ^ n dvd b*}) *q*
by *simp*
qed
def *Q* ≡ *the* (*factorization b*)
with ‹*b ≠ 0*› **have** [*simp*]: *factorization b* = *Some Q*
by *simp*
from ‹*a ≠ 0*› **have** *factorization a* =
Some (∑*p*∈*?prime-factors a*. *replicate-mset* (*Max* {*n*. *p ^ n dvd a*}) *p*)
by (*simp add*: *factorization-def*)
also have ... =
Some (∑*p*∈*insert p F*. *replicate-mset* (*Max* {*n*. *p ^ n dvd a*}) *p*)
by (*simp add*: ‹*?prime-factors a = insert p F*›)
also have ... =
Some (*replicate-mset n p* + (∑*p*∈*F*. *replicate-mset* (*Max* {*n*. *p ^ n dvd a*}) *p*))
using ‹*finite F*› ‹*p* ∉ *F*› *n-def* **by** *simp*
also have ... =
Some (*replicate-mset n p* + (∑*p*∈*F*. *replicate-mset* (*Max* {*n*. *p ^ n dvd b*}) *p*))
using *equiv* **by** *simp*
also have ... = *Some* (*replicate-mset n p* + *the* (*factorization b*))
using ‹*b ≠ 0*› **by** (*simp add*: *factorization-def* ‹*?prime-factors a = insert p F*› ‹*?prime-factors b = F*›)
finally have *fact-a*: *factorization a* =
Some (*replicate-mset n p* + *Q*)
by *simp*
moreover have *factorization b* = *Some Q* ⟷
normalize b = *msetprod Q* ∧
0 ∉# *Q* ∧
(∀*p*∈#*Q*. *is-prime p* ∧ *normalize p* = *p*)
using ‹*F = ?prime-factors b*› ‹*b ≠ 0*› **by** (*rule insert.hyps*)
ultimately have
norm-a: *normalize a* = *msetprod* (*replicate-mset n p* + *Q*) **and**
prime-Q: ∀*p*∈*set-mset Q*. *is-prime p* ∧ *normalize p* = *p*
by (*simp-all add*: *a normalize-mult normalize-power* ‹*normalize p = p*›)
show *?case* (**is** *?lhs* ⟷ *?rhs*) **proof**

assume *?lhs* **with** *fact-a*
have $P = replicate\text{-}mset\ n\ p + Q$ **by** *simp*
with ‹$n > 0$› ‹*is-prime p*› ‹*normalize p = p*› *prime-Q*
show *?rhs* **by** (*auto simp add*: *norm-a dest*: *is-prime-not-zeroI*)
next
assume *?rhs*
with ‹$n > 0$› ‹*is-prime p*› ‹*normalize p = p*› ‹$n > 0$› *prime-Q*
have *msetprod P* = *msetprod* (*replicate-mset n p* + *Q*)
and ∀ *p*∈*set-mset P*. *is-prime p* ∧ *normalize p* = *p*
and ∀ *p*∈*set-mset* (*replicate-mset n p* + *Q*). *is-prime p* ∧ *normalize p* = *p*
by (*simp-all add*: *norm-a*)
then have *P* = *replicate-mset n p* + *Q*
by (*simp only*: *msetprod-eq-iff*)
then show *?lhs*
by (*simp add*: *fact-a*)
qed
qed
qed

lemma *factorization-cases* [*case-names 0 factorization*]:
assumes *0*: $a = 0 \Longrightarrow P$
assumes *factorization*: $\bigwedge A.\ a \neq 0 \Longrightarrow factorization\ a = Some\ A \Longrightarrow msetprod\ A = normalize\ a$
$\Longrightarrow 0 \notin\#\ A \Longrightarrow (\bigwedge p.\ p \in\#\ A \Longrightarrow normalize\ p = p) \Longrightarrow (\bigwedge p.\ p \in\#\ A \Longrightarrow is\text{-}prime\ p) \Longrightarrow P$
shows *P*
proof (*cases a* = *0*)
case *True* **with** *0* **show** *P* **.**
next
case *False*
then have *factorization a* ≠ *None* **by** *simp*
then obtain *A* **where** *factorization a* = *Some A* **by** *blast*
moreover from *this* **have** *msetprod A* = *normalize a*
$0 \notin\#\ A\ \bigwedge p.\ p \in\#\ A \Longrightarrow normalize\ p = p\ \bigwedge p.\ p \in\#\ A \Longrightarrow is\text{-}prime\ p$
by (*auto simp add*: *factorization-eq-Some-iff*)
ultimately show *P* **using** ‹$a \neq 0$› *factorization* **by** *blast*
qed

lemma *factorizationE*:
assumes $a \neq 0$
obtains *A u* **where** *factorization a* = *Some A normalize a* = *msetprod A*
$0 \notin\#\ A\ \bigwedge p.\ p \in\#\ A \Longrightarrow is\text{-}prime\ p\ \bigwedge p.\ p \in\#\ A \Longrightarrow normalize\ p = p$
using *assms* **by** (*cases a rule*: *factorization-cases*) *simp-all*

lemma *prime-dvd-mset-prod-iff*:
assumes $is\text{-}prime\ p\ normalize\ p = p\ \bigwedge p.\ p \in\#\ A \Longrightarrow is\text{-}prime\ p\ \bigwedge p.\ p \in\#\ A \Longrightarrow normalize\ p = p$
shows $p\ dvd\ msetprod\ A \longleftrightarrow p \in\#\ A$
using *assms* **proof** (*induct A*)

```
  case empty then show ?case by (auto dest: is-prime-not-unit)
next
  case (add A q) then show ?case
    using is-prime-associated [of q p]
    by (simp-all add: prime-dvd-mult-iff, safe, simp-all)
qed

end

class factorial-semiring-gcd = factorial-semiring + gcd +
  assumes gcd-unfold: gcd a b =
    (if a = 0 then normalize b
     else if b = 0 then normalize a
     else msetprod (the (factorization a) #∩ the (factorization b)))
  and lcm-unfold: lcm a b =
    (if a = 0 ∨ b = 0 then 0
     else msetprod (the (factorization a) #∪ the (factorization b)))
begin

subclass semiring-gcd
proof
  fix a b
  have comm: gcd a b = gcd b a for a b
   by (simp add: gcd-unfold ac-simps)
  have gcd a b dvd a for a b
  proof (cases a rule: factorization-cases)
    case 0 then show ?thesis by simp
  next
    case (factorization A) note fact-A = this
    then have non-zero: ⋀p. p ∈#A ⟹ p ≠ 0
      using normalize-0 not-is-prime-zero by blast
    show ?thesis
    proof (cases b rule: factorization-cases)
      case 0 then show ?thesis by (simp add: gcd-unfold)
    next
      case (factorization B) note fact-B = this
      have msetprod (A #∩ B) dvd msetprod A
      using non-zero proof (induct B arbitrary: A)
        case empty show ?case by simp
      next
        case (add B p) show ?case
        proof (cases p ∈# A)
          case True then obtain C where A = C + {#p#}
            by (metis insert-DiffM2)
          moreover with True add have p ≠ 0 and ⋀p. p ∈# C ⟹ p ≠ 0
            by auto
          ultimately show ?thesis
            using True add.hyps [of C]
            by (simp add: inter-union-distrib-left [symmetric])
```

next
case *False* **with** *add.prems add.hyps* [*of A*] **show** *?thesis*
by (*simp add*: *inter-add-right1*)
qed
qed
with *fact-A fact-B* **show** *?thesis* **by** (*simp add*: *gcd-unfold*)
qed
qed
then have *gcd a b dvd a* **and** *gcd b a dvd b*
by *simp-all*
then show *gcd a b dvd a* **and** *gcd a b dvd b*
by (*simp-all add*: *comm*)
show *c dvd gcd a b* **if** *c dvd a* **and** *c dvd b* **for** *c*
proof (*cases* $a = 0 \vee b = 0 \vee c = 0$)
case *True* **with** *that* **show** *?thesis* **by** (*auto simp add*: *gcd-unfold*)
next
case *False* **then have** $a \neq 0$ **and** $b \neq 0$ **and** $c \neq 0$
by *simp-all*
then obtain *A B C* **where** *fact*:
factorization a = Some A factorization b = Some B factorization c = Some C
and *norm*: *normalize a = msetprod A normalize b = msetprod B normalize c = msetprod C*
and *A*: $0 \notin\# A$ $\bigwedge p.\ p \in\# A \Longrightarrow$ *normalize p = p* $\bigwedge p.\ p \in\# A \Longrightarrow$ *is-prime p*
and *B*: $0 \notin\# B$ $\bigwedge p.\ p \in\# B \Longrightarrow$ *normalize p = p* $\bigwedge p.\ p \in\# B \Longrightarrow$ *is-prime p*
and *C*: $0 \notin\# C$ $\bigwedge p.\ p \in\# C \Longrightarrow$ *normalize p = p* $\bigwedge p.\ p \in\# C \Longrightarrow$ *is-prime p*
by (*blast elim*!: *factorizationE*)
moreover from *that* **have** *normalize c dvd normalize a* **and** *normalize c dvd normalize b*
by *simp-all*
ultimately have *msetprod C dvd msetprod A* **and** *msetprod C dvd msetprod B*
by *simp-all*
with *A B C* **have** *msetprod C dvd msetprod* ($A \#\cap B$)
proof (*induct C arbitrary*: *A B*)
case *empty* **then show** *?case* **by** *simp*
next
case *add*: (*add C p*)
from *add.prems*
have *p*: $p \neq 0$ *is-prime p normalize p = p* **by** *auto*
from *add.prems* **have** *prems*: *msetprod C* ∗ *p dvd msetprod A msetprod C* ∗ *p dvd msetprod B*
by *simp-all*
then have *p dvd msetprod A p dvd msetprod B*
by (*auto dest*: *dvd-mult-imp-div dvd-mult-right*)
with *p add.prems* **have** $p \in\# A$ $p \in\# B$
by (*simp-all add*: *prime-dvd-mset-prod-iff*)

then obtain A' B' **where** *ABp*: $A = \{\#p\#\} + A'$ $B = \{\#p\#\} + B'$
by (*auto dest*!: *multi-member-split simp add*: *ac-simps*)
with *add.prems prems p* **have** *msetprod C dvd msetprod* (A' #∩ B')
by (*auto intro*: *add.hyps simp add*: *ac-simps*)
with *p* **have** *msetprod* ({#*p*#} + *C*) *dvd msetprod* (({#*p*#} + A') #∩ ({#*p*#} + B'))
by (*simp add*: *inter-union-distrib-right* [*symmetric*])
then show *?case* **by** (*simp add*: *ABp ac-simps*)
qed
with ‹$a \neq 0$› ‹$b \neq 0$› *that fact* **have** *normalize c dvd gcd a b*
by (*simp add*: *norm* [*symmetric*] *gcd-unfold fact*)
then show *?thesis* **by** *simp*
qed
show *normalize* (*gcd a b*) = *gcd a b*
apply (*simp add*: *gcd-unfold*)
apply *safe*
apply (*rule normalized-msetprodI*)
apply (*auto elim*: *factorizationE*)
done
show *lcm a b* = *normalize* ($a * b$) *div gcd a b*
by (*auto elim*!: *factorizationE simp add*: *gcd-unfold lcm-unfold normalize-mult union-diff-inter-eq-sup* [*symmetric*] *msetprod-diff inter-subset-eq-union*)
qed

end

instantiation *nat* :: *factorial-semiring*
begin

definition *is-prime-nat* :: *nat* ⇒ *bool*
where
is-prime-nat p ⟷ ($1 < p \wedge (\forall n.\ n\ dvd\ p \longrightarrow n = 1 \vee n = p)$)

lemma *is-prime-eq-prime*:
is-prime = *prime*
by (*simp add*: *fun-eq-iff prime-def is-prime-nat-def*)

instance proof
show ¬ *is-prime* (*0*::*nat*) **by** (*simp add*: *is-prime-nat-def*)
show ¬ *is-unit p* **if** *is-prime p* **for** *p* :: *nat*
using *that* **by** (*simp add*: *is-prime-nat-def*)
next
fix *p* :: *nat*
assume $p \neq 0$ **and** ¬ *is-unit p*
then have $p > 1$ **by** *simp*
assume *P*: $\bigwedge n.\ n\ dvd\ p \Longrightarrow \neg\ is\text{-}unit\ n \Longrightarrow p\ dvd\ n$
have $n = 1$ **if** *n dvd p* $n \neq p$ **for** *n*
proof (*rule ccontr*)
assume $n \neq 1$

with *that P* **have** *p dvd n* **by** *auto*
with ‹*n dvd p*› **have** *n* = *p* **by** (*rule dvd-antisym*)
with *that* **show** *False* **by** *simp*
qed
with ‹*p* > *1*› **show** *is-prime p* **by** (*auto simp add*: *is-prime-nat-def*)
next
fix *p m n* :: *nat*
assume *is-prime p*
then have *prime p* **by** (*simp add*: *is-prime-eq-prime*)
moreover assume *p dvd m* ∗ *n*
ultimately show *p dvd m* ∨ *p dvd n*
by (*rule prime-dvd-mult-nat*)
next
fix *n* :: *nat*
show *is-unit n* **if** ⋀*m*. *m dvd n* ⟹ ¬ *is-prime m*
using *that prime-factor-nat* **by** (*auto simp add*: *is-prime-eq-prime*)
qed *simp*

end

instantiation *int* :: *factorial-semiring*
begin

definition *is-prime-int* :: *int* ⇒ *bool*
where
is-prime-int p ⟷ *is-prime* (*nat* |*p*|)

lemma *is-prime-int-iff* [*simp*]:
is-prime (*int n*) ⟷ *is-prime n*
by (*simp add*: *is-prime-int-def*)

lemma *is-prime-nat-abs-iff* [*simp*]:
is-prime (*nat* |*k*|) ⟷ *is-prime k*
by (*simp add*: *is-prime-int-def*)

instance proof
show ¬ *is-prime* (*0*::*int*) **by** (*simp add*: *is-prime-int-def*)
show ¬ *is-unit p* **if** *is-prime p* **for** *p* :: *int*
using *that is-prime-not-unit* [*of nat* |*p*|] **by** *simp*
next
fix *p* :: *int*
assume *P*: ⋀*k*. *k dvd p* ⟹ ¬ *is-unit k* ⟹ *p dvd k*
have *nat* |*p*| *dvd n* **if** *n dvd nat* |*p*| **and** *n* ≠ *Suc 0* **for** *n* :: *nat*
proof −
from *that* **have** *int n dvd p* **by** (*simp add*: *int-dvd-iff*)
moreover from *that* **have** ¬ *is-unit* (*int n*) **by** *simp*
ultimately have *p dvd int n* **by** (*rule P*)
with *that* **have** *p dvd int n* **by** *auto*
then show *?thesis* **by** (*simp add*: *dvd-int-iff*)

```
  qed
  moreover assume p ≠ 0 and ¬ is-unit p
  ultimately have is-prime (nat |p|) by (intro is-primeI) auto
  then show is-prime p by simp
next
  fix p k l :: int
  assume is-prime p
  then have *: is-prime (nat |p|) by simp
  assume p dvd k * l
  then have nat |p| dvd nat |k * l|
    by (simp add: dvd-int-unfold-dvd-nat)
  then have nat |p| dvd nat |k| * nat |l|
    by (simp add: abs-mult nat-mult-distrib)
  with * have nat |p| dvd nat |k| ∨ nat |p| dvd nat |l|
    using is-primeD [of nat |p|] by auto
  then show p dvd k ∨ p dvd l
    by (simp add: dvd-int-unfold-dvd-nat)
next
  fix k :: int
  assume P: ⋀l. l dvd k ⟹ ¬ is-prime l
  have is-unit (nat |k|)
  proof (rule no-prime-divisorsI)
    fix m
    assume m dvd nat |k|
    then have int m dvd k by (simp add: int-dvd-iff)
    then have ¬ is-prime (int m) by (rule P)
    then show ¬ is-prime m by simp
  qed
  then show is-unit k by simp
qed simp

end

end
```